Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

Dinkum C++ Library Reference

A C++ program can call on a large number of functions from the **Standard C++ library**. These functions perform essential services such as input and output. They also provide efficient implementations of frequently used operations. Numerous function and class definitions accompany these functions to help you to make better use of the library. Most of the information about the Standard C++ library can be found in the descriptions of the **Standard C++ headers** that declare or define library entities for the program.

Table of Contents

Of the 51 Standard C++ library headers, 13 constitute the **Standard Template Library**, or **STL**. These are indicated below with the notation (STL):

<algorithm> -- (STL) for defining numerous templates that implement useful algorithms
<bitset> -- for defining a template class that administers sets of bits
<cassert> -- for enforcing assertions when functions execute
<cctype> -- for classifying characters
<cerrno> -- for testing error codes reported by library functions
<cfloat> -- for testing floating-point type properties
<ciso646> -- for programming in ISO 646 variant character sets
<climits> -- for testing integer type properties
<clocale> -- for adapting to different cultural conventions
<cmath> -- for computing common mathematical functions
<complex> -- for defining a template class that supports complex arithmetic
<csetjmp> -- for executing nonlocal *goto* statements
<csignal> -- for controlling various exceptional conditions
<cstdarg> -- for accessing a varying number of arguments
<cstddef> -- for defining several useful types and macros
<cstdio> -- for performing input and output
<cstdlib> -- for performing a variety of operations
<cstring> -- for manipulating several kinds of strings
<ctime> -- for converting between various time and date formats
<cwchar> -- for manipulating wide streams and several kinds of strings
<cwctype> -- for classifying wide characters
<deque> -- (STL) for defining a template class that implements a deque container
<exception> -- for defining several functions that control exception handling
<fstream> -- for defining several iostreams template classes that manipulate exteral files
<functional> -- (STL) for defining several templates that help construct predicates for the templates defined in <algorithm> and <numeric>
<iomanip> -- for declaring several iostreams manipulators that take an argument
<ios> -- for defining the template class that serves as the base for many iostreams classes
<iosfwd> -- for declaring several iostreams template classes before they are necessarily defined
<iostream> -- for declaring the iostreams objects that manipulate the standard streams
<istream> -- for defining the template class that performs extractions
<iterator> -- (STL) for defining several templates that help define and manipulate iterators
<limits> -- for testing numeric type properties
<list> -- (STL) for defining a template class that implements a list container
<locale> -- for defining several classes and templates that control locale-specific behavior, as in the iostreams classes
<map> -- (STL) for defining template classes that implement associative containers
<memory> -- (STL) for defining several templates that allocate and free storage for various container classes
<new> -- for declaring several functions that allocate and free storage
<numeric> -- (STL) for defining several templates that implement useful numeric functions

<ostream> -- for defining the template class that performs insertions
<queue> -- (STL) for defining a template class that implements a queue container
<set> -- (STL) for defining template classes that implement associative containers with unique elements
<sstream> -- for defining several iostreams template classes that manipulate string containers
<stack> -- (STL) for defining a template class that implements a stack container
<stdexcept> -- for defining several classes useful for reporting exceptions
<streambuf> -- for defining template classes that buffer iostreams operations
<string> -- for defining a template class that implements a string container
<strstream> -- for defining several iostreams classes that manipulate in-memory character sequences
<typeinfo> -- for defining class `type_info`, the result of the `typeid` operator
<utility> -- (STL) for defining several templates of general utility
<valarray> -- for defining several classes and template classes that support value-oriented arrays
<vector> -- (STL) for defining a template class that implements a vector container

The Standard C++ library also includes the 18 headers from the **Standard C library**, sometimes with small alterations:

<assert.h> -- for enforcing assertions when functions execute
<ctype.h> -- for classifying characters
<errno.h> -- for testing error codes reported by library functions
<float.h> -- for testing floating-point type properties
<iso646.h> -- for programming in ISO 646 variant character sets
<limits.h> -- for testing integer type properties
<locale.h> -- for adapting to different cultural conventions
<math.h> -- for computing common mathematical functions
<setjmp.h> -- for executing nonlocal *goto* statements
<signal.h> -- for controlling various exceptional conditions
<stdarg.h> -- for accessing a varying number of arguments
<stddef.h> -- for defining several useful types and macros
<stdio.h> -- for performing input and output
<stdlib.h> -- for performing a variety of operations
<string.h> -- for manipulating several kinds of strings
<time.h> -- for converting between various time and date formats
<wchar.h> -- for manipulating wide streams and several kinds of strings
<wctype.h> -- for classifying wide characters

Finally, in this implementation, the Standard C++ library also includes four headers for compatibility with traditional C++ libraries:

<fstream.h> -- for defining several iostreams template classes that manipulate exteral files
<iomanip.h> -- for declaring several iostreams manipulators that take an argument
<iostream.h> -- for declaring the iostreams objects that manipulate the standard streams

<new.h> -- for declaring several functions that allocate and free storage
<stl.h> -- for declaring several template classes that aid migration from older versions of the Standard Template Library

Other information on the Standard C++ library includes:

C++ Library Overview -- how to use the Standard C++ library
C Library Overview -- how to use the Standard C library, including what happens at program startup and at program termination
Characters -- how to write character constants and string literals, and how to convert between multibyte characters and wide characters
Files and Streams -- how to read and write data between the program and files
Formatted Output -- how to generate text under control of a format string
Formatted Input -- how to scan and parse text under control of a format string
STL Conventions -- how to read the descriptions of STL template classes and functions
Containers -- how to use an arbitrary STL container template class

A few special conventions are introduced into this document specifically for this particular **implementation** of the Standard C++ library. Because the draft C++ Standard is still changing, not all implementations support all the features described here. Hence, this implementation introduces macros, or alternative declarations, where necessary to provide reasonable substitutes for the capabilities required by the current draft C++ Standard.

See also the **Index**.

C++ Library Overview

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

All Standard C++ library entities are declared or defined in one or more standard headers. To make use of a library entity in a program, write an *include* directive that names the relevant standard header. The full set of 51 **Standard C++ headers** (along with the 18 additional Standard C headers) constitutes a **hosted implementation** of Standard C++: `<algorithm>`, `<bitset>`, `<cassert>`, `<cctype>`, `<cerrno>`, `<cfloat>`, `<ciso646>`, `<climits>`, `<clocale>`, `<cmath>`, `<complex>`, `<csetjmp>`, `<csignal>`, `<cstdarg>`, `<cstddef>`, `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, `<cwchar>`, `<cwctype>`, `<deque>`, `<exception>`, `<fstream>`, `<functional>`, `<iomanip>`, `<ios>`, `<iosfwd>`, `<iostream>`, `<istream>`, `<iterator>`, `<limits>`, `<list>`, `<locale>`, `<map>`, `<memory>`, `<new>`, `<numeric>`, `<ostream>`, `<queue>`, `<set>`, `<sstream>`, `<stack>`, `<stdexcept>`, `<streambuf>`, `<string>`, `<strstream>`, `<typeinfo>`, `<utility>`, `<valarray>`, and `<vector>`.

A **freestanding implementation** of Standard C++ provides only a subset of these headers: `<cstddef>`, `<cstdlib>` (declaring at least the functions `abort`, `atexit`, and `exit`), `<exception>`, `<limits>`, `<new>`, `<typeinfo>`, and `<cstdarg>`.

The Standard C++ headers have two broader subdivisions, iostreams headers and STL headers.

Using Standard C++ Headers

You include the contents of a standard header by naming it in an *include* directive, as in:

```
#include <iostream>  /* include I/O facilities */
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A Standard C++ header includes any other Standard C++ headers it needs to define needed types. (Always include explicitly any Standard C++ headers needed in a translation unit, however, lest you guess wrong about its actual dependencies.) A Standard C header never includes another standard header.

A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. Unlike in Standard C, the standard header never provides a masking macro, with the same name as the function, that masks the function declaration and achieves the same effect.

If an implementation supports **namespaces**, all names in the Standard C++ headers are defined in the **`std`** namespace. You refer to the name `cin`, for example, as `std::cin`. Alternatively, you can write the declaration:

```
using namespace std;
```

which promotes all library names into the current namespace. If you include one of the C standard headers, such as `<stdio.h>`, the individual names declared or defined in that header are promoted for you. Note that macro names are not subject to the rules for nesting namespaces.

C++ Library Conventions

The Standard C++ library obeys much the same conventions as the Standard C library, plus a few more outlined here.

Except for macro names, which obey no scoping rules, all names in the Standard C++ library are declared in the **`std` namespace**. Including a Standard C++ header does *not* introduce any library names into the current namespace. You must, for example, refer to the standard input stream `cin` as `std::cin`, even after including the header `<iostream>` that declares it. Alternatively, you can incorporate all members of the `std` namespace into the current namespace by writing:

```
using namespace std;
```

immediately after all *include* directives that name the standard headers. Note that the Standard C headers behave mostly as if they include no namespace declarations. If you include, for example, `<cstdlib>`, you call `std::abort()` to cause abnormal termination, but if you include `<stdlib.h>`, you call `abort()`.

An implementation has certain latitude in how it declares types and functions in the Standard C++ library:

- Names of functions in the Standard C library may have either **`extern "C++"`** or **`extern "C"`** linkage. Include the appropriate Standard C header rather than declare a library entity inline.
- A member function name in a library class may have additional function signatures over those listed in this document. You can be sure that a function call described here behaves as expected, but you cannot reliably take the address of a library member function. (The type may not be what you expect.)
- A library class may have undocumented (non-virtual) base classes. A class documented as derived from another class may, in fact, be derived from that class through other undocumented classes.
- A type defined as a synonym for some integer type may be the same as one of several different

integer types.

- A library function that has no exception specification can throw an arbitrary exception, unless its definition clearly restricts such a possibility.

On the other hand, there are some restrictions you can count on:

- The Standard C library uses no masking macros. Only specific function signatures are reserved, not the names of the functions themselves.
- A library function name outside a class will *not* have additional, undocumented, function signatures. You can reliably take its address.
- Base classes and member functions described as virtual are assuredly virtual, while those described as non-virtual are assuredly non-virtual.
- Two types defined by the Standard C++ library are always different unless this document explicitly suggests otherwise.
- Functions supplied by the library, including the default versions of replaceable functions, can throw *at most* those exceptions listed in any exception specification. (Functions in the Standard C library may propagate an exception, as when `qsort` calls a comparison function that throws an exception, but they do not otherwise throw exceptions.)

Iostreams Conventions

The **iostreams** headers support conversions between text and encoded forms, and input and output to external files: `<fstream>`, `<iomanip>`, `<ios>`, `<iosfwd>`, `<iostream>`, `<istream>`, `<ostream>`, `<sstream>`, `<streambuf>`, and `<strstream>`.

The simplest use of iostreams requires only that you include the header `<iostream>`. You can then extract values from `cin`, to read the standard input. The rules for doing so are outlined in the description of the class `basic_istream`. You can also insert values to `cout`, to write the standard output. The rules for doing so are outlined in the description of the class `basic_ostream`. Format control common to both extractors and insertors is managed by the class `basic_ios`. Manipulating this format information in the guise of extracting and inserting objects is the province of several manipulators.

You can perform the same iostreams operations on files that you open by name, using the classes declared in `<fstream>`. To convert between iostreams and objects of class `basic_string`, use the classes declared in `<sstream>`. And to do the same with C strings, use the classes declared in `<strstream>`.

The remaining headers provide support services, typically of direct interest to only the most advanced users of the iostreams classes.

C++ Program Startup and Termination

A C++ program performs the same operations as does a C program program startup and at program termination, plus a few more outlined here.

Before the target environment calls the function `main`, and after it stores any constant initial values you specify in all objects that have static duration, the program executes any remaining constructors for such static objects. The order of execution is not specified between translation units, but you can nevertheless assume that four iostreams objects are properly initialized for use by these static constructors. These control several text streams:

- **`cin`** -- for standard input
- **`cout`** -- for standard output
- **`cerr`** -- for unbuffered standard error output
- **`clog`** -- for buffered standard error output

You can also use these objects within the destructors called for static objects, during program termination.

As with C, returning from from `main` or calling `exit` calls all functions registered with `atexit` in reverse order of registry. An exception thrown from such a registered function calls `terminate()`.

See also the **Table of Contents** and the **Index**.

C Library Overview

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

Using Standard C Headers · C Library Conventions · · Program Startup and Termination

---

All Standard C library entities are declared or defined in one or more **standard headers**. To make use of a library entity in a program, write an *include* directive that names the relevant **standard header**. The full set of 18 Standard C headers constitutes a **hosted implementation**: `<assert.h>`, `<ctype.h>`, `<errno.h>`, `<float.h>`, `<iso646.h>`, `<limits.h>`, `<locale.h>`, `<math.h>`, `<setjmp.h>`, `<signal.h>`, `<stdarg.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<time.h>`, `<wchar.h>`, and `<wctype.h>`.

(The headers `<iso646.h>`, `<wchar.h>`, and `<wctype.h>` are added with **Amendment 1**, an addition to the C Standard published in 1995.)

A **freestanding implementation** of Standard C provides only a subset of these standard headers: `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`. Each freestanding implementation defines:

- how it starts the program
- what happens when the program terminates
- what library functions (if any) it provides

Using Standard C Headers

You include the contents of a standard header by naming it in an *include* directive, as in:

```
#include <stdio.h>  /* include I/O facilities */
```

You can include the standard headers in any order, a standard header more than once, or two or more standard headers that define the same macro or the same type. Do not include a standard header within a declaration. Do not define macros that have the same names as keywords before you include a standard header.

A standard header never includes another standard header. A standard header declares or defines only the entities described for it in this document.

Every function in the library is declared in a standard header. The standard header can also provide a **masking macro**, with the same name as the function, that masks the function declaration and achieves the same effect. The macro typically expands to an expression that executes faster than a call to the function of the same name. The macro can, however, cause confusion when you are tracing or debugging

the program. So you can use a standard header in two ways to declare or define a library function. To take advantage of any macro version, include the standard header so that each apparent call to the function can be replaced by a macro expansion.

For example:

```
#include <ctype.h>
char *skip_space(char *p)
    {
    while (isspace(*p))         can be a macro
        ++p;
    return (p);
    }
```

To ensure that the program calls the actual library function, include the standard header and remove any macro definition with an *undef* directive.

For example:

```
#include <ctype.h>
#undef isspace                  remove any macro definition
int f(char *p) {
    while (isspace(*p))         must be a function
        ++p;
```

You can use many functions in the library without including a standard header (although this practice is not recommended). If you do not need defined macros or types to declare and call the function, you can simply declare the function as it appears in this chapter. Again, you have two choices. You can declare the function explicitly.

For example:

```
    double sin(double x);       declared in <math.h>
    y = rho * sin(theta);
```

Or you can declare the function implicitly if it is a function returning *int* with a fixed number of arguments, as in:

```
    n = atoi(str);              declared in <stdlib.h>
```

If the function has a varying number of arguments, such as `printf`, you must declare it explicitly: Either include the standard header that declares it or write an explicit declaration.

Note also that you cannot define a macro or type definition without including its standard header because each of these varies among implementations.

C Library Conventions

A library macro that masks a function declaration expands to an expression that evaluates each of its arguments once (and only once). Arguments that have side effects evaluate the same way whether the expression executes the macro expansion or calls the function. Macros for the functions `getc` and `putc` are explicit exceptions to this rule. Their `stream` arguments can be evaluated more than once. Avoid argument expressions that have side effects with these macros.

A library function that alters a value stored in memory assumes that the function accesses no other objects that overlap the object whose stored value it alters. You cannot depend on consistent behavior from a library function that accesses and alters the same storage via different arguments. The function `memmove` is an explicit exception to this rule. Its arguments can point at objects that overlap.

An implementation has a set of **reserved names** that it can use for its own purposes. All the library names described in this document are, of course, reserved for the library. Don't define macros with the same names. Don't try to supply your own definition of a library function, unless this document explicitly says you can (only in C++). An unauthorized replacement may be successful on some implementations and not on others. Names that begin with two underscores, such as `__STDIO`, and names that begin with an underscore followed by an upper case letter, such as `_Entry`, can be used as macro names, whether or not a translation unit explicitly includes any standard headers. Names that begin with an underscore can be defined with external linkage. Avoid writing such names in a program that you wish to keep maximally portable.

Some library functions operate on **C strings**, or pointers to null-terminated strings. You designate a C string that can be altered by an argument expression that has type *pointer to char* (or type *array of char*, which converts to *pointer to char* in an argument expression). You designate a C string that cannot be altered by an argument expression that has type *pointer to const char* (or type *const array of char*). In any case, the value of the expression is the address of the first byte in an array object. The first successive element of the array that has a null character stored in it marks the end of the C string.

- A **filename** is a string whose contents meet the requirements of the target environment for naming files.
- A **multibyte string** is composed of zero or more multibyte characters, followed by a null character.
- A **wide-character string** is composed of zero or more wide characters (stored in an array of `wchar_t`), followed by a null wide character.

If an argument to a library function has a pointer type, then the value of the argument expression must be a valid address for an object of its type. This is true even if the library function has no need to access an object by using the pointer argument. An explicit exception is when the description of the library function spells out what happens when you use a null pointer.

Some examples are:

```
strcpy(s1, 0)              is INVALID
```

```
memcpy(s1, 0, 0)          is UNSAFE
realloc(0, 50)            is the same as malloc(50)
```

Program Startup and Termination

The target environment controls the execution of the program (in contrast to the translator part of the implementation, which prepares the parts of the program for execution). The target environment passes control to the program at **program startup** by calling the function **`main`** that you define as part of the program. **Program arguments** are C strings that the target environment provides, such as text from the **command line** that you type to invoke the program. If the program does not need to access program arguments, you can define `main` as:

```
extern int main(void)
    { <body of main> }
```

If the program uses program arguments, you define `main` as:

```
extern int main(int argc, char **argv)
    { <body of main> }
```

You can omit either or both of `extern int`, since these are the default storage class and type for a function definition. For program arguments:

- `argc` is a value (always greater than zero) that specifies the number of program arguments.
- `argv[0]` designates the first element of an array of C strings. `argv[argc]` designates the last element of the array, whose stored value is a null pointer.

For example, if you invoke a program by typing:

```
echo hello
```

a target environment can call `main` with:

- The value 2 for `argc`.
- The address of an array object containing `"echo"` stored in `argv[0]`.
- The address of an array object containing `"hello"` stored in `argv[1]`.
- A null pointer stored in `argv[2]`.

`argv[0]` is the name used to invoke the program. The target environment can replace this name with a null string (`""`). The program can alter the values stored in `argc`, in `argv`, and in the array objects whose addresses are stored in `argv`.

Before the target environment calls `main`, it stores the initial values you specify in all objects that have static duration. It also opens three **standard streams**, controlled by the text-stream objects designated by the macros:

- **`stdin`** -- for **standard input**

- `stdout` -- for **standard output**
- `stderr` -- for **standard error** output

If `main` returns to its caller, the target environment calls `exit` with the value returned from `main` as the status argument to `exit`. If the *return* statement that the program executes has no expression, the status argument is undefined. This is the case if the program executes the implied *return* statement at the end of the function definition.

You can also call `exit` directly from any expression within the program. In both cases, `exit` calls all functions registered with `atexit` in reverse order of registry and then begins **program termination**. At program termination, the target environment closes all open files, removes any temporary files that you created by calling `tmpfile`, and then returns control to the invoker, using the status argument value to determine the termination status to report for the program.

The program can terminate abnormally by calling `abort`, for example. Each implementation defines whether it closes files, whether it removes temporary files, and what termination status it reports when a program terminates abnormally.

See also the **Table of Contents** and the **Index**.

Preprocessing

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

The translator processes each source file in a series of phases. **Preprocessing** constitutes the earliest phases, which produce a translation unit. Preprocessing treats a source file as a sequence of text lines. You can specify **directives** and **macros** that insert, delete, and alter source text.

This document describes briefly just those aspect of preprocessing most relevant to the use of the Standard C library:

The macro **__FILE__** expands to a string literal that gives the remembered filename of the current source file. You can alter the value of this macro by writing a *line* directive.

The macro **__LINE__** expands to a decimal integer constant that gives the remembered line number within the current source file. You can alter the value of this macro by writing a *line* directive.

A ***define* directive** defines a name as a macro. Following the directive name `define`, you write one of two forms:

- a name *not* immediately followed by a left parenthesis, followed by any sequence of preprocessing tokens -- to define a macro without parameters
- a name immediately followed by a left parenthesis with *no* intervening white space, followed by zero or more distinct *parameter names* separated by commas, followed by a right parenthesis, followed by any sequence of preprocessing tokens -- to define a macro with as many parameters as names that you write inside the parentheses

You can selectively skip groups of lines within source files by writing an ***if* directive**, or one of the other **conditional directives**, *ifdef* or *ifndef*. You follow the conditional directive by the first group of lines that you want to selectively skip. Zero or more *elif* directives follow this first group of lines, each followed by a group of lines that you want to selectively skip. An optional *else* directive follows all groups of lines controlled by *elif* directives, followed by the last group of lines you want to selectively skip. The last group of lines ends with an *endif* directive.

At most one group of lines is retained in the translation unit -- the one immediately preceded by a directive whose if expression has a nonzero value. For the directive:

```
#ifdef X
```

this expression is `defined (X)`, and for the directive:

```
#ifndef X
```

this expression is `!defined (X)`.

An **if expression** is a conditional expression that the preprocessor evaluates. You can write only integer

constant expressions, with the following additional considerations:

- The expression `defined X`, or `defined (X)`, is replaced by 1 if `X` is defined as a macro, otherwise 0.
- You cannot write the *sizeof* or *type cast* operators. (The translator expands all macro names, then replaces each remaining name with 0, before it recognizes keywords.)
- The translator may be able to represent a broader range of integers than the target environment.
- The translator represents type *int* the same as *long*, and *unsigned int* the same as *unsigned long*.
- The translator can translate character constants to a set of code values different from the set for the target environment.

An **include directive** includes the contents of a standard header or another source file in a translation unit. The contents of the specified standard header or source file replace the *include* directive. Following the directive name `include`, write one of the following:

- a standard header name between angle brackets
- a filename between double quotes
- any other form that expands to one of the two previous forms after macro replacement

A ***line* directive** alters the source line number and filename used by the predefined macros `__FILE__` and `__FILE__`. Following the directive name `line`, write one of the following:

- a decimal integer (giving the new line number of the line following)
- a decimal integer as before, followed by a string literal (giving the new line number and the new source filename)
- any other form that expands to one of the two previous forms after macro replacement

Preprocessing translates each source file in a series of distinct **phases**. The first few phases of translation: terminate each line with a newline character (*NL*), convert trigraphs to their single-character equivalents, and concatenate each line ending in a backslash (\) with the line following. Later phases process include directives, expand macros, and so on to produce a **translation unit**. The translator combines separate translation units, with contributions as needed from the Standard C library, at **link time**, to form the executable **program**.

An ***undef* directive** removes a macro definition. You might want to remove a macro definition so that you can define it differently with a *define* directive or to unmask any other meaning given to the name. The name whose definition you want to remove follows the directive name `undef`. If the name is not currently defined as a macro, the *undef* directive has no effect.

See also the **Table of Contents** and the **Index**.

Files and Streams

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

Text and Binary Streams · **Byte and Wide Streams** · **Controlling Streams** · **Stream States**

A program communicates with the target environment by reading and writing **files** (ordered sequences of bytes). A file can be, for example, a data set that you can read and write repeatedly (such as a disk file), a stream of bytes generated by a program (such as a pipeline), or a stream of bytes received from or sent to a peripheral device (such as the keyboard or display). The latter two are **interactive files**. Files are typically the principal means by which to interact with a program.

You manipulate all these kinds of files in much the same way -- by calling library functions. You include the standard header `<stdio.h>` to declare most of these functions.

Before you can perform many of the operations on a file, the file must be **opened**. Opening a file associates it with a **stream**, a data structure within the Standard C library that glosses over many differences among files of various kinds. The library maintains the state of each stream in an object of type **`FILE`**.

The target environment opens three files prior to program startup. You can open a file by calling the library function `fopen` with two arguments. The first argument is a filename, a multibyte string that the target environment uses to identify which file you want to read or write. The second argument is a C string that specifies:

- whether you intend to read data from the file or write data to it or both
- whether you intend to generate new contents for the file (or create a file if it did not previously exist) or leave the existing contents in place
- whether writes to a file can alter existing contents or should only append bytes at the end of the file
- whether you want to manipulate a text stream or a binary stream

Once the file is successfully opened, you can then determine whether the stream is **byte oriented** (a **byte stream**) or **wide oriented** (a **wide stream**). Wide-oriented streams are supported only with Amendment 1. A stream is initially **unbound**. Calling certain functions to operate on the stream makes it byte oriented, while certain other functions make it wide oriented. Once established, a stream maintains its orientation until it is closed by a call to `fclose` or `freopen`.

Text and Binary Streams

A **text stream** consists of one or more **lines** of text that can be written to a text-oriented display so that they can be read. When reading from a text stream, the program reads an *NL* (newline) at the end of each line. When writing to a text stream, the program writes an *NL* to signal the end of a line. To match differing conventions among target environments for representing text in files, the library functions can alter the number and representations of characters transmitted between the program and a text stream.

Thus, positioning within a text stream is limited. You can obtain the current file-position indicator by calling `fgetpos` or `ftell`. You can position a text stream at a position obtained this way, or at the beginning or end of the stream, by calling `fsetpos` or `fseek`. Any other change of position might well be not supported.

For maximum portability, the program should not write:

- empty files
- *space* characters at the end of a line
- partial lines (by omitting the *NL* at the end of a file)
- characters other than the printable characters, *NL*, and *HT* (horizontal tab)

If you follow these rules, the sequence of characters you read from a text stream (either as byte or multibyte characters) will match the sequence of characters you wrote to the text stream when you created the file. Otherwise, the library functions can remove a file you create if the file is empty when you close it. Or they can alter or delete characters you write to the file.

A **binary stream** consists of one or more bytes of arbitrary information. You can write the value stored in an arbitrary object to a (byte-oriented) binary stream and read exactly what was stored in the object when you wrote it. The library functions do not alter the bytes you transmit between the program and a binary stream. They can, however, append an arbitrary number of null bytes to the file that you write with a binary stream. The program must deal with these additional null bytes at the end of any binary stream.

Thus, positioning within a binary stream is well defined, except for positioning relative to the end of the stream. You can obtain and alter the current file-position indicator the same as for a text stream. Moreover, the offsets used by `ftell` and `fseek` count bytes from the beginning of the stream (which is byte zero), so integer arithmetic on these offsets yields predictable results.

Byte and Wide Streams

A **byte stream** treats a file as a sequence of bytes. Within the program, the stream looks like the same sequence of bytes, except for the possible alterations described above.

By contrast, a **wide stream** treats a file as a sequence of **generalized multibyte characters**, which can have a broad range of encoding rules. (Text and binary files are still read and written as described above.) Within the program, the stream looks like the corresponding sequence of wide characters. Conversions between the two representations occur within the Standard C library. The conversion rules can, in

principle, be altered by a call to `setlocale` that alters the category `LC_CTYPE`. Each wide stream determines its conversion rules at the time it becomes wide oriented, and retains these rules even if the category `LC_CTYPE` subsequently changes.

Positioning within a wide stream suffers the same limitations as for text streams. Moreover, the file-position indicator may well have to deal with a state-dependent encoding. Typically, it includes both a byte offset within the stream and an object of type `mbstate_t`. Thus, the only reliable way to obtain a file position within a wide stream is by calling `fgetpos`, and the only reliable way to restore a position obtained this way is by calling `fsetpos`.

Controlling Streams

`fopen` returns the address of an object of type `FILE`. You use this address as the `stream` argument to several library functions to perform various operations on an open file. For a byte stream, all input takes place as if each character is read by calling `fgetc`, and all output takes place as if each character is written by calling `fputc`. For a wide stream (with Amendment 1), all input takes place as if each character is read by calling `fgetwc`, and all output takes place as if each character is written by calling `fputwc`.

You can **close** a file by calling `fclose`, after which the address of the `FILE` object is invalid.

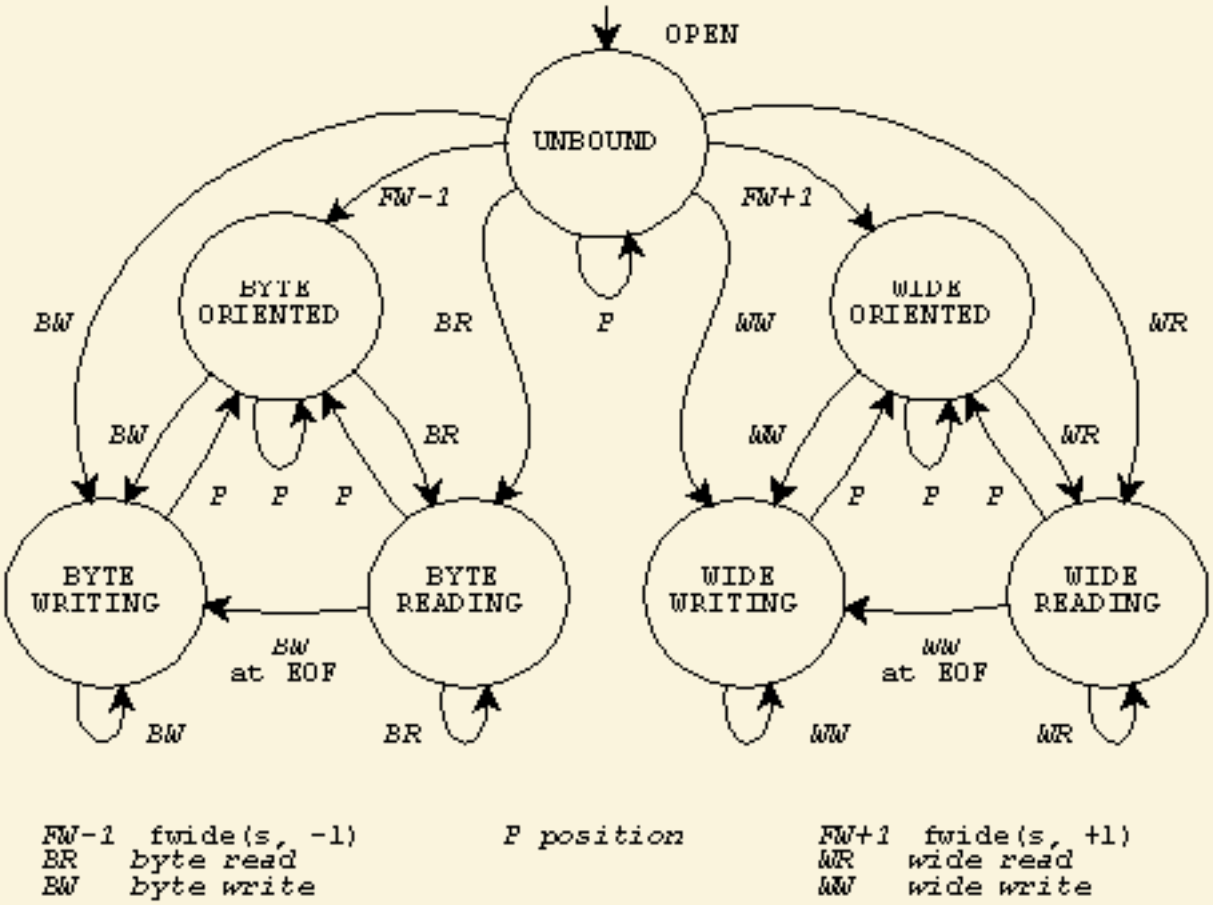
A `FILE` object stores the state of a stream, including:

- an **error indicator** -- set nonzero by a function that encounters a read or write error
- an **end-of-file indicator** -- set nonzero by a function that encounters the end of the file while reading
- a **file-position indicator** -- specifies the next byte in the stream to read or write, if the file can support positioning requests
- a **stream state** -- specifies whether the stream will accept reads and/or writes and, with Amendment 1, whether the stream is unbound, byte oriented, or wide oriented
- a **conversion state** -- remembers the state of any partly assembled or generated generalized multibyte character, as well as any shift state for the sequence of bytes in the file)
- a **file buffer** -- specifies the address and size of an array object that library functions can use to improve the performance of read and write operations to the stream

Do not alter any value stored in a `FILE` object or in a file buffer that you specify for use with that object. You cannot copy a `FILE` object and portably use the address of the copy as a `stream` argument to a library function.

Stream States

The valid states, and state transitions, for a stream are:



Each of the circles denotes a stable state. Each of the lines denotes a transition that can occur as the result of a function call that operates on the stream. Five groups of functions can cause state transitions.

Functions in the first three groups are declared in `<stdio.h>`:

- the **byte read functions** -- `fgetc`, `fgets`, `fread`, `fscanf`, `getc`, `getchar`, `gets`, `scanf`, and `ungetc`
- the **byte write functions** -- `fprintf`, `fputc`, `fputs`, `fwrite`, `printf`, `putc`, `putchar`, `puts`, `vfprintf`, and `vprintf`
- the **position functions** -- `fflush`, `fseek`, `fsetpos`, and `rewind`

Functions in the remaining two groups are declared in `<wchar.h>`:

- the **wide read functions** -- `fgetwc`, `fgetws`, `fwscanf`, `getwc`, `getwchar`, `ungetwc`, and `wscanf`,
- the **wide write functions** -- `fwprintf`, `fputwc`, `fputws`, `putwc`, `putwchar`, `vfwprintf`, `vwprintf`, and `wprintf`,

For the stream `s`, the call `fwide(s, 0)` is always valid and never causes a change of state. Any other call to `fwide`, or to any of the five groups of functions described above, causes the state transition

shown in the state diagram. If no such transition is shown, the function call is invalid.

The state diagram shows how to establish the orientation of a stream:

- The call `fwide(s, -1)`, or to a byte read or byte write function, establishes the stream as byte oriented.
- The call `fwide(s, 1)`, or to a wide read or wide write function, establishes the stream as wide oriented.

The state diagram also shows that you must call one of the position functions between most write and read operations:

- You cannot call a read function if the last operation on the stream was a write.
- You cannot call a write function if the last operation on the stream was a read, unless that read operation set the end-of-file indicator.

Finally, the state diagram shows that a position operation never *decreases* the number of valid function calls that can follow.

See also the **Table of Contents** and the **Index**.

<stdio.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define _IOFBF <integer constant expression>
#define _IOLBF <integer constant expression>
#define _IONBF <integer constant expression>
#define BUFSIZ <integer constant expression >= 256>
#define EOF <integer constant expression < 0>
typedef o-type FILE;
#define FILENAME_MAX <integer constant expression > 0>
#define FOPEN_MAX <integer constant expression >= 8>
#define L_tmpnam <integer constant expression > 0>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define SEEK_CUR <integer constant expression>
#define SEEK_END <integer constant expression>
#define SEEK_SET <integer constant expression>
#define TMP_MAX <integer constant expression >= 25>
void clearerr(FILE *stream);
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
int fgetc(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
char *fgets(char *s, int n, FILE *stream);
FILE *fopen(const char *filename, const char *mode);
typedef o-type fpos_t;
int fprintf(FILE *stream, const char *format, ...);
```

```
int fputc(int c, FILE *stream);
int fputs(const char *s, FILE *stream);
size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);
FILE *freopen(const char *filename, const char *mode, FILE *stream);
int fscanf(FILE *stream, const char *format, ...);
int fseek(FILE *stream, long offset, int mode);
int fsetpos(FILE *stream, const fpos_t *pos);
long ftell(FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nelem, FILE
*stream);
int getc(FILE *stream);
int getchar(void);
char *gets(char *s);
void perror(const char *s);
int printf(const char *format, ...);
int putc(int c, FILE *stream);
int putchar(int c);
int puts(const char *s);
int remove(const char *filename);
int rename(const char *old, const char *new);
void rewind(FILE *stream);
int scanf(const char *format, ...);
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
typedef ui-type size_t;
int sprintf(char *s, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
#define stderr <pointer to FILE rvalue>
#define stdin <pointer to FILE rvalue>
#define stdout <pointer to FILE rvalue>
FILE *tmpfile(void)
char *tmpnam(char *s);
int ungetc(int c, FILE *stream);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vprintf(const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

Include the standard header **<stdio.h>** so that you can perform input and output operations on streams and files.

_IOFBF

`#define _IOFBF <integer constant expression>`

The macro yields the value of the `mode` argument to `setvbuf` to indicate **full buffering**. (Flush the stream buffer only when it fills.)

_IOLBF

`#define _IOLBF <integer constant expression>`

The macro yields the value of the `mode` argument to `setvbuf` to indicate **line buffering**. (Flush the stream buffer at the end of a text line.)

_IONBF

`#define _IONBF <integer constant expression>`

The macro yields the value of the `mode` argument to `setvbuf` to indicate **no buffering**. (Flush the stream buffer at the end of each write operation.)

BUFSIZ

`#define BUFSIZ <integer constant expression >= 256>`

The macro yields the size of the stream buffer used by `setbuf`.

EOF

`#define EOF <integer constant expression < 0>`

The macro yields the return value used to signal the end of a stream or to report an error condition.

FILE

`typedef o-type FILE;`

The type is an object type *`o-type`* that stores all control information for a stream. The functions `fopen` and `freopen` allocate all `FILE` objects used by the read and write functions.

FILENAME_MAX

```
#define FILENAME_MAX <integer constant expression > 0>
```

The macro yields the maximum size array of characters that you must provide to hold a filename.

FOPEN_MAX

```
#define FOPEN_MAX <integer constant expression >= 8>
```

The macro yields the maximum number of files that the target environment permits to be simultaneously open (including `stderr`, `stdin`, and `stdout`).

L_tmpnam

```
#define L_tmpnam <integer constant expression > 0>
```

The macro yields the number of characters that the target environment requires for representing temporary filenames created by `tmpnam`.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

SEEK_CUR

```
#define SEEK_CUR <integer constant expression>
```

The macro yields the value of the `mode` argument to `fseek` to indicate seeking relative to the current file-position indicator.

SEEK_END

```
#define SEEK_END <integer constant expression>
```

The macro yields the value of the `mode` argument to `fseek` to indicate seeking relative to the end of the file.

SEEK_SET

```
#define SEEK_SET <integer constant expression>
```

The macro yields the value of the `mode` argument to `fseek` to indicate seeking relative to the beginning of the file.

TMP_MAX

```
#define TMP_MAX <integer constant expression >= 25>
```

The macro yields the minimum number of distinct filenames created by the function `tmpnam`.

clearerr

```
void clearerr(FILE *stream);
```

The function clears the end-of-file and error indicators for the stream `stream`.

fclose

```
int fclose(FILE *stream);
```

The function closes the file associated with the stream `stream`. It returns zero if successful; otherwise, it returns `EOF`. `fclose` writes any buffered output to the file, deallocates the stream buffer if it was automatically allocated, and removes the association between the stream and the file. Do not use the value of `stream` in subsequent expressions.

feof

```
int feof(FILE *stream);
```

The function returns a nonzero value if the end-of-file indicator is set for the stream `stream`.

ferror

```
int ferror(FILE *stream);
```

The function returns a nonzero value if the error indicator is set for the stream `stream`.

fflush

```
int fflush(FILE *stream);
```

The function writes any buffered output to the file associated with the stream `stream` and returns zero if successful; otherwise, it returns `EOF`. If `stream` is a null pointer, `fflush` writes any buffered output to all files opened for output.

fgetc

```
int fgetc(FILE *stream);
```

The function reads the next character `c` (if present) from the input stream `stream`, advances the file-position indicator (if defined), and returns `(int)(unsigned char)c`. If the function sets either the end-of-file indicator or the error indicator, it returns `EOF`.

fgetpos

```
int fgetpos(FILE *stream, fpos_t *pos);
```

The function stores the file-position indicator for the stream `stream` in `*pos` and returns zero if successful; otherwise, the function stores a positive value in `errno` and returns a nonzero value.

fgets

```
char *fgets(char *s, int n, FILE *stream);
```

The function reads characters from the input stream `stream` and stores them in successive elements of the array beginning at `s` and continuing until it stores `n-1` characters, stores an *`NL`* character, or sets the end-of-file or error indicators. If `fgets` stores any characters, it concludes by storing a null character in the next element of the array. It returns `s` if it stores any characters and it has not set the error indicator for the stream; otherwise, it returns a null pointer. If it sets the error indicator, the array contents are indeterminate.

fopen

```
FILE *fopen(const char *filename, const char *mode);
```

The function opens the file with the filename `filename`, associates it with a stream, and returns a pointer to the object controlling the stream. If the open fails, it returns a null pointer. The initial characters of `mode` determine how the program manipulates the stream and whether it interprets the stream as text or binary. The initial characters must be one of the following sequences:

- **`"r"`** -- to open an existing text file for reading

- **"w"** -- to create a text file or to open and truncate an existing text file, for writing
- **"a"** -- to create a text file or to open an existing text file, for writing. The file-position indicator is positioned at the end of the file before each write
- **"rb"** -- to open an existing binary file for reading
- **"wb"** -- to create a binary file or to open and truncate an existing binary file, for writing
- **"ab"** -- to create a binary file or to open an existing binary file, for writing. The file-position indicator is positioned at the end of the file (possibly after arbitrary null byte padding) before each write
- **"r+"** -- to open an existing text file for reading and writing
- **"w+"** -- to create a text file or to open and truncate an existing text file, for reading and writing
- **"a+"** -- to create a text file or to open an existing text file, for reading and writing. The file-position indicator is positioned at the end of the file before each write
- **"r+b"** or **"rb+"** -- to open an existing binary file for reading and writing
- **"w+b"** or **"wb+"** -- to create a binary file or to open and truncate an existing binary file, for reading and writing
- **"a+b"** or **"ab+"** -- to create a binary file or to open an existing binary file, for reading and writing. The file-position indicator is positioned at the end of the file (possibly after arbitrary null byte padding) before each write

If you open a file for both reading and writing, the target environment can open a binary file instead of a text file. If the file is not interactive, the stream is fully buffered.

fpos_t

```
typedef o-type fpos_t;
```

The type is an object type *o-type* of an object that you declare to hold the value of a file-position indicator stored by `fsetpos` and accessed by `fgetpos`.

fprintf

```
int fprintf(FILE *stream, const char *format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stream`. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

fputc

```
int fputc(int c, FILE *stream);
```

The function writes the character `(unsigned char)c` to the output stream `stream`, advances the

file-position indicator (if defined), and returns `(int)(unsigned char)c`. If the function sets the error indicator for the stream, it returns EOF.

fputs

```
int fputs(const char *s, FILE *stream);
```

The function accesses characters from the C string `s` and writes them to the output stream `stream`. The function does not write the terminating null character. It returns a nonnegative value if it has not set the error indicator; otherwise, it returns EOF.

fread

```
size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);
```

The function reads characters from the input stream `stream` and stores them in successive elements of the array whose first element has the address `(char *)ptr` until the function stores `size*nelem` characters or sets the end-of-file or error indicator. It returns `n/size`, where `n` is the number of characters it read. If `n` is not a multiple of `size`, the value stored in the last element is indeterminate. If the function sets the error indicator, the file-position indicator is indeterminate.

freopen

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

The function closes the file associated with the stream `stream` (as if by calling `fclose`); then it opens the file with the filename `filename` and associates the file with the stream `stream` (as if by calling `fopen(filename, mode)`). It returns `stream` if the open is successful; otherwise, it returns a null pointer.

fscanf

```
int fscanf(FILE *stream, const char *format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stream`. It returns the number of input items matched and assigned, or it returns EOF if the function does not store values before it sets the end-of-file or error indicator for the stream.

fseek

```
int fseek(FILE *stream, long offset, int mode);
```

The function sets the file-position indicator for the stream `stream` (as specified by `offset` and `mode`), clears the end-of-file indicator for the stream, and returns zero if successful.

For a binary stream, `offset` is a signed offset in bytes:

- If `mode` has the value `SEEK_SET`, `fseek` adds `offset` to the file-position indicator for the beginning of the file.
- If `mode` has the value `SEEK_CUR`, `fseek` adds `offset` to the current file-position indicator.
- If `mode` has the value `SEEK_END`, `fseek` adds `offset` to the file-position indicator for the end of the file (possibly after arbitrary null character padding).

`fseek` sets the file-position indicator to the result of this addition.

For a text stream:

- If `mode` has the value `SEEK_SET`, `fseek` sets the file-position indicator to the value encoded in `offset`, which is either a value returned by an earlier successful call to `ftell` or zero to indicate the beginning of the file.
- If `mode` has the value `SEEK_CUR` and `offset` is zero, `fseek` leaves the file-position indicator at its current value.
- If `mode` has the value `SEEK_END` and `offset` is zero, `fseek` sets the file-position indicator to indicate the end of the file.

The function defines no other combination of argument values.

fsetpos

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

The function sets the file-position indicator for the stream `stream` to the value stored in `*pos`, clears the end-of-file indicator for the stream, and returns zero if successful. Otherwise, the function stores a positive value in `errno` and returns a nonzero value.

ftell

```
long ftell(FILE *stream);
```

The function returns an encoded form of the file-position indicator for the stream `stream` or stores a positive value in `errno` and returns the value -1. For a binary file, a successful return value gives the number of bytes from the beginning of the file. For a text file, target environments can vary on the representation and range of encoded file-position indicator values.

fwrite

```
size_t fwrite(const void *ptr, size_t size, size_t nelem, FILE *stream);
```

The function writes characters to the output stream `stream`, accessing values from successive elements of the array whose first element has the address `(char *)ptr` until the function writes `size*nelem` characters or sets the error indicator. It returns `n/size`, where `n` is the number of characters it wrote. If the function sets the error indicator, the file-position indicator is indeterminate.

getc

```
int getc(FILE *stream);
```

The function has the same effect as `fgetc(stream)` except that a macro version of `getc` can evaluate `stream` more than once.

getchar

```
int getchar(void);
```

The function has the same effect as `fgetc(stdin)`, reading a character from the stream `stdin`

gets

```
char *gets(char *s);
```

The function reads characters from the stream `stdin` and stores them in successive elements of the array whose first element has the address `s` until the function reads an *`NL`* character (which is not stored) or sets the end-of-file or error indicator. If `gets` reads any characters, it concludes by storing a null character in the next element of the array. It returns `s` if it reads any characters and has not set the error indicator for the stream; otherwise, it returns a null pointer. If it sets the error indicator, the array contents are indeterminate. The number of characters that `gets` reads and stores cannot be limited. Use `fgets` instead.

perror

```
void perror(const char *s);
```

The function writes a line of text to the stream `stderr`. If `s` is not a null pointer, the function first writes the C string `s` (as if by calling `fputs(s, stderr)`), followed by a colon (`:`) and a *`space`*. It then writes the same message C string that is returned by `strerror(errno)`, converting the value stored in `errno`, followed by an *`NL`*.

printf

```
int printf(const char *format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stdout`. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

putc

```
int putc(int c, FILE *stream);
```

The function has the same effect as `fputc(c, stream)` except that a macro version of `putc` can evaluate `stream` more than once.

putchar

```
int putchar(int c);
```

The function has the same effect as `fputc(c, stdout)`, writing a character to the stream `stdout`.

puts

```
int puts(const char *s);
```

The function accesses characters from the C string `s` and writes them to the stream `stdout`. The function writes an *`NL`* character to the stream in place of the terminating null character. It returns a nonnegative value if it has not set the error indicator; otherwise, it returns `EOF`.

remove

```
int remove(const char *filename);
```

The function removes the file with the filename `filename` and returns zero if successful. If the file is open when you remove it, the result is implementation defined. After you remove it, you cannot open it as an existing file.

rename

```
int rename(const char *old, const char *new);
```

The function renames the file with the filename `old` to have the filename `new` and returns zero if successful. If a file with the filename `new` already exists, the result is implementation defined. After you

rename it, you cannot open the file with the filename `old`.

rewind

```
void rewind(FILE *stream);
```

The function calls `fseek(stream, 0L, SEEK_SET)` and then clears the error indicator for the stream `stream`.

scanf

```
int scanf(const char *format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stdin`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

setbuf

```
void setbuf(FILE *stream, char *buf);
```

If `buf` is not a null pointer, the function calls `setvbuf(stream, buf, __IOFBF, BUFSIZ)`, specifying full buffering with `_IOFBF` and a buffer size of `BUFSIZ` characters. Otherwise, the function calls `setvbuf(stream, 0, _IONBF, BUFSIZ)`, specifying no buffering with `_IONBF`.

setvbuf

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The function sets the buffering mode for the stream `stream` according to `buf`, `mode`, and `size`. It returns zero if successful. If `buf` is not a null pointer, then `buf` is the address of the first element of an array of *char* of size `size` that can be used as the stream buffer. Otherwise, `setvbuf` can allocate a stream buffer that is freed when the file is closed. For `mode` you must supply one of the following values:

- **`_IOFBF`** -- to indicate full buffering
- **`_IOLBF`** -- to indicate line buffering
- **`_IONBF`** -- to indicate no buffering

You must call `setvbuf` after you call `fopen` to associate a file with that stream and before you call a library function that performs any other operation on the stream.

size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type *ui-type* of an object that you declare to store the result of the *sizeof* operator.

sprintf

```
int sprintf(char *s, const char *format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a null character in the next location of the array. It returns the number of characters generated -- not including the null character.

sscanf

```
int sscanf(const char *s, const char *format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address `s`. It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a null character from the array.

stderr

```
#define stderr <pointer to FILE rvalue>
```

The macro yields a pointer to the object that controls the standard error output stream.

stdin

```
#define stdin <pointer to FILE rvalue>
```

The macro yields a pointer to the object that controls the standard input stream.

stdout

```
#define stdout <pointer to FILE rvalue>
```

The macro yields a pointer to the object that controls the standard output stream.

tmpfile

```
FILE *tmpfile(void)
```

The function creates a temporary binary file with the filename *temp-name* and then has the same effect as calling fopen(*temp-name*, "wb+"). The file *temp-name* is removed when the program closes it, either by calling fclose explicitly or at normal program termination. The filename *temp-name* does not conflict with any filenames that you create. If the open is successful, the function returns a pointer to the object controlling the stream; otherwise, it returns a null pointer.

tmpnam

```
char *tmpnam(char *s);
```

The function creates a unique filename *temp-name* and returns a pointer to the filename. If s is not a null pointer, then s must be the address of the first element of an array at least of size L_tmpnam. The function stores *temp-name* in the array and returns s. Otherwise, if s is a null pointer, the function stores *temp-name* in a static-duration array and returns the address of its first element. Subsequent calls to tmpnam can alter the values stored in this array.

The function returns unique filenames for each of the first TMP_MAX times it is called, after which its behavior is implementation defined. The filename *temp-name* does not conflict with any filenames that you create.

ungetc

```
int ungetc(int c, FILE *stream);
```

If c is not equal to EOF, the function stores (unsigned char)c in the object whose address is stream and clears the end-of-file indicator. If c equals EOF or the store cannot occur, the function returns EOF; otherwise, it returns (unsigned char)c. A subsequent library function call that reads a character from the stream stream obtains this stored value, which is then forgotten.

Thus, you can effectively **push back** a character to a stream after reading a character. (You need not push back the same character that you read.) An implementation can let you push back additional characters before you read the first one. You read the characters in reverse order of pushing them back to the stream. You cannot portably:

- push back more than one character
- push back a character if the file-position indicator is at the beginning of the file
- Call ftell for a text file that has a character currently pushed back

A call to the functions fseek, fsetpos, or rewind for the stream causes the stream to forget any pushed-back characters. For a binary stream, the file-position indicator is decremented for each character that is pushed back.

vfprintf

```
int vfprintf(FILE *stream, const char *format, va_list ap);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stream`. It returns the number of characters generated, or it returns a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

vprintf

```
int vprintf(const char *format, va_list ap);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated character to the stream `stdout`. It returns the number of characters generated, or a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

vsprintf

```
int vsprintf(char *s, const char *format, va_list ap);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a null character in the next location of the array. It returns the number of characters generated -- not including the null character.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

See also the **Table of Contents** and the **Index**.

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

Expressions

You write expressions to determine values, to alter values stored in objects, and to call functions that perform input and output. In fact, you express all computations in the program by writing expressions. The translator must evaluate some of the expressions you write to determine properties of the program. The translator or the target environment must evaluate other expressions prior to program startup to determine the initial values stored in objects with static duration. The program evaluates the remaining expressions when it executes.

This document describes briefly just those aspect of expressions most relevant to the use of the Standard C library:

An **address constant expression** specifies a value that has a pointer type and that the translator or target environment can determine prior to program startup.

A **constant expression** specifies a value that the translator or target environment can determine prior to program startup.

An **integer constant expression** specifies a value that has an integer type and that the translator can determine at the point in the program where you write the expression. (You cannot write a function call, assigning operator, or *comma* operator except as part of the operand of a *sizeof* operator.) In addition, you must write only subexpressions that have integer type. You can, however, write a floating-point constant as the operand of an integer *type cast* operator.

An **lvalue expression** An lvalue expression designates an object that has an object type other than an array type. Hence, you can access the value stored in the object. A *modifiable* lvalue expression designates an object that has an object type other than an array type or a *const* type. Hence, you can alter the value stored in the object. You can also designate objects with an lvalue expression that has an array type or an incomplete type, but you can only take the address of such an expression.

Promoting occurs for an expression whose integer type is not one of the ``computational'' types. Except when it is the operand of the *sizeof* operator, an integer rvalue expression has one of four types: *int*, *unsigned int, long,* or *unsigned long.* When you write an expression in an rvalue context and the expression has an integer type that is not one of these types, the translator *promotes* its type to one of these. If all of the values representable in the original type are also representable as type *int,* then the promoted type is *int.* Otherwise, the promoted type is *unsigned int.* Thus, for *signed char*, *short,* and any *signed bitfield* type, the promoted type is *int.* For each of the remaining integer types (*char, unsigned char*, *unsigned short,* any plain *bitfield* type, or any *unsigned bitfield* type), the effect of these rules is to favor promoting to *int* wherever possible, but to promote to *unsigned int* if necessary to preserve the original value in all possible cases.

An **rvalue expression** is an expression whose value can be determined only when the program executes. The term also applies to expressions which *need not* be determined until program execution.

You use the **sizeof** operator, as in the expression `sizeof X` to determine the size in bytes of an object whose type is the type of `X`. The translator uses the expression you write for `X` only to determine a type; it is not evaluated.

A **void expression** has type *void*.

See also the **Table of Contents** and the **Index**.

<algorithm>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class InIt, class Fun>
    Fun for_each(InIt first, InIt last, Fun f);
template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);
template<class InIt, class Pred>
    InIt find_if(InIt first, InIt last, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt>
    FwdIt adjacent_find(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
```

```
template<class InIt, class T, class Dist>
    iterator_traits<InIt>::distance_type count(InIt first, InIt last,
        const T& val, Dist& n);
template<class InIt, class Pred, class Dist>
    iterator_traits<InIt>::distance_type count_if(InIt first, InIt last,
        Pred pr, Dist& n);
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x, Pred pr);
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
template<class FwdIt1, class FwdIt2>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val, Pred pr);
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 x);
template<class T>
    void swap(T& x, T& y);
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
template<class InIt1, class InIt2, class OutIt, class Binop>
    OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,
        OutIt x, Binop bop);
template<class FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
```

```
        const T& vold, const T& vnew);
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
        Pred pr, const T& val);
template<class InIt, class OutIt, class T>
    OutIt replace_copy(InIt first, InIt last, OutIt x,
        const T& vold, const T& vnew);
template<class InIt, class OutIt, class Pred, class T>
    OutIt replace_copy_if(InIt first, InIt last, OutIt x,
        Pred pr, const T& val);
template<class FwdIt, class T>
    void fill(FwdIt first, FwdIt last, const T& x);
template<class OutIt, class Size, class T>
    void fill_n(OutIt first, Size n, const T& x);
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
template<class OutIt, class Pred, class Gen>
    void generate_n(OutIt first, Dist n, Gen g);
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);
template<class BidIt>
    void reverse(BidIt first, BidIt last);
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
template<class FwdIt>
    void rotate(FwdIt first, FwdIt middle, FwdIt last);
template<class FwdIt, class OutIt>
    OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt x);
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
```

```
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
template<class FwdIt, class Pred>
    FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle, RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle, RanIt last, Pred pr);
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2, Pred pr);
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last, Pred pr);
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
template<class FwdIt, class T>
    pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,
        const T& val, Pred pr);
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last, const T& val);
```

```
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last, const T& val,
        Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle, BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle, BidIt last, Pred pr);
template<class InIt1, class InIt2>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
```

```
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T&  x, const T& y, Pred pr);
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last, Pred pr);
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last, Pred pr);
```

```
};
```

Include the STL standard header **<algorithm>** to define numerous template functions that perform useful algorithms. The descriptions that follow make extensive use of common template parameter names (or prefixes) to indicate the least powerful category of iterator permitted as an actual argument type:

- **OutIt** -- to indicate an output iterator
- **InIt** -- to indicate an input iterator
- **FwdIt** -- to indicate a forward iterator
- **BidIt** -- to indicate a bidirectional iterator
- **RanIt** -- to indicate a random-access iterator

The descriptions of these templates employ a number of conventions common to all algorithms.

adjacent_find

```
template<class FwdIt>
    FwdIt adjacent_find(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest `N` in the range `[0, last - first)` for which `N + 1 != last - first` and the predicate `*(first + N) == *(first + N + 1)` is true. It then returns `first + N`. If no such value exists, the function returns `last`. It evaluates the predicate exactly `N + 1` times.

The second template function behaves the same, except that the predicate is `pr(*(first + N), *(first + N + 1))`.

binary_search

```
template<class FwdIt, class T>
    bool binary_search(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    bool binary_search(FwdIt first, FwdIt last, const T& val,
        Pred pr);
```

The first template function determines whether a value of `N` exists in the range `[0, last - first)` for which `*(first + N)` has equivalent ordering to `val`, where the elements designated by iterators in the range `[first, last)` form a sequence ordered by `operator<`. If so, th function returns true. If no such value exists, it returns false.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate `X < Y` at most `ceil(log(last - first)) + 2` times. Otherwise, the function evaluates the predicate a number of times proportional to `last - first`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

copy

```
template<class InIt, class OutIt>
    OutIt copy(InIt first, InIt last, OutIt x);
```

The template function evaluates `*(x + N) = *(first + N))` once for each `N` in the range `[0, last - first)`, for strictly increasing values of `N` beginning with the lowest value. It then returns `x + N`. If `x` and `first` designate regions of storage, `x` must not be in the range `[first, last)`.

copy_backward

```
template<class BidIt1, class BidIt2>
    BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 x);
```

The template function evaluates `*(x - N - 1) = *(last - N - 1))` once for each `N` in the range `[0, last - first)`, for strictly decreasing values of `N` beginning with the highest value. It then returns `x - (last - first)`. If `x` and `first` designate regions of storage, `x` must not be in the range `[first, last)`.

count

```
template<class InIt, class T>
    iterator_traits<InIt>::distance_type count(InIt first, InIt last,
        const T& val);
```

The template function sets a count `n` to zero. It then executes `++n` for each `N` in the range `[0, last - first)` for which the predicate `*(first + N) == val` is true. The function returns `n`. It evaluates the predicate exactly `last - first` times.

In this implementation, if a translator does not support partial specialization of templates, the return type is `size_t`.

count_if

```
template<class InIt, class Pred, class Dist>
    iterator_traits<InIt>::distance_type count_if(InIt first, InIt last,
        Pred pr, Dist& n);
```

The template function sets a count `n` to zero. It then executes `++n` for each `N` in the range `[0, last - first)` for which the predicate `pr(*(first + N))` is true. It evaluates the predicate exactly `last - first` times.

In this implementation, if a translator does not support partial specialization of templates, the return type is

size_t.

equal

```
template<class InIt1, class InIt2>
    bool equal(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
```

The first template function returns true only if, for each N in the range [0, last1 - first1), the predicate *(first1 + N) == *(first2 + N) is true. The function evaluates the predicate at most once for each N.

The second template function behaves the same, except that the predicate is pr(*(first1 + N), *(first2 + N)).

equal_range

```
template<class FwdIt, class T>
    pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,
        const T& val);
template<class FwdIt, class T, class Pred>
    pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,
        const T& val, Pred pr);
```

The first template function effectively returns pair(lower_bound(first, last, val), upper_bound(first, last, val)), where the elements designated by iterators in the range [first, last) form a sequence ordered by operator<. Thus, the function determines the largest range of positions over which val can be inserted in the sequence and still preserve its ordering.

If FwdIt is a random-access iterator type, the function evaluates the ordering predicate X < Y at most ceil(2 * log(last - first)) + 1. Otherwise, the function evaluates the predicate a number of times proportional to last - first.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

fill

```
template<class FwdIt, class T>
    void fill(FwdIt first, FwdIt last, const T& x);
```

The template function evaluates *(first + N) = x once for each N in the range [0, last - first).

fill_n

```
template<class OutIt, class Size, class T>
    void fill_n(OutIt first, Size n, const T& x);
```

The template function evaluates `*(first + N) = x` once for each `N` in the range `[0, n)`.

find

```
template<class InIt, class T>
    InIt find(InIt first, InIt last, const T& val);
```

The template function determines the lowest value of `N` in the range `[0, last - first)` for which the predicate `*(first + N) == val` is true. It then returns `first + N`. If no such value exists, the function returns `last`. It evaluates the predicate at most once for each `N`.

find_end

```
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the highest value of `N` in the range `[0, last1 - first1 - (last2 - first2))` such that for each `M` in the range `[0, last2 - first2)`, the predicate `*(first1 + N + M) == *(first2 + N + M)` is true. It then returns `first1 + N`. If no such value exists, the function returns `last1`. It evaluates the predicate at most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N + M), *(first2 + N + M))`.

find_first_of

```
template<class FwdIt1, class FwdIt2>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the lowest value of `N` in the range `[0, last1 - first1)` such that for some `M` in the range `[0, last2 - first2)`, the predicate `*(first1 + N) == *(first2 + M)` is true. It then returns `first1 + N`. If no such value exists, the function returns

`last1`. It evaluates the predicate at most `(last1 - first1) * (last2 - first2)` times.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N), *(first2 + M))`.

find_if

```
template<class InIt, class Pred>
    InIt find_if(InIt first, InIt last, Pred pr);
```

The template function determines the lowest value of `N` in the range `[0, last - first)` for which the predicate `pred(*(first + N))` is true. It then returns `first + N`. If no such value exists, the function returns `last`. It evaluates the predicate at most once for each `N`.

for_each

```
template<class InIt, class Fun>
    Fun for_each(InIt first, InIt last, Fun f);
```

The template function evaluates `f(*(first + N))` once for each `N` in the range `[0, last - first)`. It then returns `f`. The call `f(*(first + N))` must not alter `*(first + N)`.

generate

```
template<class FwdIt, class Gen>
    void generate(FwdIt first, FwdIt last, Gen g);
```

The template function evaluates `*(first + N) = g()` once for each `N` in the range `[0, last - first)`.

generate_n

```
template<class OutIt, class Pred, class Gen>
    void generate_n(OutIt first, Dist n, Gen g);
```

The template function evaluates `*(first + N) = g()` once for each `N` in the range `[0, n)`.

includes

```
template<class InIt1, class InIt2>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool includes(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
```

The first template function determines whether a value of `N` exists in the range `[0, last2 - first2)` such that, for each `M` in the range `[0, last1 - first1)`, `*(first + M)` and `*(first + N)` do not have equivalent ordering, where the elements designated by iterators in the ranges `[first1, last1)` and `[first2, last2)` each form a sequence ordered by `operator<`. If so, the function returns false. If no such value exists, it returns true. Thus, the function determines whether the ordered sequence designated by iterators in the range `[first2, last2)` all have equivalent ordering with some element designated by iterators in the range `[first1, last1)`.

The function evaluates the predicate at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

inplace_merge

```
template<class BidIt>
    void inplace_merge(BidIt first, BidIt middle, BidIt last);
template<class BidIt, class Pred>
    void inplace_merge(BidIt first, BidIt middle, BidIt last, Pred pr);
```

The first template function reorders the sequences designated by iterators in the ranges `[first, middle)` and `[middle, last)`, each ordered by `operator<`, to form a merged sequence of length `last - first` beginning at `first` also ordered by `operator<`. The merge occurs without altering the relative order of elements within either original sequence. Moreover, for any two elements from different original sequences that have equivalent ordering, the element from the ordered range `[first, middle)` precedes the other.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(last - first))` times. (Given enough temporary storage, it can evaluate the predicate at most `(last - first) - 1` times.)

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

iter_swap

```
template<class FwdIt1, class FwdIt2>
    void iter_swap(FwdIt1 x, FwdIt2 y);
```

The template function leaves the value originally stored in `*y` subsequently stored in `*x`, and the value originally stored in `*x` subsequently stored in `*y`.

lexicographical_compare

```
template<class InIt1, class InIt2>
    bool lexicographical_compare(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2);
template<class InIt1, class InIt2, class Pred>
    bool lexicographical_compare(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, Pred pr);
```

The first template function determines `K`, the number of elements to compare as the smaller of `last1 - first1` and `last2 - first2`. It then determines the lowest value of `N` in the range `[0, K)` for which `*(first1 + N)` and `*(first2 + N)` do not have equivalent ordering. If no such value exists, the function returns true only if `K < (last2 - first2)`. Otherwise, it returns true only if `*(first1 + N) < *(first2 + N)`. Thus, the function returns true only if the sequence designated by iterators in the range `[first1, last1)` is lexicographically less than the other sequence.

The function evaluates the ordering predicate `X < Y` at most `2 * K` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

lower_bound

```
template<class FwdIt, class T>
    FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
```

The first template function determines the lowest value of `N` in the range `[0, last - first)` such that, for each `M` in the range `[0, N)` the predicate `*(first + M) < val` is true, where the elements designated by iterators in the range `[first, last)` form a sequence ordered by `operator<`. It then returns `first + N`. If no such value exists, the function returns `last`. Thus, the function determines the lowest position before which `val` can be inserted in the sequence and still preserve its ordering.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate `X < Y` at most `ceil(log(last - first)) + 1` times. Otherwise, the function evaluates the predicate a number of times proportional to `last - first`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

make_heap

```
template<class RanIt>
    void make_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void make_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a heap ordered by `operator<`.

The function evaluates the ordering predicate `X < Y` at most `3 * (last - first)` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

max

```
template<class T>
    const T& max(const T& x, const T& y);
template<class T, class Pred>
    const T& max(const T& x, const T& y, Pred pr);
```

The first template function returns `y` if `x < y`. Otherwise it returns `x`. `T` need supply only a single-argument constructor and a destructor.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

max_element

```
template<class FwdIt>
    FwdIt max_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest value of `N` in the range `[0, last - first)` such that, for each `M` in the range `[0, last - first)` the predicate `*(first + N) < *(first + M)` is false. It then returns `first + N`. Thus, the function determines the lowest position that contains the largest value in the sequence.

The function evaluates the ordering predicate `X < Y` exactly `max((last - first) - 1, 0)` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

merge

```
template<class InIt1, class InIt2, class OutIt>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt merge(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function determines K, the number of elements to copy as (last1 - first1) + (last2 - first2). It then alternately copies two sequences, designated by iterators in the ranges [first1, last1) and [first2, last2) and each ordered by operator<, to form a merged sequence of length K beginning at x, also ordered by operator<. The function then returns x + K.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for any two elements from different sequences that have equivalent ordering, the element from the ordered range [first1, last1) precedes the other. Thus, the function merges two ordered sequences to form another ordered sequence.

If x and first1 designate regions of storage, the range [x, x + K) must not overlap the range [first1, last1). If x and first2 designate regions of storage, the range [x, x + K) must not overlap the range [first2, last2). The function evaluates the ordering predicate X < Y at most K - 1 times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

min

```
template<class T>
    const T& min(const T& x, const T& y);
template<class T, class Pred>
    const T& min(const T& x, const T& y, Pred pr);
```

The first template function returns y if y < x. Otherwise it returns x. T need supply only a single-argument constructor and a destructor.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

min_element

```
template<class FwdIt>
    FwdIt min_element(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
```

The first template function determines the lowest value of N in the range [0, last - first) such that, for each M in the range [0, last - first) the predicate *(first + M) < *(first + N) is false. It then returns first + N. Thus, the function determines the lowest position that contains the smallest value in the sequence.

The function evaluates the ordering predicate X < Y exactly max((last - first) - 1, 0) times.

The second template function behaves the same, except that it replaces operator<(X, Y) with pr(X, Y).

mismatch

```
template<class InIt1, class InIt2>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x);
template<class InIt1, class InIt2, class Pred>
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last,
        InIt2 x, Pred pr);
```

The first template function determines the lowest value of `N` in the range `[0, last1 - first1)` for which the predicate `!(*(first1 + N) == *(first2 + N))` is true. It then returns `pair(first1 + N, first2 + N)`. If no such value exists, N has the value `last1 - first1`. The function evaluates the predicate at most once for each `N`.

The second template function behaves the same, except that the predicate is `pr(*(first1 + N), *(first2 + N))`.

next_permutation

```
template<class BidIt>
    bool next_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool next_permutation(BidIt first, BidIt last, Pred pr);
```

The first template function determines a repeating sequence of permutations, whose initial permutation occurs when the sequence designated by iterators in the range `[first, last)` is ordered by `operator<`. (The elements are sorted in *ascending* order.) It then reorders the elements in the sequence, by evaluating `swap(X, Y)` for the elements `X` and `Y` zero or more times, to form the next permutation. The function returns true only if the resulting sequence is not the initial permutation. Otherwise, the resultant sequence is the one next larger lexicographically than the original sequence. No two elements may have equivalent ordering.

The function evaluates `swap(X, Y)` at most `(last - first) / 2`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

nth_element

```
template<class RanIt>
    void nth_element(RanIt first, RanIt nth, RanIt last);
template<class RanIt, class Pred>
    void nth_element(RanIt first, RanIt nth, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` such that for each `N` in the range `[0, nth - first)` and for each `M` in the range `[nth - first, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Moreover, for `N`

equal to `nth - first` and for each `M` in the range `(nth - first, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Thus, if `nth != last` the element `*nth` is in its proper position if elements of the entire sequence were sorted in *ascending* order, ordered by `operator<`. Any elements before this one belong before it in the sort sequence, and any elements after it belong after it.

The function evaluates the ordering predicate `X < Y` a number of times proportional to `last - first`, on average.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partial_sort

```
template<class RanIt>
    void partial_sort(RanIt first, RanIt middle, RanIt last);
template<class RanIt, class Pred>
    void partial_sort(RanIt first, RanIt middle, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` such that for each `N` in the range `[0, middle - first)` and for each `M` in the range `(N, last - first)` the predicate `!(*(first + M) < *(first + N))` is true. Thus, the smallest `middle - first` elements of the entire sequence are sorted in *ascending* order, ordered by `operator<`. The order of the remaining elements is otherwise unspecified.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(middle - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partial_sort_copy

```
template<class InIt, class RanIt>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2);
template<class InIt, class RanIt, class Pred>
    RanIt partial_sort_copy(InIt first1, InIt last1,
        RanIt first2, RanIt last2, Pred pr);
```

The first template function determines `K`, the number of elements to copy as the smaller of `last1 - first1` and `last2 - first2`. It then copies and reorders `K` of the sequence designated by iterators in the range `[first1, last1)` such that the `K` elements copied to `first2` are ordered by `operator<`. Moreover, for each `N` in the range `[0, K)` and for each `M` in the range `(0, last1 - first1)` corresponding to an uncopied element, the predicate `!(*(first2 + M) < *(first1 + N))` is true. Thus, the smallest `K` elements of the entire sequence designated by iterators in the range `[first1, last1)` are copied and sorted in *ascending* order to the range `[first2, first2 + K)`.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(K))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

partition

```
template<class BidIt, class Pred>
    BidIt partition(BidIt first, BidIt last, Pred pr);
```

The template function reorders the sequence designated by iterators in the range `[first, last)` and determines the value `K` such that for each `N` in the range `[0, K)` the predicate `pr(*(first + N))` is true, and for each `N` in the range `[K, last - first)` the predicate `pr(*(first + N))` is false. The function then returns `first + K`.

The predicate must not alter its operand. The function evaluates `pr(*(first + N))` exactly `last - first` times, and swaps at most `(last - first) / 2` pairs of elements.

pop_heap

```
template<class RanIt>
    void pop_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void pop_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a new heap, ordered by `operator<` and designated by iterators in the range `[first, last - 1)`, leaving the original element at `*first` subsequently at `*(last - 1)`. The original sequence must designate an existing heap, also ordered by `operator<`. Thus, `first != last` must be true and `*(last - 1)` is the element to remove from (pop off) the heap.

The function evaluates the ordering predicate `X < Y` at most `ceil(2 * log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

prev_permutation

```
template<class BidIt>
    bool prev_permutation(BidIt first, BidIt last);
template<class BidIt, class Pred>
    bool prev_permutation(BidIt first, BidIt last, Pred pr);
```

The first template function determines a repeating sequence of permutations, whose initial permutation occurs when the sequence designated by iterators in the range `[first, last)` is the *reverse* of one

ordered by `operator<`. (The elements are sorted in *descending* order.) It then reorders the elements in the sequence, by evaluating `swap(X, Y)` for the elements `X` and `Y` zero or more times, to form the next permutation. The function returns true only if the resulting sequence is not the initial permutation. Otherwise, the resultant sequence is the one next smaller lexicographically than the original sequence. No two elements may have equivalent ordering.

The function evaluates `swap(X, Y)` at most `(last - first) / 2`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

push_heap

```
template<class RanIt>
    void push_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void push_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a new heap ordered by `operator<`. Iterators in the range `[first, last - 1)` must designate an existing heap, also ordered by `operator<`. Thus, `first != last` must be true and `*(last - 1)` is the element to add to (push on) the heap.

The function evaluates the ordering predicate `X < Y` at most `ceil(log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

random_shuffle

```
template<class RanIt>
    void random_shuffle(RanIt first, RanIt last);
template<class RanIt, class Fun>
    void random_shuffle(RanIt first, RanIt last, Fun& f);
```

The first template function evaluates `swap(*(first + N), *(first + M))` once for each `N` in the range `[1, last - first)`, where `M` is a value from some uniform random distribution over the range `[0, N)`. Thus, the function randomly shuffles the order of elements in the sequence.

The second template function behaves the same, except that `M` is `(Dist)f((Dist)N)`, where `Dist` is the type `iterator_traits:: distance_type`.

remove

```
template<class FwdIt, class T>
    FwdIt remove(FwdIt first, FwdIt last, const T& val);
```

The template function effectively assigns `first` to `X`, then executes the statement:

```
if (!(*(first + N) == val))
    *X++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. It then returns `X`. Thus, the function removes from the sequence all elements for which the predicate `*(first + N) == val` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

remove_copy

```
template<class InIt, class OutIt, class T>
    OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
```

The template function effectively executes the statement:

```
if (!(*(first + N) == val))
    *x++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. It then returns `x`. Thus, the function removes from the sequence all elements for which the predicate `*(first + N) == val` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

remove_copy_if

```
template<class InIt, class OutIt, class Pred>
    OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
```

The template function effectively executes the statement:

```
if (!pr(*(first + N)))
    *x++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. It then returns `x`. Thus, the function removes from the sequence all elements for which the predicate `pr(*(first + N))` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

remove_if

```
template<class FwdIt, class Pred>
    FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
```

The template function effectively assigns `first` to `X`, then executes the statement:

```
if (!pr(*(first + N)))
    *X++ = *(first + N);
```

once for each `N` in the range `[0, last - first)`. It then returns `X`. Thus, the function removes from the sequence all elements for which the predicate `pr(*(first + N))` is true, without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence.

replace

```
template<caass FwdIt, class T>
    void replace(FwdIt first, FwdIt last,
        const T& vold, const T& vnew);
```

The template function executes the statement:

```
if (*(first + N) == vold)
    *(first + N) = vnew;
```

once for each `N` in the range `[0, last - first)`.

replace_copy

```
template<class InIt, class OutIt, class T>
    OutIt replace_copy(InIt first, InIt last, OutIt x,
        const T& vold, const T& vnew);
```

The template function executes the statement:

```
if (*(first + N) == vold)
    *(x + N) = vnew;
else
    *(x + N) = *(first + N)
```

once for each `N` in the range `[0, last - first)`.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

replace_copy_if

```
template<class InIt, class OutIt, class Pred, class T>
    OutIt replace_copy_if(InIt first, InIt last, OutIt x,
        Pred pr, const T& val);
```

The template function executes the statement:

```
if (pr(*(first + N)))
    *(x + N) = vnew;
else
    *(x + N) = *(first + N)
```

once for each `N` in the range `[0, last - first)`.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

replace_if

```
template<class FwdIt, class Pred, class T>
    void replace_if(FwdIt first, FwdIt last,
        Pred pr, const T& val);
```

The template function executes the statement:

```
if (pr(*(first + N)))
    *(first + N) = val;
```

once for each `N` in the range `[0, last - first)`.

reverse

```
template<class BidIt>
    void reverse(BidIt first, BidIt last);
```

The template function evaluates swap`(*(first + N), *(last - 1 - N)` once for each `N` in the range `[0, (last - first) / 2)`. Thus, the function reverses the order of elements in the sequence.

reverse_copy

```
template<class BidIt, class OutIt>
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
```

The template function evaluates `*(x + N) = *(last - 1 - N)` once for each `N` in the range `[0, last - first)`. It then returns `x + (last - first)`. Thus, the function reverses the order of

elements in the sequence that it copies.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

rotate

```
template<class FwdIt>
    void rotate(FwdIt first, FwdIt middle, FwdIt last);
```

The template function leaves the value originally stored in `*(first + (N + (middle - last)) % (last - first))` subsequently stored in `*(first + N)` for each `N` in the range `[0, last - first)`. Thus, if a ``left'' shift by one element leaves the element originally stored in `*(first + (N + 1) % (last - first))` subsequently stored in `*(first + N)`, then the function can be said to rotate the sequence either left by `middle - first` elements or right by `last - middle` elements. Both `[first, middle)` and `[middle, last)` must be valid ranges. The function swaps at most `last - first` pairs of elements.

rotate_copy

```
template<class FwdIt, class OutIt>
    OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt x);
```

The template function evaluates `*(x + N) = *(first + (N + (middle - first)) % (last - first))` once for each `N` in the range `[0, last - first)`. Thus, if a ``left'' shift by one element leaves the element originally stored in `*(first + (N + 1) % (last - first))` subsequently stored in `*(first + N)`, then the function can be said to rotate the sequence either left by `middle - first` elements or right by `last - middle` elements as it copies. Both `[first, middle)` and `[middle, last)` must be valid ranges.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

search

```
template<class FwdIt1, class FwdIt2>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2);
template<class FwdIt1, class FwdIt2, class Pred>
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Pred pr);
```

The first template function determines the lowest value of `N` in the range `[0, (last1 - first1) - (last2 - first2))` such that for each `M` in the range `[0, last2 - first2)`, the predicate `*(first1 + N + M) == *(first2 + M)` is true. It then returns `first1 + N`. If no such value exists, the function returns `last1`. It evaluates the predicate at most `(last2 - first2) * (last1`

- first1) times.

The second template function behaves the same, except that the predicate is pr(*(first1 + N + M), *(first2 + M)).

search_n

```
template<class FwdIt, class Dist, class T>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val);
template<class FwdIt, class Dist, class T, class Pred>
    FwdIt search_n(FwdIt first, FwdIt last,
        Dist n, const T& val, Pred pr);
```

The first template function determines the lowest value of N in the range [0, (last - first) - n) such that for each M in the range [0, n), the predicate *(first + N + M) == val is true. It then returns first + N. If no such value exists, the function returns last. It evaluates the predicate at most n * (last - first) times.

The second template function behaves the same, except that the predicate is pr(*(first + N + M), val).

set_difference

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges [first1, last1) and [first2, last2), both ordered by operator<, to form a merged sequence of length K beginning at x, also ordered by operator<. The function then returns x + K.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range [first1, last1) and skips the other. An element from one sequence that has equivalent ordering with no element from the other sequence is copied from the ordered range [first1, last1) and skipped from the other. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the difference of two sets.

If x and first1 designate regions of storage, the range [x, x + K) must not overlap the range [first1, last1). If x and first2 designate regions of storage, the range [x, x + K) must not overlap the range [first2, last2). The function evaluates the ordering predicate X < Y at most 2 * ((last1 - first1) + (last2 - first2)) - 1 times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

set_intersection

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_intersection(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges `[first1, last1)` and `[first2, last2)`, both ordered by `operator<`, to form a merged sequence of length `K` beginning at `x`, also ordered by `operator<`. The function then returns `x + K`.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range `[first1, last1)` and skips the other. An element from one sequence that has equivalent ordering with no element from the other sequence is also skipped. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the intersection of two sets.

If `x` and `first1` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first1, last1)`. If `x` and `first2` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first2, last2)`. The function evaluates the ordering predicate `X < Y` at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

set_symmetric_difference

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges `[first1, last1)` and `[first2, last2)`, both ordered by `operator<`, to form a merged sequence of length `K` beginning at `x`, also ordered by `operator<`. The function then returns `x + K`.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies neither element. An element from one sequence that has equivalent ordering

with no element from the other sequence is copied. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the symmetric difference of two sets.

If `x` and `first1` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first1, last1)`. If `x` and `first2` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first2, last2)`. The function evaluates the ordering predicate `X < Y` at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

set_union

```
template<class InIt1, class InIt2, class OutIt>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x);
template<class InIt1, class InIt2, class OutIt, class Pred>
    OutIt set_union(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2, OutIt x, Pred pr);
```

The first template function alternately copies values from two sequences designated by iterators in the ranges `[first1, last1)` and `[first2, last2)`, both ordered by `operator<`, to form a merged sequence of length `K` beginning at `x`, also ordered by `operator<`. The function then returns `x + K`.

The merge occurs without altering the relative order of elements within either sequence. Moreover, for two elements from different sequences that have equivalent ordering that would otherwise be copied to adjacent elements, the function copies only the element from the ordered range `[first1, last1)` and skips the other. Thus, the function merges two ordered sequences to form another ordered sequence that is effectively the union of two sets.

If `x` and `first1` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first1, last1)`. If `x` and `first2` designate regions of storage, the range `[x, x + K)` must not overlap the range `[first2, last2)`. The function evaluates the ordering predicate `X < Y` at most `2 * ((last1 - first1) + (last2 - first2)) - 1` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

sort

```
template<class RanIt>
    void sort(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a sequence ordered by `operator<`. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

sort_heap

```
template<class RanIt>
    void sort_heap(RanIt first, RanIt last);
template<class RanIt, class Pred>
    void sort_heap(RanIt first, RanIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a sequence that is ordered by `operator<`. The original sequence must designate a heap, also ordered by `operator<`. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log(last - first))` times.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

stable_partition

```
template<class FwdIt, class Pred>
    FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);
```

The template function reorders the sequence designated by iterators in the range `[first, last)` and determines the value `K` such that for each `N` in the range `[0, K)` the predicate `pr(*(first + N))` is true, and for each `N` in the range `[K, last - first)` the predicate `pr(*(first + N))` is false. It does so without altering the relative order of either the elements designated by indexes in the range `[0, K)` or the elements designated by indexes in the range `[K, last - first)`. The function then returns `first + K`.

The predicate must not alter its operand. The function evaluates `pr(*(first + N))` exactly `last - first` times, and swaps at most `ceil((last - first) * log(last - first))` pairs of elements. (Given enough temporary storage, it can replace the swaps with at most `2 * (last - first)` assignments.)

stable_sort

```
template<class BidIt>
    void stable_sort(BidIt first, BidIt last);
template<class BidIt, class Pred>
    void stable_sort(BidIt first, BidIt last, Pred pr);
```

The first template function reorders the sequence designated by iterators in the range `[first, last)` to form a sequence ordered by `operator<`. It does so without altering the relative order of elements that have equivalent ordering. Thus, the elements are sorted in *ascending* order.

The function evaluates the ordering predicate `X < Y` at most `ceil((last - first) * log2(last - first))` times. (Given enough temporary storage, it can evaluate the predicate at most `ceil((last - first) * log(last - first))` times.)

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

swap

```
template<class T>
    void swap(T& x, T& y);
```

The template function leaves the value originally stored in `y` subsequently stored in `x`, and the value originally stored in `x` subsequently stored in `y`.

swap_ranges

```
template<class FwdIt1, class FwdIt2>
    FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
```

The template function evaluates `swap(*(first + N), *(x + N))` once for each `N` in the range `[0, last - first)`. It then returns `x + (last - first)`. If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

transform

```
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
template<class InIt1, class InIt2, class OutIt, class Binop>
    OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,
        OutIt x, Binop bop);
```

The first template function evaluates `*(x + N) = uop(*(first + N))` once for each `N` in the range `[0, last - first)`. It then returns `x + (last - first)`. The call `uop(*(first + N))` must not alter `*(first + N)`.

The second template function evaluates `*(x + N) = bop(*(first1 + N), *(first2 + N))` once for each `N` in the range `[0, last1 - first1)`. It then returns `x + (last1 - first1)`. The call `bop(*(first1 + N), *(first2 + N))` must not alter either `*(first1 + N)` or `*(first2 + N)`.

unique

```
template<class FwdIt>
    FwdIt unique(FwdIt first, FwdIt last);
template<class FwdIt, class Pred>
    FwdIt unique(FwdIt first, FwdIt last, Pred pr);
```

The first template function effectively assigns `first` to `X`, then executes the statement:

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *X++ = V;
```

once for each `N` in the range `[0, last - first)`. It then returns `X`. Thus, the function repeatedly removes from the sequence the second of a pair of elements for which the predicate `*(first + N) == *(first + N - 1)` is true, until only the first of a sequence of equal elements survives. It does so without altering the relative order of remaining elements, and returns the iterator value that designates the end of the revised sequence. The function evaluates the predicate at most `last - first` times.

The second template function behaves the same, except that it executes the statement:

```
if (N == 0 || !pr(*(first + N), V))
    V = *(first + N), *X++ = V;
```

unique_copy

```
template<class InIt, class OutIt>
    OutIt unique_copy(InIt first, InIt last, OutIt x);
template<class InIt, class OutIt, class Pred>
    OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);
```

The first template function effectively executes the statement:

```
if (N == 0 || !(*(first + N) == V))
    V = *(first + N), *x++ = V;
```

once for each `N` in the range `[0, last - first)`. It then returns `x`. Thus, the function repeatedly removes from the sequence it copies the second of a pair of elements for which the predicate `*(first + N) == *(first + N - 1)` is true, until only the first of a sequence of equal elements survives. It does so without altering the relative order of remaining elements, and returns the iterator value that designates the end of the copied sequence.

If `x` and `first` designate regions of storage, the range `[x, x + (last - first))` must not overlap the range `[first, last)`.

The second template function behaves the same, except that it executes the statement:

```
if (N == 0 || !pr(*(first + N), V))
```

```
    V = *(first + N), *x++ = V;
```

upper_bound

```
template<class FwdIt, class T>
    FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
template<class FwdIt, class T, class Pred>
    FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
```

The first template function determines the highest value of `N` in the range `[0, last - first)` such that, for each `M` in the range `[0, N)` the predicate `*(first + M) < val` is true, where the elements designated by iterators in the range `[first, last)` form a sequence ordered by `operator<`. It then returns `first + N`. If no such value exists, the function returns `last`. Thus, the function determines the highest position before which `val` can be inserted in the sequence and still preserve its ordering.

If `FwdIt` is a random-access iterator type, the function evaluates the ordering predicate `X < Y` at most `ceil(log(last - first)) + 1` times. Otherwise, the function evaluates the predicate a number of times proportional to `last - first`.

The second template function behaves the same, except that it replaces `operator<(X, Y)` with `pr(X, Y)`.

See also the **Table of Contents** and the **Index**.

<bitset>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<size_t N>
    class bitset;
//    TEMPLATE FUNCTIONS
template<class E, class T, size_t N>
    basic_istream<E, T>& operator>>(basic_istream<E, T>& is,
        bitset<N>& x);
template<class E, class T, size_t N>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
        const bitset<N>& x);
    };
```

Include the standard header **<bitset>** to define the template class `bitset` and two supporting templates.

bitset

---

any · at · bitset · bitset_size · count · element_type · flip · none · operator!= · operator&= · operator<< · operator<<= · operator== · operator>> · operator>>= · operator[] · operator^= · operator|= · operator~ · reference · reset · set · size · test · to_string · to_ulong

---

```
template<size_t N>
    class bitset {
public:
    typedef bool element_type;
    class reference;
    bitset();
    bitset(unsigned long val);
    template<class E, class T, class A>
        explicit bitset(const string<E, T, A>& str,
            string<E, T, A>size_type pos = 0,
            string<E, T, A>size_type n = string<E, T, A>::npos);
    bitset<N>& operator&=(const bitset<N>& rhs);
```

```
bitset<N>& operator|=(const bitset<N>& rhs);
bitset<N>& operator^=(const bitset<N>& rhs);
bitset<N>& operator<<=(const bitset<N>& pos);
bitset<N>& operator>>=(const bitset<N>& pos);
bitset<N>& set();
bitset<N>& set(size_t pos, bool val = true);
bitset<N>& reset();
bitset<N>& reset(size_t pos);
bitset<N>& flip();
bitset<N>& flip(size_t pos);
reference operator[](size_t pos);
bool operator[](size_t pos) const;
reference at(size_t pos);
bool at(size_t pos) const;
unsigned long to_ulong() const;
template<class E, class T, class A>
    string to_string() const;
size_t count() const;
size_t size() const;
bool operator==(const bitset<N>& rhs) const;
bool operator!=(const bitset<N>& rhs) const;
bool test(size_t pos) const;
bool any() const;
bool none() const;
bitset<N> operator<<(size_t pos) const;
bitset<N> operator>>(size_t pos) const;
bitset<N> operator~();
static const size_t bitset_size = N;
};
```

The template class describes an object that stores a sequence of `N` bits. A bit is **set** if its value is 1, **reset** if its value is 0. To **flip** a bit is to change its value from 1 to 0 or from 0 to 1. When converting between an object of class `bitset<N>` and an object of some integral type, bit position `j` corresponds to the bit value `1 << j`. The integral value corresponding to two or more bits is the sum of their bit values.

`bitset::any`

`bool` **`any`**`() const;`

The member function returns true if any bit is set in the bit sequence.

bitset::at

```
bool at(size_type pos) const;
reference at(size_type pos);
```

The member function returns an object of class reference, which designates the bit at position pos, if the object can be modified. Otherwise, it returns the value of the bit at position pos in the bit sequence. If that position is invalid, the function throws an object of class out_of_range.

bitset::bitset

```
bitset();
bitset(unsigned long val);
template<class E, class T, class A>
    explicit bitset(const string<E, T, A>& str,
        string<E, T, A>size_type pos = 0,
        string<E, T, A>size_type n = string<E, T, A>::npos);
```

The first constructor resets all bits in the bit sequence. The second constructor sets only those bits at position j for which val & 1 << j is nonzero.

The third constructor determines the initial bit values from elements of a string determined from str. If str.size() < pos, the constructor throws out_of_range. Otherwise, the effective length of the string rlen is the smaller of n and str.size() - pos. If any of the rlen elements beginning at position pos is other than 0 or 1, the constructor throws invalid_argument. Otherwise, the constructor sets only those bits at position j for which the element at position pos + j is 1.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
explicit bitset(const string& str,
    size_t pos = 0, size_t n = -1);
```

bitset::bitset_size

```
static const size_t bitset_size = N;
```

The const static member is initialized to the template parameter N.

bitset::count

```
size_t count() const;
```

The member function returns the number of bits set in the bit sequence.

bitset::element_type

```
typedef bool element_type;
```

The type is a synonyn for `bool`.

bitset::flip

```
bitset<N>& flip();
bitset<N>& flip(size_t pos);
```

The first member function flips all bits in the bit sequence, then returns `*this`. The second member function throws `out_of_range` if `size() <= pos`. Otherwise, it flips the bit at position `pos`, then returns `*this`.

bitset::none

```
bool none() const;
```

The member function returns true if none of the bits are set in the bit sequence.

bitset::operator!=

```
bool operator !=(const bitset<N>& rhs) const;
```

The member operator function returns true only if the bit sequence stored in `*this` differs from the one stored in `rhs`.

bitset::operator&=

```
bitset<N>& operator&=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the logical AND of its previous value and the corresponding bit in `rhs`. The function returns `*this`.

bitset::operator<<

```
bitset<N> operator<<(const bitset<N>& pos);
```

The member operator function returns `bitset(*this) <<= pos`.

bitset::operator<<=

```
bitset<N>& operator<<=(const bitset<N>& pos);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the element `pos` positions earlier in the sequence. If no such earlier element exists, the function clears the bit. The function returns `*this`.

bitset::operator==

```
bool operator ==(const bitset<N>& rhs) const;
```

The member operator function returns true only if the bit sequence stored in `*this` is the same as the one stored in `rhs`.

bitset::operator>>

```
bitset<N> operator>>(const bitset<N>& pos);
```

The member operator function returns `bitset(*this) >>= pos`.

bitset::operator>>=

```
bitset<N>& operator>>=(const bitset<N>& pos);
```

The member function replaces each element of the bit sequence stored in `*this` with the element `pos` positions later in the sequence. If no such later element exists, the function clears the bit. The function returns `*this`.

bitset::operator[]

```
bool operator[](size_type pos) const;
reference operator(size_type pos);
```

The member function returns an object of class `reference`, which designates the bit at position `pos`, if the object can be modified. Otherwise, it returns the value of the bit at position `pos` in the bit sequence. If that position is invalid, the behavior is undefined.

bitset::operator^=

```
bitset<N>& operator^=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the logical EXCLUSIVE OR of its previous value and the corresponding bit in `rhs`. The function returns `*this`.

bitset::operator|=

```
bitset<N>& operator|=(const bitset<N>& rhs);
```

The member operator function replaces each element of the bit sequence stored in `*this` with the logical OR of its previous value and the corresponding bit in `rhs`. The function returns `*this`.

bitset::operator~

```
bitset<N> operator~();
```

The member operator function returns `bitset(*this).flip()`.

bitset::reference

```
class reference {
public:
    reference& operator=(bool b};
    reference& operator=(const reference& x);
    bool operator~() const;
    operator bool() const;
    reference& flip();
    };
```

The member class describes an object that designates an individual bit within the bit sequence. Thus, for `b` an object of type `bool`, `x` and `y` objects of type `bitset<N>`, and `i` and `j` valid positions within such an object, the member functions of class `reference` ensure that (in order):

- **`x[i] = b`** stores `b` at bit position `i` in `x`
- **`x[i] = y[j]`** stores the value of the bit `y[j]` at bit position `i` in `x`
- **`b = ~x[i]`** stores the flipped value of the bit `x[i]` in `b`
- **`b = x[i]`** stores the value of the bit `x[i]` in `b`
- **`x[i].flip()`** stores the flipped value of the bit `x[i]` back at bit position `i` in `x`

bitset::reset

```
bitset<N>& reset();
bitset<N>& reset(size_t pos);
```

The first member function resets all bits in the bit sequence, then returns `*this`. The second member function throws `out_of_range` if `size() <= pos`. Otherwise, it resets the bit at position `pos`, then returns `*this`.

bitset::set

```
bitset<N>& set();
bitset<N>& set(size_t pos, bool val = true);
```

The first member function resets all bits in the bit sequence, then returns `*this`. The second member function throws `out_of_range` if `size() <= pos`. Otherwise, it stores `val` in the bit at position `pos`, then returns `*this`.

bitset::size

```
size_t size() const;
```

The member function returns N.

bitset::test

```
bool test(size_t pos, bool val = true);
```

The member function throws out_of_range if size() <= pos. Otherwise, it returns true only if the bit at position pos is set.

bitset::to_string

```
template<class E, class T, class A>
    string to_string() const;
```

The member function constructs str, an object of class string. For each bit in the bit sequence, the function appends 1 if the bit is set, otherwise 0. The *last* element appended to str corresponds to bit position zero. The function returns str.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
string to_string() const;
```

bitset::to_ulong

```
unsigned long to_ulong() const;
```

The member function throws overflow_error if any bit in the bit sequence has a bit value that cannot be represented as a value of type *unsigned long*. Otherwise, it returns the sum of the bit values in the bit sequence.

operator<<

```
template<class E, class T, size_t N>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
        const bitset<N>& x);
```

The template function overloads operator<< to insert a text representation of the bit sequence in os. It effectively executes os << x.to_string<E, char_traits<E>, allocator<E> >(), then returns os.

operator>>

```
template<class E, class T, size_t N>
    basic_istream<E, T>& operator>>(basic_istream<E, T>&
        is, bitset<N>& x);
```

The template function overloads `operator>>` to store in `x` the value `bitset(str)`, where `str` is an object of type `basic_string<E, T, allocator<E> >&` extracted from `is`. The function extracts elements and appends them to `str` until:

- `N` elements have been extracted and stored
- end-of-file occurs on the input sequence
- the next input element is neither `0` not `1`, in which case the input element is not extracted

If the function stores no characters in `str`, it calls `is.setstate(ios_base::failbit)`. In any case, it returns `is`.

See also the **Table of Contents** and the **Index**.

<complex>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

abs · arg · complex · complex<double> · complex<float> · complex<long double> · conjg · cos · cosh · exp · imag · log · log10 · norm · operator!= · operator* · operator+ · operator- · operator/ · operator<< · operator== · operator>> · polar · pow · real · sin · sinh · sqrt · __STD_COMPLEX

```
namespace std {
#define __STD_COMPLEX
//    TEMPLATE CLASSES
template<class T>
    class complex;
class complex<float>;
class complex<double>;
class complex<long double>;
//    TEMPLATE FUNCTIONS
template<class T>
    complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator+(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator-(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator*(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
```

```
    complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator/(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs);
template<class T>
    bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    bool operator==(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator==(const T& lhs, const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator!=(const T& lhs, const complex<T>& rhs);
template<class E, class Ti, class T>
    basic_istream<E, Ti>& operator>>(basic_istream<E, Ti>& is,
        complex<T>& x);
template<class E, class T, class U>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
        const complex<U>& x);
template<class T>
    T real(const complex<T>& x);
template<class T>
    T imag(const complex<T>& x);
template<class T>
    T abs(const complex<T>& x);
template<class T>
    T arg(const complex<T>& x);
template<class T>
    T norm(const complex<T>& x);
template<class T>
    complex<T> conjg(const complex<T>& x);
template<class T>
    complex<T> polar(const T& rho, const T& theta = 0);
template<class T>
    complex<T> cos(const complex<T>& x);
template<class T>
    complex<T> cosh(const complex<T>& x);
```

```
template<class T>
    complex<T> exp(const complex<T>& x);
template<class T>
    complex<T> log(const complex<T>& x);
template<class T>
    complex<T> log10(const complex<T>& x);
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const T& y);
template<class T>
    complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
template<class T>
    complex<T> sin(const complex<T>& x);
template<class T>
    complex<T> sinh(const complex<T>& x);
template<class T>
    complex<T> sqrt(const complex<T>& x);
    };
```

Include the standard header **<complex>** to define template class `complex` and a host of supporting template functions. Unless otherwise specified, functions that can return multiple values return an imaginary part in the half-open interval `(-pi, pi]`.

abs

```
template<class T>
    T abs(const complex<T>& x);
```

The template function returns the magnitude of `x`.

arg

```
template<class T>
    T arg(const complex<T>& x);
```

The template function returns the phase angle of `x`.

complex

```
template<class T>
    class complex {
public:
    complex(const T& re = 0, const T& im = 0);
    template<class U>
        complex(const complex<U>& x);
public:
    typedef T value_type;
    T real() const;
    T imag() const;
    template<class U>
        complex& operator=(const complex<U>& rhs);
    template<class U>
        complex& operator+=(const complex<U>& rhs);
    template<class U>
        complex& operator-=(const complex<U>& rhs);
    template<class U>
        complex& operator*=(const complex<U>& rhs);
    template<class U>
        complex& operator/=(const complex<U>& rhs);
    complex& operator=(const T& rhs);
    complex& operator+=(const T& rhs);
    complex& operator-=(const T& rhs);
    complex& operator*=(const T& rhs);
    complex& operator/=(const T& rhs);
    friend complex<T>
        operator+(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator+(const T& lhs, const complex<T>& rhs);
    friend complex<T>
        operator-(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator-(const T& lhs, const complex<T>& rhs);
    friend complex<T>
        operator*(const complex<T>& lhs, const T& rhs);
    friend complex<T>
        operator*(const T& lhs, const complex<T>& rhs);
    friend complex<T>
        operator/(const complex<T>& lhs, const T& rhs);
    friend complex<T>
```

```
    operator/(const T& lhs, const complex<T>& rhs);
friend bool operator==(const complex<T>& lhs, const T& rhs);
friend bool operator==(const T& lhs, const complex<T>& rhs);
friend bool operator!=(const complex<T>& lhs, const T& rhs);
friend bool operator!=(const T& lhs, const complex<T>& rhs);
};
```

The template class describes an object that stores two objects of type **T**, one that represents the real part of a complex number and one that represents the imaginary part. An object of class `T`:

- has a public default constructor, destructor, copy constructor, and assignment operator -- with conventional behavior
- can be assigned integer or floating-point values, or type cast to such values -- with conventional behavior
- define the arithmetic operators defined for the floating-point types -- with conventional behavior

Explicit specializations of template class `complex` exist for the three floating-point types. In this implementation, all other types may be type cast to *double* for actual calculations, with the *double* results assigned back to the stored objects of type `T`.

complex::complex

```
complex(const T& re = 0, const T& im = 0);
template<class U>
    complex(const complex<U>& x);
```

The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The template constructor initializes the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
complex(const complex& x);
```

which is the copy constructor.

complex::imag

```
T imag() const;
```

The member function returns the stored imaginary part.

complex::operator*=

```
template<class U>
    complex& operator*=(const complex<U>& rhs);
complex& operator*=(const T& rhs);
```

The template member function replaces the stored real and imaginary parts with those corresponding to the complex product of `*this` and `rhs`. It then returns `*this`.

The second member function multiplies both the stored real part and the stored imaginary part with `rhs`. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
complex& operator*=(const complex& rhs);
```

complex::operator+=

```
template<class U>
    complex& operator+=(const complex<U>& rhs);
complex& operator+=(const T& rhs);
```

The template member function replaces the stored real and imaginary parts with those corresponding to the complex sum of `*this` and `rhs`. It then returns `*this`.

The second member function adds `rhs` to the stored real part. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
complex& operator+=(const complex& rhs);
```

complex::

```
template<class U>
    complex& operator-=(const complex<U>& rhs);
complex& operator-=(const T& rhs);
```

The template member function replaces the stored real and imaginary parts with those corresponding to the complex difference of `*this` and `rhs`. It then returns `*this`.

The second member function subtracts `rhs` from the stored real part. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
complex& operator-=(const complex& rhs);
```

complex::operator/=

```
template<class U>
    complex& operator/=(const complex<U>& rhs);
complex& operator/=(const T& rhs);
```

The template member function replaces the stored real and imaginary parts with those corresponding to the complex quotient of `*this` and `rhs`. It then returns `*this`.

The second member function multiplies both the stored real part and the stored imaginary part with `rhs`. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
complex& operator/=(const complex& rhs);
```

complex::operator=

```
template<class U>
    complex& operator=(const complex<U>& rhs);
complex& operator=(const T& rhs);
```

The template member function replaces the stored real part with `rhs.real()` and the stored imaginary part with `rhs.imag()`. It then returns `*this`.

The second member function replaces the stored real part with `rhs` and the stored imaginary part with zero. It then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
complex& operator=(const complex& rhs);
```

which is the default assignment operator.

complex::real

```
T real() const;
```

The member function returns the stored real part.

complex::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

complex<double>

```
class complex<double> {
public:
    complex(double re = 0, double im = 0);
    complex(const complex<float>& x);
    explicit complex(const complex<long double>& x);
// rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **double**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

complex<float>

```
class complex<float> {
public:
    complex(float re = 0, float im = 0);
    explicit complex(const complex<double>& x);
    explicit complex(const complex<long double>& x);
// rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **float**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

complex<long double>

```
class complex<long double> {
public:
    complex(long double re = 0, long double im = 0);
    complex(const complex<float>& x);
    complex(const complex<double>& x);
// rest same as template class complex
    };
```

The explicitly specialized template class describes an object that stores two objects of type **long double**, one that represents the real part of a complex number and one that represents the imaginary part. The explicit specialization differs only in the constructors it defines. The first constructor initializes the stored real part to `re` and the stored imaginary part to `im`. The remaining two constructors initialize the stored real part to `x.real()` and the stored imaginary part to `x.imag()`.

conjg

```
template<class T>
    complex<T> conjg(const complex<T>& x);
```

The template function returns the conjugate of `x`.

cos

```
template<class T>
    complex<T> cos(const complex<T>& x);
```

The template function returns the cosine of x.

cosh

```
template<class T>
    complex<T> cosh(const complex<T>& x);
```

The template function returns the hyperbolic cosine of x.

exp

```
template<class T>
    complex<T> exp(const complex<T>& x);
```

The template function returns the exponential of x.

imag

```
template<class T>
    T imag(const complex<T>& x);
```

The template function returns the imaginary part of x.

log

```
template<class T>
    complex<T> log(const complex<T>& x);
```

The template function returns the logarithm of x. The branch cuts are along the negative real axis.

log10

```
template<class T>
    complex<T> log10(const complex<T>& x);
```

The template function returns the base 10 logarithm of x. The branch cuts are along the negative real axis.

norm

```
template<class T>
    T norm(const complex<T>& x);
```

The template function returns the squared magnitude of `x`.

operator!=

```
template<class T>
    bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator!=(const T& lhs, const complex<T>& rhs);
```

The template operators each return true only if `real(lhs) != real(rhs) || imag(lhs) != imag(rhs)`.

operator*

```
template<class T>
    complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator*(const T& lhs, const complex<T>& rhs);
```

The template operators each convert both operands to type `complex<T>`, then return the complex product of the converted `lhs` and `rhs`.

operator+

```
template<class T>
    complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator+(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator+(const complex<T>& lhs);
```

The first three template operators each convert both operands to type `complex<T>`, then return the complex sum of the converted `lhs` and `rhs`.

The last template operator returns `lhs`.

operator-

```
template<class T>
    complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator-(const T& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator-(const complex<T>& lhs);
```

The first three template operators each convert both operands to type `complex<T>`, then return the complex difference of the converted `lhs` and `rhs`.

The last template operator returns a value whose real part is `-real(lhs)` and whose imaginary part is `-imag(lhs)`.

operator/

```
template<class T>
    complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T>
    complex<T> operator/(const T& lhs, const complex<T>& rhs);
```

The template operators each convert both operands to type `complex<T>`, then return the complex quotient of the converted `lhs` and `rhs`.

operator<<

```
template<class E, class T, class U>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T>& os,
        const complex<U>& x);
```

The template function inserts the complex value `x` in the output stream `os`, effectively by executing:

```
basic_ostringstream<E, T> ostr;
ostr.flags(os.flags());
ostr.imbue(os.imbue());
ostr.precision(os.precision());
ostr << '(' << real(x) << ','
    << imag(x) << ')';
os << ostr.str().c_str();
```

Thus, if `os.width()` is greater than zero, any padding occurs either before or after the parenthesized pair of values, which itself contains no padding. The function returns `os`.

operator==

```
template<class T>
    bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T>
    bool operator==(const complex<T>& lhs, const T& rhs);
template<class T>
    bool operator==(const T& lhs, const complex<T>& rhs);
```

The template operators each return true only if `real(lhs) == real(rhs) && imag(lhs) == imag(rhs)`.

operator>>

```
template<class E, class T, class U>
    basic_istream<E, T>& operator>>(basic_istream<E, T>& is,
        complex<U>& x);
```

The template function attempts to extract a complex value from the input stream `is`, effectively by executing:

```
is >> ch && ch == '('
    is >> re >> ch && ch == ','
    is >> im >> ch && ch == ')'
```

Here, `ch` is an object of type *char*, and `re` and `im` are objects of type `U`.

If the result of this expression is true, the function stores `re` in the real part and `im` in the imaginary part of `x`. In any event, the function returns `is`.

polar

```
template<class T>
    complex<T> polar(const T& rho, const T& theta = 0);
```

The template function returns the complex value whose magnitude is `rho` and whose phase angle is `theta`.

pow

```
template<class T>
    complex<T> pow(const complex<T>& x, int y);
template<class T>
    complex<T> pow(const complex<T>& x, const T& y);
template<class T>
    complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T>
    complex<T> pow(const T& x, const complex<T>& y);
```

The template functions each each effectively convert both operands to type `complex<T>`, then return the converted `x` to the power `y`. The branch cut for `x` is along the negative real axis.

real

```
template<class T>
    T real(const complex<T>& x);
```

The template function returns the real part of `x`.

sin

```
template<class T>
    complex<T> sin(const complex<T>& x);
```

The template function returns the imaginary sine of `x`.

sinh

```
template<class T>
    complex<T> sinh(const complex<T>& x);
```

The template function returns the hyperbolic sine of `x`.

sqrt

```
template<class T>
    complex<T> sqrt(const complex<T>& x);
```

The template function returns the square root of `x`, with phase angle in the half-open interval `(-pi/2, pi/2]`. The branch cuts are along the negative real axis.

__STD_COMPLEX

`#define __STD_COMPLEX`

The macro is defined, with an unspecified expansion, to indicate compliance with the specifications of this header.

See also the **Table of Contents** and the **Index**.

<deque>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class T, class A>
    class deque;
//    TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
template<class T, class A>
    void swap(
        const deque<T, A>& lhs,
        const deque<T, A>& rhs);
    };
```

Include the STL standard header **<deque>** to define the container template class `deque` and three supporting templates.

deque

```
template<class T, class A = allocator<T> >
    class deque {
public:
    typedef A allocator_type;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef reverse_iterator<iterator, value_type,
        reference, A::pointer, difference_type>
            reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
        const_reference, A::const_pointer, difference_type>
            const_reverse_iterator;
    explicit deque(const A& al = A());
    explicit deque(size_type n, const T& v = T(), const A& al = A());
    deque(const deque& x);
    template<class InIt>
        deque(InIt first, InIt last, const A& al = A());
    iterator begin();
    const_iterator begin() const;
    iterator end();
    iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    void resize(size_type n, T x = T());
```

```
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    reference at(size_type pos);
    const_reference at(size_type pos) const;
    reference operator[](size_type pos);
    const_reference operator[](size_type pos);
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;
    void push_front(const T& x);
    void pop_front();
    void push_back(const T& x);
    void pop_back();
    template<class InIt>
        void assign(InIt first, InIt last);
    template<class Size, class T2>
        void assign(Size n, const T2& x = T2());
    iterator insert(iterator it, const T& x = T());
    void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(deque x);
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type T**. The sequence is represented in a way that permits insertion and removal of an element at either end with a single element copy (constant time). Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

The object allocates and frees storage for the sequence it controls through a protected object named **allocator**, of **class A**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

Deque reallocation occurs when a member function must insert or erase elements of the controlled sequence. In all such cases, iterators or references that point anywhere within the controlled sequence

become **invalid**.

deque::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

deque::assign

```
template<class InIt>
    void assign(InIt first, InIt last);
template<class Size, class T2>
    void assign(Size n, const T2& x = T2());
```

The first member template function replaces the sequence controlled by `*this` with the sequence `[first, last)`. The second member template function replaces the sequence controlled by `*this` with a repetition of `n` elements of value `x`.

In this implementation, if a translator does not support member template functions, the templates are replaced by:

```
void assign(const_iterator first, const_iterator last);
void assign(size_type n, const T& x = T());
```

deque::at

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the function throws an object of class `out_of_range`.

deque::back

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

deque::begin

```
const_iterator begin() const;
iterator begin();
```

The member function returns a random-access iterator that points at the first element of the sequence (or

just beyond the end of an empty sequence).

deque::clear

```
void clear() const;
```

The member function calls erase(begin(), end()).

deque::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type T1.

deque::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

deque::const_reverse_iterator

```
typedef reverse_iterator<const_iterator, value_type,
    const_reference, A::const_pointer, difference_type>
        const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse random-access iterator for the controlled sequence.

deque::deque

```
explicit deque(const A& al = A());
explicit deque(size_type n, const T& v = T(), const A& al = A());
deque(const deque& x);
template<class InIt>
    deque(InIt first, InIt last, const A& al = A());
```

All constructors store the allocator object al (or, for the copy constructor, x.get_allocator()) in allocator and initialize the controlled sequence. The first constructor specifies an empty initial controlled sequence. The second constructor specifies a repetition of n elements of value x. The third constructor specifies a copy of the sequence controlled by x. The member template constructor specifies the sequence [first, last).

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
deque(const_iterator first, const_iterator last, const A& al = A());
```

deque::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

deque::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

deque::end

```
const_iterator end() const;
iterator end();
```

The member function returns a random-access iterator that points just beyond the end of the sequence.

deque::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

Removing `N` elements causes `N` destructor calls and an assignment for each of the elements between the insertion point and the nearer end of the sequence. Removing an element at either end invalidates only iterators and references that designate the erased elements. Otherwise, erasing an element invalidates all iterators and references.

deque::front

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

deque::get_allocator

```
A get_allocator() const;
```

The member function returns allocator.

deque::insert

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`. The member template function inserts the sequence `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(iterator it, const_iterator first, const_iterator last);
```

When inserting a single element, the number of element copies is linear in the number of elements between the insertion point and the nearer end of the sequence. When inserting a single element at either end of the sequence, the amortized number of element copies is constant. When inserting `N` elements, the number of element copies is linear in `N` plus the number of elements between the insertion point and the nearer end of the sequence -- except when the template member is specialized for `InIt` an input or forward iterator, which behaves like `N` single insertions. Inserting an element at either end invalidates all iterators, but no references, that designate existing elements. Otherwise, inserting an element invalidates all iterators and references.

deque::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

deque::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

deque::operator[]

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the behavior is undefined.

deque::pop_back

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty. Removing the element invalidates only iterators and references that designate the erased element.

deque::push_back

```
void push_back(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence. Inserting the element invalidates all iterators, but no references, to existing elements.

deque::pop_front

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty. Removing the element invalidates only iterators and references that designate the erased element.

deque::push_front

```
void push_front(const T& x);
```

The member function inserts an element with value `x` at the beginning of the controlled sequence. Inserting the element invalidates all iterators, but no references, to existing elements.

deque::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

deque::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

deque::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

deque::resize

```
void resize(size_type n, T x = T());
```

The member function ensures that `size()` henceforth returns `n`. If it must make the controlled sequence longer, it appends elements with value `x`.

deque::reverse_iterator

```
typedef reverse_iterator<iterator, value_type,
    reference, A::pointer, difference_type>
        reverse_iterator;
```

The type describes an object that can serve as a reverse random-access iterator for the controlled sequence.

deque::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

deque::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

deque::swap

```
void swap(deque& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments

and constructor calls proportional to the number of elements in the two controlled sequences.

deque::value_type

```
typedef A::value_type value_type;
```

The type is a synonym for the template parameter `T`.

operator!=

```
template<class T, class A>
    bool operator!=(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `deque`. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `deque`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const deque <T, A>& lhs,
```

```
        const deque <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

swap

```
template<class T, class A>
    void swap(
        const deque <T, A>& lhs,
        const deque <T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

See also the **Table of Contents** and the **Index**.

<exception>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    class exception;
    class bad_exception;
    typedef void (*terminate_handler)();
    typedef void (*unexpected_handler)();
    terminate_handler set_terminate(terminate_handler ph) throw();
    unexpected_handler set_unexpected(unexpected_handler ph) throw();
    void terminate();
    void unexpected();
    bool uncaught_exception();
    };
```

Include the standard header **<exception>** to define several types and functions related to the handling of exceptions.

bad_exception

```
class bad_cast : public exception {
    };
```

The class describes an exception that can be thrown from an unexpected handler. The value returned by what() is implementation-defined. None of the member functions throw any exceptions.

exception

```
class exception {
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
    };
```

The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ library. The C string value returned by **what()** is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes. None of the member functions throw any

exceptions.

set_terminate

```
terminate_handler set_terminate(terminate_handler ph) throw();
```

The function establishes a new terminate handler as the function `*ph`. Thus, `ph` must not be a null pointer. The function returns the address of the previous terminate handler.

set_unexpected

```
unexpected_handler set_unexpected(unexpected_handler ph) throw();
```

The function establishes a new unexpected handler as the function `*ph`. Thus, `ph` must not be a null pointer. The function returns the address of the previous unexpected handler.

terminate_handler

```
typedef void (*terminate_handler)();
```

The type describes a pointer to a function suitable for use as a terminate handler.

unexpected_handler

```
typedef void (*unexpected_handler)();
```

The type describes a pointer to a function suitable for use as an unexpected handler.

terminate

```
void terminate();
```

The function calls the current **terminate handler**, a function of type `void ()` called when exception handling must be abandoned for any of several reasons. A terminate handler may not return to its caller. At program startup, the terminate handler is a function that calls `abort()`.

uncaught_exception

```
bool uncaught_exception();
```

The function returns true only if a thrown exception is being currently processed.

unexpected

```
void unexpected();
```

The function calls the current **unexpected handler**, a function of type `void ()` called when control leaves a function by a thrown exception of a type not permitted by an **exception specification** for the function, as in:

```
void f() throw()      // function may throw no exceptions
    {throw "bad"; }   // unexpected throw calls unexpected()
```

An unexpected handler may not return to its caller. It may terminate execution by:

- throwing an object of a type listed in the exception specification
- throwing an object of type `bad_exception`
- calling `terminate()`, `abort()`, or `exit(int)`

At program startup, the unexpected handler is a function that calls `terminate()`.

See also the **Table of Contents** and the **Index**.

<fstream>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    template<class E, class T = char_traits<E> >
        class basic_filebuf;
    typedef basic_filebuf<char> filebuf;
    typedef basic_filebuf<wchar_t> wfilebuf;
    template<class E, class T = char_traits<E> >
        class basic_ifstream;
    typedef basic_ifstream<char> ifstream;
    typedef basic_ifstream<wchar_t> wifstream;
    template<class E, class T = char_traits<E> >
        class basic_ofstream;
    typedef basic_ofstream<char> ofstream;
    typedef basic_ofstream<wchar_t> wofstream;
    template<class E, class T = char_traits<E> >
        class basic_fstream;
    typedef basic_fstream<char> fstream;
    typedef basic_fstream<wchar_t> wfstream;
    };
```

Include the iostreams standard header **<fstream>** to define several template classes that support iostreams operations on sequences stored in external **files**.

basic_filebuf

```
template <class E, class T = char_traits<E> >
    class basic_filebuf {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    basic_filebuf();
    bool is_open() const;
    basic_filebuf *open(const char *s, ios_base::openmode mode);
```

```
    basic_filebuf *close();
protected:
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type pos,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c = T::eof());
    virtual int_type overflow(int_type c = T::eof());
    virtual int sync();
    virtual basic_streambuf<E, T> *setbuf(E *s, streamsize n);
    };
```

The template class describes a **stream buffer** that controls the transmission of elements to and from a sequence of elements stored in an external file.

An object of class `basic_filebuf<E, T>` stores a **file pointer**, which designates the `FILE` object that controls the **stream** associated with an open file. It also stores pointers to two **file conversion facets** for use by the protected member functions `overflow` and `underflow`.

basic_filebuf::basic_filebuf

```
basic_filebuf();
```

The constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores a null pointer in the file pointer.

basic_filebuf::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_filebuf::close

```
basic_filebuf *close();
```

The member function returns a null pointer if the file pointer `fp` is a null pointer. Otherwise, it calls `fclose(fp)`. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns `this` to indicate that the file was successfully closed.

basic_filebuf::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_filebuf::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_filebuf::is_open

```
bool is_open();
```

The member function returns true if the file pointer is not a null pointer.

basic_filebuf::open

```
basic_filebuf *open(const char *s, ios_base::openmode mode);
```

The member function endeavors to open the file with filename `s`, by calling `fopen(s, strmode)`. Here `strmode` is determined from `mode & ~(ate & | binary)`:

- `ios_base::in` becomes `"r"` (open existing file for reading).
- `ios_base::out` or
- `ios_base::out | ios_base::trunc` becomes `"w"` (truncate existing file or create for writing).
- `ios_base::out | app` becomes `"a"` (open existing file for appending all writes).
- `ios_base::in | ios_base::out` becomes `"r+"` (open existing file for reading and writing).
- `ios_base::in | ios_base::out | ios_base::trunc` becomes `"w+"` (truncate existing file or create for reading and writing).
- `ios_base::in | ios_base::out | ios_base::app` becomes `"a+"` (open existing file for reading and for appending all writes).

If `mode & ios_base::binary` is nonzero, the function appends `b` to `strmode` to open a binary stream instead of a text stream. It then stores the value returned by `fopen` in the file pointer `fp`. If `mode & ios_base::ate` is nonzero and the file pointer is not a null pointer, the function calls `fseek(fp, 0, SEEK_END)` to position the stream at end-of-file. If that positioning operation fails, the function calls `close(fp)` and stores a null pointer in the file pointer.

If the file pointer is not a null pointer, the function determines the **file conversion facet** `use_facet< codecvt<E, char, T::state_type> >(getloc())`, for use by `underflow` and

overflow

If the file pointer is a null pointer, the function returns a null pointer. Otherwise, it returns `this`.

basic_filebuf::overflow

```
virtual int_type overflow(int_type c = T::eof());
```

If `c != T::eof()`, the protected virtual member function endeavors to insert the element `T::to_char_type(c)` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can convert any pending output in the output buffer, followed by `c`, by using the file conversion facet `fac` to call `fac.out` as needed. Each element `x` of type *char* thus produced is written to the associated stream designated by the file pointer `fp` as if by successive calls of the form `fputc(x, fp)`. If any conversion or write fails, the function does not succeed.

If the function cannot succeed, it returns `T::eof()`. Otherwise, it returns `T::not_eof(c)`.

basic_filebuf::pbackfail

```
virtual int_type pbackfail(int_type c = T::eof());
```

The protected virtual member function endeavors to put back an element into the input buffer, then make it the current element (pointed to by the next pointer). If `c == T::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = T::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If the function can make a putback position available, it can do so, set the next pointer to point at that position, and store `x` in that position.
- If the function can push back an element onto the input stream, it can do so, such as by calling `ungetc` for an element of type *char*.

If the function cannot succeed, it returns `T::eof()`. Otherwise, it returns `T::not_eof(c)`.

basic_filebuf::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_filebuf::seekoff

```
virtual pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_filebuf<E, T>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type `pos_type` stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either `pubseekoff` or `pubseekoff`. Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for text streams, binary streams, and wide streams.

If the file pointer `fp` is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling `fseek(fp, off, way)`. If that function succeeds and the resultant position `fposn` can be determined by calling `fgetpos(fp, &fposn)`, the function succeeds. If the function succeeds, it returns a value of type `pos_type` containing `fposn`. Otherwise, it returns an invalid stream position.

basic_filebuf::seekpos

```
virtual pos_type seekpos(pos_type pos,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_filebuf<E, T>`, a stream position can be represented by an object of type `fpos_t`, which stores an offset and any state information needed to parse a wide stream. Offset zero designates the first element of the stream. (An object of type `pos_type` stores at least an `fpos_t` object.)

For a file opened for both reading and writing, both the input and output streams are positioned in tandem. To switch between inserting and extracting, you must call either `pubseekoff` or `pubseekoff`. Calls to `pubseekoff` (and hence to `seekoff`) have various limitations for text streams, binary streams, and wide streams.

If the file pointer `fp` is a null pointer, the function fails. Otherwise, it endeavors to alter the stream position by calling `fsetpos(fp, &fposn)`, where `fposn` is the `fpos_t` object stored in `pos`. If that function succeeds, the function returns `pos`. Otherwise, it returns an invalid stream position.

basic_filebuf::setbuf

```
virtual basic_streambuf<E, T> *setbuf(E *s, streamsize n);
```

The protected member function returns zero if the file pointer `fp` is a null pointer. Otherwise, it calls `setvbuf(fp, (char *)s, _IOFBF, n * sizeof (E))` to offer the array of `n` elements beginning at `s` as a buffer for the stream. If that function returns a nonzero value, the function returns a null pointer. Otherwise, it returns `this` to signal success.

basic_filebuf::sync

```
int sync();
```

The protected member function returns zero if the file pointer `fp` is a null pointer. Otherwise, it returns `fflush(fp)` to flush any pending output to the stream.

basic_filebuf::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_filebuf::underflow

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, and return the element as `T::to_int_type(c)`. It can do so in various ways:

- If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.
- It can read one or more elements of type *char*, as if by successive calls of the form `fgetc(fp)`, and convert them to an element `c` of type `E` by using the file conversion facet `fac` to call `fac.in` as needed. If any read or conversion fails, the function does not succeed.

If the function cannot succeed, it returns `T::eof()`. Otherwise, it returns `c`, converted as described above.

basic_fstream

```
template <class E, class T = char_traits<E> >
    class basic_fstream : public basic_iostream<E, T> {
public:
    typedef E char_type;
    typedef T traits_type;
```

```
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit basic_fstream();
    explicit basic_fstream(const char *s,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    basic_filebuf<E, T> *rdbuf() const;
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    void close();
    };
```

The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class `basic_filebuf<E, T>`, with elements of type `E`, whose character traits are determined by the class `T`. The object stores an object of class `basic_filebuf<E, T>`.

basic_fstream::basic_fstream

```
explicit basic_fstream();
explicit basic_fstream(const char *s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

The first constructor initializes the base class by calling `basic_iostream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_iostream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

basic_fstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_fstream::close

```
 voidclose();
```

The member function calls `rdbuf()-> close()`.

basic_fstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_fstream::is_open

```
bool is_open();
```

The member function returns `rdbuf()-> is_open()`.

basic_fstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_fstream::open

```
void open(const char *s,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

The member function calls `rdbuf()-> open(s, mode)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_fstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_fstream::rdbuf

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_filebuf<E, T>`.

basic_fstream::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_ifstream

```
template <class E, class T = char_traits<E> >
    class basic_ifstream : public basic_istream<E, T> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit basic_ifstream();
    explicit basic_ifstream(const char *s,
        ios_base::openmode mode = ios_base::in);
    basic_filebuf<E, T> *rdbuf() const;
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode = ios_base::in);
    void close();
    };
```

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_filebuf<E, T>`, with elements of type `E`, whose character traits are determined by the class `T`. The object stores an object of class `basic_filebuf<E, T>`.

basic_ifstream::basic_ifstream

```
explicit basic_ifstream();
explicit basic_ifstream(const char *s,
    ios_base::openmode mode = ios_base::in);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_filebuf<E, T>`. It also initializes `sb` by calling `basic_filebuf<E, T>()`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_filebuf<E, T>()`, then `sb.open(s, mode | ios_base::in)`. If the latter function returns a null pointer, the constructor calls `setstate(failbit)`.

basic_ifstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_ifstream::close

```
void close();
```

The member function calls rdbuf()-> close().

basic_ifstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for T::int_type.

basic_ifstream::is_open

```
bool is_open();
```

The member function returns rdbuf()-> is_open().

basic_ifstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for T::off_type.

basic_ifstream::open

```
void open(const char *s,
    ios_base::openmode mode = ios_base::in);
```

The member function calls rdbuf()-> open(s, mode | ios_base::in). If that function returns a null pointer, the function calls setstate(failbit).

basic_ifstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for T::pos_type.

basic_ifstream::rdbuf

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to basic_filebuf<E, T>.

basic_ifstream::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

basic_ofstream

```
template <class E, class T = char_traits<E> >
    class basic_ofstream : public basic_ostream<E, T> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit basic_ofstream();
    explicit basic_ofstream(const char *s,
        ios_base::openmode mode = ios_base::out | ios_base::trunc);
    basic_filebuf<E, T> *rdbuf() const;
    bool is_open() const;
    void open(const char *s,
        ios_base::openmode mode = ios_base::out | ios_base::trunc);
    void close();
    };
```

The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class basic_filebuf<E, T>, with elements of type E, whose character traits are determined by the class T. The object stores an object of class basic_filebuf<E, T>.

basic_ofstream::basic_ofstream

```
explicit basic_ofstream();
explicit basic_ofstream(const char *s,
    ios_base::openmode which = ios_base::out | ios_base::trunc);
```

The first constructor initializes the base class by calling basic_ostream(sb), where sb is the stored object of class basic_filebuf<E, T>. It also initializes sb by calling basic_filebuf<E, T>().

The second constructor initializes the base class by calling basic_ostream(sb). It also initializes sb by calling basic_filebuf<E, T>(), then sb.open(s, mode | ios_base::out). If the latter function returns a null pointer, the constructor calls setstate(failbit).

basic_ofstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_ofstream::close

```
void close();
```

The member function calls `rdbuf()-> close()`.

basic_ofstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_ofstream::is_open

```
bool is_open();
```

The member function returns `rdbuf()-> is_open()`.

basic_ofstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_ofstream::open

```
void open(const char *s,
    ios_base::openmode mode = ios_base::out | ios_base::trunc);
```

The member function calls `rdbuf()-> open(s, mode | ios_base::out)`. If that function returns a null pointer, the function calls `setstate(failbit)`.

basic_ofstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_ofstream::rdbuf

```
basic_filebuf<E, T> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to basic_filebuf<E, T>.

basic_ofstream::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

filebuf

```
typedef basic_filebuf<char, char_traits<char> > filebuf;
```

The type is a synonym for template class basic_filebuf, specialized for elements of type *char* with default character traits.

fstream

```
typedef basic_fstream<char, char_traits<char> > fstream;
```

The type is a synonym for template class basic_fstream, specialized for elements of type *char* with default character traits.

ifstream

```
typedef basic_ifstream<char, char_traits<char> > ifstream;
```

The type is a synonym for template class basic_ifstream, specialized for elements of type *char* with default character traits.

ofstream

```
typedef basic_ofstream<char, char_traits<char> > ofstream;
```

The type is a synonym for template class basic_ofstream, specialized for elements of type *char* with default character traits.

wfstream

```
typedef basic_fstream<wchar_t, char_traits<wchar_t> > wfstream;
```

The type is a synonym for template class basic_fstream, specialized for elements of type wchar_t with default character traits.

wifstream

```
typedef basic_ifstream<wchar_t, char_traits<wchar_t> > wifstream;
```

The type is a synonym for template class basic_ifstream, specialized for elements of type wchar_t with default character traits.

wofstream

```
typedef basic_ofstream<wchar_t, char_traits<wchar_t> > wofstream;
```

The type is a synonym for template class basic_ofstream, specialized for elements of type wchar_t with default character traits.

wfilebuf

```
typedef basic_filebuf<wchar_t, char_traits<wchar_t> > wfilebuf;
```

The type is a synonym for template class basic_filebuf, specialized for elements of type wchar_t with default character traits.

See also the **Table of Contents** and the **Index**.

<functional>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

binary_function · binary_negate · binder1st · binder2nd · divides · equal_to · greater · greater_equal · less · less_equal · logical_and · logical_not · logical_or · mem_fun_t · mem_fun_ref_t · mem_fun1 · mem_fun1_ref_t · minus · modulus · multiplies · negate · not_equal_to · plus · pointer_to_binary_function · pointer_to_unary_function · unary_function · unary_negate

bind1st · bind2nd · mem_fun · mem_fun_ref · mem_fun1 · mem_fun1_ref · not1 · not2 · ptr_fun

---

```
namespace std {
//  TEMPLATE CLASSES
template<class Arg, class Result>
    struct unary_function;
template<class Arg1, class Arg2, class Result>
    struct binary_function;
template<class T>
    struct plus;
template<class T>
    struct minus;
template<class T>
    struct multiplies;
template<class T>
    struct divides;
template<class T>
    struct modulus;
template<class T>
    struct negate;
template<class T>
    struct equal_to;
template<class T>
    struct not_equal_to;
template<class T>
    struct greater;
template<class T>
    struct less;
```

```
template<class T>
    struct greater_equal;
template<class T>
    struct less_equal;
template<class T>
    struct logical_and;
template<class T>
    struct logical_or;
template<class T>
    struct logical_not;
template<class Pred>
    struct unary_negate;
template<class Pred>
    struct binary_negate;
template<class Pred>
    class binder1st;
template<class Pred>
    class binder2nd;
template<class Arg, class Result>
    class pointer_to_unary_function;
template<class Arg1, class Arg2, class Result>
    class pointer_to_binary_function;
template<class R, class T>
    struct mem_fun_t;
template<class R, class T, class A>
    struct mem_fun1_t;
template<class R, class T>
    struct mem_fun_ref_t;
template<class R, class T, class A>
    struct mem_fun1_ref_t;
//  TEMPLATE FUNCTIONS
template<class Pred>
    unary_negate<Pred> not1(const Pred& pr);
template<class Pred>
    binary_negate<Pred> not2(const Pred& pr);
template<class Pred, class T>
    binder1st<Pred> bind1st(const Pred& pr, const T& x);
template<class Pred, class T>
    binder2nd<Pred> bind2nd(const Pred& pr, const T& x);
template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*)(Arg));
template<class Arg1, class Arg2, class Result>
```

```
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*)(Arg1, Arg2));
template<class R, class T>
    mem_fun_t<R, T> mem_fun(R (T::*pm)());
template<class R, class T, class A>
    mem_fun1_t<R, T, A> mem_fun1(R (T::*pm)(A arg));
template<class R, class T>
    mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)());
template<class R, class T, class A>
    mem_fun1_ref_t<R, T, A> mem_fun1_ref(R (T::*pm)(A arg));
    };
```

Include the STL standard header **<functional>** to define several templates that help construct **function objects**, objects of a class that defines `operator()`. Hence, function objects behave much like function pointers, except that the object can store additional information that can be used during a function call.

binary_function

```
template<class Arg1, class Arg2, class Result>
    struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
    };
```

The template class serves as a base for classes that define a member function of the form:

```
result_type operator()(first_argument_type, second_argument_type)
```

Hence, all such **binary functions** can refer to their first argument type as **first_argument_type**, their second argument type as **second_argument_type**, and their return type as **result_type**.

binary_negate

```
template<class Pred>
    class binary_negate
        : public binary_function<Pred::first_argument_type,
            Pred::second_argument_type, bool> {
public:
    explicit binary_negate(const Pred& pr);
    bool operator()(const first_argument_type& x,
        const second_argument_type& y) const;
    };
```

The template class stores a copy of `pr`, which must be a binary function object. It defines its member function **`operator()`** as returning `!pr(x, y)`.

bind1st

```
template<class Pred, class T>
    binder1st<Pred> bind1st(const Pred& pr, const T& x);
```

The function returns `binder1st<Pred>(pr, Pred::first_argument_type(x))`.

bind2nd

```
template<class Pred, class T>
    binder2nd<Pred> bind2nd(const Pred& pr, const T& y);
```

The function returns `binder2nd<Pred>(pr, Pred::second_argument_type(y))`.

binder1st

```
template<class Pred>
    class binder1st
        : public unary_function<Pred::second_argument_type,
            Pred::result_type> {
public:
    binder1st(const Pred& pr, const Pred::first_argument_type x);
    result_type operator()(const argument_type& y) const;
protected:
    Pred op;
    Pred::first_argument_type value;
    };
```

The template class stores a copy of `pr`, which must be a binary function object, in **`op`**, and a copy of `x` in **`value`**. It defines its member function **`operator()`** as returning `op(value, y)`.

binder2nd

```
template<class Pred>
    class binder2nd
        : public unary_function<Pred::first_argument_type,
            Pred::result_type> {
public:
    binder2nd(const Pred& pr, const Pred::second_argument_type y);
    result_type operator()(const argument_type& x) const;
```

```
protected:
    Pred op;
    Pred::second_argument_type value;
    };
```

The template class stores a copy of `pr`, which must be a binary function object, in **`op`**, and a copy of `y` in **`value`**. It defines its member function **`operator()`** as returning `op(x, value)`.

divides

```
template<class T>
    struct divides : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x / y`.

equal_to

```
template<class T>
    struct equal_to : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x == y`.

greater

```
template<class T>
    struct greater : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x > y`. The member function defines a total ordering, even if `T` is an object pointer type.

greater_equal

```
template<class T>
    struct greater_equal : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x >= y`. The member function defines a

total ordering, even if T is an object pointer type.

less

```
template<class T>
    struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x < y`. The member function defines a total ordering, even if T is an object pointer type.

less_equal

```
template<class T>
    struct less_equal : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x <= y`. The member function defines a total ordering, even if T is an object pointer type.

logical_and

```
template<class T>
    struct logical_and : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x && y`.

logical_not

```
template<class T>
    struct logical_not : public unary_function<T, bool> {
    bool operator()(const T& x) const;
    };
```

The template class defines its member function as returning `!x`.

logical_or

```
template<class T>
    struct logical_or : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x || y`.

mem_fun

```
template<class R, class T>
    mem_fun_t<R, T> mem_fun(R (T::*pm)());
```

The template function returns `mem_fun_t<R, T>(pm)`.

mem_fun_t

```
template<class R, class T>
    struct mem_fun_t : public unary_function<T *, R> {
    explicit mem_fun_t(R (T::*pm)());
    R operator()(T *p);
    };
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function **`operator()`** as returning `(p->*pm)()`.

mem_fun_ref

```
template<class R, class T>
    mem_fun_ref_t<R, T> mem_fun_ref(R (T::*pm)());
```

The template function returns `mem_fun_ref_t<R, T>(pm)`.

mem_fun_ref_t

```
template<class R, class T>
    struct mem_fun_ref_t : public unary_function<T *, R> {
    explicit mem_fun_t(R (T::*pm)());
    R operator()(T& x);
    };
```

The template class stores a copy of `pm`, which must be a pointer to a member function of class `T`, in a private member object. It defines its member function **`operator()`** as returning `(x.*Pm)()`.

mem_fun1

```
template<class R, class T, class A>
    mem_fun1_t<R, T, A> mem_fun1(R (T::*pm)(A));
```

The template function returns mem_fun1_t<R, T, A>(pm).

mem_fun1_t

```
template<class R, class T, class A>
    struct mem_fun1_t : public binary_function<T *, A, R> {
    explicit mem_fun1_t(R (T::*pm)(A));
    R operator()(T *p, A arg);
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class T, in a private member object. It defines its member function **operator()** as returning (p->*pm)(arg).

mem_fun1_ref

```
template<class R, class T, class A>
    mem_fun1_ref_t<R, T, A> mem_fun1_ref(R (T::*pm)(A));
```

The template function returns mem_fun1_ref_t<R, T, A>(pm).

mem_fun1_ref_t

```
template<class R, class T, class A>
    struct mem_fun1_ref_t : public binary_function<T *, A, R> {
    explicit mem_fun1_ref_t(R (T::*pm)(A));
    R operator()(T& x, A arg);
    };
```

The template class stores a copy of pm, which must be a pointer to a member function of class T, in a private member object. It defines its member function **operator()** as returning (x.*pm)(arg).

minus

```
template<class T>
    struct minus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x - y`.

modulus

```
template<class T>
    struct modulus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x % y`.

multiplies

```
template<class T>
    struct multiplies : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x * y`.

negate

```
template<class T>
    struct negate : public unary_function<T, T> {
    T operator()(const T& x) const;
    };
```

The template class defines its member function as returning `-x`.

not1

```
template<class Pred>
    unary_negate<Pred> not1(const Pred& pr);
```

The template function returns `unary_negate<Pred>(pr)`.

not2

```
template<class Pred>
    binary_negate<Pred> not2(const Pred& pr);
```

The template function returns `binary_negate<Pred>(pr)`.

not_equal_to

```
template<class T>
    struct not_equal_to : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x != y`.

plus

```
template<class T>
    struct plus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const;
    };
```

The template class defines its member function as returning `x + y`.

pointer_to_binary_function

```
template<class Arg1, class Arg2, class Result>
    class pointer_to_binary_function
        : public binary_function<Arg1, Arg2, Result> {
public:
    explicit pointer_to_binary_function(Result (*pf)(Arg1, Arg2));
    Result operator()(const Arg1 x, const Arg2 y) const;
    };
```

The template class stores a copy of `pf`. It defines its member function **`operator()`** as returning `(*pf)(x, y)`.

pointer_to_unary_function

```
template<class Arg, class Result>
    class pointer_to_unary_function
        : public unary_function<Arg, Result> {
public:
    explicit pointer_to_unary_function(Result (*pf)(Arg));
    Result operator()(const Arg x) const;
    };
```

The template class stores a copy of `pf`. It defines its member function **`operator()`** as returning `(*pf)(x)`.

ptr_fun

```
template<class Arg, class Result>
    pointer_to_unary_function<Arg, Result>
        ptr_fun(Result (*pf)(Arg));
template<class Arg1, class Arg2, class Result>
    pointer_to_binary_function<Arg1, Arg2, Result>
        ptr_fun(Result (*pf)(Arg1, Arg2));
```

The first template function returns `pointer_to_unary_function<Arg, Result>(pf)`.

The second template function returns `pointer_to_binary_function<Arg1, Arg2, Result>(pf)`.

unary_function

```
template<class Arg, class Result>
    struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
    };
```

The template class serves as a base for classes that define a member function of the form:

```
result_type operator()(argument_type)
```

Hence, all such **unary functions** can refer to their sole argument type as **`argument_type`** and their return type as **`result_type`**.

unary_negate

```
template<class Pred>
    class unary_negate
        : public unary_function<Pred::argument_type, bool> {
public:
    explicit unary_negate(const Pred& pr);
    bool operator()(const argument_type& x) const;
    };
```

The template class stores a copy of `pr`, which must be a unary function object. It defines its member function **`operator()`** as returning `!pr(x)`.

See also the **Table of Contents** and the **Index**.

<iomanip>

```
namespace std {
//      MANIPULATORS
T1 resetiosflags(ios_base::fmtflags mask);
T2 setiosflags(ios_base::fmtflags mask);
T3 setbase(int base);
template<class E>
    T4 setfill(E c);
T5 setprecision(int n);
T6 setw(int n);
    };
```

Include the iostreams standard header **<iomanip>** to define several **manipulators** that each take a single argument. Each of these manipulators returns an unspecified type, called `T1` through `T6` here, that overloads both `basic_istream<E, T>::operator>>` and `basic_ostream<E, T>::operator<<`. Thus, you can write extractors and inserters such as:

```
cin >> setbase(8);
cout << setbase(8);
```

resetiosflags

```
T1 resetiosflags(ios_base::fmtflags mask);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(ios_base:: fmtflags(), mask)`, then returns `str`.

setiosflags

```
T2 setiosflags(ios_base::fmtflags mask);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(mask)`, then returns `str`.

setbase

```
T3 setbase(int base);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.setf(mask, ios_base::basefield)`, then returns `str`. Here, `mask` is determined as follows:

- If `base` is 8, then `mask` is `ios_base::oct`
- If `base` is 10, then `mask` is `ios_base::dec`
- If `base` is 16, then `mask` is `ios_base::hex`
- If `base` is any other value, then `mask` is `ios_base::fmtflags(0)`

setfill

```
template<class E>
    T4 setfill(E fillch);
```

The template manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.fill(fillch)`, then returns `str`. The type `E` must be the same as the element type for the stream `str`.

setprecision

```
T5 setprecision(int prec);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.precision(prec)`, then returns `str`.

setw

```
T6 setw(int wide);
```

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.width(wide)`, then returns `str`.

See also the **Table of Contents** and the **Index**.

<ios>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

basic_ios · fpos · ios · ios_base · streamoff · streampos · streamsize · wios · wstreampos

boolalpha · dec · fixed · hex · internal · left · noboolalpha · noshowbase · noshowpoint · noshowpos · noskipws · nounitbuf · nouppercase · oct · right · scientific · showbase · showpoint · showpos · skipws · unitbuf · uppercase

---

```
namespace std {
    typedef T1 streamoff;
    typedef T2 streamsize;
    class ios_base;
//      TEMPLATE CLASSES
    template <class E, class T = char_traits<E> >
        class basic_ios;
    typedef basic_ios<char, char_traits<char> > ios;
    typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
    template <class St>
        class fpos;
    typedef fpos<mbstate_t> streampos;
    typedef fpos<mbstate_t> wstreampos;
//      MANIPULATORS
    ios_base& boolalpha(ios_base& str);
    ios_base& noboolalpha(ios_base& str);
    ios_base& showbase(ios_base& str);
    ios_base& noshowbase(ios_base& str);
    ios_base& showpoint(ios_base& str);
    ios_base& noshowpoint(ios_base& str);
    ios_base& showpos(ios_base& str);
    ios_base& noshowpos(ios_base& str);
    ios_base& skipws(ios_base& str);
    ios_base& noskipws(ios_base& str);
    ios_base& unitbuf(ios_base& str);
    ios_base& nounitbuf(ios_base& str);
    ios_base& uppercase(ios_base& str);
```

```
    ios_base& nouppercase(ios_base& str);
    ios_base& internal(ios_base& str);
    ios_base& left(ios_base& str);
    ios_base& right(ios_base& str);
    ios_base& dec(ios_base& str);
    ios_base& hex(ios_base& str);
    ios_base& oct(ios_base& str);
    ios_base& fixed(ios_base& str);
    ios_base& scientific(ios_base& str);
    };
```

Include the iostreams standard header **<ios>** to define several types and functions basic to the operation of iostreams. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

A large group of functions are **manipulators**. The manipulators declared in `<ios>` alter the values stored in its argument object of class `ios_base`. Other manipulators perform actions on streams controlled by objects of a type derived from this class, such as a specialization of one of the template classes `basic_istream` or `basic_ostream`. For example, `noskipws(istr)` clears the format flag `ios_base::skipws` in the object `istr`, which might be of type `istream`.

You can also call a manipulator by inserting it into an output stream or extracting it from an input stream, thanks to some special machinery supplied in the classes derived from `ios_base`. For example:

```
istr >> noskipws;
```

calls `noskipws(istr)`.

basic_ios

---

bad · **basic_ios** · **char_type** · **clear** · **copyfmt** · **eof** · **exceptions** · **init** · **fail** · **good** · **imbue** · **init** · **int_type** · **narrow** · **off_type** · **operator!** · **operator void *** · **pos_type** · **rdbuf** · **rdstate** · **setstate** · **tie** · **widen**

```
template <class E, class T = char_traits<E> >
    class basic_ios : public ios_base {
public:
    typedef E char_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
```

```
    explicit basic_ios(basic_streambuf<E, T>* sb);
    virtual ~basic_ios();
    operator void *() const;
    bool operator!() const;
    iostate rdstate() const;
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;
    iostate exceptions() const;
    iostate exceptions(iostate except);
    basic_ios& copyfmt(const basic_ios& rhs);
    E fill() const;
    E fill(E ch);
    basic_ostream<E, T> *tie() const;
    basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
    basic_streambuf<E, T> *rdbuf() const;
    basic_streambuf<E, T> *rdbuf(basic_streambuf<E, T> *sb);
    basic_ios& copyfmt(const basic_ios& rhs);
    locale imbue(const locale& loc);
    E widen(char ch);
    char narrow(E ch, char dflt);
protected:
    basic_ios();
    void init(basic_streambuf<E, T>* sb);
    };
```

The template class describes the storage and member functions common to both input streams (of template class `basic_istream`) and output streams (of template class `basic_ostream`) that depend on the template parameters. (The class `ios_base` describes what is common and *not* dependent on template parameters. An object of class `basic_ios<E, T>` helps control a stream with elements of type `E`, whose character traits are determined by the class `T`.

An object of class `basic_ios<E, T>` stores:

- **formatting information** and
- **stream state information** in a base object of type `ios_base`
- a **fill character** in an object of type *E*
- a **tie pointer** to an object of type `basic_ostream<E, T>`
- a **stream buffer pointer** to an object of type `basic_streambuf<E, T>`

basic_ios::bad

```
bool bad() const;
```

The member function returns true if rdstate() & badbit.

basic_ios::basic_ios

```
explicit basic_ios(basic_streambuf<E, T>* sb);
basic_ios();
```

The first constructor initializes its member objects by calling init(sb). The second (protected) constructor leaves its member objects uninitialized. A later call to init *must* initialize the object before it can be safely destroyed.

basic_ios::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

basic_ios::clear

```
void clear(iostate state = goodbit);
```

The member function replaces the stored stream state information with state | (rdbuf() != 0 ? goodbit : badbit). If state & exceptions() is nonzero, it then throws an object of class failure.

basic_ios::copyfmt

```
basic_ios& copyfmt(const basic_ios& rhs);
```

The member function reports the callback event erase_event. It then copies from rhs into *this the fill character, the tie pointer, and the formatting information. Before altering the exception mask, it reports the callback event copyfmt_event. If, after the copy is complete, state & exceptions() is nonzero, the function effectively calls clear with the argument rdstate(). It returns *this.

basic_ios::eof

```
bool eof() const;
```

The member function returns true if rdstate() & eofbit.

basic_ios::exceptions

```
iostate exceptions() const;
iostate exceptions(iostate except);
```

The first member function returns the stored exception mask. The second member function stores `except` in the exception mask and returns its previous stored value. Note that storing a new exception mask can throw an exception just like the call `clear( rdstate())`.

basic_ios::fail

```
bool fail() const;
```

The member function returns true if `rdstate() & failbit`.

basic_ios::fill

```
E fill() const;
E fill(E ch);
```

The first member function returns the stored fill character. The second member function stores `ch` in the fill character and returns its previous stored value.

basic_ios::good

```
bool good() const;
```

The member function returns true if `rdstate() == goodbit` (no state flags are set).

basic_ios::imbue

```
locale imbue(const locale& loc);
```

If `rdbuf` is not a null pointer, the member function calls `rdbuf()->pubimbue(loc)`. In any case, it returns `ios_base::imbue(loc)`.

basic_ios::init

```
void init(basic_streambuf<E, T>* sb);
```

The member function stores values in all member objects, so that:

- `rdbuf()` returns `sb`
- `tie()` returns a null pointer
- `rdstate()` returns `goodbit` if `sb` is nonzero; otherwise, it returns `badbit`
- `exceptions()` returns `goodbit`

- `flags()` returns `skipws | dec`
- `width()` returns zero
- `precision()` returns 6
- `fill()` returns the space character
- `getloc()` returns `locale::classic()`
- `iword` returns zero and `pword` returns a null pointer for all argument value

basic_ios::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_ios::narrow

```
char narrow(E ch, char dflt);
```

The member function returns `use_facet< ctype<E> >( getloc()). narrow(ch, dflt)`.

basic_ios::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_ios::operator void *

```
operator void *() const;
```

The operator returns a null pointer only if `fail()`.

basic_ios::operator!

```
bool operator!() const;
```

The operator returns `fail()`.

basic_ios::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_ios::rdbuf

```
basic_streambuf<E, T> *rdbuf() const;
basic_streambuf<E, T> *rdbuf(basic_streambuf<E, T> *sb);
```

The member function returns the stored stream buffer pointer.

basic_ios::rdstate

```
iostate rdstate() const;
```

The member function returns the stored stream state information.

basic_ios::setstate

```
void setstate(iostate state);
```

The member function effectively calls `clear(state | rdstate())`.

basic_ios::tie

```
basic_ostream<E, T> *tie() const;
basic_ostream<E, T> *tie(basic_ostream<E, T> *str);
```

The first member function returns the stored tie pointer. The second member function stores `str` in the tie pointer and returns its previous stored value.

basic_ios::widen

```
E narrow(char ch);
```

The member function returns `use_facet< ctype<E> >( getloc()). widen(ch)`.

boolalpha

```
ios_base& boolalpha(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: boolalpha)`, then returns `str`.

dec

```
ios_base& dec(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: dec, ios_base:: basefield)`, then returns `str`.

fixed

```
ios_base& fixed(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: fixed, ios_base:: floatfield)`, then returns `str`.

fpos

```
template <class St>
    class fpos {
public:
    fpos(St state, fpos_t fposn);
    fpos(streamoff off);
    fpos_t get_fpos_t() const;
    St state() const;
    void state(St state);
    operator streamoff() const;
    streamoff operator-(const fpos<St>& rhs) const;
    fpos<St>& operator+=(streamoff off);
    fpos<St>& operator-=(streamoff off);
    fpos<St> operator+(streamoff off) const;
    fpos<St> operator-(streamoff off) const;
    bool operator==(const fpos<St>& rhs) const;
    bool operator!=(const fpos<St>& rhs) const;
    };
```

The template class describes an object that can store all the information needed to restore an arbitrary file-position indicator within any stream. An object of class `fpos<St>` effectively stores three member objects:

- a byte offset, of type `streamoff`
- an arbitrary file position, for use by an object of class `basic_filebuf`, of type `fpos_t`
- a conversion state, for use by an object of class `basic_filebuf`, of type **`St`**, typically `mbstate_t`

For an environment with limited file size, however, `streamoff` and `fpos_t` may sometimes be used interchangeably. And for an environment with no streams that have a state-dependent encoding, `mbstate_t` may actually be unused. So the number of member objects stored may well vary from one to three.

fpos::fpos

```
fpos(St state, fpos_t fposn);
fpos(streamoff off);
```

The first constructor stores a zero offset and the objects `state` and `fposn`. The second constructor stores the offset `off`, relative to the beginning of file and in the initial conversion state (if that matters). If `off` is -1, the resulting object represents an invalid stream position.

fpos::get_fpos_t

```
fpos_t get_fpos_t() const;
```

returns the value stored in the `fpos_t` member object.

fpos::operator!=

```
bool operator!=(const fpos<St>& rhs) const;
```

The member function returns `!(*this == rhs)`.

fpos::operator+

```
fpos<St> operator+(streamoff off) const;
```

The member function returns `fpos<St>(*this) += off`.

fpos::operator+=

```
fpos<St>& operator+=(streamoff off);
```

The member function adds `off` to the stored offset member object, then returns `*this`. For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

fpos::operator-

```
streamoff operator-(const fpos<St>& rhs) const;
fpos<St> operator-(streamoff off) const;
```

The first member function returns `(streamoff)*this - (streamoff)rhs`. The second member function returns `fpos<St>(*this) -= off`.

fpos::operator-=

```
fpos<St>& operator-=(streamoff off);
```

The member function returns `fpos<St>(*this) -= off`. For positioning within a file, the result is generally valid only for binary streams that do not have a state-dependent encoding.

fpos::operator==

```
bool operator==(const fpos<St>& rhs) const;
```

The member function returns `(streamoff)*this == (streamoff)rhs`.

fpos::operator streamoff

```
operator streamoff() const;
```

The member function returns the stored offset member object, plus any additional offset stored as part of the `fpos_t` member object.

fpos::state

```
St state() const;
void state(St state);
```

The first member function returns the value stored in the `St` member object. The second member function stores `state` in the `St` member object.

hex

```
ios_base& hex(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: hex, ios_base:: basefield)`, then returns `str`.

internal

```
ios_base& internal(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: internal, ios_base:: adjustfield)`, then returns `str`.

ios

```
typedef basic_ios<char, char_traits<char> > ios;
```

The type is a synonym for template class basic_ios, specialized for elements of type *char* with default character traits.

ios_base

---

event · event_callback · failure · flags · fmtflags · getloc · imbue · Init · ios_base · iostate · iword · openmode · operator= · precision · pword · register_callback · seekdir · setf · sync_with_stdio · unsetf · width · xalloc

---

```
class ios_base {
public:
    class failure;
    typedef T1 fmtflags;
    static const fmtflags boolalpha, dec, fixed, hex, internal,
        left, oct, right, scientific, showbase, showpoint,
        showpos, skipws, unitbuf, uppercase, adjustfield,
        basefield, floatfield;
    typedef T2 iostate;
    static const iostate badbit, eofbit, failbit, goodbit;
    typedef T3 openmode;
    static const openmode app, ate, binary, in, out, trunc;
    typedef T4 seekdir;
    static const seekdir beg, cur, end;
    typedef T5 event;
    static const event copyfmt_event, erase_event,
        copyfmt_event;
    class Init;
    ios_base& operator=(const ios_base& rhs);
    fmtflags flags() const;
    fmtflags flags(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl);
    fmtflags setf(fmtflags fmtfl, fmtflags mask);
    void unsetf(fmtflags mask);
    streamsize precision() const;
    streamsize precision(streamsize prec);
    streamsize width() const;
```

```
    stramsize width(streamsize wide);
    locale imbue(const locale& loc);
    locale getloc() const;
    static int xalloc();
    long& iword(int idx);
    void *& pword(int idx);
    typedef void *(event_callback(event ev, ios_base& ios, int idx);
    void register_callback(event_callback pfn, int idx);
    static bool sync_with_stdio(bool sync = true);
protected:
    ios_base();
    };
```

The class describes the storage and member functions common to both input and output streams that does not depend on the template parameters. (The template class basic_ios describes what is common and *is* dependent on template parameters.

An object of class ios_base stores **formatting information**, which consists of:

- **format flags** in an object of type fmtflags
- an **exception mask** in an object of type iostate
- a **field width** in an object of type *int*
- a **display precison** in an object of type *int*
- a **locale object** in an object of type locale
- two **extensible arrays**, with elements of type *long* and *void* pointer

An object of class ios_base also stores **stream state information**, in an object of type iostate, and a **callback stack**.

ios_base::event

```
typedef T5 event;
static const event copyfmt_event, erase_event,
    imbue_event;
```

The type is an enumerated type T5 that describes an object that can store the **callback event** used as an argument to a function registered with register_callback. The distinct event values are:

- **copyfmt_event**, to identify a callback that occurs near the end of a call to copyfmt, just before the exception mask is copied.
- **erase_event**, to identify a callback that occurs at the beginning of a call to copyfmt, or at the beginning of a call to the

destructor for *this.

- **imbue_event**, to identify a callback that occurs at the end of a call to imbue, just before the function returns.

ios_base::event_callback

```
typedef void *(event_callback(event ev, ios_base& ios, int idx);
```

The type describes a pointer to a function that can be registered with register_callback.

ios_base::failure

```
class failure : public exception {
public:
    explicit failure(const string& what_arg) {
    };
```

The member class serves as the base class for all exceptions thrown by the member function clear in template class basic_ios. The value returned by what() is what_arg.data().

ios_base::flags

```
fmtflags flags() const;
fmtflags flags(fmtflags fmtfl);
```

The first member function returns the stored format flags. The second member function stores fmtfl in the format flags and returns its previous stored value.

ios_base::fmtflags

```
typedef T1 fmtflags;
static const fmtflags boolalpha, dec, fixed, hex, internal,
    left, oct, right, scientific, showbase, showpoint,
    showpos, skipws, unitbuf, uppercase, adjustfield,
    basefield, floatfield;
```

The type is an enumerated type T1 that describes an object that can store format flags. The distinct flag values are:

- **boolalpha**, to insert or extract objects of type *bool* as names (such as true and false) rather than as numeric values
- **dec**, to insert or extract integer values in decimal format
- **fixed**, to insert floating-point values in fixed-point format (with

no exponent field)
- **hex**, to insert or extract integer values in hexadecimal format
- **internal**, to pad to a field width as needed by inserting fill characters at a point internal to a generated numeric field
- **left**, to pad to a field width as needed by inserting fill characters at the end of a generated field (left justification)
- **oct**, to insert or extract integer values in octal format
- **right**, to pad to a field width as needed by inserting fill characters at the beginning of a generated field (right justification)
- **scientific**, to insert floating-point values in scientific format (with an exponent field)
- **showbase**, to insert a prefix that reveals the base of a generated integer field
- **showpoint**, to insert a decimal point unconditionally in a generated floating-point field
- **showpos**, to insert a plus sign in a non-negative generated numeric field
- **skipws**, to skip leading white space before certain extractions
- **unitbuf**, to flush output after each insertion
- **uppercase**, to insert uppercase equivalents of lowercase letters in certain insertions

In addition, several useful values are:
- **adjustfield**, internal | left | right
- **basefield**, dec | hex | oct
- **floatfield**, fixed | scientific

ios_base::getloc

locale **getloc**() const;

The member function returns the stored locale object.

ios_base::imbue

locale **imbue**(const locale& loc);

The member function stores loc in the locale object, then reports the callback event imbue event. It returns the previous stored value.

ios_base::Init

```
class Init {
    };
```

The nested class describes an object whose construction ensures that the standard iostreams objects are properly constructed, even during the execution of a constructor for an arbitrary static object.

ios_base::ios_base

```
ios_base();
```

The (protected) constructor does nothing. A later call to basic_ios::init *must* initialize the object before it can be safely destroyed. Thus, the only safe use for class ios_base is as a base class for template class basic_ios.

ios_base::iostate

```
typedef T2 iostate;
static const iostate badbit, eofbit, failbit, goodbit;
```

The type is an enumerated type T2 that describes an object that can store stream state information. The distinct flag values are:

- **badbit**, to record a loss of integrity of the stream buffer
- **eofbit**, to record end-of-file while extracting from a stream
- **failbit**, to record a failure to extract a valid field from a strea

In addition, a useful value is:

- **goodbit**, no bits set

ios_base::iword

```
long& iword(int idx);
```

The member function returns a reference to element idx of the extensible array with elements of type *long*. All elements are effectively present and initially store the value zero. The returned reference is invalid after the next call to iword for the object, after the object is altered by a call to basic_ios::copyfmt, or after the object is destroyed.

To obtain a unique index, for use across all objects of type ios_base, call xalloc.

ios_base::openmode

typedef T3 **openmode**;
static const openmode **app**, **ate**, **binary**, **in**, **out**, **trunc**;

The type is an enumerated type T3 that describes an object that can store the **opening mode** for several iostreams objects. The distinct flag values are:

- **app**, to seek to the end of a stream before each insertion
- **ate**, to seek to the end of a stream when its controlling object is first created
- **binary**, to read a file as a binary stream, rather than as a text stream
- **in**, to permit extraction from a stream
- **out**, to permit insertion to a stream
- **trunc**, to truncate an existing file when its controlling object is first created

ios_base::operator=

ios_base& **operator=**(const ios_base& rhs) const;

The operator copies the stored formatting information, making a new copy of any extensible arrays. It then returns *this. Note that the callback stack is *not* copied.

ios_base::precision

streamsize **precision**() const;
streamsize **precision**(streamsize prec);

The first member function returns the stored display precision. The second member function stores prec in the display precision and returns its previous stored value.

ios_base::pword

void *& **pword**(int idx);

The member function returns a reference to element idx of the extensible array with elements of type *void* pointer. All elements are effectively present and initially store the null pointer. The returned reference is invalid after the next call to pword for the object, after the object is altered by a call to basic_ios::copyfmt, or after

the object is destroyed.

To obtain a unique index, for use across all objects of type ios_base, call xalloc.

ios_base::register_callback

void **register_callback**(event_callback pfn, int idx);

The member function pushes the pair {pfn, idx} onto the stored callback stack. When a callback event ev is reported, the functions are called, in reverse order of registry, by the expression (*pfn)(ev, *this, idx).

ios_base::seekdir

typedef T4 **seekdir**;
static const seekdir **beg**, **cur**, **end**;

The type is an enumerated type T4 that describes an object that can store the **seek mode** used as an argument to the member functions of several iostreams classes. The distinct flag values are:

- **beg**, to seek (alter the current read or write position) relative to the beginning oc a sequence (array, stream, or file)
- **cur**, to seek relative to the current position within a sequence
- **end**, to seek relative to the end of a sequence

ios_base::setf

void **setf**(fmtflags mask);
fmtflags **setf**(fmtflags fmtfl, fmtflags mask);

The first member function effectively calls flags(mask | flags()) (set selected bits), then returns the previous format flags. The second member function effectively calls flags(mask & fmtfl, flags() & ~mask) (replace selected bits under a mask), then returns the previous format flags.

ios_base::sync_with_stdio

static bool **sync_with_stdio**(bool sync = true);

The static member function stores a **stdio sync flag**, which is initially true. When true, this flag ensures that operations on the same file are properly synchronized between the iostreams functions and those defined in the Standard C library. Otherwise,

synchronization may or may not be guaranteed, but performance may be improved. The function stores sync in the stdio sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

ios_base::unsetf

```
void unsetf(fmtflags mask);
```

The member function effectively calls `flags(~mask & flags())` (clear selected bits).

ios_base::width

```
streamsize width() const;
streamsize width(streamsize wide);
```

The first member function returns the stored field width. The second member function stores `wide` in the field width and returns its previous stored value.

ios_base::xalloc

```
static int xalloc();
```

The static member function returns a stored static value, which it increments on each call. You can use the return value as a unique index argument when calling the member functions `iword` or `pword`.

left

```
ios_base& left(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: left, ios_base:: adjustfield)`, then returns `str`.

noboolalpha

```
ios_base& noboolalpha(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: boolalpha)`, then returns `str`.

noshowbase

```
ios_base& noshowbase(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: showbase)`, then returns `str`.

noshowpoint

```
ios_base& noshowpoint(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: showpoint)`, then returns `str`.

noshowpos

```
ios_base& noshowpos(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: showpos")`, then returns `str`.

noskipws

```
ios_base& noskipws(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: skipws)`, then returns `str`.

nounitbuf

```
ios_base& nounitbuf(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: unitbuf)`, then returns `str`.

nouppercase

```
ios_base& nouppercase(ios_base& str);
```

The manipulator effectively calls `str.unsetf(ios_base:: uppercase)`, then returns `str`.

oct

```
ios_base& oct(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base:: oct, ios_base:: basefield)`, then returns `str`.

right

```
ios_base& right(ios_base& str);
```

The maiipulator effectively calls `str.setf(ios_base:: right, ios_base:: adjustfield)`, then returns `str`.

scientific

```
ios_base& scientific(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::scientific, ios_base::floatfield)`, then returns `str`.

showbase

```
ios_base& showbase(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showbase)`, then returns `str`.

showpoint

```
ios_base& showpoint(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showpoint)`, then returns `str`.

showpos

```
ios_base& showpos(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::showpos)`, then returns `str`.

skipws

```
ios_base& skipws(ios_base& str);
```

The manipulator effectively calls `str.setf(ios_base::skipws)`, then returns `str`.

streamoff

```
typedef T1 streamoff;
```

The type is a signed integer type `T1` that describes an object that can store a byte offset involved in various stream positioning operations. Its representation has at least 32 value bits. It is *not* necessarily large enough to represent an arbitrary byte position within a stream. The value `streamoff(-1)` generally indicates an erroneous offset.

streampos

```
typedef fpos<mbstate_t> streampos;
```

The type is a synonym for fpos< mbstate_t>.

streamsize

```
typedef T2 streamsize;
```

The type is a signed integer type T3 that describes an object that can store a count of the number of elements involved in various stream operations. Its representation has at least 16 bits. It is *not* necessarily large enough to represent an arbitrary byte position within a stream.

unitbuf

```
ios_base& unitbuf(ios_base& str);
```

The manipulator effectively calls str.setf(ios_base:: unitbuf), then returns str.

uppercase

```
ios_base& uppercase(ios_base& str);
```

The manipulator effectively calls str.setf(ios_base:: uppercase), then returns str.

wios

```
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
```

The type is a synonym for template class basic_ios, specialized for elements of type wchar_t with default character traits.

wstreampos

```
typedef fpos<mbstate_t> wstreampos;
```

The type is a synonym for fpos< mbstate_t>.

See also the **Table of Contents** and the **Index**.

<iosfwd>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
        // TYPE DEFINITIONS
typedef T1 streamoff;
typedef T2 streampos;
        // TEMPLATES
template<class E>
    class char_traits;
class char_traits<char>;
class char_traits<wchar_t>;
template<class E, class T = char_traits<E> >
    class basic_ios;
template<class E, class T = char_traits<E> >
    class istreambuf_iterator;
template<class E, class T = char_traits<E> >
    class ostreambuf_iterator;
template<class E, class T = char_traits<E> >
    class basic_streambuf;
template<class E, class T = char_traits<E> >
    class basic_istream;
template<class E, class T = char_traits<E> >
    class basic_ostream;
template<class E, class T = char_traits<E> >
    class basic_iostream;
template<class E, class T = char_traits<E> >
    class basic_stringbuf;
template<class E, class T = char_traits<E> >
    class basic_istringstream;
template<class E, class T = char_traits<E> >
    class basic_ostringstream;
template<class E, class T = char_traits<E> >
    class basic_stringstream;
template<class E, class T = char_traits<E> >
    class basic_filebuf;
template<class E, class T = char_traits<E> >
    class basic_ifstream;
template<class E, class T = char_traits<E> >
    class basic_ofstream;
template<class E, class T = char_traits<E> >
    class basic_fstream;
```

```
        // char TYPE DEFINITIONS
typedef basic_ios<char, char_traits<char> > ios;
typedef basic_streambuf<char, char_traits<char> > streambuf;
typedef basic_istream<char, char_traits<char> > istream;
typedef basic_ostream<char, char_traits<char> > ostream;
typedef basic_iostream<char, char_traits<char> > iostream;
typedef basic_stringbuf<char, char_traits<char> > stringbuf;
typedef basic_istringstream<char, char_traits<char> > istringstream;
typedef basic_ostringstream<char, char_traits<char> > ostringstream;
typedef basic_stringstream<char, char_traits<char> > stringstream;
typedef basic_filebuf<char, char_traits<char> > filebuf;
typedef basic_ifstream<char, char_traits<char> > ifstream;
typedef basic_ofstream<char, char_traits<char> > ofstream;
typedef basic_fstream<char, char_traits<char> > fstream;
        // wchar_t TYPE DEFINITIONS
typedef basic_ios<wchar_t, char_traits<wchar_t> > wios;
typedef basic_streambuf<wchar_t, char_traits<wchar_t> > wstreambuf;
typedef basic_istream<wchar_t, char_traits<wchar_t> > wistream;
typedef basic_ostream<wchar_t, char_traits<wchar_t> > wostream;
typedef basic_iostream<wchar_t, char_traits<wchar_t> > wiostream;
typedef basic_stringbuf<wchar_t, char_traits<wchar_t> > wstringbuf;
typedef basic_istringstream<wchar_t, char_traits<wchar_t> > wistringstream;
typedef basic_ostringstream<wchar_t, char_traits<wchar_t> > wostringstream;
typedef basic_stringstream<wchar_t, char_traits<wchar_t> > wstringstream;
typedef basic_filebuf<wchar_t, char_traits<wchar_t> > wfilebuf;
typedef basic_ifstream<wchar_t, char_traits<wchar_t> > wifstream;
typedef basic_ofstream<wchar_t, char_traits<wchar_t> > wofstream;
typedef basic_fstream<wchar_t, char_traits<wchar_t> > wfstream;
    };
```

Include the iostreams standard header **<iosfwd>** to declare forward references to several template classes used throughout iostreams. All such template classes are defined in other standard headers. You include this header explicitly only when you need one of the above declarations, but not its definition.

See also the **Table of Contents** and the **Index**.

<iostream>

```
namespace std {
    extern istream cin;
    extern ostream cout;
    estern ostream cerr;
    extern ostream clog;
    extern wistream wcin;
    extern wostream wcout;
    extern wostream wcerr;
    extern wostream wclog;
    };
```

Include the iostreams standard header **<iostream>** to declare several objects that control reading from and writing to the standard streams. This is often the *only* header you need include to perform input and output from a C++ program.

The objects fall into two groups:

- `cin`, `cout`, `cerr`, and `clog` are **byte oriented**, performing conventional byte-at-a-time transfers
- `wcin`, `wcout`, `wcerr`, and `wclog` are **wide oriented**, translating to and from the wide characters that the program manipulates internally

Once you perform certain operations on a stream, such as the standard input, you cannot perform operations of a different orientation on the same stream. Hence, a program cannot operate interchangeably on both `cin` and `wcin`, for example.

All the objects declared in this header share a peculiar property -- you can assume they are **constructed** before any static objects you define, in a translation unit that includes `<iostreams>`. Equally, you can assume that these objects are **not destroyed** before the destructors for any such static objects you define. (The output streams are, however, flushed during program termination.) Hence, you can safely read from or write to the standard streams prior to program startup and after program termination.

This guarantee is *not* universal, however. A static constructor may call a function in another translation unit. The called function cannot assume that the objects declared in this header have been constructed, given the uncertain order in which translation units participate in static construction. To use these objects in such a context, you must first construct an object of class **`ios_base::Init`**, as in:

```
#include <iostream>
void marker()
    {    // called by some constructor
```

```
    ios_base::Init unused_name;
    cout <<: "called fun" << endl;
    }
```

cerr

```
extern ostream cerr;
```

The object controls unbuffered insertions to the standard error output as a byte stream. Once the object is constructed, the expression `cerr.flags() & unitbuf` is nonzero.

cin

```
extern istream cin;
```

The object controls extractions from the standard input as a byte stream. Once the object is constructed, the call `cin.tie()` returns `&cout`.

clog

```
extern ostream clog;
```

The object controls buffered insertions to the standard error output as a byte stream.

cout

```
extern ostream cout;
```

The object controls insertions to the standard output as a byte stream.

wcerr

```
extern wostream wcerr;
```

The object controls unbuffered insertions to the standard error output as a wide stream. Once the object is constructed, the expression `wcerr.flags() & unitbuf` is nonzero.

wcin

```
extern wistream wcin;
```

The object controls extractions from the standard input as a wide stream. Once the object is constructed, the call `wcin.tie()` returns `&wcout`.

wclog

```
extern wostream wclog;
```

The object controls buffered insertions to the standard error output as a wide stream.

wcout

```
extern wostream wcout;
```

The object controls insertions to the standard output as a wide stream.

See also the **Table of Contents** and the **Index**.

<istream>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    template<class E, class T = char_traits<E> >
        class basic_istream;
    typedef basic_istream<char, char_traits<char> > istream;
    typedef basic_istream<wchar_t, char_traits<wchar_t> > wistream;
    template<class E, class T = char_traits<E> >
        class basic_iostream;
    typedef basic_iostream<char, char_traits<char> > iostream;
    typedef basic_iostream<wchar_t, char_traits<wchar_t> > wiostream;
//      EXTRACTORS
    template<class E, class T>
        basic_istream<E, T>& operator>>(basic_istream<E, T> is, E *s);
    template<class E, class T>
        basic_istream<E, T>& operator>>(basic_istream<E, T> is, E& c);
    template<class T>
        basic_istream<char, T>& operator>>(basic_istream<char, T> is, signed char
*s);
    template<class T>
        basic_istream<char, T>& operator>>(basic_istream<char, T> is, signed char&
c);
    template<class T>
        basic_istream<char, T>& operator>>(basic_istream<char, T> is, unsigned char
*s);
    template<class T>
        basic_istream<char, T>& operator>>(basic_istream<char, T> is, unsigned char&
c);
//      MANIPULATOR
    template class<E, T>
        basic_istream<E, T>& ws(basic_istream<E, T> is);
    };
```

Include the iostreams standard header **<istream>** to define template class basic_istream, which mediates extractions for the iostreams, and the template class. basic_iostream, which mediates both insertions and extractions. The header also defines a related manipulator. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

basic_iostream

```
template <class E, class T = char_traits<E> >
    class basic_iostream : public basic_istream<E, T>,
        public  basic_ostream<E, T> {
public:
    typedef T traits_type;
    explicit basic_iostream(basic_streambuf<E, T> *sb);
    virtual ~basic_iostream();
```

```
};
```

The template class describes an object that controls insertions, through its base object basic_ostream<E, T>, and extractions, through its base object basic_istream<E, T>. The two objects share a common virtual base object basic_ios<E, T>. They also manage a common stream buffer, with elements of type E, whose character traits are determined by the class T. The constructor initializes its base objects via basic_istream(sb) and basic_ostream(sb).

basic_istream

---

basic_istream · char_type · gcount · get · getline · ignore · int_type · ipfx · isfx · off_type · operator>> · peek · pos_type · putback · read · readsome · seekg · sentry · sync · tellg · traits_type · unget

---

```
template <class E, class T = char_traits<E> >
    class basic_istream : virtual public basic_ios<E, T> {
public:
    typedef T traits_type;
    typedef T::char_type char_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    class sentry;
    explicit basic_istream(basic_streambuf<E, T> *sb);
    virtual ~istream();
    bool ipfx(bool noskip = false);
    void isfx();
    basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));
    basic_istream& operator>>(basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
    basic_istream& operator>>(ios_base<E, T>& (*pf)(ios_base<E, T>&));
    basic_istream& operator>>(basic_streambuf<E, T> *sb);
    basic_istream& operator>>(bool& n);
    basic_istream& operator>>(short& n);
    basic_istream& operator>>(unsigned short& n);
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(unsigned int& n);
    basic_istream& operator>>(long& n);
    basic_istream& operator>>(unsigned long& n);
    basic_istream& operator>>(void *& n);
    basic_istream& operator>>(float& n);
    basic_istream& operator>>(double& n);
    basic_istream& operator>>(long double& n);
    streamsize gcount() const;
    int_type get();
    basic_istream& get(E& c);
    basic_istream& get(E *s, streamsize n);
    basic_istream& get(E *s, streamsize n, E delim);
    basic_istream& get(basic_streambuf<E, T> *sb);
```

```
basic_istream& get(baiic_streambuf<E, T> *sb, E delim);
basic_istream& getline(E *s, streamsize n)E
basic_istream& getline(E *s, streamsize n, E delim);
basic_istream& ignore(streamsize n = 1,
    int_type delim = T::eof());
int_type peek();
basic_istream& read(E *s, streamsize n);
streamsize readsome(E *s, streamsize n);
basic_istream& putback(E c);
basic_istream& unget();
basic_istream& tellg();
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off, ios_base::seek_dir way);
int sync();
};
```

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer with elements of type `E`, whose character traits are determined by the class `T`.

Most of the member functions that overload `operator>>` are **formatted input functions**. They follow the pattern:

```
iostate state = goodbit;
const sentry ok(*this);
if (ok)
    {try
        {extract elements and convert
        accumulate flags in state
        store a successful conversion}
    catch (...)
        {if (exceptions() & badbit)
            throw;
        setstate(badbit); }}
setstate(state);
return (*this);
```

Many other member functions are **unformatted input functions**. They follow the pattern:

```
iostate state = goodbit;
count = 0;    // the value returned by gcount
const sentry ok(*this, true);
if (ok)
    {try
        {extract elements and deliver
        count extracted elements in count
        accumulate flags in state}
    catch (...)
        {if (rdstate() & badbit)
            throw;
        setstate(badbit); }}
setstate(state);
```

Both groups of functions call `setstate(eofbit)` if they encounter end-of-file while extracting elements.

An object of class `basic_istream<E, T>` stores:

- a virtual public base object of class **`basic_ios<E, T>`**

- an **extraction count** for the last unformatted input operation (called `count` in the code above

basic_istream::basic_istream

```
explicit basic_istream(basic_streambuf<E, T> *sb);
```

The constructor initializes the base class by calling `init(sb)`. It also stores zero in the extraction count.

basic_istream::char_type

```
typedef T::char_type char_type;
```

The type describes an element of the controlled sequence. Typically, it is the same as the template parameter `E`. In this implementation, however, if `wchar_t` is not a unique type, then **`char_type`** is defined as an encapsulated wchar_t, so that `operator>>:` can be overloaded on `char_type&`.

basic_istream::gcount

```
streamsize gcount() const;
```

The member function returns the extraction count.

basic_istream::get

```
int_type get();
basic_istream& get(E& c);
basic_istream& get(E *s, streamsize n);
basic_istream& get(E *s, streamsize n, E delim);
basic_istream& get(basic_streambuf<E, T> *sb);
basic_istream& get(basic_streambuf<E, T> *sb, E delim);
```

The first of these unformatted input functions extracts an element, if possible, as if by returning `rdbuf()->sbumpc()`. Otherwise, it returns `T::eof()`. If the function extracts no element, it calls `setstate(failbit)`.

The second function extracts the `int_type` element `x` the same way. If `x` compares equal to `T::eof(x)`, the function calls `setstate(failbit)`. Otherwise, it stores `T::to_char_type(x)` in `c`. The function returns `*this`.

The third function returns `get(s, n, widen('\n'))`.

The fourth function extracts up to `n - 1` elements and stores them in the array beginning at `s`. It always stores `E(0)` after any extracted elements it stores. Extraction stops early on end-of-file or on an element that compares equal to `delim` (which is not extracted). If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

The fifth function returns `get(sb, widen('\n'))`.

The sixth function extracts elements and inserts them in `sb`. Extraction stops on end-of-file or on an element that compares equal to `delim` (which is not extracted). It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

basic_istream::getline

```
basic_istream& getline(E *s, streamsize n);
basic_istream& getline(E *s, streamsize n, E delim);
```

The first of these unformatted input functions returns `getline(s, n, widen('\n'))`.

The second function extracts up to `n - 1` elements and stores them in the array beginning at `s`. It always stores `E(0)` after any extracted elements it stores. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence
3. after the function extracts `is.max_size()` elements

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::ignore

```
basic_istream& ignore(streamsize n = 1,
    int_type delim = T::eof());
```

The unformatted input function extracts up to `n` elements and discards them. If `n` equals `numeric_limits<int>::max()`, however, it is taken as arbitrarily large. Extraction stops early on end-of-file or on an element `x` such that `T::to_int_type(x)` compares equal to `delim` (which is also extracted). The function returns `*this`.

basic_istream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_istream::ipfx

```
bool ipfx(bool noskip = false);
```

The member function prepares for formatted or unformatted input. If `good()` is true, the function:

- calls `tie-> flush()` if `tie()` is not a null pointer
- effectively calls `ws(*this)` if `flags() & skipws` is nonzero

If, after any such preparation, `good()` is false, the function calls `setstate(failbit)`. In any case, the function returns `good()`.

You should not call `ipfx` directly. It is called as needed by an object of class `sentry`.

basic_istream::isfx

```
void isfx();
```

The member function has no official duties, but an implementation may depend on a call to `isfx` by a formatted or unformatted input function to tidy up after an extraction. You should not call `isfx` directly. It is called as needed by an object of class `sentry`.

basic_istream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_istream::operator>>

```
basic_istream& operator>>(
    basic_istream& (*pf)(basic_istream&));
basic_istream& operator>>(
    basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_istream& operator>>(
    ios_base<E, T>& (*pf)(ios_base<E, T>&));
basic_istream& operator>>(
    basic_streambuf<E, T> *sb);
basic_istream& operator>>(bool& n);
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

The first member function ensures that an expression of the form `istr >>: ws` calls `ws(istr)`, then returns `*this`. The second and third functions ensure that other manipulators, such as `hex` behave similarly. The remaining functions constitute the formatted input functions.

The function:

```
basic_istream& operator>>(
    basic_streambuf<E, T> *sb);
```

extracts elements, if `sb` is not a null pointer, and inserts them in `sb`. Extraction stops on end-of-file. It also stops, without extracting the element in question, if an insertion fails or throws an exception (which is caught but not rethrown). If the function extracts no elements, it calls `setstate(failbit)`. In any case, the function returns `*this`.

The function:

```
basic_istream& operator>>(bool& n);
```

extracts a field and converts it to a boolean value by calling `use_facet<num_get<E, InIt>(getloc()).get(InIt( rdbuf()), Init(0), *this, getloc(), n)`. Here, `InIt` is defined as `istreambuf_iterator<E, T>`. The function returns `*this`.

The functions:

```
basic_istream& operator>>(short& n);
basic_istream& operator>>(unsigned short& n);
basic_istream& operator>>(int& n);
basic_istream& operator>>(unsigned int& n);
basic_istream& operator>>(long& n);
basic_istream& operator>>(unsigned long& n);
basic_istream& operator>>(void *& n);
```

each extract a field and convert it to a numeric value by calling `use_facet<num_get<E, InIt>(getloc()).get(InIt( rdbuf()), Init(0), *this, getloc(), x)`. Here, `InIt` is defined as `istreambuf_iterator<E, T>`, and `x` has type *long*, *unsigned long*, or *void* * as needed. If the converted value cannot be represented as the type of `n`, the function calls `setstate(failbit)`. In any case, it returns `*this`.

The functions:

```
basic_istream& operator>>(float& n);
basic_istream& operator>>(double& n);
basic_istream& operator>>(long double& n);
```

each extract a field and convert it to a numeric value by calling `use_facet<num_get<E, InIt>(getloc()).get(InIt( rdbuf()), Init(0), *this, getloc(), x)`. Here, `InIt` is defined as `istreambuf_iterator<E, T>`, and `x` has type *double* or *long double* as needed. If the converted value cannot be represented as the type of `n`, the function calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::peek

```
int_type peek();
```

The unformatted input function extracts an element, if possible, as if by returning `rdbuf()->sgetc()`. Otherwise, it returns `T::eof()`.

basic_istream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_istream::putback

```
basic_istream& putback(E c);
```

The unformatted input function puts back `c`, if possible, as if by calling `rdbuf()->sputbackc()`. If `rdbuf()` is a null pointer, or if the call to `sputbackc` returns `T::eof()`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

basic_istream::read

```
basic_istremm& read(E *s, streamsize n);
```

The unformatted input function extracts up to `n` elements and stores them in the array beginning at `s`. Extraction stops early on end-of-file, in which case the function calls `setstate(failbit)`. In any case, it returns `*this`.

basic_istream::readsome

```
readsome readsome(E *s, streamsize n);
```

The member function extracts up to `n` elements and stores them in the array beginning at `s`. If `rdbuf()` is a null pointer, the function calls `setstate(failbit)`. Otherwise, it assigns the value of `rdbuf()-<in_avail()` to `N`. if `N < 0`, the function calls `setstate(eofbit)`. Otherwise, it replaces the value stored in `N` with the smaller of `n` and `N`, then calls `read(s, N)`. In any case, the function returns `gcount()`.

basic_istream::seekg

```
basic_istream& seekg(pos_type pos);
basic_istream& seekg(off_type off, ios_base::seek_dir way);
```

If `fail()` is false, the first member function calls `rdbuf()-> pubseekpos(pos)`. If `fail()` is false, the second function calls `rdbuf()-> pubseekoff(off, way)`. Both functions return `*this`.

basic_istream::sentry

```
class sentry {
public:
    explicit sentry(basic_istream<E, T>& is, bool noskip = false);
    operator bool() const;
    };
```

The nested class describes an object whose declaration structures the formatted input functions and the unformatted input functions. The constructor effectively calls `is.ipfx(noskip)` and stores the return value. `operator bool()` delivers this return value. The destructor effectively calls `is.isfx()`.

basic_istream::sync

```
int sync();
```

If `rdbuf()` is a null pointer, the function returns -1. Otherwise, it calls `rdbuf()->pubsync()`. If that returns -1, the function calls `setstate(badbit)` and returns -1. Otherwise, the function returns zero.

basic_istream::tellg

```
basic_istream& tellg();
```

If `fail()` is false, the member function returns `rdbuf()-> pubseekoff(0, cur, in)`. Otherwise, it returns `streampos(-1)`.

basic_istream::traits_type

```
typedef T traits_type;
```

basic_istream::unget

```
basic_istream& unget();
```

The unformatted input function puts back the previous element in the stream, if possible, as if by calling `rdbuf()->sungetc()`. If `rdbuf()` is a null pointer, or if the call to `sungetc` returns `T::eof()`, the function calls `setstate(badbit)`. In any case, it returns `*this`.

iostream

```
typedef basic_iostream<char, char_traits<char> > iostream;
```

The type is a synonym for template class `basic_iostream`, specialized for elements of type *char* with default character traits.

istream

```
typedef basic_istream<char, char_traits<char> > istream;
```

The type is a synonym for template class `basic_istream`, specialized for elements of type *char* with default character traits.

operator>>

```
template<class E, class T>
    basic_istream<E, T>& operator>>(basic_istream<E, T> is, E *s);
template<class E, class T>
    basic_istream<E, T>& operator>>(basic_istream<E, T> is, E& c);
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, signed char *s);
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, signed char& c);
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, unsigned char *s);
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, unsigned char& c);
```

The template function:

```
template<class E, class T>
    basic_istream<E, T>& operator>>(basic_istream<E, T>& is, E *s);
```

extracts up to `n - 1` elements and stores them in the array beginning at `s`. If `is.width()` is greater than zero, `n` is `is.width()`; otherwise it is the largest array of `E` that can be declared. The function always stores `E(0)` after any extracted elements it stores. Extraction stops early on end-of-file or on any element (which is not extracted) that would be discarded by `ws`. If the function extracts no elements, it calls `is.setstate(failbit)`. In any case, it calls `is.width(0)` and returns `is`.

The template function:

```
template<class E, class T>
    basic_istream<E, T>& operator>>(basic_istream<E, T>& is, char& c);
```

extracts an element, if possible, and stores it in `c`. Otherwise, it calls `is.setstate(failbit)`. In any case, it returns `is`.

The template function:

```
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, signed char *s);
```

returns `is >> (char *)s`.

The template function:

```
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, signed char& c);
```

returns `is >> (char&)c`.

The template function:

```
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, unsigned char *s);
```

returns `is >> (char *)s`.

The template function:

```
template<class T>
    basic_istream<char, T>& operator>>(basic_istream<char, T> is, unsigned char& c);
```

returns `is >> (char&)c`.

wiostream

```
typedef basic_iostream<wchar_t, char_traits<wchar_t> > wiostream;
```

The type is a synonym for template class basic_iostream, specialized for elements of type wchar_t with default character traits.

wistream

```
typedef basic_istream<wchar_t, char_traits<wchar_t> > wistream;
```

The type is a synonym for template class basic_istream, specialized for elements of type wchar_t with default character traits.

ws

```
template class<E, T>
    basic_istream<E, T>& ws(basic_istream<E, T> is);
```

The manipulator extracts and discards any elements x for which use_facet< ctype<E> >(getloc()). is(ctype<E>::space, x) is true. It calls setstate(eofbit) if it encounters end-of-file while extracting elements. The function returns is.

See also the **Table of Contents** and the **Index**.

<iterator>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

advance · back_insert_iterator · back_inserter · bidirectional_iterator_tag · distance · forward_iterator_tag · front_insert_iterator · front_inserter · input_iterator_tag · insert_iterator · inserter · istream_iterator · istreambuf_iterator · iterator · iterator_traits · operator!= · operator== · operator< · operator<= · operator> · operator>= · operator+ · operator- · ostream_iterator · ostreambuf_iterator · output_iterator_tag · random_access_iterator_tag · reverse_bidirectional_iterator · reverse_iterator

---

```
namespace std {
struct input_iterator_tag;
struct output_iterator_tag;
struct forward_iterator_tag;
struct bidirectional_iterator_tag;
struct random_access_iterator_tag;
//    TEMPLATE CLASSES
template<class C, class T, class Dist>
    struct iterator;
template<class It>
    struct iterator_traits;
template<class T>
    struct iterator_traits<T *>
template<class BidIt, class T, class Ref,
    class Ptr, class Dist>
    class reverse_bidirectional_iterator;
template<class RanIt, class T, class Ref,
    class Ptr, class Dist>
    class reverse_iterator;
template<class Cont>
    class back_insert_iterator;
template<class Cont>
    class front_insert_iterator;
template<class Cont>
    class insert_iterator;
template<class T, class Dist>
    class istream_iterator;
template<class T>
    class ostream_iterator;
template<class E, class T>
    class istreambuf_iterator;
template<class E, class T>
    class ostreambuf_iterator;
```

```
//    TEMPLATE FUNCTIONS
template<class BidIt, class T, class Ref, class Ptr, class Dist>
    bool operator==(
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& lhs,
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator==(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class T, class Dist>
    bool operator==(
        const istream_iterator<T, Dist>& lhs,
        const istream_iterator<T, Dist>& rhs);
template<class E, class T>
    bool operator==(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class BidIt, class T, class Ref, class Ptr, class Dist>
    bool operator!=(
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& lhs,
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator!=(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class T, class Dist>
    bool operator!=(
        const istream_iterator<T, Dist>& lhs,
        const istream_iterator<T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator<(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>&mmp; lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator>(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>&mmp; lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator<=(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>&mmp; lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
```

```
    bool operator>=(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>&mmp; lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    Dist operator-(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    reverse_iterator<RanIt, T, Ref, Ptr, Dist> operator+(
        Dist n,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
template<class Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
template<class Init, class Dist>
    iterator_traits<InIt>::distance_type
        distance(InIt first, InIt last);
    };
```

Include the STL standard header **<iterator>** to define a number of classes, template classes, and template functions that aid in the declaration and manipulation of iterators.

advance

```
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
```

The template function effectively advances `it` by incrementing it `n` times. If `InIt` is a random-access iterator type, the function evaluates the expression `it += n`. Otherwise, it performs each increment by evaluating `++it`. If `InIt` is an input or forward iterator type, `n` must not be negative.

back_insert_iterator

```
template<class Cont>
    class back_insert_iterator
        : public iterator<output_iterator_tag, void, void> {
public:
    typedef Cont container_type;
    typedef Cont::value_type value_type;
    explicit back_insert_iterator(Cont& x);
    back_insert_iterator& operator=(const Cont::value_type& val);
    back_insert_iterator& operator*();
    back_insert_iterator& operator++();
    back_insert_iterator operator++(int);
```

```
protected:
    Cont& container;
    };
```

The template class describes an output iterator object. It inserts elements into a container of type **Cont**, which it accesses via the protected reference object it stores called **container**. The container must define:

- the member type **value_type**, which is the type of an element of the sequence controlled by the container
- the member function **push_back**(value_type c), which appends a new element with value c to the end of the sequence

back_insert_iterator::back_insert_iterator

```
explicit back_insert_iterator(Cont& x);
```

The constructor initializes container with x.

back_insert_iterator::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter Cont.

back_insert_iterator::operator*

```
back_insert_iterator& operator*();
```

The member function returns *this.

back_insert_iterator::operator++

```
back_insert_iterator& operator++();
back_insert_iterator operator++(int);
```

The member functions both return *this.

back_insert_iterator::operator=

```
back_insert_iterator& operator=(const Cont::value_type& val);
```

The member function evaluates container. push_back(val), then returns *this.

back_insert_iterator::value_type

```
typedef Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

back_inserter

```
template<class Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);
```

The template member function returns back_insert_iterator<Cont>(x).

bidirectional_iterator_tag

```
struct bidirectional_iterator_tag
    : public forward_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a bidirectional iterator.

distance

```
template<class Init, class Dist>
    iterator_traits<InIt>::distance_type distance(InIt first, InIt last);
```

The template function sets a count `n` to zero. It then effectively advances `first` and increments `n` until `first == last`. If `InIt` is a random-access iterator type, the function evaluates the expression `n += last - first`. Otherwise, it performs each iterator increment by evaluating `++first`.

In this implementation, if a translator does not support partial specialization of templates, the return type is `ptrdiff_t`. If you are not certain this type is adequate, use the template function:

```
template<class InIt, class Dist>
    void _Distance(InIt first, InIt last, Dist& n0);
```

which adds `n` to the value stored in `n0`.

forward_iterator_tag

```
struct forward_iterator_tag
    : public input_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a forward iterator.

front_insert_iterator

```
template<class Cont>
    class front_insert_iterator
        : public iterator<output_iterator_tag, void, void> {
public:
    typedef Cont container_type;
    typedef Cont::value_type value_type;
    explicit front_insert_iterator(Cont& x);
    front_insert_iterator& operator=(const Cont::value_type& val);
    front_insert_iterator& operator*();
    front_insert_iterator& operator++();
    front_insert_iterator operator++(int);
protected:
    Cont& container;
```

```
};
```

The template class describes an output iterator object. It inserts elements into a container of type **`Cont`**, which it accesses via the protected reference object it stores called **`container`**. The container must define:

- the member type **`value_type`**, which is the type of an element of the sequence controlled by the container
- the member function **`push_front`**`(value_type c)`, which prepends a new element with value `c` to the beginning of the sequence

front_insert_iterator::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

front_insert_iterator::front_insert_iterator

```
explicit front_insert_iterator(Cont& x);
```

The constructor initializes `container` with `x`.

front_insert_iterator::operator*

```
front_insert_iterator& operator*();
```

The member function returns `*this`.

front_insert_iterator::operator++

```
front_insert_iterator& operator++();
front_insert_iterator operator++(int);
```

The member functions both return `*this`.

front_insert_iterator::operator=

```
front_insert_iterator& operator=(const Cont::value_type& val);
```

The member function evaluates `container. push_front(val)`, then returns `*this`.

front_insert_iterator::value_type

```
typedef Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

front_inserter

```
template<class Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);
```

The template member function returns `front_insert_iterator`<Cont>(x).

input_iterator_tag

```
struct input_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as an input iterator.

insert_iterator

```
template<class Cont>
    class insert_iterator
        : public iterator<output_iterator_tag, void, void> {
public:
    typedef Cont container_type;
    typedef Cont::value_type value_type;
    explicit insert_iterator(Cont& x, Cont::iterator it);
    insert_iterator& operator=(const Cont::value_type& val);
    insert_iterator& operator*();
    insert_iterator& operator++();
    insert_iterator& operator++(int);
protected:
    Cont& container;
    Cont::iterator iter;
    };
```

The template class describes an output iterator object. It inserts elements into a container of type **`Cont`**, which it accesses via the protected reference object it stores called **`container`**. It also stores the protected iterator object, of class `Cont::iterator`, called **`iter`**. The container must define:

- the member type **`iterator`**, which is the type of an iterator for the container
- the member type **`value_type`**, which is the type of an element of the sequence controlled by the container
- the member function **`insert`**`(iterator it, value_type c)`, which inserts a new element with value `c` immediately before the element designated by `it` in the controlled sequence, then returns an iterator that designates the inserted element

insert_iterator::container_type

```
typedef Cont container_type;
```

The type is a synonym for the template parameter `Cont`.

insert_iterator::insert_iterator

```
explicit insert_iterator(Cont& x, Cont::iterator it);
```

The constructor initializes `container` with `x`, and `iter` with `it`.

insert_iterator::operator*

```
insert_iterator& operator*();
```

The member function returns `*this`.

insert_iterator::operator++

```
insert_iterator& operator++();
insert_iterator& operator++(int);
```

The member functions both return `*this`.

insert_iterator::operator=

```
insert_iterator& operator=(const Cont::value_type& val);
```

The member function evaluates `iter = container. insert(iter, val)`, then returns `*this`.

insert_iterator::value_type

```
typedef Cont::value_type value_type;
```

The type describes the elements of the sequence controlled by the associated container.

inserter

```
template<class Cont, class Iter>
    insert_iterator<Cont> inserter(Cont& x, Iter it);
```

The template member function returns `insert_iterator<Cont>(x, it)`.

istream_iterator

```
template<class U, class E = char, class T = char_traits<E> >
    class istream_iterator
        : public iterator<input_iterator_tag, U, ptrdiff_t> {
public:
    typedef U value_type;
    typedef E char_type;
    typedef T traits_type;
    typedef basic_istream<E, T> istream_type;
    istream_iterator();
    istream_iterator(istream_type& is);
    const U& operator*() const;
    const U *operator->() const;
    istream_iterator<U, E, T>& operator++();
    istream_iterator<U, E, T> operator++(int);
    };
```

The template class describes an input iterator object. It extracts objects of class **U** from an **input stream**, which it

accesses via an object it stores, of type pointer to `basic_istream<E, T>`. After constructing or incrementing an object of class `istream_iterator` with a non-null stored pointer, the object attempts to extract and store an object of type `U` from the associated input stream. If the extraction fails, the object effectively replaces the stored pointer with a null pointer (thus making an end-of-sequence indicator).

istream_iterator::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

istream_iterator::istream_iterator

```
istream_iterator();
istream_iterator(istream_type& is);
```

The first constructor initializes the input stream pointer with a null pointer. The second constructor initializes the input stream pointer with `&is`, then attempts to extract and store an object of type `U`.

istream_iterator::istream_type

```
typedef basic_istream<E, T> istream_type;
```

The type is a synonym for `basic_istream<E, T>`.

istream_iterator::operator*

```
const U& operator*() const;
```

The operator returns the stored object of type `U`.

istream_iterator::operator->

```
const U *operator->() const;
```

The operator returns `&**this`. In this implementation, if a translator always requires a return value that designates an object with members, this operator is not available.

istream_iterator::operator++

```
istream_iterator<U, E, T>& operator++();
istream_iterator<U, E, T> operator++(int);
```

The first operator attempts to extract and store an object of type `U` from the associated input stream. The second operator makes a copy of the object, increments the object, then returns the copy.

istream_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

istream_iterator::value_type

```
typedef U value_type;
```

The type is a synonym for the template parameter U.

istreambuf_iterator

```
template<class E, class T = char_traits<E> >
    class istreambuf_iterator
        : public iterator<input_iterator_tag, T, Dist> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef basic_streambuf<E, T> streambuf_type;

    typedef basic_istream<E, T> istream_type;
    istreambuf_iterator(streambuf_type *sb = 0) throw();
    istreambuf_iterator(istream_type& is) throw();
    const E& operator*() const;
    const E *operator->();
    istreambuf_iterator& operator++();
    istreambuf_iterator operator++(int);
    bool equal(const istreambuf_iterator& rhs);
    };
```

The template class describes an input iterator object. It extracts elements of class **E** from an **input stream buffer**, which it accesses via an object it stores, of type pointer to `basic_streambuf<E, T>`. After constructing or incrementing an object of class `istreambuf_iterator` with a non-null stored pointer, the object effectively attempts to extract and store an object of type `E` from the associated itput stream. (The extraction may be delayed, however, until the object is actually dereferenced or copied.) If the extraction fails, the object effectively replaces the stored pointer with a null pointer (thus making an end-of-sequence indicator).

istreambuf_iterator::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

istreambuf_iterator::equal

```
bool equal(const istreambuf_iterator& rhs);
```

The member function returns true only if the stored streambbuffer pointers for the object and `rhs` are both null pointers or are both non-null pointers.

istreambuf_iterator::int_type

```
typedef T:int_type int_type;
```

The type is a synonym for `T::int_type">int_type`.

istreambuf_iterator::istream_type

```
typedef basic_istream<E, T> istream_type;
```

The type is a synonym for `basic_istream<E, T>`.

istreambuf_iterator::istreambuf_iterator

```
istreambuf_iterator(streambuf_type *sb = 0) throw();
istreambuf_iterator(istream_type& is) throw();
```

The first constructor initializes the input stream-buffer pointer with `sb`. The second constructor initializes the input stream-buffer pointer with `is.rdbuf()`, then (eventually) attempts to extract and store an object of type `E`.

istreambuf_iterator::operator*

```
const E& operator*() const;
```

The operator returns the stored object of type `E`.

istreambuf_iterator::operator++

```
istreambuf_iterator& operator++();
istreambuf_iterator operator++(int);
```

The first operator (eventually) attempts to extract and store an object of type `E` from the associated input stream. The second operator makes a copy of the object, increments the object, then returns the copy.

istreambuf_iterator::operator->

```
const E *operator->() const;
```

The operator returns `&**this`. In this implementation, if a translator always requires a return value that designates an object with members, this operator is not available.

istreambuf_iterator::streambuf_type

```
typedef basic_streambuf<E, T> streambuf_type;
```

The type is a synonym for `basic_streambuf<E, T>`.

istreambuf_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

iterator

```
template<class C, class T, class Dist = ptrdiff_t>
    struct iterator {
    typedef C iterator_category;
    typedef T value_type;
    typedef Dist distance_type;
    };
```

The template class serves as a base type for all iterators. It defines the member types `iterator_category` (a synonym for the template parameter `C`), `value_type` (a synonym for the template parameter `T`), and `distance_type` (a synonym for the template parameter `Dist`).

iterator_traits

```
template<class It>
    struct iterator_traits {
    typedef It::iterator_category iterator_category;
    typedef It::value_type value_type;
    typedef It::distance_type distance_type;
    };
template<class T>
    struct iterator_traits<T *> {
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t distance_type;
    };
```

The template class determines several critical types associated with the iterator type `It`. It defines the member types `iterator_category` (a synonym for `It::iterator_category`), `value_type` (a synonym for `It::value_type`), and `distance_type` (a synonym for `It::distance_type`).

The partial specialization determines the critical types associated with an object pointer type `T *`. In this implementation, if a translator does not support partial specialization of templates, you should use the template functions:

```
template<class C, class T, class Dist>
    C _Iter_cat(const iterator<C, T, Dist>&);
template<class T>
    random_access_iterator_tag _Iter_cat(const T *);

template<class C, class T, class Dist>
    T *_Val_type(const iterator<C, T, Dist>&);
template<class T>
    T *_Val_type(const T *);

template<class C, class T, class Dist>
    Dist *_Dist_type(const iterator<C, T, Dist>&);
template<class T>
    ptrdiff_t *_Dist_type(const T *);
```

which determine the same types a bit more indirectly. You use these functions as arguments on a function call. Their sole purpose is to supply a useful template class parameter to the called function.

operator!=

```
template<class BidIt, class T, class Ref, class Ptr, class Dist>
    bool operator!=(
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& lhs,
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator!=(
        const reverse_iterator<BidIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<BidIt, T, Ref, Ptr, Dist>& rhs);
template<class T, class Dist>
    bool operator!=(
        const istream_iterator<T, Dist>& lhs,
        const istream_iterator<T, Dist>& rhs);
template<class E, class T>
    bool operator!=(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
```

The template operator returns `!(lhs == rhs)`.

operator==

```
template<class BidIt, class T, class Ref, class Ptr, class Dist>
    bool operator==(
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& lhs,
        const reverse_bidirectional_iterator<BidIt, T, Ref,
            Ptr, Dist>& rhs);
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator==(
        const reverse_iterator<BidIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<BidIt, T, Ref, Ptr, Dist>& rhs);
template<class T, class Dist>
    bool operator==(
        const istream_iterator<T, Dist>& lhs,
        const istream_iterator<T, Dist>& rhs);
template<class E, class T>
    bool operator==(
        const istreambuf_iterator<E, T>& lhs,
        const istreambuf_iterator<E, T>& rhs);
```

The first two template operators each return true only if `lhs.current == rhs.current`. The third template operator returns true only if both `lhs` and `rhs` store the same stream pointer. The fourth template operator returns `lhs.equal(rhs)`.

operator<

```
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator<(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
```

The template operator returns `rhs.current < lhs.current` [sic].

operator<=

```
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator<=(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
```

The template operator returns `!(rhs < lhs)`.

operator>

```
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator>(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
```

The template operator returns `rhs < lhs`.

operator>=

```
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    bool operator>=(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
```

The template operator returns `!(lhs < rhs)`.

operator+

```
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    reverse_iterator<RanIt, T, Ref, Ptr, Dist> operator+(
        Dist n,
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
```

The template operator returns `rhs + n`.

operator-

```
template<class RanIt, class T, class Ref, class Ptr, class Dist>
    Dist operator-(
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& lhs,
```

```
        const reverse_iterator<RanIt, T, Ref, Ptr, Dist>& rhs);
```

The template operator returns `rhs.current - lhs.current` [sic].

ostream_iterator

```
template<class T>
    class ostream_iterator
        : public iterator<output_iterator_tag, void, void> {
public:
    typedef U value_type;
    typedef E char_type;
    typedef T traits_type;
    typedef basic_ostream<E, T> ostream_type;
    ostream_iterator(ostream_type& os);
    ostream_iterator(ostream_type& os, const E *delim);
    ostream_iterator<U, E, T>& operator=(const U& val);
    ostream_iterator<U, E, T>& operator*();
    ostream_iterator<U, E, T>& operator++();
    ostream_iterator<U, E, T> operator++(int);
    };
```

The template class describes an output iterator object. It inserts objects of class **U** into an **output stream**, which it accesses via an object it stores, of type pointer to `basic_ostream<E, T>`. It also stores a pointer to a **delimiter string**, a null-terminated string of elements of type `E`, which is appended after each insertion. (Note that the string itself is *not* copied by the constructor.

ostream_iterator::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

ostream_iterator::operator*

```
ostream_iterator<U, E, T>& operator*();
```

The operator returns `*this`.

ostream_iterator::operator++

```
ostream_iterator<U, E, T>& operator++();
ostream_iterator<U, E, T> operator++(int);
```

The operators both return `*this`.

ostream_iterator::operator=

```
ostream_iterator<U, E, T>& operator=(const U& val);
```

The operator inserts `val` into the output stream associated with the object, then returns `*this`.

ostream_iterator::ostream_iterator

```
ostream_iterator(ostream_type& os);
ostream_iterator(ostream_type& os, const E *delim);
```

The first constructor initializes the output stream pointer with `&os`. The delimiter string pointer designates an empty string. The second constructor initializes the output stream pointer with `&os` and the delimiter string pointer with `delim`.

ostream_iterator::ostream_type

```
typedef basic_ostream<E, T> ostream_type;
```

The type is a synonym for `basic_ostream<E, T>`.

ostream_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

ostream_iterator::value_type

```
typedef U value_type;
```

The type is a synonym for the template parameter `U`.

ostreambuf_iterator

```
template<class E, class T = char_traits<E> >
    class ostreambuf_iterator
        : public iterator<output_iterator_tag, void, void> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef basic_streambuf<E, T> streambuf_type;
    typedef basic_ostream<E, T> ostream_type;
    ostreambuf_iterator(streambuf_type *sb) throw();
    ostreambuf_iterator(ostream_type& os) throw();
    ostreambuf_iterator& operator=(E x);
    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    T1 operator++(int);
    bool failed() const throw();
    };
```

The template class describes an output iterator object. It inserts elements of class **E** into an **output stream buffer**, which it accesses via an object it stores, of type pointer to `basic_streambuf<E, T>`.

ostreambuf_iterator::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

ostreambuf_iterator::failed

```
bool failed() const throw();
```

The member function returns true only if no insertion into the output stream buffer has earlier failed.

ostreambuf_iterator::operator*

```
ostreambuf_iterator& operator*();
```

The operator returns *this.

ostreambuf_iterator::operator++

```
ostreambuf_iterator& operator++();
T1 operator++(int);
```

The first operator returns *this. The second operator returns an object of some type T1 that can be converted to ostreambuf_iterator<E, T>.

ostreambuf_iterator::operator=

```
ostreambuf_iterator& operator=(E x);
```

The operator inserts x into the associated stream buffer, then returns *this.

ostreambuf_iterator::ostream_type

```
typedef basic_ostream<E, T> ostream_type;
```

The type is a synonym for basic_ostream<E, T>.

ostreambuf_iterator::ostreambuf_iterator

```
ostreambuf_iterator(streambuf_type *sb) throw();
ostreambuf_iterator(ostream_type& is) throw();
```

The first conttructor initializes the output stream-buffer pointer with sb. The second constructor initializes the output stream-buffer pointer with is.rdbuf(). (The stored pointer must not be a null pointer.)

ostreambuf_iterator::streambuf_type

```
typedef basic_streambuf<E, T> streambuf_type;
```

The type is a synonym for basic_streambuf<E, T>.

ostreambuf_iterator::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

output_iterator_tag

```
struct output_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a output iterator.

random_access_iterator_tag

```
struct random_access_iterator_tag
    : public bidirectional_iterator_tag {
    };
```

The type is the same as `iterator<It>::iterator_category` when `It` describes an object that can serve as a random-access iterator.

reverse_bidirectional_iterator

```
template<class BidIt,
    class T = iterator_traits<BidIt>::value_type,
    class Ref = T&,
    class Ptr = T *, class Dist = ptrdiff_t>
    class reverse_bidirectional_iterator
        : public iterator<bidirectional_iterator_tag, T, Dist> {
public:
    typedef BidIt iter_type;
    typedef T value_type;
    typedef Ref reference_type;
    typedef Ptr pointer_type;
    typedef Dist distance_type;
    reverse_bidirectional_iterator();
    explicit reverse_bidirectional_iterator(BidIt x);
    BidIt base() const;
    Ref operator*() const;
    Ptr operator->() const;
    reverse_bidirectional_iterator& operator++();
    reverse_bidirectional_iterator operator++(int);
    reverse_bidirectional_iterator& operator--();
    reverse_bidirectional_iterator operator--();
protected:
    BidIt current;
    };
```

The template class describes an object that behaves like a bidirectional iterator of class iterator<bidirectional_iterator_tag, **T**, **Dist**>. It stores a bidirectional iterator of type **BidIt** in the protected object **current**. Incrementing the object x of type reverse_bidirectional_iterator decrements x.current, and decrementing x increments x.current. Moreover, the expression *x evaluates to *--(tmp = current) (where tmp is a temporary object of class BidIt), of type **Ref**. Typically, Ref is type T&.

Thus, you can use an object of class reverse_bidirectional_iterator to access in reverse order a sequence that is traversed in order by a bidirectional iterator.

reverse_bidirectional_iterator::base

```
BidIt base() const;
```

The member function returns current.

reverse_bidirectional_iterator::distance_type

```
typedef Dist distance_type;
```

The type is a synonym for the template parameter Ref.

reverse_bidirectional_iterator::iter_type

```
typedef BidIt iter_type;
```

The type is a synonym for the template parameter BidIt.

reverse_bidirectional_iterator::operator*

```
Ref operator*() const;
```

The operator assigns current to a temporary object tmp of class BidIt, then returns *--tmp.

reverse_bidirectional_iterator::operator++

```
reverse_bidirectional_iterator& operator++();
reverse_bidirectional_iterator operator++(int);
```

The first (preincrement) operator evaluates --current. then returns *this.

The second (postincrement) operator makes a copy of *this, evaluates --current, then returns the copy.

reverse_bidirectional_iterator::operator--

```
reverse_bidirectional_iterator& operator--();
reverse_bidirectional_iterator operator--();
```

The first (predecrement) operator evaluates ++current. then returns *this.

The second (postdecrement) operator makes a copy of *this, evaluates ++current, then returns the copy.

reverse_bidirectional_iterator::operator->

```
Ptr operator->() const;
```

The operator returns `&**this`. In this implementation, if a translator always requires a return value that designates an object with members, this operator is not available.

reverse_bidirectional_iterator::pointer_type

```
typedef Ptr pointer_type;
```

The type is a synonym for the template parameter `Ref`.

reverse_bidirectional_iterator::reference_type

```
typedef Ref reference_type;
```

The type is a synonym for the template parameter `Ref`.

reverse_bidirectional_iterator::reverse_bidirectional_iterator

```
reverse_bidirectional_iterator();
explicit reverse_bidirectional_iterator(BidIt x);
```

The first constructor initializes `current` with its default constructor. The second constructor initializes `current` with `current(x)`.

reverse_bidirectional_iterator::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

reverse_iterator

```
template<class RanIt,
    class T = iterator_traits<RanIt>::value_type,
    class Ref = T&,
    class Ptr = T *, class Dist = ptrdiff_t>
    class reverse_iterator
        : public iterator<random_access_iterator_tag, T, Dist> {
public:
    typedef BidIt iter_type;
    typedef T value_type;
    typedef Ref reference_type;
    typedef Ptr pointer_type;
    typedef Dist distance_type;
    reverse_iterator();
    explicit reverse_iterator(RanIt x);
    RanIt base() const;
    Ref operator*() const;
```

```
    Ptr operator->() const;
    reverse_iterator& operator++();
    reverse_iterator operator++(int);
    reverse_iterator& operator--();
    reverse_iterator operator--();
    reverse_iterator& operator+=(Dist n);
    reverse_iterator operator+(Dist n) const;
    reverse_iterator& operator-=(Dist n);
    reverse_iterator operator-(Dist n) const;
    Ref operator[](Dist n) const;
protected:
    RanIt current;
    };
```

The template class describes an object that behaves like a random-access iterator of class `iterator<random_access_iterator_tag, T, Dist>`. It stores a random-access iterator of type **`RanIt`** in the protected object **`current`**. Incrementing the object `x` of type `reverse_iterator` decrements `x.current`, and decrementing `x` increments `x.current`. Moreover, the expression `*x` evaluates to `*(current - 1)`, of type **`Ref`**. Typically, `Ref` is type `T&`.

Thus, you can use an object of class `reverse_iterator` to access in reverse order a sequence that is traversed in order by a random-access iterator.

reverse_iterator::base

```
RanIt base() const;
```

The member function returns `current`.

reverse_iterator::distance_type

```
typedef Dist distance_type;
```

The type is a synonym for the template parameter `Ref`.

reverse_iterator::iter_type

```
typedef BidIt iter_type;
```

The type is a synonym for the template parameter `BidIt`.

reverse_iterator::operator*

```
Ref operator*() const;
```

The operator returns `*(current - 1)`.

reverse_iterator::operator+

```
reverse_iterator operator+(Dist n) const;
```

The operator returns `reverse_iterator(*this) += n`.

reverse_iterator::operator++

```
reverse_iterator& operator++();
reverse_iterator operator++(int);
```

The first (preincrement) operator evaluates `--current`. then returns `*this`.

The second (postincrement) operator makes a copy of `*this`, evaluates `--current`, then returns the copy.

reverse_iterator::operator+=

```
reverse_iterator& operator+=(Dist n);
```

The operator evaluates `current - n`. then returns `*this`.

reverse_iterator::operator-

```
reverse_iterator operator-(Dist n) const;
```

The operator returns `reverse_iterator(*this) -= n`.

reverse_iterator::operator--

```
reverse_iterator& operator--();
reverse_iterator operator--();
```

The first (predecrement) operator evaluates `++current`. then returns `*this`.

The second (postdecrement) operator makes a copy of `*this`, evaluates `++current`, then returns the copy.

reverse_iterator::operator-=

```
reverse_iterator& operator-=(Dist n);
```

The operator evaluates `current + n`. then returns `*this`.

reverse_iterator::operator->

```
Ptr operator->() const;
```

The operator returns `&**this`. In this implementation, if a translator always requires a return value that designates an object with members, this operator is not available.

reverse_iterator::operator[]

```
Ref operator[](Dist n) const;
```

The operator returns `*(*this + n)`.

reverse_iterator::pointer_type

```
typedef Ptr pointer_type;
```

The type is a synonym for the template parameter `Ref`.

reverse_iterator::reference_type

```
typedef Ref reference_type;
```

The type is a synonym for the template parameter `Ref`.

reverse_iterator::reverse_iterator

```
reverse_iterator();
explicit reverse_iterator(RanIt x);
```

The first constructor initializes `current` with its default constructor. The second constructor initializes `current` with `current(x)`.

reverse_iterator::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

See also the **Table of Contents** and the **Index**.

<limits>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    enum float_round_style;
    template<class T>
        class numeric_limits;
    };
```

Include the standard header **<limits>** to define the template class `numeric_limits`. Explicit specializations of this class describe many arithmetic properties of the scalar types (other than pointers).

float_round_style

```
enum float_round_style {
    round_indeterminate = -1,
    round_toward_zero = 0,
    round_to_nearest = 1,
    round_toward_infinity = 2,
    round_toward_neg_infinity = 3
    };
```

The enumeration describes the various methods that an implementation can choose for rounding a floating-point value to an integer value:

- **round_indeterminate** -- rounding method cannot be determined
- **round_toward_zero** -- round toward zero
- **round_to_nearest** -- round to nearest integer
- **round_toward_infinity** -- round away from zero
- **round_toward_neg_infinity** -- round to more negative integer

numeric_limits

```
template<class T>
    class numeric_limits {
public:
    static const bool has_denorm = false;
    static const bool has_denorm_loss = false;
    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
```

```
    static const bool has_signaling_NaN = false;
    static const bool is_bounded = false;
    static const bool is_exact = false;
    static const bool is_iec559 = false;
    static const bool is_integer = false;
    static const bool is_modulo = false;
    static const bool is_signed = false;
    static const bool is_specialized = false;
    static const bool tinyness_before = false;
    static const bool traps = false;
    static const float_round_style round_style = round_toward_zero;
    static const int digits = 0;
    static const int digits10 = 0;
    static const int max_exponent = 0;
    static const int max_exponent10 = 0;
    static const int min_exponent = 0;
    static const int min_exponent10 = 0;
    static const int radix = 0;
    static T denorm_min() throw();
    static T epsilon() throw();
    static T infinity() throw();
    static T max() throw();
    static T min() throw();
    static T quiet_NaN() throw();
    static T round_error() throw();
    static T signaling_NaN() throw();
    };
```

The template class describes many arithmetic properties of its parameter type `T`. The header defines explicit specializations for the types `wchar_t`, *bool*, *char*, *signed char*, *unsigned char*, *short*, *unsigned short*, *int*, *unsigned int*, *long*, *unsigned long*, *float*, *double*, and *long double*. For all these explicit specializations, the member `is_specialized` is true, and all relevant members have meaningful values. The program can supply additional explicit specializations.

For an arbitrary specialization, *no* members have meaningful values. A member object that does not have a meaningful value stores zero (or false) and a member function that does not return a meaningful value returns `T(0)`.

numeric_limits::denorm_min

```
static T denorm_min() throw();
```

The function returns the minimum value for the type (which is the same as `min`() if `has_denorm` is

false).

numeric_limits::digits

```
static const int digits = 0;
```

The member stores the number of radix digits that the type can represent without change (which is the number of bits other than any sign bit for a predefined integer type, or the number of mantissa digits for a predefined floating-point type).

numeric_limits::digits10

```
static const int digits10 = 0;
```

The member stores the number of decimal digits that the type can represent without change.

numeric_limits::epsilon

```
static T epsilon() throw();
```

The function returns the difference between 1 and the smallest value greater than 1 that is representable for the type (which is the value FLT_EPSILON for type *float*).

numeric_limits::has_denorm

```
static const bool has_denorm = false;
```

The member stores true for a floating-point type that has denormalized values (effectively a variable number of exponent bits).

numeric_limits::has_denorm_loss

```
static const bool has_denorm_loss = false;
```

The member stores true for a type that determines whether a value has lost accuracy because it is delivered as a denormalized result (too small to represent as a normalized value) or because it is inexact (not the same as a result not subject to limitations of exponent range and precision), an option with IEC 559 floating-point representations that can affect some results.

numeric_limits::has_infinity

```
static const bool has_infinity = false;
```

The member stores true for a type that has a representation for positive infinity. True if is_iec559 is true.

numeric_limits::has_quiet_NaN

```
static const bool has_quiet_NaN = false;
```

The member stores true for a type that has a representation for a **quiet NaN**, an encoding that is ``Not a Number'' which does not signal its presence in an expression. True if `is_iec559` is true.

numeric_limits::has_signaling_NaN

```
static const bool has_signaling_NaN = false;
```

The member stores tree for a type that has a representation for a **signaling NaN**, an encoding that is ``Not a Number'' which signals itd presence in an expression by reporting an exception. True if `is_iec559` is true.

numeric_limits::infinity

```
static T infinity() throw();
```

The function returns the representation of positive infinity for the type. The return value is meaningful only if `has_infinity` is true.

numeric_limits::is_bounded

```
static const bool is_bounded = false;
```

The member stores true for a type that has a bounded set of represtntable values (which is the case for all predefined types).

numeric_limits::is_exact

```
static const bool is_exact = false;
```

The member stores true for a type that has exact representations for all its values (which is the case for all predefined integer types). A fixed-point or rational representation is also considered exact, but not a floating-point representation.

numeric_limits::is_iec559

```
static const bool is_iec559 = false;
```

The member stores true for a type that has a representation conforming to **IEC 559**, an international standard for representing floating-point values (also known as **IEEE 754** in the USA).

numeric_limits::is_integer

```
static const bool is_integer = false;
```

The member stores true for a type that has an integer representation (which is the case for all predefined integer types).

numeric_limits::is_modulo

```
static const bool is_modulo = false;
```

The member stores true for a type that has a **modulo representation**, where all results are reduced modulo some value (which is the case for all predefined unsigned integer types).

numeric_limits::is_signed

```
static const bool is_signed = false;
```

The member stores true for a type that has a signed representation (which is the case for all predefined floating-point and signed integer types.)

numeric_limits::is_specialized

```
static const bool is_specialized = false;
```

The member stores true for a type that has an explicit specialization defined for template class `numeric_limits` (which is the case for all scalar types other than pointers).

numeric_limits::max

```
static T max() throw();
```

The function returns the maximum finite value for the type (which is `INT_MAX` for type *int* and `FLT_MAX` for type *float*). The return value is meaningful if `is_bounded` is true.

numeric_limits::max_exponent

```
static const int max_exponent = 0;
```

The member stores the maximum positive integer such that the type can represent as a finite value `radix` raised to that power (which is the value `FLT_MAX_EXP` for type *float*). Meaningful only for floating-point types.

numeric_limits::max_exponent10

```
static const int max_exponent10 = 0;
```

The member stores the maximum positive integer such that the type can represent as a finite value 10 raised to that power (which is the value FLT_MAX_10_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::min

```
static T min() throw();
```

The function returns the minimum normalized value for the type (which is INT_MIN for type *int* and FLT_MIN for type *float*). The return value is meaningful if is_bounded is true or is_signed is false.

numeric_limits::min_exponent

```
static const int min_exponent = 0;
```

The member stores the minimum negative integer such that the type can represent as a normalized value radix raised to that power (which is the value FLT_MIN_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::min_exponent10

```
static const int min_exponent10 = 0;
```

The member stores the minimum negative integer such that the type can represent as a normalized value 10 raised to that power (which is the value FLT_MIN_10_EXP for type *float*). Meaningful only for floating-point types.

numeric_limits::quiet_NaN

```
static T quiet_NaN() throw();
```

The function returns a representation of a quiet NaN for the type. The return value is meaningful only if has_quiet_NaN is true.

numeric_limits::radix

```
static const int radix = 0;
```

The member stores the base of the representation for the type (which is 2 for the predefined integer types, and and the base to which the exponent is raised, or FLT_RADIX, for the predefined floating-point types).

numeric_limits::round_error

```
static T round_error() throw();
```

The function returns the maximum rounding error for the type.

numeric_limits::round_style

```
static const float_round_style round_style = round_toward_zero;
```

The member stores a value that describes the vaious methods that an implementation can choose for rounding a floating-point value to an integer value.

numeric_limits::signaling_NaN

```
static T signaling_NaN() throw();
```

The function returns a representation of a signaling NaN for the type. The return value is meaningful only if `has_signaling_NaN` is true.

numeric_limits::tinyness_before

```
static const bool tinyness_before = false;
```

The member stores true for a type that determines whether a value is ``tiny'' (too small to represent as a normalized value) before rounding, an option with IEC 559 floating-point representations that can affect some results.

numeric_limits::traps

```
static const bool traps = false;
```

The member stores true for a type that generates some kind of signal to report certain arithmetic exceptions.

See also the **Table of Contents** and the **Index**.

<list>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class T, class A>
    class list;
//    TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
template<class T, class A>
    void swap(
        const list<T, A>& lhs,
        const list<T, A>& rhs);
    };
```

Include the STL standard header **<list>** to define the container template class `list` and three supporting templates.

list

```
template<class T, class A = allocator<T> >
    class list {
public:
    typedef A allocator_type;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef reverse_bidirectional_iterator<iterator,
        value_type, reference, A::pointer,
            difference_type> reverse_iterator;
    typedef reverse_bidirectional_iterator<const_iterator,
        value_type, const_reference, A::const_pointer,
             difference_type> const_reverse_iterator;
    explicit list(const A& al = A());
    explicit list(size_type n, const T& v = T(), const A& al = A());
    list(const list& x);
    template<class InIt>
        list(InIt first, InIt last, const A& al = A());
    iterator begin();
    const_iterator begin() const;
    iterator end();
    iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
```

```
void resize(size_type n, T x = T());
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
template<class InIt>
    void assign(InIt first, InIt last);
template<class Size, class T2>
    void assign(Size n, const T2& x = T2());
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
void clear();
void swap(list x);
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first, iterator last);
void remove(const T& x);
templace<class Pred>
    void remove_if(Pred pr);
void unique();
template<class Pred>
    void unique(Pred pr);
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
void sort();
template<class Pred>
    void sort(Pred pr);
void reverse();
```

```
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type `T`**. The sequence is stored as a bidirectional linked list of elements, each containing a member of type `T`.

The object allocates and frees storage for the sequence it controls through a protected object named **`allocator`**, of **class `A`**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

List reallocation occurs when a member function must insert or erase elements of the controlled sequence. In all such cases, only iterators or references that point at erased portions of the controlled sequence become **invalid**.

list::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

list::assign

```
template<class InIt>
    void assign(InIt first, InIt last);
template<class Size, class T2>
    void assign(Size n, const T2& x = T2());
```

The first member template function replaces the sequence controlled by `*this` with the sequence `[first, last)`. The second member template function replaces the sequence controlled by `*this` with a repetition of `n` elements of value `x`.

In this implementation, if a translator does not support member template functions, the templates are replaced by:

```
void assign(const_iterator first, const_iterator last);
void assign(size_type n, const T& x = T());
```

list::back

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

list::begin

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

list::clear

```
void clear() const;
```

The member function calls erase(begin(), end()).

list::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type T1.

list::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

list::const_reverse_iterator

```
typedef reverse_bidirectional_iterator<const_iterator,
    value_type, const_reference, A::const_pointer,
        difference_type> const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

list::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

list::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

list::end

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

list::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

Erasing `N` elements causes `N` destructor calls. No reallocation occurs, so iterators and references become invalid only for the erased elements.

list::front

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

list::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

list::insert

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`. The member template function inserts the sequence `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(iterator it, const_iterator first, const_iterator last);
void insert(iterator it, const T *first, const T *last);
```

Inserting `N` elements causes `N` copies. No reallocation occurs, so no iterators or references become invalid.

list::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

list::list

```
explicit list(const A& al = A());
explicit list(size_type n, const T& v = T(), const A& al = A());
list(const list& x);
template<class InIt>
    list(InIt first, InIt last, const A& al = A());
```

All constructors store the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence. The first constructor specifies an empty initial controlled sequence. The second constructor specifies a repetition of `n` elements of value `x`. The third constructor specifies a copy of the sequence controlled by `x`. The member template constructor specifies the sequence `[first, last)`. None of the constructors perform any interim reallocations.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
list(const_iterator first, const_iterator last, const A& al = A());
```

list::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

list::merge

```
void merge(list& x);
template<class Pred>
    void merge(list& x, Pred pr);
```

Both member functions remove all elements from the sequence controlled by `x` and insert them in the controlled sequence. Both sequences must be ordered by the same predicate, described below. The resulting sequence is also ordered by that predicate.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in *ascending* order.) The member template function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`.

No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence. If a pair of elements in the resulting controlled sequence compares equal (`!(*Pi < *Pj) && !(*Pj < *Pi)`), an element from the original controlled sequence appears before an element from the sequence controlled by `x`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void merge(list& x, greater<T> pr);
```

list::pop_back

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

list::push_back

```
void push_back(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

list::pop_front

```
void pop_front();
```

The member function removes the first element of the controlled sequence, which must be non-empty.

list::push_front

```
void push_front(const T& x);
```

The member function inserts an element with value `x` at the beginning of the controlled sequence.

list::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

list::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

list::remove

```
void remove(const T& x);
```

The member function removes from the controlled sequence all elements, designated by the iterator `P`, for which `*P == x`.

list::remove_if

```
templace<class Pred>
    void remove_if(Pred pr);
```

The member template function removes from the controlled sequence all elements, designated by the iterator `P`, for which `pr(*P)` is true.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void remove_if(binder2nd< not_equal_to<T> > pr);
```

list::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

list::resize

```
void resize(size_type n, T x = T());
```

The member function ensures that `size()` henceforth returns `n`. If it must make the controlled sequence

longer, it appends elements with value `x`.

list::reverse

```
void reverse();
```

The member function reverses the order in which elements appear in the controlled sequence.

list::reverse_iterator

```
typedef reverse_bidirectional_iterator<iterator,
    value_type, reference, A::pointer,
        difference_type> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

list::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

list::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

list::sort

```
void sort();
template<class Pred>
    void sortsort(Pred pr);
```

Both member functions order the elements in the controlled sequence by a predicate, described below.

For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the first member function imposes the order `!(*Pj < *Pi)` whenever `i < j`. (The elements are sorted in *ascending* order.) The member template function imposes the order `!pr(*Pj, *Pi)` whenever `i < j`. No pairs of elements in the original controlled sequence are reversed in the resulting controlled sequence.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void sort(greater<T> pr);
```

list::splice

```
void splice(iterator it, list& x);
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first, iterator last);
```

The first member function inserts the sequence controlled by `x` before the element in the controlled sequence pointed to by `it`. It also removes all elements from `x`. (`&x` must not equal `this`.)

The second member function removes the element pointed to by `first` in the sequence controlled by `x` and inserts it before the element in the controlled sequence pointed to by `it`. (If `it == first || it == ++first`, no change occurs.)

The third member function inserts the subrange designated by `[first, last)` from the sequence controlled by `x` before the element in the controlled sequence pointed to by `it`. It also removes the original subrange from the sequence controlled by `x`. (If `&x == this`, the range `[first, last)` must not include the element pointed to by `it`.)

If the third member function inserts `N` elements, and `&x != this`, an object of class `iterator` is incremented `N` times. For all `splice` member functions, If `allocator != str.allocator`, a copy and a destructor call also occur for each inserted element.

list::swap

```
void swap(list& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

list::unique

```
void unique();
template<class Pred>
    void unique(Pred pr);
```

The first member function removes from the controlled sequence every element that compares equal to its preceding element. For the iterators `Pi` and `Pj` designating elements at positions `i` and `j`, the template member function removes every element for which `i + 1 == j && pr(*Pi, *Pj)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void unique(not_equal_to<T> pr);
```

For a controlled sequence of length `N` (> 0), the predicate `pr(*Pi, *Pj)` is evaluated `N - 1` times.

list::value_type

```
typedef A::value_type value_type;
```

The type is a synonym for the template parameter `T`.

operator!=

```
template<class T, class A>
    bool operator!=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `list`. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `list`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

swap

```
template<class T, class A>
    void swap(
        const list <T, A>& lhs,
        const list <T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

See also the **Table of Contents** and the **Index**.

<locale>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

codecvt · codecvt_base · codecvt_byname · collate · collate_byname · ctype · ctype<char> · ctype_base · ctype_byname · has_facet · locale · messages · messages_base · messages_byname · money_base · money_get · money_put · moneypunct · moneypunct_byname · num_get · num_put · numpunct · numpunct_byname · time_base · time_get · time_get_byname · time_put · time_put_byname · use_facet

isalnum · isalpha · iscntrl · isdigit · isgraph · islower · isprint · ispunct · isspace · isupper · isxdigit · tolower · toupper

---

```
namespace std {
    class locale;
    class ctype_base;
    template<class E>
        class ctype;
    class ctype<char>;
    template<class E>
        class ctype_byname;
    class codecvt_base;
    template<class From, class To, class State>
        class codecvt;
    template<class From, class To, class State>
        class codecvt_byname;
    template<class E, class InIt>
        class num_get;
    template<class E, class OutIt>
        class num_put;
    template<class E>
        class numpunct;
    template<class E>
        class numpunct_byname;
    template<class E>
        class collate;
    template<class E>
        class collate_byname;
    class time_base;
    template<class E, class InIt>
        class time_get;
```

```
    template<class E, class InIt>
        class time_get_byname;
    template<class E, class OutIt>
        class time_put;
    template<class E, class OutIt>
        class time_put_byname;
    class money_base;
    template<class E, bool Intl, class InIt>
        class money_get;
    template<class E, bool Intl, class OutIt>
        class money_put;
    template<class E, bool Intl>
        class moneypunct;
    template<class E, bool Intl>
        class moneypunct_byname;
    class messages_base;
    template<class E>
        class messages;
    template<class E>
        class messages_byname;
//    TEMPLATE FUNCTIONS
    template<class E>
        bool isspace(E c, const locale& loc) const;
    template<class E>
        bool isprint(E c, const locale& loc) const;
    template<class E>
        bool iscntrl(E c, const locale& loc) const;
    template<class E>
        bool isupper(E c, const locale& loc) const;
    template<class E>
        bool islower(E c, const locale& loc) const;
    template<class E>
        bool isalpha(E c, const locale& loc) const;
    template<class E>
        bool isdigit(E c, const locale& loc) const;
    template<class E>
        bool ispunct(E c, const locale& loc) const;
    template<class E>
        bool isxdigit(E c, const locale& loc) const;
    template<class E>
        bool isalnum(E c, const locale& loc) const;
    template<class E>
        bool isgraph(E c, const locale& loc) const;
    template<class E>
        E toupper(E c, const locale& loc) const;
```

```
    template<class E>
        E tolower(E c, const locale& loc) const;
    };
```

Include the standard header **<locale>** to define a host of template classes and functions that encapsulate and manipulate locales.

codecvt

```
template<class From, class To, class State>
    class codecvt : public locale::facet, public codecvt_base {
public:
    typedef From from_type;
    typedef To to_type;
    typedef State state_type;
    explicit codecvt(size_t refs = 0);
    result in(State& state,
        const To *first1, const To *last1, const To *next1,
        From *first2, From *last2, From *next2);
    result out(State& state,
        const From *first1, const From *last1, const From *next1,
        To *first2, To *last2, To *next2);
    bool always_noconv() const throw();
    int max_length() const throw();
    int length(State& state,
        From *first1, const From *last1, size_t _N2) const throw();
    int encoding() const throw();
    static locale::id id;
protected:
    ~codecvt();
    virtual result do_in(State& state,
        const To *first1, const To *last1, const To *next1,
        From *first2, From *last2, From *next2);
    virtual result do_out(State& state,
        const From *first1, const From *last1, const From *next1,
        To *first2, To *last2, To *next2);
    virtual bool do_always_noconv() const throw();
    virtual int do_max_length() const throw();
    virtual int do_encoding() const throw();
    virtual int do_length(State& state,
        From *first1, const From *last1, size_t len2) const throw();
    };
```

The template class describes an object that can serve as a locale facet, to control conversions between a sequence of values of type `From` and a sequence of values of type `To`. The class `State` characterizes the transformation -- and an object of class `State` stores any necessary state information during a conversion.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The template versions of `do_in` `do_out` always return `codecvt_base::noconv`. The Standard C++ library defines an explicit specialization, however, that is more useful:

```
codecvt<wchar_t, char, mbstate_t
```

which converts between `wchar_t` and *char* sequences.

codecvt::always_noconv

```
bool always_noconv() const throw();
```

The member function returns `do_always_noconv()`.

codecvt::codecvt

```
explicit codecvt(size_t refs = 0);
```

The constructor initializes its `locale::facet` base object with `locale::facet(refs)`.

codecvt::do_always_noconv

```
virtual bool do_always_noconv() const throw();
```

The protected virtual member function returns true only if every call to `do_in` or `do_out` returns `noconv`. The template version always returns true.

codecvt::do_encoding

```
virtual int do_encoding() const throw();
```

The protected virtual member function returns:

- -1, if the encoding of sequences of type `to_type` is state dependent
- 0, if the encoding involves sequences of varying lengths
- `n`, if the encoding involves only sequences of length `n`

codecvt::do_in

```
virtual result do_in(State state&,
    const To *first1, const To *last1, const To *next1,
    From *first2, From *last2, From *next2);
```

The protected virtual member function endeavors to convert the source sequence at `[first1, last1)` to a destination sequence that it stores within `[first2, last2)`. It always stores in `next1` a pointer to the first unconverted element in the source sequence, and it always stores in `next2` a pointer to the first unaltered element in the destination sequence.

`state` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

The function returns:

- `codecvt_base::error` if the source sequence is ill formed
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the source is insufficient, or if the destination is not large enough, for the conversion to succeed

The template version always returns `noconv`.

codecvt::do_length

```
virtual int do_length(State state&,
    From *first1, const From *last1, size_t len2) const throw();
```

The protected virtual member function effectively calls `do_out(state, first1, last1, next1, buf, buf + len2, next2)` for some buffer `buf` and pointer `next2`, then returns `next2 - buf`. (Thus, it is roughly analogous to the function `mbrlen`, at least when `From` is type `char`.)

The template version always returns the lesser of `last1 - first1` and `len2`.

codecvt::do_max_length

```
virtual int do_max_length() const throw();
```

The protected virtual member function returns the largest permissible value that can be returned by `do_length(first1, last1, 1)`, for arbitrary valid values of `first1` and `last1`. (Thus, it is roughly analogous to the macro `MB_CUR_MAX`, at least when `From` is type `char`.)

The template version always returns 1.

codecvt::do_out

```
virtual result do_out(State state&,
    const From *first1, const From *last1, const From *next1,
    To *first2, To *last2, To *next2);
```

The protected virtual member function endeavors to convert the source sequence at `[first1, last1)` to a destination sequence that it stores within `[first2, last2)`. It always stores in `next1` a pointer to the first unconverted element in the source sequence, and it always stores in `next2` a pointer to the first unaltered element in the destination sequence.

`state` must represent the initial conversion state at the beginning of a new source sequence. The function alters its stored value, as needed, to reflect the current state of a successful conversion. Its stored value is otherwise unspecified.

The function returns:

- `codecvt_base::error` if the source sequence is ill formed
- `codecvt_base::noconv` if the function performs no conversion
- `codecvt_base::ok` if the conversion succeeds
- `codecvt_base::partial` if the source is insufficient, or if the destination is not large enough, for the conversion to succeed

The template version always returns `noconv`.

codecvt::from_type

```
typedef From from_type;
```

The type is a synonym for the template parameter `From`.

codecvt::in

```
result in(State state&,
    const To *first1, const To *last1, const To *next1,
    From *first2, From *last2, From *next2);
```

The member function returns `do_in(state, first1, last1, next1, first2, last2, next2)`.

codecvt::length

```
int length(State state&,
    From *first1, const From *last1, size_t len2) const throw();
```

The member function returns `do_length(first1, last1, len2)`.

codecvt::encoding

```
int encoding() const throw();
```

The member function returns `do_encoding()`.

codecvt::max_length

```
int max_length() const throw();
```

The member function returns `do_max_length()`.

codecvt::out

```
result out(State state&,
    const From *first1, const From *last1, const From *next1,
    To *first2, To *last2, To *next2);
```

The member function returns do_out(state, first1, last1, next1, first2, last2, next2).

codecvt::state_type

```
typedef State state_type;
```

The type is a synonym for the template parameter State.

codecvt::to_type

```
typedef To to_type;
```

The type is a synonym for the template parameter To.

codecvt_base

```
class codecvt_base {
public:
    enum result {ok, partial, error, noconv};
    };
```

The class describes an enumeration common to all specializations of template class codecvt. The enumeration **result** describes the possible return values from do_in or do_out:

- **error** if the source sequence is ill formed
- **noconv** if the function performs no conversion
- **ok** if the conversion succeeds
- **partial** if the destination is not large enough for the conversion to succeed

codecvt_byname

```
template<class From, class To, class State>
    class codecvt_byname : public codecvt<From, To, State> {
public:
    explicit codecvt_byname(const char *s, size_t refs = 0);
protected:
    ~codecvt_byname();
    };
```

The template class describes an object that can serve as a locale facet of type codecvt<From, To, State>. Its behavior is determined by the named locale s. The constructor initializes its base object with codecvt<From, To, State>(refs).

collate

```
template<class E>
    class collate : public locale::facet {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit collate(size_t refs = 0);
    int compare(const E *first1, const E *last1,
        const E *first2, const E *last2) const;
    string_type transform(const E *first, const E *last) const;
    long hash(const E *first, const E *last) const;
    static locale::id id;
protected:
    ~collate();
    virtual int do_compare(const E *first1, const E *last1,
        const E *first2, const E *last2) const;
    virtual string_type do_transform(const E *first, const E *last) const;
    virtual long do_hash(const E *first, const E *last) const;
    };
```

The template class describes an object that can serve as a locale facet, to control comparisons of sequences of type `E`.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

collate::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

collate::collate

```
explicit collate(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

collate::compare

```
int compare(const E *first1, const E *last1,
    const E *first2, const E *last2) const;
```

The member function returns `do_compare(first1, last1, first2, last2)`.

collate::do_compare

```
virtual int do_compare(const E *first1, const E *last1,
    const E *first2, const E *last2) const;
```

The protected virtual member function compares the sequence at `[first1, last1)` with the sequence at `[first2, last2)`. It compares values by applying `operator<` between pairs of corresponding elements of type `E`. The first sequence compares less if it has the smaller element in the earliest unequal pair in the sequences, or if no unequal pairs exist but the first sequence is shorter.

If the first sequence compares less than the second sequence, the function returns -1. If the second sequence compares less, the function returns +1. Otherwise, the function returns zero.

collate::do_transform

```
virtual string_type do_transform(const E *first, const E *last) const;
```

The protected virtual member function returns an object of class `string_type` whose controlled sequence is a copy of the sequence `[first, last)`. If a class derived from `collate<E>` overrides `do_compare`, it should also override `do_transform` to match. Put simply, two transformed strings should yield the same result, when passed to `collate::compare`, that you would get from passing the untransformed strings to `compare` in the derived class.

collate::do_hash

```
virtual long do_hash(const E *first, const E *last) const;
```

The protected virtual member function returns an integer derived from the values of the elements in the sequence `[first, last)`. Such a **hash** value can be useful, for example, in distributing sequences pseudo randomly across an array of lists.

collate::hash

```
long hash(const E *first, const E *last) const;
```

The member function returns `do_hash(first, last)`.

collate::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` whose objects can store copies of the source sequence.

collate::transform

```
string_type transform(const E *first, const E *last) const;
```

The member function returns `do_transform(first, last)`.

collate_byname

```
template<class E>
    class collate_byname : public collate<E> {
public:
    explicit collate_byname(const char *s, size_t refs = 0);
protected:
    ~collate_byname();
    };
```

The template class describes an object that can serve as a locale facet of type `collate<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `collate<E>(refs)`.

ctype

---

char_type · ctype · do_is · do_narrow · do_scan_is · do_scan_not · do_tolower · do_toupper · do_widen · is · narrow · scan_is · scan_not · tolower · toupper · widen

---

```
template<class E>
    class ctype : public locale::facet, public ctype_base {
public:
    typedef E char_type;
    explicit ctype(size_t refs = 0);
    bool is(mask msk, E ch) const;
    const E *is(const E *first, const E *last, mask *dst) const;
    const E *scan_is(mask msk, const E *first, const E *last) const;
    const E *scan_not(mask msk, const E *first, const E *last) const;
    E toupper(E ch) const;
    const E *toupper(E *first, E *last) const;
    E tolower(E ch) const;
    const E *tolower(E *first, E *last) const;
    E widen(char ch) const;
    const char *widen(char *first, char *last, E *dst) const;
    char narrow(E ch, char dflt) const;
    const E *narrow(const E *first, const E *last,
        char dflt, char *dst) const;
    static locale::id id;
protected:
    ~ctype();
    virtual bool do_is(mask msk, E ch) const;
    virtual const E *do_is(const E *first, const E *last,
        mask *dst) const;
```

```
    virtual const E *do_scan_is(mask msk, const E *first,
        const E *last) const;
    virtual const E *do_scan_not(mask msk, const E *first,
        const E *last) const;
    virtual E do_toupper(E ch) const;
    virtual const E *do_toupper(E *first, E *last) const;
    virtual E do_tolower(E ch) const;
    virtual const E *do_tolower(E *first, E *last) const;
    virtual E do_widen(char ch) const;
    virtual const char *do_widen(char *first, char *last, E *dst) const;
    virtual char do_narrow(E ch, char dflt) const;
    virtual const E *do_narrow(const E *first, const E *last,
        char dflt, char *dst) const;
    };
```

The template class describes an object that can serve as a locale facet, to characterize various properties of a ``character'' (element) of type `E`. Such a facet also converts between sequences of `E` elements and sequences of *char*.

An object of class `ctype<E>` stores a pointer to the first element of a **ctype mask table**, an array of `UCHAR_MAX + 1` elements of typ `ctype_base::mask`. It also stores a boolean object that indicates whether the array should be deleted when the `ctype<E>` object is destroyed.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The Standard C++ library defines two explicit specializations of this template class:

- `ctype<char>`, whose differences are described spparately
- `ctype<wchar_t<`, which treats elements as wide characters

In this implementation, other specializations of template class `ctype<E>`:

- convert a value `ch` of type `E` to a value of type *char* with the expression `(char)ch`
- convert a value `c` of type *char* to a value of type `E` with the expression `E(c)`

All other operations are performed on *char* values the same as for the specialization `ctype<char>`.

ctype::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

ctype::ctype

```
explicit ctype(size_t refs = 0);
```

The type is a synonym for the template parameter `E`.

ctype::do_is

```
virtual bool do_is(mask msk, E ch) const;
virtual const E *do_is(const E *first, const E *last,
    mask *dst) const;
```

The first protected member template function returns true if `table[(unsigned char)(char)ch] & msk` is nonzero, where `table` is the stored pointer to the ctype mask table.

The second protected member template function stores in `dst[I]` the value `table[(unsigned char)(char)first[I]] & msk`, where `I` ranges over the interval `[0, last - first)`.

ctype::do_narrow

```
virtual char do_narrow(E ch, char dflt) const;
virtual const E *do_narrow(const E *first, const E *last,
    char dflt, char *dst) const;
```

The first protected member template function returns `(char)ch`, or `dflt` if that expression is undefined.

The second protected mem er template function stores in `dst[I]` the value `do_narrow(first[I], dflt)`, for `I` in the interval `[0, last - first)`.

ctype::do_scan_is

```
virtual const E *do_scan_is(mask msk, const E *first,
    const E *last) const;
```

The protected member function returns the smallest pointer `p` in the range `[first, last)` for which `do_is(msk, *p)` is true. If no such value xxists, the function returns `last`.

ctype::do_scan_not

```
virtual const E *do_scan_not(mask msk, const E *first,
    const E *last) const;
```

The protected member function returns the smallest pointer `p` in the range `[first, last)` for which `do_is(msk, *p)` is false. If no such value exists, the function returns `last`.

ctype::do_tolower

```
virtual E do_tolower(E ch) const;
virtual const E *do_tolower(E *first, E *last) const;
```

The first protected member template function returns the lowercase character corresponding to `ch`, if such a character exists. Otherwise, it returns `ch`.

The second protected member template function replaces each element `first[I]`, for `I` in the interval `[0, last - first)`, with `do_tolower(first[I])`.

ctype::do_toupper

```
virtual E do_toupper(E ch) const;
virtual const E *do_toupper(E *first, E *last) const;
```

The first protected member template function returns the uppercase character corresponding to `ch`, if such a character exists. Otherwise, it returns `ch`.

The second protected member template function replaces each element `first[I]`, for `I` in the interval `[0, last - first)`, with `do_toupper(first[I])`.

ctype::do_widen

```
virtual E do_widen(char ch) const;
virtual const char *do_widen(char *first, char *last, E *dst) const;
```

The first protected member template function returns `E(ch)`.

The second protected member template function stores in `dst[I]` the value `do_widen(first[I])`, for `I` in the interval `[0, last - first)`.

ctype::is

```
bool is(mask msk, E ch) const;
const E *is(const E *first, const E *last, mask *dst) const;
```

The first member function returns `do_is(msk, ch)`. The second member function returns `do_is(first, last, dst)`.

ctype::narrow

```
char narrow(E ch, char dflt) const;
const E *narrow(const E *first, const E *last,
    char dflt, char *dst) const;
```

The first member function returns `do_narrow(ch, dflt)`. The second member function returns `do_narrow(first, last, dflt, dst)`.

ctype::scan_is

```
const E *scan_is(mask msk, const E *first, const E *last) const;
```

The member function returns `do_scan_is(msk, first, last)`.

ctype::scan_not

```
const E *scan_not(mask msk, const E *first, const E *last) const;
```

The member function returns `do_scan_not(msk, first, last)`.

ctype::tolower

```
E tolower(E ch) const;
const E *tolower(E *first, E *last) const;
```

The member function returns do_tolower(first, last).

ctype::toupper

```
E toupper(E ch) const;
const E *toupper(E *first, E *last) const;
```

The member function returns do_toupper(first, last).

ctype::widen

```
E widen(char ch) const;
const char *widen(char *first, char *last, E *dst) const;
```

The member function returns do_widen(first, last, dst).

ctype<char>

```
class ctype<char> : public locale::facet, public ctype_base {
public:
    typedef char char_type;
    explicit ctype(const mask *tab = 0, bool del = false,
        size_t refs = 0);
    bool is(mask msk, char ch) const;
    const char *is(const char *first, const char *last,
        mask *dst) const;
    const char *scan_is(mask msk,
        const char *first, const char *last) const;
    const char *scan_not(mask msk,
        const char *first, const char *last) const;
    char toupper(char ch) const;
    const char *toupper(char *first, char *last) const;
    char tolower(char ch) const;
    const char *tolower(char *first, char *last) const;
    char widen(char ch) const;
    const char *widen(char *first, char *last, char *dst) const;
    char narrow(char ch, char dflt) const;
    const char *narrow(const char *first, const char *last,
        char dflt, char *dst) const;
    static locale::id id;
protected:
    ~ctype();
    virtual char do_toupper(char ch) const;
```

```
    virtual const char *do_toupper(char *first, char *last) const;
    virtual char do_tolower(char ch) const;
    virtual const char *do_tolower(char *first, char *last) const;
    const mask *table() const throw();
    static const mask *classic_table() const throw();
    static const size_t table_size;
    };
```

The class is an explicit specialization of template class `ctype` for type *char*. Hence, it describes an object that can serve as a locale facet, to characterize various properties of a ``character'' (element) of type *char*. The explicit specialization differs from the template class in several ways:

- Its sole public constructor lets you specify `tab`, the ctype mask table, and `del`, the boolean object that is true if the array should be deleted when the `ctype<char>` object is destroyed -- as well as the usual reference-count parameter `refs`.
- The protected member function **`table()`** returns the stored ctype mask table.
- The static member object **`table_size`** specifies the minimum number of elements in a ctype mask table.
- The protected static member function **`classic_table()`** returns the ctype mask table appropriate to the `"C"` locale.
- There are no protected virtual member functions `do_is`, `do_narrow`, `do_scan_is`, `do_scan_not`, or `do_widen`. The corresponding public member functions perform the equivalent operations themselves.
- The member functions `narrow` and `widen` simply copy elements unaltered.

ctype_base

```
class ctype_base {
public:
    enum mask;
    static const mask space, print, cntrl,
        upper, lower, digit, punct, xdigit,
        alpha, alnum, graph;
    };
```

The class serves as a base class for facets of template class `ctype`. It defines just the enumerated type **`mask`** and several constants of this type. Each of the constants characterizes a different way to classify characters, as defined by the functions with similar names declared in the header `<ctype.h>`. The constants are:

- **`space`** (function `isspace`)
- **`print`** (function `isprint`)
- **`cntrl`** (function `iscntrl`)
- **`upper`** (function `isupper`)
- **`lower`** (function `islower`)

- **digit** (function isdigit)
- **punct** (function ispunct)
- **xdigit** (function isxdigit)
- **alpha** (function isalpha)
- **alnum** (function isalnum)
- **graph** (function isgraph)

You can charaterize a combination of classifications by ORing these constants. In particular, it is always true that `alnum == (alpha | digit)` and `graph == (alnum | punct)`.

ctype_byname

```
template<class E>
    class ctype_byname : public ctype<E> {
public:
    explicit ctype_byname(const char *s, size_t refs = 0);
protected:
    ~ctype_byname();
    };
```

The template class describes an object that can serve as a locale facet of type `ctype<E>`. Its behavior is determined by the named locale `s`. The constructor initializes its base object with `ctype<E>(refs)` (or the equivalent for base class `ctype<char>`).

has_facet

```
template<class Facet>
    bool has_facet(const locale& loc) const;
```

The template function returns true if a locale facet of class `Facet` is listed within the locale object `loc`.

In this implementation, you should write **`_HAS(loc, Facet)`** in place of `has_facet<Facet>(loc)`, which not all translators currently support.

isalnum

```
template<class E>
    bool isalnum(E c, const locale& loc) const;
```

The template function returns `use_facet< ctype<E> >(loc). is(ctype<E>:: alnum, c)`.

isalpha

```
template<class E>
    bool isalpha(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: alpha, c).

iscntrl

```
template<class E>
    bool iscntrl(E c, const locale& loc) const;
```

The eemplate function returns use_facet< ctype<E> >(loc). is(ctype<E>:: cntrl, c).

isdigit

```
template<class E>
    bool isdigit(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: digit, c).

isgraph

```
template<class E>
    bool isgraph(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: graph, c).

islower

```
template<class E>
    bool islower(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: lower, c).

isprint

```
template<class E>
    bool isprint(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: print, c).

ispunct

```
template<class E>
    bool ispunct(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: punct, c).

isspace

```
template<class E>
    bool isspace(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: space, c).

isupper

```
template<class E>
    bool isupper(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: upper, c).

isxdigit

```
template<class E>
    bool isxdigit(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). is(ctype<E>:: xdigit, c).

locale

---

category · classic · facet · global · id · locale · name · operator!= · operator() · operator==

```
class locale {
public:
    class facet;
    class id;
    typedef int category;
    static const category none, collate, ctype, monetary,
        numeric, time, messages, all;
    locale();
    explicit locale(const char *s);
    locale(const locale& x, const locale& y,
```

```
    category cat);
  locale(const locale& x, const char *s, category cat);
  template<class Facet>
    locale(const locale& x, Facet *fac);
  template<class Facet>
    locale(const locale& x, const locale& y);
  string name() const;
  bool operator==(const locale& x) const;
  bool operator!=(const locale& x) const;
  template<class E>
    bool operator()(const basic_string<E> lhs,
        const basic_string<E> rhs) const;
  static locale global(const locale& x);
  static const locale& classic();
  };
```

The class describes a **locale object** that encapsulates a locale. It represents culture-specific information as a list of **facets**. A facet is a pointer to an object of a class derived from class `facet` that has a public object of the form:

```
static locale::id id;
```

You can define an open-ended set of these facets. You can also construct a locale object that designates an arbitrary number of facets.

Predefined groups of these facets represent the locale categories traditionally managed in the Standard C library by the function `setlocale`.

Category **`collate`** (`LC_COLLATE`) includes the facets:

```
collate<char>
collate<wchar_t>
```

Category **`ctype`** (`LC_CTYPE`) includes the facets:

```
ctype<char>
ctype<wchar_t>
codecvt<char, char, mbstate_t>
codecvt<wchar_t, char, mbstate_t>
```

Category **`monetary`** (`LC_MONETARY`) includes the facets:

```
moneypunct<char, false>
moneypunct<wchar_t, false>
moneypunct<char, true>
moneypunct<wchar_t, true>
money_get<char, istreambuf_iterator<char> >
money_get<wchar_t, istreambuf_iterator<wchar_t> >
money_put<char, ostreambuf_iterator<char> >
money_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **numeric** (LC_NUMERIC) includes the facets:

```
num_get<char, istreambuf_iterator<char> >
num_get<wchar_t, istreambuf_iterator<wchar_t> >
num_put<char, ostreambuf_iterator<char> >
num_put<wchar_t, ostreambuf_iterator<wchar_t> >
numpunct<char>
numpunct<wchar_t>
```

Category **time** (LC_TIME) includes the facets:

```
time_get<char, istreambuf_iterator<char> >
time_get<wchar_t, istreambuf_iterator<wchar_t> >
time_put<char, ostreambuf_iterator<char> >
time_put<wchar_t, ostreambuf_iterator<wchar_t> >
```

Category **messages** [sic] (LC_MESSAGE) includes the facets:

```
messages<char>
messages<wchar_t>
```

(The last category is required by Posix, but not the C Standard.)

Some of these predefined facets are used by the iostreams classes, to control the conversion of numeric values to and from text sequences.

An object of class `locale` also stores a **locale name** as an object of class `string`. Using an invalid locale name to construct a locale facet or a locale object throws an object of class `runtime_error`. If the stored locale name is `"*"`, no C-style locale corresponds exactly to that represented by the object. Otherwise, you can establish a matching locale within the Standard C library by calling `setlocale( LC_ALL, x.name. c_str())`.

In this implementation, you can also call the static member function:

```
static locale empty();
```

to construct a locale object that has no facets. It is also a **transparent locale** -- the template function `use_facet` consults the global locale if it cannot find the requested facet in a transparent locale. Thus, you can write:

```
cout.imbue(locale::empty());
```

Subsequent insertions to `cout` are mediated by the current state of the global locale. You can even write:

```
locale loc(locale::empty(), locale("C"), locale::numeric);
cout.imbue(loc);
```

Numeric formatting rules remain the same as in the C locale even as the global locale supplies changing rules for inserting dates and monetary amounts.

locale::category

```
typedef int category;
static const category none, collate, ctype, monetary,
    numeric, time, messages, all;
```

The type is a synonym for *int*, so that it can represent any of the C locale categories. It can also represent a group of constants local to class `locale`:

- **none**, corresponding to none of the the C categories
- **collate**, corresponding to the C category `LC_COLLATE`
- **ctype**, corresponding to the C category `LC_CTYPE`
- **monetary**, corresponding to the C category `LC_MONETARY`
- **numeric**, corresponding to the C category `LC_NUMERIC`
- **time**, corresponding to the C category `LC_TIME`
- **messages**, corresponding to the Posix category `LC_MESSAGE`
- **all**, corresponding to the C union of all categories `LC_ALL`

You can represent an arbitrary group of categories by ORing these constants, as in `monetary | time`.

locale::classic

```
static const locale& classic();
```

The static member function returns a locale object that represents the C locale.

locale::facet

```
class facet {
protected:
    explicit facet(size_t refs = 0);
    virtual ~facet();
private:
    facet(const facet&)          // not defined
    void operator=(const facet&) // not defined
    };
```

The member class serves as the base class for all locale facets. Note that you can neither copy nor assign an object of class `facet`. You can construct and destroy objects derived from class `locale::facet`, but not objects of the base class proper. Typically, you construct an object `myfac` derived from `facet` when you construct a locale, as in:

```
locale loc(locale::classic(), new myfac);
```

In such cases, the constructor for the base class `facet` should have a zero `refs` argument. When the object is no longer needed, it is deleted. Thus, you supply a nonzero `refs` argument only in those rare cases where you take responsibility for the lifetime of the object.

locale::global

```
static locale global(const locale& x);
```

The static member function stores a copy of `x` as the **global locale**. It also calls `setlocale( LC_ALL, x.name. c_str())`, to establishing a matching locale within the Standard C library. The function then returns the previous global locale. At program startup, the global locale represents the C locale.

locale::id

```
class id {
protected:
    id();
private:
    id(const id&)              // not defined
    void operator=(const id&) // not defined
    };
```

The member class describes the static member object required by each unique locale facet. Note that you can neither copy nor assign an object of class `id`.

locale::locale

```
locale();
explicit locale(const char *s);
locale(const locale& x, const locale& y,
    category cat);
locale(const locale& x, const char *s, category cat);
template<class Facet>
    locale(const locale& x, Facet *fac);
template<class Facet>
    locale(const locale& x, const locale& y);
```

The first constructor initializes the object to match the global locale. The second constructor initializes all the locale categories to have behavior consistent with the locale name `s`. The remaining constructors copy `x`, with the exceptions noted:

```
locale(const locale& x, const locale& y,
    category cat);
```

replaces from `y` those facets corresponding to a category `c` for which `c & cat` is nonzero.

```
locale(const locale& x, const char *s, category cat);
```

replaces from `locale(s, all)` those facets corresponding to a category `c` for which `c & cat` is nonzero.

```
template<class Facet>
    locale(const locale& x, Facet *fac);
```

replaces (or adds) the facet `Facet` with `fac`, if `fac` is not a null pointer.

```
template<class Facet>
    locale(const locale& x, const locale& y);
```

replaces (or adds) the facet `Facet` listed in `y`.

If a locale name `s` is a null pointer or otherwise invalid, the function throws `runtime_error`.

In this implementation, you should write **`_ADDFAC(loc, Facet)`** to return a new locale that adds the facet `Facet` to the locale `loc`, since not all translators currently support member templates.

locale::name

```
string name() const;
```

The member function returns the stored locale name.

locale::operator!=

```
bool operator!=(const locale& x) const;
```

The member function returns `!(*this == x)`.

locale::operator()

```
template<class E>
    bool operator()(const basic_string<E> lhs,
        const basic_string<E> rhs) const;
```

The member function effectively executes:

```
const collate<E>& fac = use_fac<collate<E> >(*this);
return (fac.compare(lhs.begin(), lhs.end(),
    rhs.begin(), rhs.end()) < 0);
```

Thus, you can use a locale object as a function object.

locale::operator==

```
bool operator==(const locale& x) const;
```

The member function returns true only if *this and `x` are copies of the same locale or have the same name (other than `"*"`).

messages

```
template<class E>
    class messages : public locale::facet, public messages_base {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
```

```
    explicit messages(size_t refs = 0);
    catalog open(const string& name,
        const locale& loc) const;
    string_type get(catalog cat, int set, int msg,
        const string_type& dflt) const;
    void close(catalog cat) const;
    static locale::id id;
protected:
    ~messages();
    virtual catalog do_open(const string& name,
        const locale& loc) const;
    virtual string_type do_get(catalog cat, int set, int msg,
        const string_type& dflt) const;
    virtual void do_close(catalog cat) const;
    };
```

The template class describes an object that can serve as a locale facet, to characterize various properties of a **message catalog** that can supply messages represented as sequences of elements of type `E`.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

messages::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

messages::close

```
void close(catalog cat) const;
```

The member function calls `do_close(cat);`.

messages::do_close

```
virtual void do_close(catalog cat) const;
```

The protected member function closes the `message catalog`, which must have been opened by an earlier call to `do_open`.

messages::do_get

```
virtual string_type do_get(catalog cat, int set, int msg,
    const string_type& dflt) const;
```

The protected member function endeavors to obtain a message sequence from the `message catalog` `cat`. It may make use of `set`, `msg`, and `dflt` in doing so. It returns a copy of `dflt` on failure. Otherwise, it

returns a copy of the specified message sequence.

In this implementation, the function returns a locale-specific version of the sequence **no** if `msg` is zero. It returns a locale-specific version of the sequence **yes** if `msg` is one. Otherwise, it returns `dflt`.

messages::do_open

```
virtual catalog do_open(const string& name,
    const locale& loc) const;
```

The protected member function endeavors to open a `message catalog` whose name is `name`. It may make use of the locale `loc` in doing so. It returns a value that compares less than zero on failure. Otherwise, the returned value can be used as the first argument on a later call to `get`. It should in any case be used as the argument on a later call to `close`.

In this implementation, the function always returns zero.

messages::get

```
string_type get(catalog cat, int set, int msg,
     const string_type& dflt) const;
```

The member function returns `do_get(cat, set, msg, dflt);`.

messages::messages

```
explicit messages(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

messages::open

```
catalog open(const string& name,
    const locale& loc) const;
```

The member function returns `do_open(name, loc);`.

messages::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` whose objects can store copies of the message sequences.

messages_base

```
class messages_base {
    typedef int catalog;
    };
```

The class describes a type common to all specializations of template class messages. The type **catalog** is a synonym for type *int* that describes the possible return values from messages::do_open.

messages_byname

```
template<class E>
    class messages_byname : public messages<E> {
public:
    explicit messages_byname(const char *s, size_t refs = 0);
protected:
    ~messages_byname();
    };
```

The template class describes an object that can serve as a locale facet of type messages<E>. Its behavior is determined by the named locale s. The constructor initializes its base object with messages<E>(refs).

money_base

```
class money_base {
    enum part {none, sign, space,
        symbol, value};
    struct pattern {
        char field[4];
        };
    };
```

The class describes an enumeration and a structure common to all specializations of template class moneypunct. The enumeration **part** describes the possible values in elements of the array **field** in the structure **pattern**. The values of part are:

- **none** to match zero or more spaces or generate nothing
- **sign** to match or generate a positive or negative sign
- **space** to match zero or more spaces or generate a space
- **symbol** to match or generate a currency symbol
- **value** to match or generate a monetary value

money_get

```
template<class E,
    class InIt = istreambuf_iterator<E> >
    class money_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    typedef basic_string<E> string_type;
```

```
    explicit money_get(size_t refs = 0);
    iter_type get(iter_type first, iter_type last, bool intl,
        ios_base& x, ios_base::iostate& st, long double& val) const;
    iter_type get(iter_type first, iter_type last, bool intl,
        ios_base& x, ios_base::iostate& st, string_type& val) const;
    static locale::id id;
protected:
    ~money_get();
    virtual iter_type do_get(iter_type first, iter_type last, bool intl,
        ios_base& x, ios_base::iostate& st, string_type& val) const;
    virtual iter_type do_get(iter_type first, iter_type last, bool intl,
        ios_base& x, ios_base::iostate& st, long double& val) const;
    };
```

The template class describes an object that can serve as a locale facet, to control conversions of sequences of type `E` to monetary values.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

money_get::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

money_get::do_get

```
virtual iter_type do_get(iter_type first, iter_type last, bool intl,
    ios_base& x, ios_base::iostate& st, string_type& val) const;
virtual iter_type do_get(iter_type first, iter_type last, bool intl,
    ios_base& x, ios_base::iostate& st, long double& val) const;
```

The first virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **monetary input field**. If successful, it converts this field to a sequence of one or more decimal digits, optionally preceded by a minus sign (`-`), to represent the amount and stores the result in the `string_type` object `val`. It returns an iterator designating the first element beyond the monetary input field. Otherwise, the function stores an empty sequence in `val` and sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid monetary input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The second virtual protected member function behaves the same as the first, except that if successful it converts the optionally-signed digit sequence to a value of type *long double* and stores that value in `val`.

The format of a monetary input field is determined by the locale facet `fac` returned by the (effective) call `use_facet <moneypunct<E, intl>(x. getloc())`. Specifically:

- `fac.neg_format()` determines the order in which components of the field occur

- `fac.curr_symbol()` determines the sequence of elements that constitutes a currency symbol
- `fac.positive_sign()` determines the sequence of elements that constitutes a positive sign
- `fac.negative_sign()` determines the sequence of elements that constitutes a negative sign
- `fac.grouping()` determines how digits are grouped to the left of any decimal point
- `fac.thousands_sep()` determines the element that separates groups of digits to the left of any decimal point
- `fac.decimal_point()` determines the element that separates the integer digits from the fraction digits
- `fac.frac_digits()` determines the number of significant fraction digits to the right of any decimal point

If the sign string (`fac.negative_sign` or `fac.positive_sign`) has more than one element, only the first element is matched where the element equal to **`money_base::sign`** appears in the format pattern (`fac.neg_format`). Any remaining elements are matched at the end of the monetary input field. If neither string has a first element that matches the next element in the monetary input field, the sign string is taken as empty and the sign is positive.

If `x.flags() & showbase` is nonzero, the string `fac.curr_symbol` *must* match where the element equal to **`money_base::symbol`** appears in the format pattern. Otherwise, if `money_base::symbol` occurs at the end of the format pattern, and if no elements of the sign string remain to be matched, the currency symbol is *not* matched. Otherwise, the currency symbol is *optionally* matched.

If no instances of `fac.thousands_sep()` occur in the value portion of the monetary input field (where the element equal to **`money_base::value`** appears in the format pattern), no grouping constraint is imposed. Otherwise, any grouping constraints imposed by `fac.grouping()` is enforced. Note that the resulting digit sequence represents an integer whose low-order `fac.frac_digits()` decimal digits are considered to the right of the decimal point.

Arbitrary white space is matched where the element equal to **`money_base::space`** appears in the format pattern, if it appears other than at the end of the format pattern. Otherwise, no internal white space is matched. An element `c` is considered white space if `use_facet <ctype<E>(x.getloc()).is(ctype_base::space, c)` is true.

`money_get::get`

```
iter_type get(iter_type first, iter_type last, bool intl,
    ios_base& x, ios_base::iostate& st, long double& val) const;
iter_type get(iter_type first, iter_type last, bool intl,
    ios_base& x, ios_base::iostate& st, string_type& val) const;
```

Both member functions return `do_get(first, last, intl, x, st, val)`.

money_get::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `InIt`.

money_get::money_get

```
explicit money_get(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

money_get::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` whose objects can store sequences of elements from the source sequence.

money_put

```
template<class E,
    class OutIt = ostreambuf_iterator<E> >
    class money_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    typedef basic_string<E> string_type;
    explicit money_put(size_t refs = 0);
    iter_type put(iter_type next, bool intl, ios_base& x,
        E fill, long double& val) const;
    iter_type put(iter_type next, bool intl, ios_base& x,
        E fill, string_type& val) const;
    static locale::id id;
protected:
    ~money_put();
    virtual iter_type do_put(iter_type next, bool intl,
        ios_base& x, E fill, string_type& val) const;
    virtual iter_type do_put(iter_type next, bool intl,
        ios_base& x, E fill, long double& val) const;
    };
```

The template class describes an object that can serve as a locale facet, to control conversions of monetary values to sequences of type `E`.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

money_put::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

money_put::do_put

```
virtual iter_type do_put(iter_type next, bool intl,
    ios_base& x, E fill, string_type& val) const;
virtual iter_type do_put(iter_type next, bool intl,
    ios_base& x, E fill, long double& val) const;
```

The first virtual protected member function generates sequential elements beginning at `next` to produce a **monetary output field** from the `string_type` object `val`. The sequence controlled by `val` must begin with one or more decimal digits, optionally preceded by a minus sign (`-`), which represents the amount. The function returns an iterator designating the first element beyond the generated monetary output field.

The second virtual protected member function behaves the same as the first, except that it effectively first converts `val` to a sequence of decimal digits, optionally preceded by a minus sign, then converts that sequence as above.

The format of a monetary output field is determined by the locale facet `fac` returned by the (effective) call `use_facet <moneypunct<E, intl>(x. getloc())`. Specifically:

- `fac.pos_format()` determines the order in which components of the field are generated for a non-negative value
- `fac.neg_format()` determines the order in which components of the field are generated for a negative value
- `fac.curr_symbol()` determines the sequence of elements to generate for a currency symbol
- `fac.positive_sign()` determines the sequence of elements to generate for a positive sign
- `fac.negative_sign()` determines the sequence of elements to generate for a negative sign
- `fac.grouping()` determines how digits are grouped to the left of any decimal point
- `fac.thousands_sep()` determines the element that separates groups of digits to the left of any decimal point
- `fac.decimal_point()` determines the element that separates the integer digits from any fraction digits
- `fac.frac_digits()` determines the number of significant fraction digits to the right of any decimal point

If the sign string (`fac.negative_sign` or `fac.positive_sign`) has more than one element, only the first element is generated where the element equal to **`money_base::sign`** appears in the format pattern (`fac.neg_format` or `fac.pos_format`). Any remaining elements are generated at the end of the monetary output field.

If `x.flags() & showbase` is nonzero, the string `fac.curr_symbol` is generated where the element equal to **`money_base::symbol`** appears in the format pattern. Otherwise, no currency symbol is generated.

If no grouping constraints are imposed by `fac.grouping()` (its first element has the value CHAR_MAX) then no instances of `fac.thousands_sep()` are generated in the value portion of the monetary output field (where the element equal to **`money_base::value`** appears in the format pattern). If `fac.frac_digits()` is zero, then no instance of `fac.decimal_point()` is generated after the decimal digits. Otherwise, the resulting monetary output field places the low-order `fac.frac_digits()` decimal digits to the right of the decimal point.

Padding occurs as for any numeric output field, except that if `x.flags() & x.internal` is nonzero, any internal padding is generated where the element equal to **`money_base::space`** appears in the format pattern, if it does appear. Otherwise, internal padding occurs before the generated sequence. The padding character is `fill`.

The function calls `x.width(0)` to reset the field width to zero.

money_put::put

```
iter_type put(iter_type next, bool intl, ios_base& x,
    E fill, long double& val) const;
iter_type put(iter_type iter_type next, bool intl, ios_base& x,
    E fill, string_type& val) const;
```

Both member functions return `do_put(next, intl, x, fill, val)`.

money_put::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter `OutIt`.

money_put::money_put

```
explicit money_put(size_t refs = 0);
```

The constructor initializes its base object with `locale::facet(refs)`.

money_put::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class `basic_string` whose objects can store sequences of elements from the source sequence.

moneypunct

---

char_type · **curr_symbol** · **decimal_point** · **do_curr_symbol** · **do_decimal_point** · **do_frac_digits** · **do_grouping** · **do_neg_format** · **do_negative_sign** · **do_pos_format** · **do_positive_sign** · **do_thousands_sep** ·

```
template<class E, bool Intl>
    class moneypunct : public locale::facet, public money_base {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit moneypunct(size_t refs = 0);
    E decimal_point() const;
    E thousands_sep() const;
    string grouping() const;
    string_type curr_symbol() const;
    string_type positive_sign() const;
    string_type negative_sign() const;
    int frac_digits() const;
    pattern pos_format(  oonst;
    pattern neg_format() const;
    static const bool intl = Intl;
    static locale::id id;
protected:
    ~moneypunct();
    virtual E do_decimal_point() const;
    virtual E do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_curr_symbol() const;
    virtual string_type do_positive_sign() const;
    virtual string_type do_negative_sign() const;
    virtual int do_frac_digits() const;
    virtual pattern do_pos_format() const;
    virtual pattern do_neg_format() const;
    };
```

The template class describes an object that can serve as a locale facet, to desceibe the sequences of type `E` used to represent a **monetary input field** or a **monetary output field**. If the template parameter `Intl` is true, international conventions are observed.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

The const static object **`intl`** stores the value of the template parameter `Intl`.

moneypunct::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

moneypunct::curr_symbol

```
string_type curr_symbol() const;
```

The member function returns do_curr_symbol().

moneypunct::decimal_point

```
E decimal_point() const;
```

The member function returns do_decimal_point().

moneypunct::do_curr_symbol

```
string_type do_curr_symbol() const;
```

The protected virtual member function returns a locale-specific sequence of elements to use as a currency symbol.

moneypunct::do_decimal_point

```
E do_decimal_point() const;
```

The protected virtual member function returns a locale-specific element to use as a decimal-point.

moneypunct::do_frac_digits

```
int do_frac_digits() const;
```

The protected virtual member function returns a locale-specific count of the number of digits to display to the right of any decimal point.

moneypunct::do_grouping

```
string do_grouping() const;
```

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for lconv::grouping.

moneypunct::do_neg_format

```
pattern do_neg_format() const;
```

The protected virtual member function returns a locale-specific rule for determining how to generate a

monetary output field for a neeative amount. Each of the four elements of `pattern::field` can have the values:

- **`none`** to match zero or more spaces or generate nothing
- **`sign`** to match or generate a positive or negative sign
- **`space`** to match zero or more spaces or generate a space
- **`symbol`** to match or generate a currency symbol
- **`value`** to match or generate a monetary value

Components of a monetary output field are generated (and components of a monetary input field are matched) in the order in which these elements appear in `pattern::field`. Each of the values `sign`, `symbol`, `value`, and either `none` or `space` must appear exactly once. The value `none` must not appear first. The value `space` must not appear first or last. If `Intl` is true, the order is `symbol`, `sign`, `none`, then `value`.

The template version of `moneypunct<E, Intl>` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

moneypunct::do_negative_sign

```
string_type do_negative_sign() const;
```

The protected virtual member function returns a locale-specific sequence of elements to use as a negative sign.

moneypunct::do_pos_format

```
pattern do_pos_format() const;
```

The protected virtual member function returns a locale-specific rule for determining how to generate a monetary output field for a positive amount. (It also determines how to match the components of a monetary input field.) The encoding is the same as for `do_neg_format`.

The template version of `moneypunct<E, Intl>` returns `{money_base::symbol, money_base::sign, money_base::value, money_base::none}`.

moneypunct::do_positive_sign

```
string_type do_positive_sign() const;
```

The protected virtual member function returns a locale-specific sequence of elements to use as a positive sign.

moneypunct::do_thousands_sep

```
E do_thousands_sep() const;
```

The protected virtual member function returns a locale-specific element to use as a group separator to the left of any decimal point.

moneypunct::frac_digits

```
int frac_digits() const;
```

The member function returns do_frac_digits().

moneypunct::grouping

```
string grouping() const;
```

The member function returns do_grouping().

moneypunct::moneypunct

```
explicit moneypunct(size_t refs = 0);
```

The constructor initializes its base object with locale::facet(refs).

moneypunct::neg_format

```
pattern neg_format() const;
```

The member function returns do_neg_format().

moneypunct::negative_sign

```
string_type negative_sign() const;
```

The member function returns do_negative_sign().

moneypunct::pos_format

```
pattern pos_format() const;
```

The member function returns do_pos_format().

moneypunct::positive_sign

```
string_type positive_sign() const;
```

The member function returns do_positive_sign().

moneypunct::string_type

```
typedef basic_string<E> string_type;
```

The type describes a specialization of template class basic_string whose objects can store copies of the punctuation sequences.

moneypunct::thousands_sep

```
E thousands_sep() const;
```

The member function returns do_thousands_sep().

moneypunct_byname

```
template<class E, bool Intl>
    class moneypunct_byname : public moneypunct<E, Intl> {
public:
    explicit moneypunct_byname(const char *s, size_t refs = 0);
protected:
    ~moneypunct_byname();
    };
```

The template class describes an object that can serve as a locale facet of type moneypunct<E, Intl>. Its behavior is determined by the named locale s. The constructor initializes its base object with moneypunct<E, Intl>(refs).

num_get

```
template<class E, class InIt = istreambuf_iterator<E> >
    class num_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    explicit num_get(size_t refs = 0);
    iter_type get(iter_type first, iter_type last, ios_base& x,
        ios_base::iostate& st, long& val) const;
    iter_type get(iter_type first, iter_type last, ios_base& x,
        ios_base::iostate& st, unsigned long& val) const;
    iter_type get(iter_type first, iter_type last, ios_base& x,
        ios_base::iostate& st, double& val) const;
    iter_type get(iter_type first, iter_type last, ios_base& x,
        ios_base::iostate& st, long double& val) const;
    iter_type get(iter_type first, iter_type last, ios_base& x,
        ios_base::iostate& st, void *& val) const;
    iter_type get(iter_type first, iter_type last, ios_base& x,
        ios_base::iostate& st, bool& val) const;
    static locale::id id;
protected:
    ~num_get();
    virtual iter_type do_get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, long& val) const;
```

```
    virtual iter_type do_get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, unsigned long& val) const;
    virtual iter_type do_get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, double& val) const;
    virtual iter_type do_get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, long double& val) const;
    virtual iter_type do_get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, void *& val) const;
    virtual iter_type do_get(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, bool& val) const;
    };
```

The template class describes an object that can serve as a locale facet, to control conversions of sequences of type `E` to numeric values.

As with any locale facet, the static object **`id`** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in `id`.

num_get::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

num_get::do_get

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, long& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, unsigned long& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, double& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, long double& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, void *& val) const;
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, bool& val) const;
```

The first virtual protected member function endeavors to match sequential elements beginning at `first` in the sequence `[first, last)` until it has recognized a complete, nonempty **integer input field**. If successful, it converts this field to its equivalent value as type *long*, and stores the result in `val`. It returns an iterator designating the first element beyond the numeric input field. Otherwise, the function stores nothing in `val` and sets `ios_base::failbit` in `st`. It returns an iterator designating the first element beyond any prefix of a valid integer input field. In either case, if the return value equals `last`, the function sets `ios_base::eofbit` in `st`.

The integer input field is converted by the same rules used by the scan functions for matching and converting a series of *char* elements from a file. (Each such *char* element is assumed to map to an equivalent element of

type E by a simple, one-to-one, mapping.) The equivalent scan conversion specification is determined as follows:

- If x.flags() & ios_base::basefield == ios_base::oct, the conversion specification is lo.
- If x.flags() & ios_base::basefield == ios_base::hex, the conversion specification is lx.
- If x.flags() & ios_base::basefield == 0, the conversion specification is li.
- Otherwise, the conversion specification is ld.

The format of an integer input field is further determined by the locale facet fac returned by the call use_facet <numpunct<E>(x. getloc()). Specifically:

- fac.grouping() determines how digits are grouped to the left of any decimal point
- fac.thousands_sep() determines the sequence that separates groups of digits to the left of any decimal point

If no instances of fac.thousands_sep() occur in the numeric input field, no grouping constraint is imposed. Otherwise, any grouping constraints imposed by fac.grouping() is enforced and separators are removed before the scan conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, unsigned long& val) const;
```

behaves the same as the first, except that it replaces a conversion specification of ld with lu. If successful it converts the numeric input field to a value of type *unsigned long* and stores that value in val.

The third virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, double& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty **floating-point input field**. fac.decimal_point() determines the sequence that separates the integer digits from the fraction digits. The equivalent scan conversion specifier is lf.

The fourth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, long double& val) const;
```

behaves the same the third, except that the equivalent scan conversion specifier is Lf.

The fifth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, void *& val) const;
```

behaves the same the first, except that the equivalent scan conversion specifier is p.

The sixth virtual protected member function:

```
virtual iter_type do_get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, bool& val) const;
```

behaves the same as the first, except that it endeavors to match a complete, nonempty **boolean input field**. If successful it converts the boolean input field to a value of type bool and stores that value in val.

A boolean input field takes one of two forms. If x.flags() & ios_base::boolalpha is false, it is the same as an integer input field, except that the converted value must be either 0 (for false) or 1 (for true). Otherwise, the sequence must match either fac.falsename() (for false), or fac.truename() (for true).

in this implementation, if bool is not a distinct type, an argument val of type bool must be replaced by (_**Bool**&)val.

num_get::get

```
iter_type get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, long& val) const;
iter_type get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, unsigned long& val) const;
iter_type get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, double& val) const;
iter_type get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, long double& val) const;
iter_type get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, void *& val) const;
iter_type get(iter_type first, iter_type last, ios_base& x,
    ios_base::iostate& st, bool& val) const;
```

All member functions return do_get(first, last, x, st, val).

in this implementation, if bool is not a distinct type, an argument val of type bool must be replaced by (_Bool&)val.

num_get::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter InIt.

num_get::num_get

```
explicit num_get(size_t refs = 0);
```

The constructor initializes its base object with locale::facet(refs).

num_put

```
template<class E, class OutIt = ostreambuf_iterator<E> >
    class num_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    explicit num_put(size_t refs = 0);
    iter_type put(iter_type next, ios_base& x,
        E fill, long val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, unsigned long val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, double val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, long double val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, const void *val) const;
    iter_type put(iter_type next, ios_base& x,
        E fill, bool val) const;
    static locale::id id;
protected:
    ~num_put();
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, long val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, unsigned long val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, double val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, long double val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, const void *val) const;
    virtual iter_type do_put(iter_type next, ios_base& x,
        E fill, bool val) const;
    };
```

The template class describes an object that can serve as a locale facet, to control conversions of numeric values to sequences of type E.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in id.

num_put::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

num_put::do_put

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, double val) const;
virtual iter_type do_put(iter_type nextp ios_base& x,
    E fill, long double val) const;
virtual iter_type do_put(iter_type nextp ios_base& x,
    E fill, const void *val) const;
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

The first virtual protected member function generates sequential elements beginning at next to produce an **integer output field** from the value of val. The function returns an iterator designating the next place to insert an element beyond the generated integer output field.

The integer output field is generated by the same rules used by the print functions for generating a series of *char* elements to a file. (Each such *char* element is assumed to map to an equivalent element of type E by a simple, one-to-one, mapping.) Where a print function pads a field with either spaces or the digit 0, however, do_put instead uses fill. The equivalent print conversion specification is determined as follows:

- If x.flags() & ios_base::basefield == ios_base::oct, the conversion specification is lo.
- If x.flags() & ios_base::basefield == ios_base::hex, the conversion specification is lx.
- Otherwise, the conversion specification is ld.

If x.width() is nonzero, a field width of this value is prepended. The function then calls x.width(0) to reset the field width to zero.

Padding occurs only if the minimum number of elements N required to specify the output field is less than x.width(). Such padding consists of a sequence of N - width() copies of fill. Padding then occurs as follows:

- If x.flags() & ios_base::adjustfield == ios_base::left, the flag - is prepended. (Padding occurs after the generated text.)
- If x.flags() & ios_base::adjustfield == ios_base::internal, the flag 0 is prepended. (For a numeric output field, padding occurs where the print functions pad with 0.)
- Otherwise, no additional flag is prepended. (Padding occurs before the generated sequence.)

Finally:

- If x.flags() & ios_base::showpos is nonzero, the flag + is prepended to the conversion specification.
- If x.flags() & ios_base::showbase is nonzero, the flag # is prepended to the conversion specification.

The format of an integer output field is further determined by the locale facet fac returned by the call use_facet <numpunct<E>(x. getloc()). Specifically:

- fac.grouping() determines how digits are grouped to the left of any decimal point
- fac.thousands_sep() determines the sequence that separates groups of digits to the left of any decimal point

If no grouping constraints are imposed by fac.grouping() (its first element has the value CHAR_MAX) then no instances of fac.thousands_sep() are generated in the output field. Otherwise, separators are inserted after the print conversion occurs.

The second virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
```

behaves the same as the first, except that it replaces a conversion specification of ld with lu.

The third virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, double val) const;
```

behaves the same as the first, except that it produces a **floating-point output field** from the value of val. fac.decimal_point() determines the sequence that separates the integer digits from the fraction digits. The equivalent print conversion specifier is determined as follows:

- If x.flags() & ios_base::floatfield == ios_base::fixed, the conversion specification is lf.
- If x.flags() & ios_base::floatfield == ios_base::scientific, the conversion specification is le. If x.flags() & ios_base::uppercase is

nonzero, e is replaced with E.

- Otherwise, the conversion specification is lg. If x.flags() & ios_base::uppercase is nonzero, g is replaced with G.

If x.flags() & ios_base::fixed is nonzero, or if x.precision() is greater than zero, a precision with the value x.precision() is prepended to the conversion specification. Any padding behaves the same as for an integer output field. The padding character is fill. Finally:

- If x.flags() & ios_base::showpos is nonzero, the flag + is prepended to the conversion specification.
- If x.flags() & ios_base::showpoint is nonzero, the flag # is prepended to the conversion specification.

The fourth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, long double val) const;
```

behaves the same the third, except that the qualifier l in the conversion specification is replaced with L.

The fifth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, const void *val) const;
```

behaves the same the first, except that the conversion specification is p, plus any qualifier needed to specify padding.

The sixth virtual protected member function:

```
virtual iter_type do_put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

behaves the same as the first, except that it generates a **boolean output field** from val.

A boolean output field takes one of two forms. If x.flags() & ios_base::boolalpha is false, the generated sequence is either 0 (for false) or 1 (for true). Otherwise, the generated sequence is either fac.falsename() (for false), or fac.truename() (for true).

in this implementation, if bool is not a distinct type, an argument val of type bool must be replaced by (_Bool)val.

num_put::put

```
iter_type put(iter_type next, ios_base& x,
    E fill, long val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, unsigned long val) const;
iter_type put(iter_type iter_type next, ios_base& x,
```

```
    E fill, double val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, long double val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, const void *val) const;
iter_type put(iter_type next, ios_base& x,
    E fill, bool val) const;
```

All member functions return do_put(next, x, fill, val).

in this implementation, if bool is not a dastinct type, an argument val of type bool must be replaced by (_Bool)val.

num_put::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter OutIt.

num_put::num_put

```
explicit num_put(size_t refs = 0);
```

The constructor initializes its base object with locale::facet(refs).

numpunct

---

char_type · decimal_point · do_decimal_point · do_falsename · do_grouping · do_truename · do_thousands_sep · falsename · grouping · numpunct · string_type · thousands_sep · truename

---

```
template<class E, class numpunct : public locale::facet {
public:
    typedef E char_type;
    typedef basic_string<E> string_type;
    explicit numpunct(size_t refs = 0);
    E decimal_point() const;
    E thousands_sep() const;
    string grouping() const;
    string_type truename() const;
    string_type falsename() const;
    static locale::id id;
protected:
    ~numpunct();
    virtual E do_decimal_point() const;
```

```
    virtual E do_thousands_sep() const;
    virtual string do_grouping() const;
    virtual string_type do_truename() const;
    virtual string_type do_falsename() const;
    };
```

The template class describes an object that can serve as a locale facet, to desceibe the sequences of type E used to represent the input fields matched by num_get or the output fields generated by num_get.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in id.

numpunct::char_type

typedef E **char_type**;

The type is a synonym for the template parameter E.

numpunct::decimal_point

E **decimal_point**() const;

The member function returns do_decimal_point().

numpunct::do_decimal_point

E **do_decimal_point**() const;

The protected virtual member function returns a locale-specific element to use as a decimal-point.

numpunct::do_falsename

string_type **do_falsename**() const;

The protected virtual member function returns a locale-specific sequence to use as a text representation of the value false.

numpunct::do_grouping

string **do_grouping**() const;

The protected virtual member function returns a locale-specific rule for determining how digits are grouped to the left of any decimal point. The encoding is the same as for lconv::grouping.

numpunct::do_thousands_sep

E **do_thousands_sep**() const;

The protected virtual member function returns a locale-specific element to use as a group separator to the left of any decimal point.

numpunct::do_truename

string_type **do_truename**() const;

The protected virtual member function returns a locale-specific sequence to use as a text representation of the value true.

numpunct::falsename

string_type **falsename**() const;

The member function returns do_falsename().

numpunct::grouping

string **grouping**() const;

The member function returns do_grouping().

numpunct::numpunct

explicit **numpunct**(size_t refs = 0);

The constructor initializes its base object with locale::facet(refs).

numpunct::string_type

typedef basic_string<E> **string_type**;

The type describes a specialization of template class basic_string whose objects can store copies of the punctuation sequences.

numpunct::thousands_sep

E **thousands_sep**() const;

The mmmber function returns do_thousands_sep().

numpunct::truename

string_type **falsename**() const;

The member function returns do_truename().

numpunct_byname

```
template<class E>
    class numpunct_byname : public numpunct<E> {
public:
    explicit numpunct_byname(const char *s, size_t refs = 0);
protected:
    ~numpunct_byname();
    };
```

The template class describes an object that can serve as a locale facet of type numpunct<E>. Its behavior is determined by the named locale s. The constructor initializes its base object with numpunct<E>(refs).

time_base

```
class time_base {
public:
    enum dateorder {no_order, dmy, mdy, ymd, ydm};
    };
```

The class serves as a base class for facets of template class time_get. It defines just the enumerated type **dateorder** and several constants of this type. Each of the constants characterizes a different way to order the components of a date. The constants are:

- **no_order** specifies no particular order.
- **dmy** specifies the order day, month, then year, as in 2 December 1979.
- **mdy** specifies the order month, day, then year, as in December 2, 1979.
- **ymd** specifies the order year, month, then day, as in 1979/12/2.
- **ydm** specifies the order year, day, then month, as in 1979: 2 Dec.

time_get

```
template<class E, class InIt = istreambuf_iterator<E> >
    class time_get : public locale::facet {
public:
    typedef E char_type;
    typedef InIt iter_type;
    explicit time_get(size_t refs = 0);
    dateorder date_order() const;
    iter_type get_time(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_date(iter_type first, iter_type last,
```

```
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_weekday(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_month(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    iter_type get_year(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    static locale::id id;
protected:
    ~time_get();
    virtual dateorder do_date_order() const;
    virtual iter_type do_get_time(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type do_get_date(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type do_get_weekday(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type do_get_month(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    virtual iter_type do_get_year(iter_type first, iter_type last,
        ios_base& x, ios_base::iostate& st, tm *pt) const;
    };
```

The template class describes an object that can serve as a locale facet, to control conversions of sequences of type E to time values.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in id.

time_get::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

time_get::date_order

```
dateorder date_order() const;
```

The member function returns date_order().

time_get::do_date_order

```
virtual dateorder do_date_order() const;
```

The virtual protected member function returns a value of type time_base::dateorder, which describes the order in which date components

are matched by do_get_date. In this implementation, the value is time_base::mdy, corresponding to dates of the form December 2, 1979.

time_get::do_get_date

```
virtual iter_type do_get_date(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty **date input field**. If successful, it converts this field to its equivalent value as the components tm::tm_mon, tm::tm_day, and tm::tm_year, and stores the results in pt->tm_mon, pt->tm_day and pt->tm_year, respectively. It returns an iterator designating the first element beyond the date input field. Otherwise, the function sets ios_base::failbit in st. It returns an iterator designating the first element beyond any prefix of a valid date input field. In either case, if the return value equals last, the function sets ios_base::eofbit in st.

In this implementation, the date input field has the form MMM DD, YYYY, where:

- MMM is matched by calling get_month, giving the month.
- DD is a sequence of decimal digits whose corresponding numeric value must be in the range [1, 31], giving the day of the month.
- YYYY is matched by calling get_year, giving the year.
- The literal spaces and commas must match corresponding elements in the input sequence.

time_get::do_get_month

```
virtual iter_type do_get_month(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty **month input field**. If successful, it converts this field to its equivalent value as the component tm::tm_mon, and stores the result in pt->tm_mon. It returns an iterator designating the first element beyond the month input field. Otherwise, the function sets ios_base::failbit in st. It returns an iterator designating the first element beyond any prefix of a valid month input field. In either case, if the return value equals last, the function sets ios_base::eofbit in st.

The month input field is a sequence that matches the longest of a set of locale-specific sequences, such as: Jan, January, Feb, February, etc. The converted value is the number of months since January.

time_get::do_get_time

```
virtual iter_type do_get_time(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty **time input field**. If successful, it converts this field to its equivalent value as the components tm::tm_hour, tm::tm_min, and tm::tm_sec, and stores the results in pt->tm_hour, pt->tm_min and pt->tm_sec, respectively. It returns an iterator designating the first element beyond the time input field. Otherwise, the function sets ios_base::failbit in st. It returns an iterator designating the first element beyond any prefix of a valid time input field. In either case, if the return value equals last, the function sets ios_base::eofbit in st.

In this implementation, the time input field has the form HH:MM:SS, where:

- HH is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 24), giving the hour of the day.
- MM is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the minutes past the hour.
- SS is a sequence of decimal digits whose corresponding numeric value must be in the range [0, 60), giving the seconds past the minute.
- The literal colons must match corresponding elements in the input sequence.

time_get::do_get_weekday

```
virtual iter_type do_get_weekday(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty **weekday input field**. If successful, it converts this field to its equivalent value as the component tm::tm_wday, and stores the result in pt->tm_wday. It returns an iterator designating the first element beyond the weekday input field. Otherwise, the function sets ios_base::failbit in st. It returns an iterator designating the first element beyond any prefix of a valid weekday input field. In either case, if the return value equals last, the function sets ios_base::eofbit in st.

The weekday input field is a sequence that matches the longest of a set of locale-specific sequences, such as: Sun, Sunday, Mon, Monday, etc. The converted value is the number of days since Sunday.

time_get::do_get_year

```
virtual iter_type do_get_year(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The virtual protected member function endeavors to match sequential elements beginning at first in the sequence [first, last) until it has recognized a complete, nonempty **year input field**. If successful, it converts this field to its equivalent value as the component tm::tm_year, and stores the result in pt->tm_year. It returns an iterator designating the first element beyond the year input field. Otherwise, the function sets ios_base::failbit in st. It returns an iterator designating the first element beyond any prefix of a valid year input field. In either case, if the return value equals last, the function sets ios_base::eofbit in st.

The year input field is a sequence of decimal digits whose corresponding numeric value must be in the range [1900, 2036). The stored value is this value minus 1900. In this implementation, a numeric value in the range [0, 136) is also permissible. It is stored unchanged.

time_get::get_date

```
iter_type get_date(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_date(first, last, x, st, pt).

time_get::get_month

```
iter_type get_month(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_month(first, last, x, st, pt).

time_get::get_time

```
iter_type get_time(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_time(first, last, x, st, pt).

time_get::get_weekday

```
iter_type get_weekday(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_weekday(first, last, x, st, pt).

time_get::get_year

```
iter_type get_year(iter_type first, iter_type last,
    ios_base& x, ios_base::iostate& st, tm *pt) const;
```

The member function returns do_get_year(first, last, x, st, pt).

time_get::iter_type

```
typedef InIt iter_type;
```

The type is a synonym for the template parameter InIt.

time_get::time_get

```
explicit time_get(size_t refs = 0);
```

The constructor initializes its base object with locale::facet(refs).

time_get_byname

```
template<class E, class InIt>
    class time_get_byname : public time_get<E, InIt> {
public:
    explicit time_get_byname(const char *s, size_t refs = 0);
protected:
    ~time_get_byname();
    };
```

The template class describes an object that can serve as a locale facet of type time_get<E, InIt>. Its behavior is determined by the named locale s. The constructor initializes its base object with time_get<E, InIt>(refs).

time_put

```
template<class E, class OutIt = ostreambuf_iterator<E> >
    class time_put : public locale::facet {
public:
    typedef E char_type;
    typedef OutIt iter_type;
    explicit time_put(size_t refs = 0);
    iter_type put(iter_type next, ios_base& x,
        tm *pt, char fmt, char mod = 0) const;
    iter_type put(iter_type next, ios_base& x,
        tm *pt, const E *first, const E *last) const;
    static locale::id id;
protected:
```

```
    ~time_put();
    virtual iter_type do_put(iter_type next, ios_base& x,
        tm *pt, char fmt, char mod = 0) const;
    };
```

The template class describes an object that can serve as a locale facet, to control conversions of time values to sequences of type E.

As with any locale facet, the static object **id** has an initial stored value of zero. The first attempt to access its stored value stores a unique positive value in id.

time_put::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

time_put::do_put

```
virtual iter_type do_put(iter_type next, ios_base& x,
    tm *pt, char fmt, char mod = 0) const;
```

The virtual protected member function generates sequential elements beginning at next from time values stored in the object *pt, of type tm</CODE>. The function returns an iterator designating the next place to insert an element beyond the generated output.

The output is generated by the same rules used by strftime, with a last argument of pt, for generating a series of *char* elements into an array. (Each such *char* element is assumed to map to an equivalent element of type E by a simple, one-to-one, mapping.) If mod equals zero, the effective format is "%F", where F equals fmt. Otherwise, the effective format is "%MF", where M equals mod.

time_put::put

```
iter_type put(iter_type next, ios_base& x,
    tm *pt, char fmt, char mod = 0) const;
iter_type put(iter_type next, ios_base& x,
    tm *pt, const E *first, const E *last) const;
```

The first member function returns do_put(next, x, pt, fmt, mod). The second member function copies to *next++ any element in the interval [first, last) other than a percent (%). For a percent followed by a character C in the interval [first, last), the function instead evaluates next = do_put(next, x, pt, C, 0) and skips past C. If, however, C is a qualifier character from the set EOQ#, followed by a character C2 in the interval [first, last), the function instead evaluates next = do_put(next, x, pt, C2, C) and skips past C2.

time_put::iter_type

typedef InIt **iter_type**;

The type is a synonym for the template parameter OutIt.

time_put::time_put

explicit **time_put**(size_t refs = 0);

The constructor initializes its base object with locale::facet(refs).

time_put_byname

```
template<class E, class OutIt>
    class time_put_byname : public time_put<E, OutIt> {
public:
    explicit time_put_byname(const char *s, size_t refs = 0);
protected:
    ~time_put_byname();
    };
```

The template class describes an object that can serve as a locale facet of type time_put<E, OutIt>. Its behavior is determined by the named locale s. The constructor initializes its base object with time_put<E, OutIt>(refs).

tolower

```
template<class E>
    E tolower(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). tolower(c).

toupper

```
template<class E>
    E toupper(E c, const locale& loc) const;
```

The template function returns use_facet< ctype<E> >(loc). toupper(c).

use_facet

```
template<class Facet>
    const Facet& use_facet(const locale& loc) const;
```

The template function returns a reference to the locale facet of class

Facet listed within the locale object loc. If no such object is listed, the function throws an object of class bad_cast.

In this implementation, you should write **_USE(loc, Facet)**, or **_USEFAC(loc, Facet)**. in place of use_facet<Facet>(loc), which not all translators currently support. The former is strongly preferred when looking up a facet that should always be present -- it generates the requested facet on demand, if necessary. The latter will report that the locale initially constructed by locale() has no facets.

See also the **Table of Contents** and the **Index**.

<map>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class Key, class T, class Pred, class A>
    class map;
template<class Key, class T, class Pred, class A>
    class multimap;
//    TEMPLATE FUNCTIONS
template<class Key, class T, class Pred, class A>
    bool operator==(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator==(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
```

```
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        const map<Key, T, Pred, A>& lhs,
        const map<Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        const multimap<Key, T, Pred, A>& lhs,
        const multimap<Key, T, Pred, A>& rhs);
    };
```

Include the STL standard header **<map>** to define the container template classes `map` and `multimap`, and their supporting templates.

map

---

allocator_type · begin · clear · const_iterator · const_reference · const_reverse_iterator · count · difference_type · empty · end · equal_range · erase · find · get_allocator · insert · iterator · key_comp · key_compare · key_type · lower_bound · map · max_size · operator[] · rbegin · reference · referent_type · rend · reverse_iterator · size · size_type · swap · upper_bound · value_comp · value_compare · value_type

---

```
template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
    class map {
public:
    typedef Key key_type;
    typedef T referent_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::rebind<value_type>::other::reference reference;
```

```
typedef A::rebind<value_type>::other::const_reference const_reference;
typedef T0 iterator;
typedef T1 const_iterator;
typedef reverse_bidirectional_iterator<iterator,
    value_type, reference, A::pointer,
        difference_type> reverse_iterator;
typedef reverse_bidirectional_iterator<const_iterator,
    value_type, const_reference, A::const_pointer,
        difference_type> const_reverse_iterator;
explicit map(const Pred& comp = Pred(), const A& al = A());
map(const map& x);
template<class InIt>
    map(InIt first, InIt last, const Pred& comp = Pred(),
        const A& al = A());
iterator begin();
const_iterator begin() const;
iterator end();
iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
A::reference operator[](const Key& key);
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
void clear();
void swap(map x);
key_compare key_comp() const;
value_compare value_comp() const;
iterator find(const Key& key);
const_iterator find(const Key& key) const;
size_type count(const Key& key) const;
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

```
    pair<iterator, iterator> equal_range(const Key& key);
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type pair<const Key, T>**. The first element of each pair is the **sort key** and the second is its associated **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total order on sort keys. For any element `x` that precedes `y` in the sequence, `key_comp()(y.first, x.first)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `multimap`, an object of template class `map` ensures that `key_comp()(x.first, y.first)` is true. (Each key is unique.)

The object allocates and frees storage for the sequence it controls through a protected object named **allocator**, of **class A**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

map::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

map::begin

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

map::clear

```
void clear() const;
```

The member function calls `erase( begin(), end())`.

map::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T1`.

map::const_reference

```
typedef A::rebind<value_type>::other::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

map::const_reverse_iterator

```
typedef reverse_bidirectional_iterator<const_iterator,
    value_type, const_reference, A::const_pointer,
        difference_type> const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

map::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_bound(key))`.

map::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

map::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

map::end

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

map::equal_range

```
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

map::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the interval `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

map::find

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key equals `key`. If no such element exists, the iterator equals `end()`.

map::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

map::insert

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function determines whether an element `y` exists in the sequence whose key matches that of `x`. (The keys match if `!key_comp()(x. first, y.first) && !key_comp()(y.first, x.first)`.) If not, it creates such an element `y` and initializes it with `x`. The function then determines the iterator `it` that designates `y`. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values in the range `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(const value_type *first, const value_type *last);
```

map::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

map::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object detines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

map::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of any two elements in the controlled sequence.

map::key_type

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

map::lower_bound

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x. first, key)` is false.

If no such element exists, the function returns `end()`.

map::map

```
explicit map(const Pred& comp = Pred(), const A& al = A());
map(const map& x);
template<class InIt>
    map(InIt first, InIt last, const Pred& comp = Pred(),
        const A& al = A());
```

The constructors with an argument named `comp` store the function object so that it can be later returned by calling `key_comp()`. All constructors also store the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence. The first constructor specifies an

empty initial controlled sequence. The second constructor specifies a copy of the sequence controlled by `x`. The member template constructor specifies the sequence of element values `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
map(const value_type *first, const value_type *last,
    const Pred& comp = Pred(), const A& al = A());
```

map::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

map::operator[]

```
A::reference operator[](const Key& key);
```

The member function determines the iterator `it` as the return value of `insert( value_type(key, T())`. (It inserts an element with the specified key if no such element exists.) It then returns a reference to `(*it).second`.

map::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

map::reference

```
typedef A::rebind<value_type>::other::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

map::referent_type

```
typedef T referent_type;
```

The type is a synonym for the template parameter `T`.

map::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

map::reverse_iterator

```
typedef reverse_bidirectional_iterator<iterator,
    value_type, reference, A::pointer,
        difference_type> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

map::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

map::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

map::swap

```
void swap(map& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

map::upper_bound

```
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x.first)` is true.

If no such element exists, the function returns `end()`.

map::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

map::value_compare

```
class value_compare
    : public binary_function<value_type, value_type, bool> {
public:
    bool operator()(const value_type& x, const value_type& y)
        {return (comp(x.first, x.second)); }
protected:
```

```
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
    };
```

The type describes a function object that can compare the sort keys in two elements to determine their relative order in the controlled sequence. The function object stores an object **comp** of type key_type. The member function **operator()** uses this object to compare the sort-key components of two element.

map::value_type

```
typedef pair<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

multimap

---

allocator_type · begin · clear · const_iterator · const_reference · const_reverse_iterator · count · difference_type · empty · end · equal_range · erase · find · get_allocator · insert · iterator · key_comp · key_compare · key_type · lower_bound · max_size · multimap · rbegin · reference · referent_type · rend · reverse_iterator · size · size_type · swap · upper_bound · value_comp · value_compare · value_type

---

```
template<class Key, class T, class Pred = less<Key>, class A = allocator<T> >
    class multimap {
public:
    typedef Key key_type;
    typedef T referent_type;
    typedef Pred key_compare;
    typedef A allocator_type;
    typedef pair<const Key, T> value_type;
    class value_compare;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::rebind<value_type>::other::reference reference;
    typedef A::rebind<value_type>::other::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef reverse_bidirectional_iterator<iterator,
        value_type, reference, A::pointer,
            difference_type> reverse_iterator;
    typedef reverse_bidirectional_iterator<const_iterator,
        value_type, const_reference, A::const_pointer,
            difference_type> const_reverse_iterator;
    explicit multimap(const Pred& comp = Pred(), const A& al = A());
```

```
    multimap(const multimap& x);
    template<class InIt>
        multimap(InIt first, InIt last, const Pred& comp = Pred(),
            const A& al = A());
    iterator begin();
    const_iterator begin() const;
    iterator end();
    iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    iterator insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(multimap x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    iterator find(const Key& key);
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    iterator lower_bound(const Key& key);
    const_iterator lower_bound(const Key& key) const;
    iterator upper_bound(const Key& key);
    const_iterator upper_bound(const Key& key) const;
    pair<iterator, iterator> equal_range(const Key& key);
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type** **pair<const Key, T>**. The first element of each pair is the **sort key** and the second is its associated **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point

at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total order on sort keys. For any element `x` that precedes `y` in the sequence, `key_comp()(y.first, x.first)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `map`, an object of template class `multimap` does not ensure that `key_comp()(x.first, y.first)` is true. (Keys need not be unique.)

The object allocates and frees storage for the sequence it controls through a protected object named **`allocator`**, of **class `A`**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

multimap::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

multimap::begin

```
const_iterator begin() const;
iterator begin();
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

multimap::clear

```
void clear() const;
```

The member function calls `erase( begin(), end())`.

multimap::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T1`.

multimap::const_reference

```
typedef A::rebind<value_type>::other::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

multimap::const_reverse_iterator

```
typedef reverse_bidirectional_iterator<const_iterator,
    value_type, const_reference, A::const_pointer,
        difference_type> const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

multimap::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_bound(key))`.

multimap::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

multimap::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

multimap::end

```
const_iterator end() const;
iterator end();
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

multimap::equal_range

```
pair<iterator, iterator> equal_range(const Key& key);
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

multimap::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

multimap::find

```
iterator find(const Key& key);
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key equals `key`. If no such element exists, the iterator equals `end()`.

multimap::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

multimap::insert

```
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function inserts the element `x` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values in the range `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(const value_type *first, const value_type *last);
```

multimap::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

multimap::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

multimap::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of any two elements in the controlled sequence.

multimap::key_type

```
typedef Key key_type;
```

The type describes the sort key object stored in each element of the controlled sequence.

multimap::lower_bound

```
iterator lower_bound(const Key& key);
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x. first, key)` is false.

If no such element exists, the function returns `end()`.

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

multimap::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

multimap::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

multimap::multimap

```
explicit multimap(const Pred& comp = Pred(), const A& al = A());
multimap(const multimap& x);
template<class InIt>
    multimap(InIt first, InIt last, const Pred& comp = Pred(),
        const A& al = A());
```

The constructors with an argument named `comp` store the function object so that it can be later returned by calling `key_comp()`. All constructors also store the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence. The first constructor specifies an empty initial controlled sequence. The second constructor specifies a copy of the sequence controlled by `x`. The

member template constructor specifies the sequence of element values `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
multimap(const value_type *first, const value_type *last,
    const Pred& comp = Pred(), const A& al = A());
```

multimap::reference

```
typedef A::rebind<value_type>::other::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

multimap::referent_type

```
typedef T referent_type;
```

The type is a synonym for the template parameter `T`.

multimap::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

multimap::reverse_iterator

```
typedef reverse_bidirectional_iterator<iterator,
    value_type, reference, A::pointer,
        difference_type> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

multimap::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

multimap::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

multimap::swap

```
void swap(multimap& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator ==`

`str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

multimap::upper_bound

```
iterator upper_bound(const Key& key);
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x.first)` is true.

If no such element exists, the function returns `end()`.

multimap::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

multimap::value_compare

```
class value_compare
    : public binary_function<value_type, value_type, bool> {
public:
    bool operator()(const value_type& x, const value_type& y)
        {return (comp(x.first, x.second)); }
protected:
    value_compare(key_compare pr)
        : comp(pr) {}
    key_compare comp;
    };
```

The type describes a function object that can compare the sort keys in two elements to determine their relative order in the controlled sequence. The function object stores an object **`comp`** of type `key_type`. The member function **`operator()`** uses this object to compare the sort-key components of two element.

multimap::value_type

```
typedef pair<const Key, T> value_type;
```

The type describes an element of the controlled sequence.

operator!=

```
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap <Key, T, Pred, A>& lhs,
```

```
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class Key, class T, class Pred, class A>
    bool operator==(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator==(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The first template function overloads `operator==` to compare two objects of template class `multimap`. The second template function overloads `operator==` to compare two objects of template class `multimap`. Both functions return `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class Key, class T, class Pred, class A>
    bool operator<(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The first template function overloads `operator<` to compare two objects of template class `multimap`. The second template function overloads `operator<` to compare two objects of template class `multimap`. Both functions return `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator<=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class Key, class T, class Pred, class A>
    bool operator>(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator>(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class Key, class T, class Pred, class A>
    bool operator>=(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    bool operator!=(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class Key, class T, class Pred, class A>
    void swap(
        const map <Key, T, Pred, A>& lhs,
        const map <Key, T, Pred, A>& rhs);
template<class Key, class T, class Pred, class A>
    void swap(
        const multimap <Key, T, Pred, A>& lhs,
        const multimap <Key, T, Pred, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

See also the **Table of Contents** and the **Index**.

<memory>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
//  TEMPLATE CLASSES
template<class T>
    class allocator;
class allocator<void>;
template<class FwdIt, class T>
    class raw_storage_iterator;
template<class T>
    class auto_ptr;
//  TEMPLATE OPERATORS
template<class T>
    bool operator==(allocator<T>& lhs,
        allocator<T>& rhs);
template<class T>
    bool operator!=(allocator<T>& lhs,
        allocator<T>& rhs);
template<class T>
    void operator delete(void *p, size_t n, allocator& al);
template<class T>
    void operator delete[](void *p, size_t n, allocator& al);
template<class T>
    void *operator new(size_t n, allocator& al);
template<class T>
    void *operator new[](size_t n, allocator& al);
//  TEMPLATE FUNCTIONS
template<class T>
    pair<T *, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template<class T>
    void return_temporary_buffer(T *p);
template<class InIt, class FwdIt>
    FwdIt uninitialized_copy(InIt first, InIt last, FwdIt result);
template<class FwdIt, class T>
    void uninitialized_fill(FwdIt first, FwdIt last, const T& x);
template<class FwdIt, class Size, class T>
    void uninitialized_fill_n(FwdIt first, Size n, const T& x);
    };
```

Include the STL standard header **<memory>** to define a class, an operator, and several templates that help allocate and free objects.

allocator

```
template<class T>
    class allocator {
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T *pointer;
    typedef const T *const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;
    pointer address(reference x) const;
    const_pointer address(const_reference x) const;
    template<class U>
        struct rebind;
    allocator();
    template<class U>
        allocator(const allocator<U>);
    template<class U>
        operator=(const allocator<U>);
    template<class U>
        pointer allocate(size_type n, const U *hint);
    void deallocate(pointer p, size_type n);
    void construct(pointer p, const T& val);
    void destroy(pointer p);
    size_type max_size() const;
    };
```

The template class describes an object that manages storage allocation and freeing for arrays of objects of type `T`. An object of class `allocator` is the default **allocator object** specified in the constructors for several container template classes in the Standard C++ library.

Template class `allocator` supplies several type definitions that are rather pedestrian. They hardly seem worth defining. But another class with the same members might choose more interesting alternatives. Constructing a container with an allocator object of such a class gives individual control over allocation and freeing of elements controlled by that container.

For example, an allocator object might allocate storage on a **private heap**. Or it might allocate storage on a **far heap**, requiring nonstandard pointers to access the allocated objects. Or it might specify, through the type definitions it supplies, that elements be accessed through special **accessor objects** that manage **shared memory**, or perform automatic **garbage collection**. Hence, a class that allocates storage using an

allocator object should use these types religiously for declaring pointer and reference objects (as do the containers in the Standard C++ library).

Thus, an allocator defines the types (among others):

- `pointer` -- behaves like a pointer to `T`
- `const_pointer` -- behaves like a const pointer to `T`
- `reference` -- behaves like a reference to `T`
- `const_reference` -- behaves like a const reference to `T`

These types specify the form that pointers and references must take for allocated elements. (`allocator::types<T>::pointer` is not necessarily the same as `T *` for all allocator objects, even though it has this obvious definition for class `allocator`.)

allocator::address

```
pointer address(reference x) const;
const_pointer address(const_reference x) const;
```

The member functions return the address of `x`, in the form that pointers must take for allocated elements.

allocator::allocate

```
template<class U>
    pointer allocate(size_type n, const U *hint);
```

The member template function allocates storage for an array of `n` elements of type `T`, by calling `operator new(n)`. It returns a pointer to the allocated object. The `hint` argument helps some allocators in improving locality of reference -- a valid choice is the address of an object earlier allocated by the same allocator object, and not yet deallocated. To supply no hint, use a null pointer argument instead.

allocator::allocator

```
allocator();
template<class U>
    allocator(const allocator<U>);
```

The constructors do nothing. In general, however, an allocator object constructed from another allocator object should compare equal to it (and hence permit intermixing of object allocation and freeing between the two allocator objects).

In this implementation, if a translator does not support member template functions, the template constructor is replaced by:

```
allocator(const allocator<T>);
```

allocator::const_pointer

```
typedef const T *pointer;
```

The pointer type describes an object `p` that can designate, via the expression `*p`, any const object that an object of template class `allocator` can allocate.

allocator::const_reference

```
typedef const T& const_reference;
```

The reference type describes an object `x` that can designate any const object that an object of template class `allocator` can allocate.

allocator::construct

```
void construct(pointer p, const T& val);
```

The member function constructs an object of type `T` at `p` by evaluating the placement `new` expression `new ((void *)p) T(val)`.

allocator::deallocate

```
void deallocate(pointer p, size_type n);
```

The member function frees storage for the array of `n` objects of type `T` beginning at `p`, by calling `operator delete(p)`. The pointer `p` must have been earlier returned by a call to `allocate` for an allocator object that compares equal to `*this`, allocating an array object of the same size and type.

allocator::destroy

```
void destroy(pointer p);
```

The member function destroys the object designated by `p`, by calling `p->T::~T()`.

allocator::difference_type

```
typedef ptrdiff_t difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in a sequence that an object of template class `allocator` can allocate.

allocator::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence of elements of type `T` that an object of class `allocator` *might* be able to allocate.

allocator::operator=

```
template<class U>
    allocator<T> operator=(const allocator<U>);
```

The template assignment operator does nothing. In general, however, an allocator object assigned to another allocator object should compare equal to it (and hence permit intermixing of object allocation and freeing between the two allocator objects).

In this implementation, if a translator does not support member template functions, the template assignment operator is replaced by:

```
allocator<T> operator=(const allocator<U>);
```

allocator::pointer

```
typedef T *pointer;
```

The pointer type describes an object `p` that can designate, via the expression `*p`, any object that an object of template class `allocator` can allocate.

allocator::rebind

```
template<class U>
    struct rebind {
    typedef allocator<U> other;
    };
```

The member template class defines the type **`other`**. Its sole purpose is to provide the type name `allocator<U>` given the type name `allocator<U>`.

For example, given al allocator object `al` of type `A`, you can allocate an object of type `U` with the expression:

```
A::rebind<U>::other(al).allocate(1, 0)
```

Or, you can simply name its pointer type by writing the type:

```
A::rebind<U>::other::pointer
```

In this implementation, if a translator does not support member template functions, how you write an allocator is constrained. A container may need to allocate and free objects other than type `T`, but cannot use the `rebind` mechanism to derive a suitable allocator object. Thus, you cannot write an allocator that uses any pointer or reference types that differ from those used by `allocator`, and you must supply the member function:

```
char *_Charalloc(size_type n);
```

which allocates an object of size `n` bytes and returns a pointer to its first byte.

allocator::reference

```
typedef T& reference;
```

The reference type describes an object `x` that can designate any object that an object of template class `allocator` can allocate.

allocator::size_type

```
typedef size_t size_type;
```

The unsigned integer type describes an object that can represent the length of any sequence that an object of template class `allocator` can allocate.

allocator::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

allocator<void>

```
class allocator<void> {
    typedef void *pointer;
    typedef const void *const_pointer;
    typedef void value_type;
    template<class U>
        struct rebind;
    allocator();
    template<class U>
        allocator(const allocator<U>);
    template<class U>
        operator=(const allocator<U>);
    };
```

The class explicitly specializes template class allocator for type *void*. It defines only the types `const_pointer`, `pointer`, `value_type`, and the nested template class `rebind`.

auto_ptr

```
template<class T>
    class auto_ptr {
public:
    typedef T element_type;
    explicit auto_ptr(T *p = 0) throw();
```

```
    template<class U>
        auto_ptr(const auto_ptr<U>& rhs) throw();
    template<class U>
        auto_ptr<T>& operator=(auto_ptr<U>& rhs) throw();
    ~auto_ptr();
    T& operator*() const throw();
    T *operator->() const throw();
    T *get() const throw();
    T *release() const throw();
    };
```

The class describes an object that stores a pointer to an allocated object of type `T`. The stored pointer must either be null or designate an object allocated by a `new` expression. The object also stores an **ownership indicator**. An object constructed with a non-null pointer owns the pointer. It transfers ownership if its stored value is assigned to another object. The destructor for `auto_ptr<T>` deletes the allocated object if it owns it. Hence, an object of class `auto_ptr<T>` ensures that an allocated object is automatically deleted when control leaves a block, even via a thrown excepiton.

auto_ptr::auto_ptr

```
explicit auto_ptr(T *p = 0) throw();
template<class U>
    auto_ptr(const auto_ptr(auto_ptr<U>& rhs) throw();
```

The first constructor stores `p` as the pointer to the allocated object. It stores true as the ownership indicator only if `p != 0`. The second (template) constructor transfers ownership of the pointer stored in `rhs`, by storing both the pointer value and the ownership indicator from `rhs` in the constructed object. It effectively releases the pointer by calling `rhs.release()`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
auto_ptr(const auto_ptr(auto_ptr<T>& rhs);
```

auto_ptr::~auto_ptr

```
~auto_ptr();
```

If the ownership indicator is true, the destructor deletes the object designated by the stored pointer `p` by evaluating the `delete` expression `delete p`.

auto_ptr::element_type

```
typedef T element_type;
```

The type is a synonym for the template parameter `T`.

auto_ptr::get

```
T *get() const throw();
```

The member function returns the stored pointer.

auto_ptr::operator=

```
template<class U>
    auto_ptr<T>& operator=(auto_ptr<U>& rhs) throw();
```

The template assignment operator deletes any pointer `p` that it owns, by evaluating the `delete` expression `delete p`. It then transfers ownership of the pointer stored in `rhs`, by storing both the pointer value and the ownership indicator from `rhs` in `*this`. It effectively releases the pointer by calling `rhs.release()`. The function returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
auto_ptr<T>& operator=(auto_ptr<T>& rhs);
```

auto_ptr::operator*

```
T& operator*() const throw();
```

The indirection operator effectively returns `*get()`. Hence, the stored pointer must not be null.

auto_ptr::operator->

```
T *operator->() const throw();
```

The selection operator effectively returns `get()`, so that the expression `al->m` behaves the same as `(al.get())->m`, where `al` is an object of class `auto_ptr<T>`. Hence, the stored pointer must not be null, and `T` must be a class, structure, or union type.

auto_ptr::release

```
T *release() throw();
```

The member function sets the ownership indicator to false, then returns the stored pointer.

get_temporary_buffer

```
template<class T>
    pair<T *, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
```

The template function allocates storage for a sequence of at most `n` elements of type `T`, from an unspecified source (which may well be the standard heap used by `operator new`). It returns a value `pr`, of type `pair<T *, ptrdiff_t>`. If the function allocates storage, `pr.first` designates the allocated storage and `pr.second` is the number of elements in the longest sequence the storage can hold. Otherwise, `pr.first` is a null pointer.

In this implementation, you should write **`get_temporary_buffer(n, (T *)0)`** in place of `get_temporary_buffer<T>(n)`, which not all translators currently support.

operator!=

```
template<class T>
    bool operator!=(allocator<T>& lhs,
        allocator<T>& rhs);
```

The template operator returns false.

operator==

```
template<class T>
    bool operator==(allocator<T>& lhs,
        allocator<T>& rhs);
```

The template operator returns true. (Two allocator objects should compare equal only if an object allocated through one can be deallocated through the other. If the value of one object is determined from another by assignment or by construction, the two object should compare equal.)

operator delete

```
template<class T>
    void operator delete(void *p, size_t n, allocator<T>& al);
```

The template operator function lets you write a placement `delete` expression that deallocates storage under control of the allocator object `al`, as in `delete (n, al) p`. The function effectively calls `al.deallocate(p, n)`.

operator delete[]

```
template<class T>
    void operator delete[](void *p, size_t n, allocator<T>& al);
```

The template operator function lets you write a placement `delete[]` expression that deallocates storage under control of the allocator object `al`, as in `delete[] (n, al) p`. The function effectively calls `al.deallocate(p, n)`.

operator new

```
template<class T>
    void *operator new(size_t n, allocator<T>& al);
```

The template operator function lets you write a placement `new` expression that allocates storage under control of the allocator object `al`, as in `new(al) U` to allocate and construct a new object of type `U`. The function effectively returns `allocator<T>::rebind<char>::other(al).allocate(n, 0)`.

operator new[]

```
template<class T>
    void *operator new[](size_t n, allocator<T>& al);
```

The template operator function lets you write a placement `new[]` expression that allocates storage under control of the allocator object `al`, as in `new(al) T[N]` to allocate and construct a new object of type `T`. The function effectively returns `operator new(n, al)`.

raw_storage_iterator

```
template<class FwdIt, class T>
    class raw_storage_iterator
          : public iterator<output_iterator_tag, void, void> {
public:
    typedef FwdIt iterator_type;
    typedef T element_type;
    explicit raw_storage_iterator(FwdIt it);
    raw_storage_iterator<FwdIt, T>& operator*();
    raw_storage_iterator<FwdIt, T>& operator=(const T& val);
    raw_storage_iterator<FwdIt, T>& operator++();
    raw_storage_iterator<FwdIt, T> operator++(int);
    };
```

The class describes an output iterator that constructs objects of type `T` in the sequence it generates. An object of class `raw_storage_iterator<FwdIt, T>` accesses storage through a forward iterator object, of class `FwdIt`, that you specify when you construct the object. For an object `it` of class `FwdIt`, the expression `&*it` must designate unconstructed storage for the next object (of type `T`) in the generated sequence.

raw_storage_iterator::element_type

```
typedef T element_type;
```

The type is a synonym for the template parameter `T`.

raw_storage_iterator::iterator_type

```
typedef FwdIt iterator_type;
```

The type is a synonym for the template parameter `FwdIt`.

raw_storage_iterator::operator*

```
raw_storage_iterator<FwdIt, T>& operator*();
```

The indirection operator returns `*this` (so that `operator=(const T&)` can perform the actual store in an expression such as `*x = val`).

raw_storage_iterator::operator=

```
raw_storage_iterator<FwdIt, T>& operator=(const T& val);
```

The assignment operator constructs the next object in the output sequence using the stored iterator value `it`, by evaluating the placement `new` expression `new ((void *)&*it) T(val)`. The function returns `*this`.

raw_storage_iterator::operator++

```
raw_storage_iterator<FwdIt, T>& operator++();
raw_storage_iterator<FwdIt, T> operator++(int);
```

The first (preincrement) operator increments the stored output iterator object, then returns `*this`.

The second (postincrement) operator makes a copy of `*this`, increments the stored output iterator object, then returns the copy.

raw_storage_iterator::raw_storage_iterator

```
explicit raw_storage_iterator(FwdIt it);
```

The constructor stores `it` as the output iterator object.

return_temporary_buffer

```
template<class T>
    void return_temporary_buffer(T *p);
```

The template function frees the storage designated by `p`, which must be earlier allocated by a call to `get_temporary_buffer`.

uninitialized_copy

```
template<class InIt, class FwdIt>
    FwdIt uninitialized_copy(InIt first, InIt last, FwdIt result);
```

The template function effectively executes:

```
while (first != last)
    new ((void *)&*result++) T(*first++);
```

where `T` is the type of `*first`.

uninitialized fill

```
template<class FwdIt, class T>
    void uninitialized_fill(FwdIt first, FwdIt last, const T& x);
```

The template function effectively executes:

```
while (first != last)
    new ((void *)&*first++) T(x);
```

uninitialized_fill_n

```
template<class FwdIt, class Size, class T>
    void uninitialized_fill_n(FwdIt first, Size n, const T& x);
```

The template function effectively executes:

```
while (0 < n--)
    new ((void *)&*first++) T(x);
```

See also the **Table of Contents** and the **Index**.

<new>

```
namespace std {
    typedef void (*new_handler)();
    class bad_alloc;
    class nothrow_t;
    extern const nothrow_t nothrow;
        // FUNCTIONS
    new_handler set_new_handler(new_handler ph) throw();
    void operator delete(void *p) throw();
    void operator delete(void *, void *) throw();
    void operator delete(void *p, const nothrow_t&) throw();
    void operator delete[](void *p) throw();
    void operator delete[](void *, void *) throw();
    void operator delete[](void *p, const nothrow_t&) throw();
    void *operator new(size_t n) throw(bad_alloc);
    void *operator new(size_t n, const nothrow_t&) throw();
    void *operator new(size_t n, void *p) throw();
    void *operator new[](size_t n) throw(bad_alloc);
    void *operator new[](size_t n, const nothrow_t&) throw();
    void *operator new[](size_t n, void *p) throw();
    };
```

Include the standard header **<new>** to define several types and functions that control allocation and freeing of storage under program control.

Some of the functions declared in this header are **replaceable**. The implementation supplies a default version, whose behavior is described in this document. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements described in this document.

bad_alloc

```
class bad_alloc : public exception {
    };
```

The class describes an exception thrown to indicate that an allocation request did not succeed. The value returned by what() is implementation-defined. None of the member functions throw any exceptions.

new_handler

```
typedef void (*new_handler)();
```

The type points to a function suitable for use as a new handler.

nothrow

```
extern const nothrow_t nothrow;
```

The object is used as a function argument to match the parameter type `nothrow_t`.

nothrow_t

```
class nothrow_t {};
```

The class is used as a function parameter to indicate that the function should never throw an exception.

operator delete

```
void operator delete(void *p) throw();
void operator delete(void *, void *);
void operator delete(void *p, const nothrow_t&) throw();
```

The first function is called by a **`delete` expression** to render the value of `p` invalid. The program can define a function with this function signature that replaces the default version defined by the Standard C++ library. The required behavior is to accept a value of `p` that is null or that was returned by an earlier call to `operator new(size_t)`.

The default behavior for a null value of `p` is to do nothing. Any other value of `p` must be a value returned earlier by a call as described above. The default behavior for such a non-null value of `p` is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)`, or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a **placement `delete` expression** corresponding to a `new` expression of the form `new(void *)`. It does nothing.

The third function is called by a placement `delete` expression corresponding to a `new` expression of the form `new(void *, const nothrow_t&)`. It calls `delete(p)`.

operator delete[]

```
void operator delete[](void *p) throw();
void operator delete[](void *, void *);
```

The first function is called by a **`delete[]` expression** to render the value of `p` invalid. The program can define a function with this function signature that replaces the default version defined by the Standard C++ library.

The required behavior is to accept a value of `p` that is null or that was returned by an earlier call to `operator new[](size_t)`.

The default behavior for a null value of `p` is to do nothing. Any other value of `ptr` must be a value returned earlier by a call as described above. The default behavior for such a non-null value of `p` is to reclaim storage allocated by the earlier call. It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to `operator new(size_t)`, or to any of `calloc(size_t)`, `malloc(size_t)`, or `realloc(void*, size_t)`.

The second function is called by a **placement `delete[]` expression** corresponding to a `new[]` expression of the form `new[](void *)`. It does nothing.

The third function is called by a placement `delete` expression corresponding to a `new[]` expression of the form `new[](void *, const nothrow_t&)`. It calls `delete[](p)`.

operator new

```
void *operator new(size_t n) throw(bad_alloc);
void *operator new(size_t n, const nothrow_t&) throw();
void *operator new(size_t n, void *p);
```

The first function is called by a **`new` expression** to allocate `n` bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ library.

The required behavior is to return a non-null pointer only if storage can be allocated as requested. Each such allocation yields a pointer to storage disjoint from any other allocated storage. The order and contiguity of storage allocated by successive calls is unspecified. The initial stored value is unspecified. The returned pointer points to the start (lowest byte address) of the allocated storage. If `n` is zero, the value returned does not compare equal to any other value returned by the function.

The default behavior is to execute a loop. Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to `malloc(size_t)` is unspecified. If the attempt is successful, the function returns a pointer to the allocated storage. Otherwise if the stored new handler pointer is null, the result is implementation-defined. Otherwise, the function calls the designated

new handler. If the called function returns, the loop repeats. The loop terminates when an attempt to allocate the requested storage is successful or when a called function does not return.

The required behavior of a **new handler** is to perform one of the following operations:

- make more storage available for allocation and then return
- throw an object of type `bad_alloc`
- call either `abort()` or `exit(int)`

The default behavior of a new handler is to throw an object of type `bad_alloc`

The order and contiguity of storage allocated by successive calls to `operator new(size_t)` is unspecified, as are the initial values stored there.

The second function:

```
void *operator new(size_t n, const nothrow_t&) throw();
```

is called by a placement `new` expression to allocate `n` bytes of storage suitably aligned to represent any object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ library.

The default behavior is to return `operator new(n)` if that function succeeds. Otherwise, it returns a null pointer.

The third function:

```
void *operator new(size_t n, void *p);
```

is called by a **placement `new` expression**, of the form `new (args) T`. Here, *`args`* consists of a single object pointer. The function returns `p`.

`operator new[]`

```
void *operator new[](size_t n) throw(bad_alloc);
void *operator new[](size_t n, const nothrow_t&) throw();
void *operator new[](size_t n, void *p);
```

The first function is called by a **`new[]` expression** to allocate `n` bytes of storage suitably aligned to represent any array object of that size or smaller. The program can define a function with this function signature that replaces the default version defined by the Standard C++ library.

The required behavior is the same as for `operator new(size_t)`. The default behavior is to return `operator new(n)`.

The second function is called by a placement `new[]` expression to allocate `n` bytes of storage suitably aligned to represent any array object of that size. The program can define a function with this function signature that replaces the default version defined by the Standard C++ library.

The default behavior is to return `operator new(n)` if that function succeeds. Otherwise, it returns a null pointer.

The third function is called by a **placement `new[]` expression**, of the form `new (args) T[N]`. Here, `args` consists of a single object pointer. The function returns `p`.

`set_new_handler`

```
new_handler set_new_handler(new_handler ph) throw();
```

The function stores `ph` in a static new handler pointer that it maintains, then returns the value previously stored in the pointer. The new handler is used by `operator new(size_t)`.

See also the **Table of Contents** and the **Index**.

<numeric>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);
template<class InIt1, class InIt2, class T>
    T product(InIt1 first1, InIt1 last1,
        Init2 first2, T val);
template<class InIt1, class InIt2, class T,
    class Pred1, class Pred2>
    T product(InIt1 first1, InIt1 last1,
        Init2 first2, T val, Pred1 pr1, Pred2 pr2);
template<class InIt, class OutIt>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result, Pred pr);
template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
    };
```

Include the STL standard header **<numeric>** to define several template functions useful for computing numeric values. The descriptions of these templates employ a number of conventions common to all algorithms.

accumulate

```
template<class InIt, class T>
    T accumulate(InIt first, InIt last, T val);
template<class InIt, class T, class Pred>
    T accumulate(InIt first, InIt last, T val, Pred pr);
```

The first template function repeatedly replaces `val` with `val + *I`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. It then returns `val`.

The second template function repeatedly replaces `val` with `pr(val, *I)`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. It then returns `val`.

adjacent_difference

```
template<class InIt, class OutIt>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt adjacent_difference(InIt first, InIt last,
        OutIt result, Pred pr);
```

The first template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value stored is `*I - val`, and `val` is replaced by `*I`. The function returns `result` incremented `last - first` times.

The second template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value stored is `pr(*I, val)`, and `val` is replaced by `*I`. The function returns `result` incremented `last - first` times.

inner_product

```
template<class InIt1, class InIt2, class T>
    T product(InIt1 first1, InIt1 last1,
        Init2 first2, T val);
template<class InIt1, class InIt2, class T,
    class Pred1, class Pred2>
    T product(InIt1 first1, InIt1 last1,
        Init2 first2, T val, Pred1 pr1, Pred2 pr2);
```

The first template function repeatedly replaces `val` with `val + (*I1 * *I2)`, for each value of the `InIt1` iterator `I1` in the interval `[first1, last2)`. In each case, the `InIt2` iterator `I2` equals `first2 + (I1 - first1)`. The function returns `val`.

The first template function repeatedly replaces `val` with `pr1(val, pr2(*I1, *I2))`, for each value of the `InIt1` iterator `I1` in the interval `[first1, last2)`. In each case, the `InIt2` iterator `I2` equals `first2 + (I1 - first1)`. The function returns `val`.

partial_sum

```
template<class InIt, class OutIt>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result);
template<class InIt, class OutIt, class Pred>
    OutIt partial_sum(InIt first, InIt last,
        OutIt result, Pred pr);
```

The first template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value `val` stored is `val + *I`. The function returns `result` incremented `last - first` times.

The second template function stores successive values beginning at `result`, for each value of the `InIt` iterator `I` in the interval `[first, last)`. The first value `val` stored (if any) is `*I`. Each subsequent value `val` stored is `pr(val, *I)`. The function returns `result` incremented `last - first` times.

See also the **Table of Contents** and the **Index**.

<ostream>


```
namespace std {
    template<class E, class T = char_traits<E> >
        class basic_ostream;
    typedef basic_ostream<char, char_traits<char> > ostream;
    typedef basic_ostream<wchar_t, char_traits<wchar_t> > wostream;
//      INSERTERS
    template<class E, class T>
        basic_ostream<E, T>& operator<<(basic_ostream<E, T> os, const E *s);
    template<class E, class T>
        basic_ostream<E, T>& operator<<(basic_ostream<E, T> os, E c);
    template<class T>
        basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, const signed
char *s);
    template<class T>
        basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, signed char c);
    template<class T>
        basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, const unsigned
char *s);
    template<class T>
        basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, unsigned char
c);
//      MANIPULATORS
    template class<E, T>
        basic_ostream<E, T>& endl(basic_ostream<E, T> os);
    template class<E, T>
        basic_ostream<E, T>& ends(basic_ostream<E, T> os);
    template class<E, T>
        basic_ostream<E, T>& flush(basic_ostream<E, T> os);
    };
```

Include the iostreams standard header **<ostream>** to define template class basic_ostream, which mediates extractions for the iostreams classes. The header also defines several related manipulators. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

basic_ostream

basic_ostream · char_type · flush · int_type · off_type · operator<< · opfx · osfx · pos_type · put · seekp · sentry · tellp · traits_type · write

```
template <class E, class T = char_traits<E> >
    class basic_ostream {
public:
    typedef T traits_type;
    typedef T::char_type char_type;
```

```
typedef T::int_type int_type;
typedef T::pos_type pos_type;
typedef T::off_type off_type;
class sentry;
explicit basic_ostream(basic_streambuf<E, T> *sb);
virtual ~ostream();
bool opfx();
void osfx();
basic_ostream& operator<<(basic_ostream& (*pf)(basic_ostream&));
basic_ostream& operator<<(basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_ostream& operator<<(ios_base<E, T>& (*pf)(ios_base<E, T>&));
basic_ostream& operator<<(basic_streambuf<E, T> *sb);
basic_ostream& operator<<(const char *s);
basic_ostream& operator<<(char c);
basic_ostream& operator<<(bool n);
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
basic_ostream& operator<<(void * n);
basic_ostream& put(E c);
basic_ostream& write(E *s, streamsize n);
basic_ostream& flush();
basic_ostream& tellp();
basic_ostream& seekp(pos_type pos);
basic_ostream& seekp(off_type off, ios_base::seek_dir way);
};
```

The template class describes an object that controls insertion of elements and encoded objects into a stream buffer with elements of type E, whose character traits are determined by the class T.

Most of the member functions that overload operator<< are **formatted output functions**. They follow the pattern:

```
iostate state = goodbit;
const sentry ok(*this);
if (ok)
    {try
        {convert and insert elements
        accumulate flags in state}
    catch (...)
        {if (exceptions() & badbit)
            throw;
        setstate(badbit); }}
width(0);    // except for operator<<(E)
setstate(state);
return (*this);
```

Two other member functions are **unformatted output functions**. They follow the pattern:

```
    iostate state = goodbit;
    const sentry ok(*this);
    if (!ok)
        state |= badbit;
    else
        {try
            {obtain and insert elements
            accumulate flags in state}
        catch (...)
            {if (rdstate() & badbit)
                throw;
            setstate(badbit); }}
    setstate(state);
    return (*this);
```

Both groups of functions call `setstate(badbit)` if they encounter a failure while inserting elements.

An object of class `basic_ostream<E, T>` stores only a virtual public base object of class **`basic_ios<E, T>`**.

basic_ostream::basic_ostream

```
explicit basic_ostream(basic_streambuf<E, T> *sb);
```

The constructor initializes the base class by calling `init(sb)`.

basic_ostream::char_type

```
typedef T::char_type char_type;
```

The type describes an element of the controlled sequence. Typically, it is the same as the template parameter `E`. In this implementation, however, if `wchar_t` is not a unique type, then **`char_type`** is defined as an encapsulated wchar_t, so that `operator<<:` can be overloaded on `char_type&`.

basic_ostream::flush

```
basic_ostream& flush();
```

If `rdbuf()` is not a null pointer, the function calls `rdbuf()->pubsync()`. If that returns -1, the function calls `setstate(badbit)`. It returns `*this`.

basic_ostream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_ostream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_ostream::operator<<

```
basic_ostream& operator<<(
    basic_ostream& (*pf)(basic_ostream&));
basic_ostream& operator<<(
```

```
    basic_ios<E, T>& (*pf)(basic_ios<E, T>&));
basic_ostream& operator<<(
    ios_base<E, T>& (*pf)(ios_base<E, T>&));
basic_ostream& operator<<(
    basic_streambuf<E, T> *sb);
basic_ostream& operator<<(const char *s);
basic_ostream& operator<<(char c);
basic_ostream& operator<<(bool n);
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
basic_ostream& operator<<(void *n);
```

The first member function ensures that an expression of the form `ostr <<: endl` calls `endl(ostr)`, then returns `*this`. The second and third functions ensure that other manipulators, such as `hex` behave similarly. The remaining functions are all formatted output functions.

The function:

```
basic_ostream& operator<<(
    basic_streambuf<E, T> *sb);
```

extracts elements from `sb`, if `sb` is not a null pointer, and inserts them. Extraction stops on end-of-file, or if an extraction throws an exception (which is rethrown). It also stops, without extracting the element in question, if an insertion fails. If the function inserts no elements, or if an extraction throws an exception, the function calls `setstate(failbit)`. In any case, the function returns `*this`.

The function:

```
basic_ostream& operator<<(const char *s);
```

determines the length `n = strlen(s)` of the sequence beginning at `s`, and inserts the widened sequence. Each element `c` of the sequence is widened by calling `use_facet< ctype<E> >( getloc()). widen(c)`. If `n < width()`, then the function also inserts a repetition of `width() - n` fill characters. The repetition precedes the sequence if `(flags() & adjustfield != left`. Otherwise, the repetition follows the sequence.

The function:

```
basic_ostream& operator<<(char c);
```

inserts the widened element `use_facet< ctype<E> >( getloc()). widen(c)`. It returns `*this`.

The function:

```
basic_ostream& operator<<(bool n);
```

converts `n` to a boolean field and inserts it by calling `use_facet<num_put<E, OutIt>(getloc()). put(OutIt( rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

The functions:

```
basic_ostream& operator<<(short n);
basic_ostream& operator<<(unsigned short n);
basic_ostream& operator<<(int n);
```

```
basic_ostream& operator<<(unsigned int n);
basic_ostream& operator<<(long n);
basic_ostream& operator<<(unsigned long n);
basic_ostream& operator<<(void *n);
```

each convert `n` to a numberic field and insert it by calling `use_facet<num_put<E, OutIt>(getloc()).put(OutIt( rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

The functions:

```
basic_ostream& operator<<(float n);
basic_ostream& operator<<(double n);
basic_ostream& operator<<(long double n);
```

each convert `n` to a numberic field and insert it by calling `use_facet<num_put<E, OutIt>(getloc()).put(OutIt( rdbuf()), *this, getloc(), n)`. Here, `OutIt` is defined as `ostreambuf_iterator<E, T>`. The function returns `*this`.

basic_ostream::opfx

```
bool opfx();
```

If `good()` is true, and `tie()` is not a null pointer, the member function calls `tie->flush()`. It returns `good()`.

You should not call `opfx` directly. It is called as needed by an object of class `sentry`.

basic_ostream::osfx

```
void osfx();
```

If `flags() & unitbuf` is nonzero, the member function calls `flush()`. You should not call `osfx` directly. It is called as needed by an object of class `sentry`.

basic_ostream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_ostream::put

```
basic_ostream& put(E c);
```

The unformatted output function inserts the element `c`. It returns `*this`.

basic_ostream::seekp

```
basic_ostream& seekp(pos_type pos);
basic_ostream& seekp(off_type off, ios_base::seek_dir way);
```

If `fail()` is false, the first member function calls `rdbuf()-> pubseekpos(pos)`. If `fail()` is false, the second function calls `rdbuf()-> pubseekoff(off, way)`. Both functions return `*this`.

basic_ostream::sentry

```
class sentry {
public:
    explicit sentry(basic_ostream<E, T>& os);
    operator bool() const;
    };
```

The nested class describes an object whose declaration structures the formatted output functions and the unformatted output functions. The constructor effectively calls `os.opfx()` and stores the return value. `operator bool()` delivers this return value. The destructor effectively calls `os.osfx()`, but only if `uncaught_exception()` returns false.

basic_ostream::tellp

```
basic_ostream& tellp();
```

If `fail()` is false, the member function returns `rdbuf()-> pubseekoff(0, cur, in)`. Otherwise, it returns `streampos(-1)`.

basic_ostream::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_ostream::write

```
basic_ostream& write(const E *s, streamsize n);
```

The unformatted output function inserts the sequence of `n` elements beginning at `s`.

endl

```
template class<E, T>
    basic_ostream<E, T>& endl(basic_ostream<E, T> os);
```

The manipulator calls `os.put(os. widen('\n'))`, then calls `os.flush()`. It returns `os`.

ends

```
template class<E, T>
    basic_ostream<E, T>& ends(basic_ostream<E, T> os);
```

The manipulator calls `os.put(E('\0'))`. It returns `os`.

flush

```
template class<E, T>
    basic_ostream<E, T>& flush(basic_ostream<E, T> os);
```

The manipulator calls `os.flush()`. It returns `os`.

operator<<

```
template<class E, class T>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T> os, const E *s);
template<class E, class T>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T> os, E c);
template<class T>
    basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, const signed char
*s);
template<class T>
   basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, signed char c);
template<class T>
    basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, const unsigned char
*s);
template<class T>
   basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, unsigned char c);
```

The template function:

```
template<class E, class T>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T> os, const E *s);
```

determines the length `n = T::length(s)` of the sequence beginning at `s`, and inserts the sequence. If `n < os.width()`, then the function also inserts a repetition of `width() - n` fill characters. The repetition precedes the sequence if `(os.flags() & adjustfield != left`. Otherwise, the repetition follows the sequence. The function returns `os`.

The template function:

```
template<class E, class T>
    basic_ostream<E, T>& operator<<(basic_ostream<E, T> os, E c);
```

is an formatted output functions that inserts the element `c`. It returns `os`.

The template function:

```
template<class T>
    basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, const signed char
*s);
```

returns `os << (const char *)s`.

The template function:

```
template<class T>
    basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, signed char c);
```

returns `os << (char)c`.

The template function:

```
template<class T>
    basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, const unsigned char
*s);
```

returns `os << (const char *)s`.

The template function:

```
template<class T>
    basic_ostream<char, T>& operator<<(basic_ostream<char, T> os, unsigned char c);
```

returns `os << (char)c`.

ostream

```
typedef basic_ostream<char, char_traits<char> > ostream;
```

The type is a synonym for template class basic_ostream, specialized for elements of type *char* with default character traits.

wostream

```
typedef basic_ostream<wchar_t, char_traits<wchar_t> > wostream;
```

The type is a synonym for template class basic_ostream, specialized for elements of type `wchar_t` with default character traits.

See also the **Table of Contents** and the **Index**.

<queue>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class T, class Cont>
    class queue;
template<class T, class Cont, class Pred>
    class priority_queue;
//    TEMPLATE FUNCTIONS
template<class T, class Cont>
    bool operator==(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator!=(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator<(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator>(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator<=(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
template<class T, class Cont>
    bool operator>=(const queue<T, Cont>& lhs,
        const queue<T, Cont>&);
    };
```

Include the STL standard header **<queue>** to define the template classes `priority_queue` and `queue`, and two supporting templates.

operator!=

```
template<class T, class Cont>
    bool operator!=(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class Cont>
    bool operator==(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `queue`. The function returns `lhs.c == rhs.c`.

operator<

```
template<class T, class Cont>
    bool operator<(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `queue`. The function returns `lhs.c < rhs.c`.

operator<=

```
template<class T, class Cont>
    bool operator<=(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class Cont>
    bool operator>(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class Cont>
    bool operator>=(const queue <T, Cont>& lhs,
        const queue <T, Cont>& rhs);
```

The template function returns `!(lhs < rhs)`.

priority_queue

```
template<class T,
    class Cont = vector<T>,
    class Pred = less<Cont::value_type> >
    class priority_queue {
public:
    typedef Cont::allocator_type allocator_type;
    typedef Cont::value_type value_type;
    typedef Cont::size_type size_type;
    explicit priority_queue(const Pred& pr = Pred(),
        const allocator_type& al = allocator_type());
    template<class InIt>
        priority_queue(InIt first, InIt last,
            const Pred& pr = Pred(), const allocator_type& al = allocator_type());
    bool empty() const;
    size_type size() const;
```

```
    allocator_type get_allocator() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    Pred comp;
    };
```

The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of **class** `Cont`. The type `T` of elements in the controlled sequence must match `value_type`.

The sequence is ordered using a protected object named **comp**. After each insertion or removal of the top element (at position zero), for the iterators `P0` and `Pi` designating elements at positions `0` and `i`, `comp(*P0, *Pi)` is false. (For the default template parameter `less<Cont::value_type>` the top element of the sequence compares largest, or highest priority.)

An object of class `Cont` must supply random-access iterators and several public members defined the same as for `deque` and `vector` (both of which are suitable candidates for class `Cont`). The required members are:

```
    typedef T value_type;
    typedef T0 size_type;
    Cont(const A& al);
    Cont(InIt first, InIt last, const allocator_type& al);
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
    const value_type& front() const;
    value_type& front();
    void push_back(const value_type& x);
    void pop_back();
```

Here, `T0` is an unspecified type that meets the stated requirements.

priority_queue::allocator_type

```
typedef Cont::allocator_type allocator_type;
```

The type is a synonym for `Cont::allocator_type`.

priority_queue::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

priority_queue::get_allocator

```
allocator_type get_allocator() const;
```

The member function returns `c.get_allocator()`.

priority_queue::pop

```
void pop();
```

The member function removes the first element of the controlled sequence, which must be non-empty, then reorders it.

priority_queue::priority_queue

```
explicit priority_queue(const Pred& pr = Pred(),
    const allocator_type& al = allocator_type());
template<class InIt>
    priority_queue(InIt first, InIt last,
        const Pred& pr = Pred(), const allocator_type& al = allocator_type());
```

Both constructors store `pr` in `comp` and effectively initialize the stored object with `c(al)`, to specify an empty initial controlled sequence. The template constructor then calls `push(x)` for `x` an iterator of class `InIt` in the range `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
priority_queue(const value_type *first, const value_type *last, const Pred& pr =
Pred(), const allocator_type& al = allocator_type());
```

priority_queue::push

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence, then reorders it.

priority_queue::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

priority_queue::size_type

```
typedef Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

priority_queue::top

```
value_type& top();
const value_type& top() const;
```

The member function returns a reference to the first (highest priority) element of the controlled sequence, which must be non-empty.

priority_queue::value_type

```
typedef Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

queue

```
template<class T,
    class Cont = deque<T> >
    class queue {
public:
    typedef Cont::allocator_type allocator_type;
    typedef Cont::value_type value_type;
    typedef Cont::size_type size_type;
    explicit queue(const allocator_type& al = allocator_type()) const;
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    };
```

The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of **class** `Cont`. The type `T` of elements in the controlled sequence must match `value_type`.

An object of class `Cont` must supply several public members defined the same as for `deque` and `list` (both of which are suitable candidates for class `Cont`). The required members are:

```
    typedef T value_type;
    typedef T0 size_type;
    Cont(const allocator_type& al);
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
    value_type& front();
    const value_type& front() const;
    value_type& back();
    const value_type& back() const;
    void push_back(const value_type& x);
    void pop_front();
```

Here, `T0` is an unspecified type that meets the stated requirements.

queue::allocator_type

```
typedef Cont::allocator_type allocator_type;
```

The type is a synonym for `Cont::allocator_type`.

queue::back

```
value_type& back();
const value_type& back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

queue::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

queue::front

```
value_type& front();
const value_type& front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

queue::get_allocator

```
allocator_type get_allocator() const;
```

The member function returns `c.get_allocator()`.

queue::pop

```
void pop();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

queue::push

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

queue::queue

```
explicit queue(const allocator_type& al = allocator_type());
```

The constructor initializes the stored object with `c(al)`, to specify an empty initial controlled sequence.

queue::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

queue::size_type

```
typedef Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

queue::top

```
value_type& top();
const value_type& top() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

queue::value_type

```
typedef Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

See also the **Table of Contents** and the **Index**.

<set>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class Key, class Pred, class A>
    class set;
template<class Key, class Pred, class A>
    class multiset;
//    TEMPLATE FUNCTIONS
template<class Key, class Pred, class A>
    bool operator==(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator==(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
```

```
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        const set<Key, Pred, A>& lhs,
        const set<Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        const multiset<Key, Pred, A>& lhs,
        const multiset<Key, Pred, A>& rhs);
    };
```

Include the STL standard header **<set>** to define the container template classes `set` and `multiset`, and their supporting templates.

multiset

---

allocator_type · begin · clear · const_iterator · const_reference · const_reverse_iterator · count · difference_type · empty · end · equal_range · erase · find · get_allocator · insert · iterator · key_comp · key_compare · key_type · lower_bound · max_size · multiset · rbegin · reference · rend · reverse_iterator · size · size_type · swap · upper_bound · value_comp · value_compare · value_type

---

```
template<class Key, class Pred = less<Key>, class A = allocator<T> >
    class multiset {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
```

```
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::rebind<value_type>::other::const_reference reference;
    typedef A::rebind<value_type>::other::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef reverse_bidirectional_iterator<iterator,
        value_type, reference, A::const_pointer,
            difference_type> reverse_iterator;
    typedef reverse_bidirectional_iterator<const_iterator,
        value_type, const_reference, A::pointer,
            difference_type> const_reverse_iterator;
    explicit multiset(const Pred& comp = Pred(), const A& al = A());
    multiset(const multiset& x);
    template<class InIt>
        multiset(InIt first, InIt last, const Pred& comp = Pred(),
        const A& al = A());
    const_iterator begin() const;
    iterator end() const;
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    iterator insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(multiset x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    const_iterator lower_bound(const Key& key) const;
    const_iterator upper_bound(const Key& key) const;
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
protected:
    A allocator;
```

```
};
```

The template class describes an object that controls a varying-length sequence of elements of **type `const Key`**. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this stored object by calling the member function `key_comp()`. Such a function object must impose a total order on sort keys. For any element `x` that precedes `y` in the sequence, `key_comp()(y, x)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `set`, an object of template class `multiset` does not ensure that `key_comp()(x, y)` is true. (Keys need not be unique.)

The object allocates and frees storage for the sequence it controls through a protected object named **`allocator`**, of **class `A`**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

`multiset::allocator_type`

```
yypedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

`multiset::begin`

```
const_iterator begin() const;
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

`multiset::clear`

```
void clear() const;
```

The member function calls `erase( begin(), end())`.

`multiset::const_iterator`

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T1`.

`multiset::const_reference`

```
typedef A::rebind<value_type>::other::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

multiset::const_reverse_iterator

```
typedef reverse_bidirectional_iterator<const_iterator,
    value_type, const_reference, A::const_pointer,
        difference_type> const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

multiset::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_b3und(key))`.

multiset::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

multiset::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

multiset::end

```
const_iterator end() const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

multiset::equal_range

```
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

multiset::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second

member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

multiset::find

```
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key equals `key`. If no such element exists, the iterator equals `end()`.

multiset::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

multiset::insert

```
iterator insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function inserts the element `x` in the controlled sequence, then returns the iterator that designates the inserted element. The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values in the range `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(const value_type *first, const value_type *last);
```

multiset::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

multiset::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

multiset::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of any two elements in the controlled sequence.

multiset::key_type

```
typedef Key key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

multiset::lower_bound

```
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

multiset::multiset

```
explicit multiset(const Pred& comp = Pred(), const A& al = A());
multiset(const multiset& x);
template<class InIt>
    multiset(InIt first, InIt last, const Pred& comp = Pred(),
        const A& al = A());
```

The constructors with an argument named `comp` store the function object so that it can be later returned by calling `key_comp()`. All constructors also store the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence. The first constructor specifies an empty initial controlled sequence. The second constructor specifies a copy of the sequence controlled by `x`. The member template constructor specifies the sequence of element values `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
multiset(const value_type *first, const value_type *last,
    const Pred& comp = Pred(), const A& al = A());
```

multiset::max_size

multiset::rbegin

```
const_reverse_iterator rbegin() const;
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

multiset::reference

```
typedef A::rebind<value_type>::other::const_reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

multiset::rend

```
const_reverse_iterator rend() const;
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

multiset::reverse_iterator

```
typedef reverse_bidirectional_iterator<iterator,
    value_type, reference, A::pointer,
        difference_type> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

multiset::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

multiset::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

multiset::swap

```
void swap(multiset& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

multiset::upper_bound

```
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x)` is true.

If no such element exists, the function returns `end()`.

multiset::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

multiset::value_compare

```
typedef Pred value_compare;
```

The type describes a function object that can compare two elements as sort keys to determine their relative order in the controlled sequence.

multiset::value_type

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

operator!=

```
template<class Key, class Pred, class A>
    bool operator!=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator!=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class Key, class Pred, class A>
    bool operator==(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
```

```
    bool operator==(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The first template function overloads `operator==` to compare two objects of template class `multiset`. The second template function overloads `operator==` to compare two objects of template class `multiset`. Both functions return `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class Key, class Pred, class A>
    bool operator<(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The first template function overloads `operator<` to compare two objects of template class `multiset`. The second template function overloads `operator<` to compare two objects of template class `multiset`. Both functions return `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class Key, class Pred, class A>
    bool operator<=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator<=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class Key, class Pred, class A>
    bool operator>(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>(
```

```
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class Key, class Pred, class A>
    bool operator>=(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    bool operator>=(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

set

---

allocator_type · begin · clear · const_iterator · const_reference · const_reverse_iterator · count · difference_type · empty · end · equal_range · erase · find · get_allocator · insert · iterator · key_comp · key_compare · key_type · lower_bound · set · max_size · rbegin · reference · rend · reverse_iterator · size · size_type · swap · upper_bound · value_comp · value_compare · value_type

---

```
template<class Key, class Pred = less<Key>, class A = allocator<T> >
    class set {
public:
    typedef Key key_type;
    typedef Pred key_compare;
    typedef Key value_type;
    typedef Pred value_compare;
    typedef A allocator_type;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::rebind<value_type>::other::const_reference reference;
    typedef A::rebind<value_type>::other::const_reference const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef reverse_bidirectional_iterator<iterator,
        value_type, reference, A::pointer,
            difference_type> reverse_iterator;
```

```
    typedef reverse_bidirectional_iterator<const_iterator,
        value_type, const_reference, A::const_pointer,
            difference_type> const_reverse_iterator;
    explicit set(const Pred& comp = Pred(), const A& al = A());
    set(const set& x);
    template<class InIt>
        set(InIt first, InIt last, const Pred& comp = Pred(),
            const A& al = A());
    const_iterator begin() const;
    iterator end() const;
    const_reverse_iterator rbegin() const;
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator it, const value_type& x);
    template<class InIt>
        void insert(InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    size_type erase(const Key& key);
    void clear();
    void swap(set x);
    key_compare key_comp() const;
    value_compare value_comp() const;
    const_iterator find(const Key& key) const;
    size_type count(const Key& key) const;
    const_iterator lower_bound(const Key& key) const;
    const_iterator upper_bound(const Key& key) const;
    pair<const_iterator, const_iterator>
        equal_range(const Key& key) const;
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type `const Key`**. Each element serves as both a **sort key** and a **value**. The sequence is represented in a way that permits lookup, insertion, and removal of an arbitrary element with a number of operations proportional to the logarithm of the number of elements in the sequence (logarithmic time). Moreover, inserting an element invalidates no iterators, and removing an element invalidates only those iterators which point at the removed element.

The object orders the sequence it controls by calling a stored **function object** of type `Pred`. You access this

stored object by calling the member function `key_comp()`. Such a function object must impose a total order on sort keys. For any element `x` that precedes `y` in the sequence, `key_comp()(y, x)` is false. (For the default function object `less<Key>`, sort keys never decrease in value.) Unlike template class `multiset`, an object of template class `set` ensures that `key_comp()(x, y)` is true. (Each key is unique.)

The object allocates and frees storage for the sequence it controls through a protected object named **`allocator`**, of **class `A`**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

set::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

set::begin

```
const_iterator begin() const;
```

The member function returns a bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

set::clear

```
void clear() const;
```

The member function calls `erase( begin(), end())`.

set::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant bidirectional iterator for the controlled sequence. It is desciibed here as a synonym for the unspecified type `T1`.

set::const_reference

```
typedef A::rebind<value_type>::other::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

set::const_reverse_iterator

```
typedef reverse_bidirectional_iterator<const_iterator,
    value_type, const_reference, A::const_pointer,
        difference_type> const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse bidirectional iterator for the controlled sequence.

set::count

```
size_type count(const Key& key) const;
```

The member function returns the number of elements `x` in the range `[lower_bound(key), upper_bound(key))`.

set::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

set::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

set::end

```
const_iterator end() const;
```

The member function returns a bidirectional iterator that points just beyond the end of the sequence.

set::equal_range

```
pair<const_iterator, const_iterator>
    equal_range(const Key& key) const;
```

The member function returns a pair of iterators `x` such that `x.first == lower_bound(key)` and `x.second == upper_bound(key)`.

set::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member removes the elements with sort keys in the range `[lower_bound(key), upper_bound(key))`. It returns the number of elements it removes.

set::find

```
const_iterator find(const Key& key) const;
```

The member function returns an iterator that designates the earliest element in the controlled sequence whose sort key equals `key`. If no such element exists, the iterator equals `end()`.

set::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

set::insert

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
template<class InIt>
    void insert(InIt first, InIt last);
```

The first member function determines whether an element `y` exists in the sequence whose key matches that of `x`. (The keys match if `!key_comp()(x, y) && !key_comp()(y, x)`.) If not, it creates such an element `y` and initializes it with `x`. The function then determines the iterator `it` that designates `y`. If an insertion occurred, the function returns `pair(it, true)`. Otherwise, it returns `pair(it, false)`.

The second member function returns `insert(x)`, using `it` as a starting place within the controlled sequence to search for the insertion point. (Insertion can occur in amortized constant time, instead of logarithmic time, if the insertion point immediately follows `it`.) The third member function inserts the sequence of element values in the range `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(const value_type *first, const value_type *last);
```

set::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a bidirectional iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

set::key_comp

```
key_compare key_comp() const;
```

The member function returns the stored function object that determines the order of elements in the controlled sequence. The stored object defines the member function:

```
bool operator(const Key& x, const Key& y);
```

which returns true if `x` strictly precedes `y` in the sort order.

set::key_compare

```
typedef Pred key_compare;
```

The type describes a function object that can compare two sort keys to determine the relative order of any two elements in the controlled sequence.

set::key_type

```
typedef Key key_type;
```

The type describes the sort key object which constitutes each element of the controlled sequence.

set::lower_bound

```
const_iterator lower_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(x, key)` is false.

If no such element exists, the function returns `end()`.

set::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

set::rbegin

```
const_reverse_iterator rbegin() const;
```

The member function returns a reverse bidirectional iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

set::reference

```
typedef A::rebind<value_type>::other::const_reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

set::rend

```
const_reverse_iterator rend() const;
```

The member function returns a reverse bidirectional iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

set::reverse_iterator

```
typedef reverse_bidirectional_iterator<iterator,
    value_type, reference, A::pointer,
        difference_type> reverse_iterator;
```

The type describes an object that can serve as a reverse bidirectional iterator for the controlled sequence.

set::set

```
explicit set(const Pred& comp = Pred(), const A& al = A());
set(const set& x);
template<class InIt>
    set(InIt first, InIt last, const Pred& comp = Pred(),
        const A& al = A());
```

The constructors with an argument named `comp` store the function object so that it can be later returned by calling `key_comp()`. All constructors also store the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence. The first constructor specifies an empty initial controlled sequence. The second constructor specifies a copy of the sequence controlled by `x`. The member template constructor specifies the sequence of element values `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
set(const value_type *first, const value_type *last,
    const Pred& comp = Pred(), const A& al = A());
```

set::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

set::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

set::swap

```
void swap(set& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

set::upper_bound

```
const_iterator upper_bound(const Key& key) const;
```

The member function returns an iterator that designates the earliest element `x` in the controlled sequence for which `key_comp()(key, x)` is true.

If no such element exists, the function returns `end()`.

set::value_comp

```
value_compare value_comp() const;
```

The member function returns a function object that determines the order of elements in the controlled sequence.

set::value_compare

```
typedef Pred value_compare;
```

The type describes a function object that can compare two elements as sort keys to determine their relative order in the controlled sequence.

set::value_type

```
typedef Key value_type;
```

The type describes an element of the controlled sequence.

swap

```
template<class Key, class Pred, class A>
    void swap(
        const multiset <Key, Pred, A>& lhs,
        const multiset <Key, Pred, A>& rhs);
template<class Key, class Pred, class A>
    void swap(
        const set <Key, Pred, A>& lhs,
        const set <Key, Pred, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

See also the **Table of Contents** and the **Index**.

<sstream>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    template<class E,
        class T = char_traits<E>,
        class A = allocator<E> >
        class basic_stringbuf;
    typedef basic_stringbuf<char> stringbuf;
    typedef basic_stringbuf<wchar_t> wstringbuf;
    template<class E,
        class T = char_traits<E>,
        class A = allocator<E> >
        class basic_istringstream;
    typedef basic_istringstream<char> istringstream;
    typedef basic_istringstream<wchar_t> wistringstream;
    template<class E,
        class T = char_traits<E>,
        class A = allocator<E> >
        class basic_ostringstream;
    typedef basic_ostringstream<char> ostringstream;
    typedef basic_ostringstream<wchar_t> wostringstream;
    template<class E,
        class T = char_traits<E>,
        class A = allocator<E> >
        class basic_stringstream;
    typedef basic_stringstream<char> stringstream;
    typedef basic_stringstream<wchar_t> wstringstream;
    };
```

Include the iostreams standard header **<sstream>** to define several template classes that support iostreams operations on sequences stored in an allocated array object. Such sequences are easily converted to and from objects of template class basic_string.

basic_stringbuf

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_stringbuf {
public:
    typedef T traits_type;
    typedef E char_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    basic_stringbuf(ios_base::openmode mode =
        ios_base::in | ios_base::out);
```

```
    basic_stringbuf(basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    basic_string<E, T, A> str() const;
    void str(basic_string<E, T, A>& x);
protected:
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    virtual int_type underflow();
    virtual int_type pbackfail(int_type c = T::eof());
    virtual int_type overflow(int_type c = T::eof());
    };
```

The template class describes a **stream buffer** that controls the transmission of elements to and from a sequence of elements stored in an array object. The object is allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `basic_stringbuf<E, T, A>` stores a copy of the `ios_base::openmode` argument from its constructor as its **stringbuf mode** `mode`:

- If `mode & ios_base::in` is nonzero, the input buffer is accessible.
- If `mode & ios_base::out` is nonzero, the output buffer is accessible.

basic_stringbuf::basic_stringbuf

```
basic_stringbuf(ios_base::openmode mode =
    ios_base::in | ios_base::out);
basic_stringbuf(basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

The first constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores `mode` as the stringbuf mode.

The second constructor allocates a copy of the sequence controlled by `x`, an object of class `basic_string<E, T, A>`. If `mode & ios_base::in` is nonzero, it sets the input buffer to begin reading at the start of the sequence. If `mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the start of the sequence. It also stores `mode` as the stringbuf mode.

basic_stringbuf::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_stringbuf::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_stringbuf::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_stringbuf::overflow

```
virtual int_type overflow(int_type c = T::eof());
```

If `c` does not compare equal to `T::eof()`, the protected virtual member function endeavors to insert the element `T::to_char_type(c)` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer. (Extending the output buffer this way also extends any associated input buffer.)

If the function cannot succeed, it returns `T::eof()`. Otherwise, it returns `T::not_eof(c)`.

basic_stringbuf::pbackfail

```
virtual int_type pbackfail(int_type c = T::eof());
```

The protected virtual member function endeavors to put back an element into the input buffer, then make it the current element (pointed to by the next pointer). If `c` compares equal to `T::eof()`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = T::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If a putback position is available, and if the stringbuf mode permits the sequence to be altered (`mode & ios_base::out` is nonzero), it can store `x` into the putback position and decrement the next pointer for the input buffer.

If the function cannot succeed, it returns `T::eof()`. Otherwise, it returns `T::not_eof(c)`.

basic_stringbuf::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_stringbuf::seekoff

```
virtual pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_stringbuf<E, T, A>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus `off`.
- If `way == ios_base::cur`, the new position is the current stream position plus `off`.
- If `way == ios_base::end`, the new position is the end of the stream plus `off`.

If `mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. If the function affects both stream positions, `way` must be `ios_base::beg` or `ios_base::end` and both streams are positioned at the same element. Otherwise (or if neither position is affected) the positioning operation fails.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

basic_stringbuf::seekpos

```
virtual pos_type seekpos(pos_type sp,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `basic_stringbuf<E, T, A>`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `sp`.

If `mode & ios_base::in` is nonzero, the function alters the next position to read in the input buffer. If `mode & ios_base::out` is nonzero, the function alters the next position to write in the output buffer. For a stream to be affected, its buffer must exist. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence. Otherwise (or if neither position is affected) the positioning operation fails.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

basic_stringbuf::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns an object of class `basic_string<E, T, allocator>`, whose controlled sequence is a copy of the sequence controlled by `*this`. The sequence copied depends on the stored stringbuf mode `mode`:

- If `mode & ios_base::out` is nonzero and an output buffer exists, the sequence is the entire output buffer (`epptr() - pbase()` elements beginning with `pbase()`).
- Otherwise, if `mode & ios_base::in` is nonzero and an input buffer exists, the sequence is the entire input buffer (`egptr() - eback()` elements beginning with `eback()`).
- Otherwise, the copied sequence is empty.

The second member function deallocates any sequence currently controlled by `*this`. It then allocates a copy of the sequence controlled by `x`. If `mode & ios_base::in` is nonzero, it sets the input buffer to begin reading at the beginning of the sequence. If `mode & ios_base::out` is nonzero, it sets the output buffer to begin writing at the beginning of the sequence.

basic_stringbuf::traits_type

```
typedef T traits_type;
```

basic_stringbuf::underflow

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input buffer, then advance the current stream position, and return the element as `T::to_int_type(c)`. It can do so in only one way: If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.

If the function cannot succeed, it returns `T::eof()`. Otherwise, it returns the current element in the input stream, converted as described above.

basic_istringstream

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_istringstream : public basic_istream<E, T> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit basic_istringstream(ios_base::openmode mode = ios_base::in);
    explicit basic_istringstream(const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::in);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
    };
```

The template class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

basic_istringstream::basic_istringstream

```
explicit basic_istringstream(ios_base::openmode mode = ios_base::in);
explicit basic_istringstream(const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::in);
```

The first constructor initializes the base class by calling `basic_istream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode | ios_base::in)`.

The second constructor initializes the base class by calling `basic_istream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode | ios_base::in)`.

basic_istringstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_istringstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_istringstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_istringstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_istringstream::rdbuf

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `basic_stringbuf<E, T, A>`.

basic_istringstream::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns `rdbuf()-> str()`. The second member function calls `rdbuf()-> str(x)`.

basic_istringstream::traits_type

```
typedef T traits_type;
```

basic_ostringstream

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_ostringstream : public basic_ostream<E, T> {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit basic_ostringstream(ios_base::openmode mode = ios_base::out);
    explicit basic_ostringstream(const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::out);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
    };
```

The template class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

basic_ostringstream::basic_ostringstream

```
explicit basic_ostringstream(ios_base::openmode mode = ios_base::out);
explicit basic_ostringstream(const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::out);
```

The first constructor initializes the base class by calling basic_ostream(sb), where sb is the stored object of class basic_stringbuf<E, T, A>. It also initializes sb by calling basic_stringbuf<E, T, A>(mode | ios_base::out).

The second constructor initializes the base class by calling basic_ostream(sb). It also initializes sb by calling basic_stringbuf<E, T, A>(x, mode | ios_base::out).

basic_ostringstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter E.

basic_ostringstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for T::int_type.

basic_ostringstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for T::off_type.

basic_ostringstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for T::pos_type.

basic_ostringstream::rdbuf

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to basic_stringbuf<E, T, A>.

basic_ostringstream::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns rdbuf()-> str(). The second member function calls rdbuf()-> str(x).

basic_ostringstream::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

basic_stringstream

```
template <class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_stringstream : public basic_iostream<E, T> {
```

```
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit basic_stringstream(ios_base::openmode mode = ios_base::in |
ios_base::out);
    explicit basic_stringstream(const basic_string<E, T, A>& x,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    basic_stringbuf<E, T, A> *rdbuf() const;
    basic_string<E, T, A>& str();
    void str(const basic_string<E, T, A>& x);
    };
```

The template class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class `basic_stringbuf<E, T, A>`, with elements of type `E`, whose character traits are determined by the class `T`, and whose elements are allocated by an allocator of class `A`. The object stores an object of class `basic_stringbuf<E, T, A>`.

basic_stringstream::basic_stringstream

```
explicit basic_stringstream(ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_stringstream(const basic_string<E, T, A>& x,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

The first constructor initializes the base class by calling `basic_iostream(sb)`, where `sb` is the stored object of class `basic_stringbuf<E, T, A>`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(mode)`.

The second constructor initializes the base class by calling `basic_ostream(sb)`. It also initializes `sb` by calling `basic_stringbuf<E, T, A>(x, mode)`.

basic_stringstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_stringstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_stringstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_stringstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_stringstream::rdbuf

```
basic_stringbuf<E, T, A> *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to basic_stringbuf<E, T, A>.

basic_stringstream::str

```
basic_string<E, T, A> str() const;
void str(basic_string<E, T, A>& x);
```

The first member function returns rdbuf()-> str(). The second member function calls rdbuf()-> str(x).

basic_stringstream::traits_type

```
typedef T traits_type;
```

istringstream

```
typedef basic_istringstream<char> istringstream;
```

The type is a synonym for template class basic_istringstream, specialized for elements of type *char*.

ostringstream

```
typedef basic_ostringstream<char> ostringstream;
```

The type is a synonym for template class basic_ostringstream, specialized for elements of type *char*.

stringbuf

```
typedef basic_stringbuf<char> stringbuf;
```

The type is a synonym for template class basic_stringbuf, specialized for elements of type *char*.

stringstream

```
typedef basic_stringstream<char> stringstream;
```

The type is a synonym for template class basic_stringstream, specialized for elements of type *char*.

wistringstream

```
typedef basic_istringstream<wchar_t> wistringstream;
```

The type is a synonym for template class basic_istringstream, specialized for elements of type wchar_t.

wostringstream

```
typedef basic_ostringstream<wchar_t> wostringstream;
```

The type is a synonym for template class basic_ostringstream, specialized for elements of type wchar_t.

wstringbuf

```
typedef basic_stringbuf<wchar_t> wstringbuf;
```

The type is a synonym for template class `basic_stringbuf`, specialized for elements of type `wchar_t`.

wstringstream

```
typedef basic_stringstream<wchar_t> wstringstream;
```

The type is a synonym for template class `basic_stringstream`, specialized for elements of type `wchar_t`.

See also the **Table of Contents** and the **Index**.

<stack>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class T, class Cont>
    class stack;
//    TEMPLATE FUNCTIONS
template<class T, class Cont>
    bool operator==(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator!=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator<(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator>(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator<=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
template<class T, class Cont>
    bool operator>=(const stack<T, Cont>& lhs,
        const stack<T, Cont>&);
    };
```

Include the STL standard header **<stack>** to define the template class `stack` and two supporting templates.

operator!=

```
template<class T, class Cont>
    bool operator!=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class Cont>
    bool operator==(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `stack`. The function returns `lhs.c == rhs.c`.

operator<

```
template<class T, class Cont>
    bool operator<(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `stack`. The function returns `lhs.c < rhs.c`.

operator<=

```
template<class T, class Cont>
    bool operator<=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class Cont>
    bool operator>(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class Cont>
    bool operator>=(const stack <T, Cont>& lhs,
        const stack <T, Cont>& rhs);
```

The template function returns `!(lhs < rhs)`.

stack

```
template<class T,
    class Cont = deque<T> >
    class stack {
public:
    typedef Cont::allocator_type allocator_type;
    typedef Cont::value_type value_type;
    typedef Cont::size_type size_type;
    explicit stack(const allocator_type& al = allocator_type()) const;
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);
    void pop();
protected:
    Cont c;
    };
```

The template class describes an object that controls a varying-length sequence of elements. The object allocates and frees storage for the sequence it controls through a protected object named **c**, of **class Cont**. The type T of elements in the controlled sequence must match value_type.

An object of class Cont must supply several public members defined the same as for deque, list, and vector (all of which are suitable candidates for class Cont). The required members are:

```
    typedef T value_type;
    typedef T0 size_type;
    Cont(const allocator_type& al);
    bool empty() const;
    size_type size() const;
    allocator_type get_allocator() const;
    value_type& back();
    const value_type& back() const;
    void push_back(const value_type& x);
    void pop_back();
```

Here, T0 is an unspecified type that meets the stated requirements.

stack::allocator_type

```
typedef Cont::allocator_type allocator_type;
```

The type is a synonym for `Cont::allocator_type`.

stack::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

stack::get_allocator

```
allocator_type get_allocator() const;
```

The member function returns `c.get_allocator()`.

stack::pop

```
void pop();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

stack::push

```
void push(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

stack::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

stack::size_type

```
typedef Cont::size_type size_type;
```

The type is a synonym for `Cont::size_type`.

stack::stack

```
explicit stack(const allocator_type& al = allocator_type());
```

The constructor initializes the stored object with `c(al)`, to specify an empty initial controlled sequence.

stack::top

```
value_type& top();
const value_type& top() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

stack::value_type

```
typedef Cont::value_type value_type;
```

The type is a synonym for `Cont::value_type`.

See also the **Table of Contents** and the **Index**.

<stdexcept>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
    };
```

Include the standard header **<stdexcept>** to define several classes used for reporting exceptions. The classes form a derivation hierarchy, as indicated by the indenting above, all derived from class exception.

domain_error

```
class domain_error : public logic_error {
public:
    domain_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report a domain error. The value returned by what() is what_arg.data().

invalid_argument

```
class invalid_argument : public logic_error {
public:
    invalid_argument(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an invalid argument. The value returned by what() is what_arg.data().

length_error

```
class length_error : public logic_error {
public:
    length_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an attempt to generate an object too long to be specified. The value returned by what() is what_arg.data().

logic_error

```
class logic_error : public exception {
public:
    logic_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions. The value returned by what() is what_arg.data().

out_of_range

```
class out_of_range : public logic_error {
public:
    out_of_range(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an argument that is out of its valid range. The value returned by what() is what_arg.data().

overflow_error

```
class overflow_error : public runtime_error {
public:
    overflow_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an arithmetic overflow. The value returned by what() is what_arg.data().

range_error

```
class range_error : public runtime_error {
public:
    range_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report a range error. The value returned by `what()` is `what_arg.data()`.

runtime_error

```
class runtime_error : public exception {
public:
    runtime_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report errors presumably detectable only when the program executes. The value returned by `what()` is `what_arg.data()`.

underflow_error

```
class underflow_error : public runtime_error {
public:
    underflow_error(const string& what_arg);
    };
```

The class serves as the base class for all exceptions thrown to report an arithmetic underflow. The value returned by `what()` is `what_arg.data()`.

See also the **Table of Contents** and the **Index**.

<streambuf>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    template<class E, class T = char_traits<E> >
        class basic_streambuf;
    typedef basic_streambuf<char, char_traits<char> > streambuf;
    typedef basic_streambuf<wchar_t, char_traits<wchar_t> > wstreambuf;
    };
```

Include the iostreams standard header **<streambuf>** to define template class basic_streambuf, which is basic to the operation of the iostreams classes. (This header is typically included for you by another of the iostreams headers. You seldom have occasion to include it directly.)

basic_streambuf

---

char_type · eback · egptr · epptr · gbump · getloc · gptr · imbue · in_avail · int_type · off_type · overflow · pbackfail · pbase · pbump · pos_type · pptr · pubimbue · pubseekoff · pubseekpos · pubsetbuf · pubsync · sbumpc · seekoff · seekpos · setbuf · setg · setp · sgetc · sgetn · showmanyc · snextc · sputbackc · sputc · sputn · sungetc · sync · traits_type · uflow · underflow · xsgetn · xsputn

---

```
template <class E, class T = char_traits<E> >
    class basic_streambuf {
public:
    typedef E char_type;
    typedef T traits_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    virtual ~streambuf();
    locale pubimbue(const locale& loc);
    locale getloc() const;
    basic_streambuf *pubsetbuf(E *s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
```

```
        ios_base::openmode which = ios_base::in | ios_base::out);
    int pubsync();
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    streamsize sgetn(E *s, streamsize n);
    int_type sputbackc(E c);
    int_type sungetc();
    int_type sputc(E c);
    streamsize sputn(const E *s, streamsize n);
protected:
    basic_streambuf();
    E *eback() const;
    E *gptr() const;
    E *egptr() const;
    void gbump(int n);
    void setg(E *gbeg, E *gnext, E *gend);
    E *pbase() const;
    E *pptr() const;
    E *epptr() const;
    void pbump(int n);
    void setp(E *pbeg, E *pend);
    virtual void imbue(const locale &loc);
    virtual basic_streambuf *setbuf(E *s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int sync();
    virtual int showmanyc();
    virtual streamsize xsgetn(E *s, streamsize n);
    virtual int_type underflow();
    virtual int_type uflow();
    virtual int_type pbackfail(int_type c = T::eof());
    virtual streamsize xsputn(const E *s, streamsize n);
    virtual int_type overflow(int_type c = T::eof());
    };
```

The template class describes an abstract base class for deriving a **stream buffer**, which controls the transmission of elements to and from a specific representation of a stream. An object of class `basic_streambuf<E, T>` helps control a stream with elements of type `E`, whose character traits are

determined by the class T.

Every stream buffer conceptually controls two independent streams, in fact, one for extractions (input) and one for insertions (output). A specific representation may, however, make either or both of these streams inaccessible. It typically maintains some relationship between the two streams. What you insert into the output stream of a basic_stringbuf<E, T> object, for example, is what you later extract from its input stream. And when you position one stream of a basic_filebuf<E, T> object, you position the other stream in tandem.

The public interface to template class basic_streambuf supplies the operations common to all stream buffers, however specialized. The protected interface supplies the operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. Each of the derived stream buffers in the Standard C++ library describes how it specializes the behavior of its protected virtual member functions. Documented here is the **default behavior** for the base class, which is often to do nothing.

The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An **input buffer**, for example, is characterized by:

- eback(), a pointer to the beginning of the buffer
- gptr(), a pointer to the next element to read
- egptr(), a pointer just past the end of the buffer

Similarly, an **output buffer** is characterized by:

- pbase(), a pointer to the beginning of the buffer
- pptr(), a pointer to the next element to write
- epptr(), a pointer just past the end of the buffer

For any buffer, the protocol is:

- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. (They can be safely compared for order.)
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the **write position** designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the **read position** designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the **putback position** designated by the decremented next pointer.

Any protected virtual member functions you write for a class derived from basic_streambuf<E, T> must cooperate in maintaining this protocol.

An object of class basic_streambuf<E, T> stores the six pointers described above. It also stores a **locale object** in an object of type locale for potential use by a derived stream buffer.

basic_streambuf::basic_streambuf

```
basic_streambuf();
```

The protected constructor stores a null pointer in all the pointers controlling the input buffer and the output buffer. It also stores `locale::classic()` in the locale object.

basic_streambuf::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_streambuf::eback

```
E *eback() const;
```

The member function returns a pointer to the beginning of the input buffer.

basic_streambuf::egptr

```
E *egptr() const;
```

The member function returns a pointer just past the end of the input buffer.

basic_streambuf::epptr

```
E *epptr() const;
```

The member function returns a pointer just past the end of the output buffer.

basic_streambuf::gbump

```
void gbump(int n);
```

The member function adds `n` to the next pointer for the input buffer.

basic_streambuf::getloc

```
locale getloc() const;
```

The member function returns the stored locale object.

basic_streambuf::gptr

```
E *gptr() const;
```

The member function returns a pointer to the next element of the input buffer.

basic_streambuf::imbue

```
virtual void imbue(const locale &loc);
```

The default behavior is to do nothing.

basic_streambuf::in_avail

```
streamsize in_avail();
```

If a read position is available, the member function returns `egptr() - gptr()`. Otherwise, it returns `showmanyc()`.

basic_streambuf::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_streambuf::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_streambuf::overflow

```
virtual int_type overflow(int_type c = T::eof());
```

If `c` does not compare equal to `T::eof()`, the protected virtual member function endeavors to insert the element `T::to_char_type(c)` into the output stream. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- It can make a write position available by allocating new or additional storage for the output buffer.
- It can make a write position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

If the function cannot succeed, it returns `T::eof()` or throws an exception. Otherwise, it returns Otherwise, it returns `T::not_eof(c)`. The default behavior is to return `T::eof()`.

basic_streambuf::pbackfail

```
virtual int_type pbackfail(int_type c = T::eof());
```

The protected virtual member function endeavors to put back an element into the input stream, then make it the current element (pointed to by the next pointer). If `c` compares equal to `T::eof()`, the element to

push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `T::to_char_type(c)`. The function can put back an element in various ways:

- If a putback position is available, it can store the element into the putback position and decrement the next pointer for the input buffer.
- It can make a putback position available by allocating new or additional storage for the input buffer.
- For a stream buffer with common input and output streams, it can make a putback position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer.

If the function cannot succeed, it returns `T::eof()` or throws an exception. Otherwise, it returns some other value. The default behavior is to return `T::eof()`.

basic_streambuf::pbase

```
E *pbase() const;
```

The member function returns a pointer to the beginning of the output buffer.

basic_streambuf::pbump

```
void pbump(int n);
```

The member function adds `n` to the next pointer for the output buffer.

basic_streambuf::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

basic_streambuf::pptr

```
E *pptr() const;
```

The member function returns a pointer to the next element of the output buffer.

basic_streambuf::pubimbue

```
locale pubimbue(const locale& loc);
```

The member function stores `loc` in the locale object, calls `imbue()`, then returns the previous value stored in the locale object.

basic_streambuf::pubseekoff

```
pos_type pubseekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The member function returns seekoff(off, way, which).

basic_streambuf::pubseekpos

```
pos_type pubseekpos(pos_type sp,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The member function returns seekpos(sp, which).

basic_streambuf::pubsetbuf

```
basic_streambuf *pubsetbuf(E *s, streamsize n);
```

The member function returns stbuf(s, n).

basic_streambuf::pubsync

```
int pubsync();
```

The member function returns sync().

basic_streambuf::sbumpc

```
int_type sbumpc();
```

If a read position is available, the member function returns T::to_int_type(*gptr()) and increments the next pointer for the input buffer. Otherwise, it returns uflow().

basic_streambuf::seekoff

```
virtual pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. The new position is determined as follows:

- If way == ios_base::beg, the new position is the beginning of the stream plus off.
- If way == ios_base::cur, the new position is the current stream position plus off.
- If way == ios_base::end, the new position is the end of the stream plus off.

Typically, if which & ios_base::in is nonzero, the input stream is affected, and if which & ios_base::out is nonzero, the output stream is affected. Actual use of this parameter varies among

derived stream buffers, however.

If the function succeeds in altering the stream position(s), it returns the resultant stream position (or one of them). Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::seekpos

```
virtual pos_type seekpos(pos_type sp,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. The new position is `sp`.

Typically, if `which & ios_base::in` is nonzero, the input stream is affected, and if `which & ios_base::out` is nonzero, the output stream is affected. Actual use of this parameter varies among derived stream buffers, however.

If the function succeeds in altering the stream position(s), it returns the resultant stream position (or one of them). Otherwise, it returns an invalid stream position. The default behavior is to return an invalid stream position.

basic_streambuf::setbuf

```
virtual basic_streambuf *setbuf(E *s, streamsize n);
```

The protected virtual member function performs an operation peculiar to each derived stream buffer. (See, for example, `basic_filebuf`.) The default behavior is to return `this`.

basic_streambuf::setg

```
void setg(E *gbeg, E *gnext, E *gend);
```

The member function stores `gbeg` in the beginning pointer, `gnext` in the next pointer, and `gend` in the end pointer for the input buffer.

basic_streambuf::setp

```
void setp(E *pbeg, E *pend);
```

The member function stores `pbeg` in the beginning pointer, `pnext` in the next pointer, and `pend` in the end pointer for the output buffer.

basic_streambuf::sgetc

```
int_type sgetc();
```

If a read position is available, the member function returns `T::to_int_type( *gptr())`

Otherwise, it returns `underflow()`.

basic_streambuf::sgetn

```
streamsize sgetn(E *s, streamsize n);
```

The member function returns `sgetn(s, n)`.

basic_streambuf::showmanyc

```
virtual int showmanyc();
```

The protected virtual member function returns a count of the number of characters that can be extracted from the input stream with no fear that the program will suffer an indefinite wait. The default behavior is to return zero.

basic_streambuf::snextc

```
int_type snextc();
```

The member function calls `sbumpc()` and, if that function returns `T::eof()`, returns `T::eof()`. Otherwise, it returns `sgetc()`.

basic_streambuf::sputbackc

```
int_type sputbackc(E c);
```

If a putback position is available and `c` compares equal to the character stored in that position, the member function decrements the next pointer for the input buffer and returns `ch`, which is the value `T::to_int_type(c)`. Otherwise, it returns `pbackfail(ch)`.

basic_streambuf::sputc

```
int_type sputc(E c);
```

If a write position is available, the member function stores `c` in the write position, increments the next pointer for the output buffer, and returns `ch`, which is the value `T::to_int_type(c)`. Otherwise, it returns `overflow(ch)`.

basic_streambuf::sputn

```
streamsize sputn(const E *s, streamsize n);
```

The member function returns `sputn(s, n)`.

basic_streambuf::sungetc

```
int_type sungetc();
```

If a putback position is available, the member function decrements the next pointer for the input buffer and returns `T::to_int_type( *gptr())`. Otherwise it returns `pbackfail()`.

basic_streambuf::sync

```
virtual int sync();
```

The protected virtual member function endeavors to synchronize the controlled streams with any associated external streams. Typically, this involves writing out any elements between the beginning and next pointers for the output buffer. It does *not* involve putting back any elements between the next and end pointers for the input buffer. If the function cannot succeed, it returns -1. The default behavior is to return zero.

basic_streambuf::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter `T`.

basic_streambuf::uflow

```
virtual int_type uflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, then advance the current stream position, and return the element as `T::to_int_type(c)`. It can do so in various ways:

- If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.
- It can read an element directly, from some external source, and deliver it as the value `c`.
- For a stream buffer with common input and output streams, it can make a read position available by writing out, to some external destination, some or all of the elements between the beginning and next pointers for the output buffer. Or it can allocate new or additional storage for the input buffer. The function then reads in, from some external source, one or more elements.

If the function cannot succeed, it returns `T::eof()`, or throws an exception. Otherwise, it returns the current element `c` in the input stream, converted as described above, and advances the next pointer for the input buffer. The default behavior is to call `underflow()` and, if that function returns `T::eof()`, to return `T::eof()`. Otherwise, the function returns the current element `c` in the input stream, converted as described above, and advances the next pointer for the input buffer.

basic_streambuf::underflow

```
virtual int_type underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input stream, without advancing the current stream position, and return it as `T::to_int_type(c)`. It can do so in various ways:

- If a read position is available, `c` is the element stored in the read position.
- It can make a read position available by allocating new or additional storage for the input buffer, then reading in, from some external source, one or more elements.

If the function cannot succeed, it returns `T::eof()`, or throws an exception. Otherwise, it returns the current element in the input stream, converted as described above. The default behavior is to return `T::eof()`.

basic_streambuf::xsgetn

```
virtual streamsize xsgetn(E *s, streamsize n);
```

The protected virtual member function extracts up to `n` elements from the input stream, as if by repeated calls to `sbumpc`, and stores them in the array beginning at `s`. It returns the number of elements actually extracted.

basic_streambuf::xsputn

```
virtual streamsize xsputn(const E *s, streamsize n);
```

The protected virtual member function inserts up to `n` elements into the output stream, as if by repeated calls to `sputc`, from the array beginning at `s`. It returns the number of elements actually inserted.

streambuf

```
typedef basic_streambuf<char, char_traits<char> > streambuf;
```

The type is a synonym for template class `basic_streambuf`, specialized for elements of type *char* with default character traits.

wstreambuf

```
typedef basic_streambuf<wchar_t, char_traits<wchar_t> > wstreambuf;
```

The type is a synonym for template class `basic_streambuf`, specialized for elements of type `wchar_t` with default character traits.

See also the **Table of Contents** and the **Index**.

<string>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

basic_string · char_traits · char_traits<char> · char_traits<wchar_t> · getline · operator+ · operator!= · operator== · operator< · operator<< · operator<= · operator> · operator>= · operator>> · string · swap · wstring

---

```
namespace std {
//    TEMPLATE CLASSES
template<class E>
    struct char_traits;
struct char_traits<char>;
struct char_traits<wchar_t>;
template<class E,
    class T = char_traits<E>,
    class A = allocator<E> >
    class basic_string;
typedef basic_string<char> string;
typedef basic_string>wchar_t> wstring;
//    TEMPLATE FUNCTIONS
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        E rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        E lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
```

```
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator==(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator!=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
```

```
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    void swap(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_ostream<E>& operator<<(
        basic_ostream <E>& os,
        const basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E>& operator>>(
        basic_istream <E>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream <E, T>& is,
        basic_string<E, T, A>& str);
 template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream <E, T>& is,
        basic_string<E, T, A>& str,
        E delim);
   };
```

Include the standard header **<string>** to define the container template class `basic_string` and various supporting templates.

basic_string

```
template<class E,
    class T = char_traits<E>,
    class A = allocator<T> >
    class basic_string {
public:
    typedef T traits_type;
    typedef A allocator_type;
    typedef T::char_type char_type;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::pointer pointer;
    typedef A::const_pointer const_pointer;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef A::value_type value_type;
    typedef T0 iterator;
    typedef T1 const_iterator;
    typedef reverse_iterator<iterator, value_type,
        reference, pointer, difference_type>
            reverse_iterator;
    typedef reverse_iterator<const_iterator, value_type,
        const_reference, const_pointer, difference_type>
            const_reverse_iterator;
    static const size_type npos = -1;
    explicit basic_string(const A& al = A());
    basic_string(const basic_string& rhs);
    basic_string(const basic_string& rhs, size_type pos, size_type n,
        const A& al = A());
    basic_string(const E *s, size_type n, const A& al = A());
    basic_string(const E *s, const A& al = A());
    basic_string(size_type n, E c, const A& al = A());
    template <class InIt>
```

```
    basic_string(InIt first, InIt last, const A& al = A());
basic_string& operator=(const basic_string& rhs);
basic_string& operator=(const E *s);
basic_string& operator=(E c);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
const_reference at(size_type pos) const;
reference at(size_type pos);
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const E *c_str() const;
const E *data() const;
size_type length() const;
size_type size() const;
size_type max_size() const;
void resize(size_type n, E c = E());
size_type capacity() const;
void reserve(size_type n = 0);
bool empty() const;
basic_string& operator+=(cosst basic_string& rhs);
basic_string& operator+=(c=nst E *s);
basic_string& operator+=(E c);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const E *s, size_type n);
basic_string& append(const E *s);
basic_string& append(size_type n, E c);
template<class InIt>
    basic_string& append(InIt first, InIt last);
basic_string& assign(const basic_string& str);
basic_string& assign(const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const E *s, size_type n);
basic_string& assign(const E *s);
basic_string& assign(size_type n, E c);
template<class InIt>
    basic_string& assign(InIt first, InIt last);
```

```
basic_string& insert(size_type p0,
    const basic_string& str);
basic_string& insert(size_type p0,
    const basic_string& str, size_type pos, size_type n);
basic_string& insert(size_type p0,
    const E *s, size_type n);
basic_string& insert(size_type p0, const E *s);
basic_string& insert(size_type p0, size_type n, E c);
iterator insert(iterator it, E c);
void insert(iterator it, size_type n, E c);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
basic_string& erase(size_type p0 = 0, size_type n = npos);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const E *s, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const E *s);
basic_string& replace(size_type p0, size_type n0,
    size_type n, E c);
basic_string& replace(iterator first0, iterator last0,
    const basic_string& str);
basic_string& replace(iterator first0, iterator last0,
    const E *s, size_type n);
basic_string& replace(iterator first0, iterator last0,
    const E *s);
basic_string& replace(iterator first0, iterator last0,
    size_type n, E c);
template<class InIt>
    basic_string& replace(iterator first0, iterator last0,
        InIt first, InIt last);
size_type copy(E *s, size_type n, size_type pos = 0) const;
void swap(basic_string& str);
size_type find(const basic_string& str,
    size_type pos = 0) const;
size_type find(const E *s, size_type pos, size_type n) const;
size_type find(const E *s, size_type pos = 0) const;
size_type find(E c, size_type pos = 0) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;
size_type rfind(const E *s, size_type pos,
```

```
        size_type n = npos) const;
    size_type rfind(const E *s, size_type pos = npos) const;
    size_type rfind(E c, size_type pos = npos) const;
    size_type find_first_of(const basic_string& str,
        size_type pos = 0) const;
    size_type find_first_of(const E *s, size_type pos,
        size_type n) const;
    size_type find_first_of(const E *s, size_type pos = 0) const;
    size_type find_first_of(E c, size_type pos = 0) const;
    size_type find_last_of(const basic_string& str,
        size_type pos = npos) const;
    size_type find_last_of(const E *s, size_type pos,
        size_type n = npos) con/t;
    size_type find_last_of(const E *s, size_type pos = npos) const;
    size_type find_last_of(E c, size_type pos = npos) const;
    size_type find_first_not_of(const basic_string& str,
        size_type pos = 0) const;
    size_type find_first_not_of(const E *s, size_type pos,
        size_type n) const;
    size_type find_first_not_of(const E *s, size_type pos = 0) const;
    size_type find_first_not_of(E c, size_type pos = 0) const;
    size_type find_last_not_of(const basic_string& str,
        size_type pos = npos) const;
    size_type find_last_not_of(const E *s, size_type pos,
         size_type n) const;
    size_type find_last_not_of(const E *s,
        size_type pos = npos) const;
    size_type find_last_not_of(E c, size_type pos = npos) const;
    basic_string substr(size_type pos = 0, size_type n = npos) const;
    int compare(const basic_string& str) const;
    int compare(size_type p0, size_type n0,
        const basic_string& str);
    int compare(size_type p0, size_type n0,
        const basic_string& str, size_type pos, size_type n);
    int compare(const E *s) const;
    int compare(size_type p0, size_type n0,
        const E *s) const;
    int compare(size_type p0, size_type n0,
        const E *s, size_type pos) const;
    A get_allocator() const;
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type E**. Such an element type must not require explicit construction or destruction, and it must be suitable for use as the E

parameter to basic_istream or basic_ostream. (A ``plain old data structure," or **POD**, from C generally meets this criterion.) The Standard C++ library provides two specializations of this template class, with the type definitions string, for elements of type *char*, and wstring, for elements of type wchar_t.

Various important properties of the elements in a basic_string specialization are described by the **class T**. A class that specifies these character traits must have the same external interface as an object of template class char_traits.

The object allocates and frees storage for the sequence it controls through a protected object named **allocator**, of **class A**. Such an allocator object must have the same external interface as an object of template class allocator. (Class char_traits has no provision for alternate addressing schemes, such as might be required to implement a far heap.) Note that allocator is *not* copied when the object is assigned.

The sequences controlled by an object of template class basic_string are usually called **strings**. These objects should not be confused, however, with the null-terminated C strings used throughout the Standard C++ library.

Many member functions require an **operand sequence** of elements of type E. You can specify such an operand sequence several ways:

- c -- a sequence of one element with value c
- n, c -- a repetition of n elements each with value c
- s -- a null-terminated sequence (such as a C string, for E of type *char*) beginning at s (which must not be a null pointer), where the terminating element is the value E(0) and is not part of the operand sequence
- s, n -- a sequence of n elements beginning at s (which must not be a null pointer)
- str -- the sequence specified by the basic_string object str
- str, pos, n -- the substring of the basic_string object str with up to n elements (or through the end of the string, whichever comes first) beginning at position pos
- first, last -- a sequence of elements delimited by the iterators first and last, in the range [first, last)

If a **position argument** (such as pos above) is beyond the end of the string on a call to a basic_string member function, the function reports an **out-of-range error** by throwing an object of class out_of_range.

If a function is asked to generate a sequence longer than max_size() elements, the function reports a **length error** by throwing an object of class length_error.

basic_string::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

basic_string::append

```
basic_string& append(const E *s);
basic_string& append(const E *s, size_type n);
basic_string& append(const basic_string& str,
    size_type pos, size_type n);
basic_string& append(const basic_string& str);
```

```
basic_string& append(size_type n, E c);
template<class InIt>
    basic_string& append(InIt first, InIt last);
```

The member template function appends the operand sequence to the end of the sequence controlled by `*this`, then returns `*this`.

In this implementation, if a translator does not support member template functions, hhe template is replaced by:

```
basic_string& append(const_iterator first, const_iterator last);
```

basic_string::assign

```
basic_string& assign(const E *s);
basic_string& assign(const E *s, size_type n);
basic_string& assign(const basic_string& str,
    size_type pos, size_type n);
basic_string& assign(const basic_string& str);
basic_string& assign(size_type n, E c);
template<class InIt>
    basic_string& assign(InIt first, InIt last);
```

The member functions each replaces the sequence controlled by `*this` with the operand sequence, then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
basic_string& assign(const_iterator first, const_iterator last);
```

basic_string::at

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

The member functions each returns a reference to the element of the controlled sequence at position `pos`, or it reports an out-of-range error.

basic_string::basic_string

```
basic_string(const E *s, const A& al = A());
basic_string(const E *s, size_type n, const A& al = A());
basic_string(const basic_string& rhs);
basic_string(const basic_string& rhs, size_type pos, size_type n,
    const A& al = A());
basic_string(size_type n, E c, const A& al = A());
explicit basic_string(const A& al = A());
template <class InIt>
    basic_string(InIt first, InIt last, const A& al = A());
```

The constructors each stores the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initializes the controlled sequence to a copy of the operand sequence specified by the

remaining operands. The `explicit` constructor specifies an empty initial controlled sequence.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
basic_string(const_iterator first, const_iterator last, const A& al = A());
```

basic_string::begin

```
const_iterator begin() const;
iterator begin();
```

The member functions each returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

basic_string::c_str

```
const E *c_str() const;
```

The member function returns a pointer to a non-modifiable C string constructed by adding a terminating null element (`E(0)`) to the controlled sequence. Calling any non-const member function for `*this` can invalidate the pointer.

basic_string::capacity

```
size_type capacity() const;
```

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as `size()`.

basic_string::char_type

```
typedef T::char_type char_type;
```

The type is a synonym for the template parameter `E`.

basic_string::compare

```
int compare(const basic_string& str) const;
int compare(size_type p0, size_type n0,
    const basic_string& str);
int compare(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
int compare(const E *s) const;
int compare(size_type p0, size_type n0,
    const E *s) const;
int compare(size_type p0, size_type n0,
    const E *s, size_type pos) const;
```

The member functions each compares up to `n0` elements of the controlled sequence beginning with position `p0`, or the entire controlled sequence if these arguments are not supplied, to the operand sequence. The function returns:

- a negative value if the first differing element in the controlled sequence compares less than the corresponding element in the operand sequence (as determined by `T::compare`), or if the two have a common prefix but the operand sequence is longer
- zero if the two compare equal element by element and are the same length
- a positive value otherwise

basic_string::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T1`.

basic_string::const_pointer

```
typedef A::const_pointer const_pointer;
```

The type describes an object that can serve as a constant pointer to an element of the controlled sequence.

basic_string::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

basic_string::const_reverse_iterator

```
typedef reverse_iterator<const_iterator, value_type,
    const_reference, const_pointer, difference_type>
        const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

basic_string::copy

```
size_type copy(E *s, size_type n, size_type pos = 0) const;
```

The member function copies up to `n` elements from the controlled sequence, beginning at position `pos`, to the array of `E` beginning at `s`. It returns the number of elements actually copied.

basic_string::data

```
const E *data() const;
```

The member function returns a pointer to the first element of the sequence (or, for an empty sequence, a non-null pointer that cannot be dereferenced).

basic_string::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

basic_string::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

basic_string::end

```
const_iterator end() const;
iterator end();
```

The member functions each returns a random-access iterator that points just beyond the end of the sequence.

basic_string::erase

```
iterator erase(iterator first, iterator last);
iterator erase(iterator it);
basic_string& erase(size_type p0 = 0, size_type n = npos);
```

The first member function removes the elements of the controlled sequence in the range `[first, last)`. The second member function removes the element of the controlled sequence pointed to by `it`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes up to `n` elements of the controlled sequence beginning at position `p0`, then returns `*this`.

basic_string::find

```
size_type find(E c, size_type pos = 0) const;
size_type find(const E *s, size_type pos = 0) const;
size_type find(const E *s, size_type pos, size_type n) const;
size_type find(const basic_string& str, size_type pos = 0) const;
```

The member functions each finds the first (lowest beginning position) subsequence in the controlled sequence, beginning on or after position `pos`, that matches the operand sequence specified by the remaining operands. If it succeeds, it returns the position where the matching subsequence begins. Otherwise, the function returns `npos`.

basic_string::find_first_not_of

```
size_type find_first_not_of(E c, size_type pos = 0) const;
size_type find_first_not_of(const E *s, size_type pos = 0) const;
size_type find_first_not_of(const E *s, size_type pos,
    size_type n) const;
size_type find_first_not_of(const basic_string& str,
```

```
    size_type pos = 0) const;
```

The member functions each finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches *none* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns `npos`.

basic_string::find_first_of

```
size_type find_first_of(E c, size_type pos = 0) const;
size_type find_first_of(const E *s, size_type pos = 0) const;
size_type find_first_of(const E *s, size_type pos, size_type n) const;
size_type find_first_of(const basic_string& str,
    size_type pos = 0) const;
```

The member functions each finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches *any* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns `npos`.

basic_string::find_last_not_of

```
size_type find_last_not_of(E c, size_type pos = npos) const;
size_type find_last_not_of(const E *s, size_type pos = npos) const;
size_type find_last_not_of(const E *s, size_type pos,  size_type n) const;
size_type find_last_not_of(const basic_string& str,
    size_type pos = npos) const;
```

The member functions each finds the last (highest position) element of the controlled sequence, at or before position `pos`, that matches *none* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns `npos`.

basic_string::find_last_of

```
size_type find_last_of(E c, size_type pos = npos) const;
size_type find_last_of(const E *s, size_type pos = npos) const;
size_type find_last_of(const E *s, size_type pos, size_type n = npos) const;
size_type find_last_of(const basic_string& str,
    size_type pos = npos) const;
```

The member functions each finds the last (highest position) element of the controlled sequence, at or before position `pos`, that matches *any* of the elements in the operand sequence specified by the remaining operands. If it succeeds, it returns the position. Otherwise, the function returns `npos`.

basic_string::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

basic_string::insert

```
basic_string& insert(size_type p0, const E *s);
basic_string& insert(size_type p0, const E *s, size_type n);
basic_string& insert(size_type p0,
    const basic_string& str);
basic_string& insert(size_type p0,
    const basic_string& str, size_type pos, size_type n);
basic_string& insert(size_type p0, size_type n, E c);
iterator insert(iterator it, E c);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
void insert(iterator it, size_type n, E c);
```

The member functions each inserts, before position `p0` or before the element pointed to by `it` in the controlled sequence, the operand sequence specified by the remaining operands. A function that returns a value returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(iterator it, const_iterator first, const_iterator last);
```

basic_string::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

basic_string::length

```
size_type length() const;
```

The member function returns the length of the controlled sequence (same as `size()`).

basic_string::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

basic_string::npos

```
static const size_type npos = -1;
```

The constant is the largest representable value of type `size_type`. It is assuredly larger than `max_size()`, hence it serves as either very large value or as a special code.

basic_string::operator+=

```
basic_string& operator+=(E c);
basic_string& operator+=(const E *s);
basic_string& operator+=(const basic_string& rhs);
```

The operators each appends the operand sequence to the end of the sequence controlled by `*this`, then returns `*this`.

basic_string::operator=

```
basic_string& operator=(E c);
basic_string& operator=(const E *s);
basic_string& operator=(const basic_string&nmp; rhs);
```

The operators each replaces the sequence controlled by `*this` with the operand sequence, then returns `*this`.

basic_string::operator[]

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member functions each returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the behavior is undefined.

basic_string::pointer

```
typedef A::pointer pointer;
```

The type describes an object that can serve as a pointer to an element of the controlled sequence.

basic_string::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

basic_string::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

basic_string::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member functions each returns a reverse iterator that points at the first element of the sequence (or just

beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

basic_string::replace

```
basic_string& replace(size_type p0, size_type n0,
    const E *s);
basic_string& replace(size_type p0, size_type n0,
    const E *s, size_type n);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str);
basic_string& replace(size_type p0, size_type n0,
    const basic_string& str, size_type pos, size_type n);
basic_string& replace(size_type p0, size_type n0,
    size_type n, E c);
basic_string& replace(iterator first0, iterator last0,
    const E *s);
basic_string& replace(iterator first0, iterator last0,
    const E *s, size_type n);
basic_string& replace(iterator first0, iterator last0,
    const basic_string& str);
basic_string& replace(iterator first0, iterator last0,
    size_type n, E c);
template<class InIt>
    basic_string& replace(iterator first0, iterator last0,
        InIt first, InIt last);
```

The member functions each replaces up to `n0` elements of the controlled sequence beginning with position `p0`, or the elements of the controlled sequence beginning with the one pointed to by `first`, up to but not including `last`. The replacement is the operand sequence specified by the remaining operands. The function then returns `*this`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
basic_string& replace(iterator first0, iterator last0, const_iterator first,
const_iterator last);
```

basic_string::reserve

```
void reserve(size_type n = 0);
```

The member function ensures that `capacity()` henceforth returns at least `n`.

basic_string::resize

```
void resize(size_type n, E c = E());
```

The member function ensures that `size()` henceforth returns `n`. If it must make the controlled sequence longer, it appends elements with value `c`.

basic_string::reverse_iterator

```
typedef reverse_iterator<iterator, value_type,
    reference, pointer, difference_type>
        reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence.

basic_string::rfind

```
size_type rfind(E c, size_type pos = npos) const;
size_type rfind(const E *s, size_type pos = npos) const;
size_type rfind(const E *s, size_type pos, size_type n = npos) const;
size_type rfind(const basic_string& str,
    size_type pos = npos) const;
```

The member functions each finds the last (highest beginning position) subsequence in the controlled sequence, beginning on or before position `pos`, that matches the operand sequence specified by the remaining operands. If it succeeds, it returns the position where the matching subsequence begins. Otherwise, the function returns `npos`.

basic_string::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

basic_string::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

basic_string::substr

```
basic_string substr(size_type pos = 0,
    size_type n = npos) const;
```

The member function returns an object whose controlled sequence is a copy of up to `n` elements of the controlled sequence beginning at position `pos`.

basic_string::swap

```
void swap(basic_string& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

basic_string::traits_type

```
typedef T traits_type;
```

The type is a synonym for the template parameter T.

basic_string::value_type

```
typedef A::value_type value_type;
```

The type is a synonym for the template parameter E.

char_traits

```
struct char_traits<E> {
    typedef E char_type;
    typedef T1 int_type;
    typedef T2 pos_type;
    typedef T3 off_type;
    typedef T4 state_type;
    static void assign(E& x, const E& y);
    static E *assign(E *x, size_t n, const E& y);
    static bool eq(const E& x, const E& y);
    static bool lt(const E& x, const E& y);
    static int compare(const E *x, const E *y, size_t n);
    static size_t length(const E *x);
    static E *copy(E *x, const E *y, size_t n);
    static E *move(E *x, const E *y, size_t n);
    static const E *find(const E *x, size_t n, const E& y);
    static E to_char_type(const int_type& ch);
    static int_type to_int_type(const E& c);
    static bool eq_int_type(const int_type& ch1, const int_type& ch2);
    static int_type eof();
    static int_type not_eof(const int_type& ch);
    };
```

The template class describes various **character traits** for type E. The template class basic_string as well as several iostreams template classes, including basic_ios, use this information to manipulate elements of type E. Such an element type must not require explicit construction or destruction. A bitwise copy has the same effect as an assignment.

char_traits::assign

```
static void assign(E& x, const E& y);
static E *assign(E *x, size_t n, const E& y);
```

The first static member function assigns y to x. The second static member function assigns y to each element

`X[N]` for `N` in the range `[0, N)`.

char_traits::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

char_traits::compare

```
static int compare(const E *x, const E *y, size_t n);
```

The static member function compares the sequence of length `n` beginning at `x`to the sequence of the same length beginning at `y`. The function returns:

- a negative value if the first differing element in `x` (as determined by `eq`) compares less than the corresponding element in `y` (as determined by `lt`)
- zero if the two compare equal element by element
- a positive value otherwise

char_traits::copy

```
static E *copy(E *x, const E *y, size_t n);
```

The static member function copies the sequence of `n` elements beginning at `y` to the array beginning at `x`, then returns `x`. The source and destination must not overlap.

char_traits::eof

```
static int_type eof();
```

The static member function returns a value that represents end-of-file (such as `EOF` or `WEOF`). If the value is also representable as type `E`, it must correspond to no *valid* value of that type.

char_traits::eq

```
static bool eq(const E& x, const E& y);
```

The static member function returns true if `x` compares equal to `y`.

char_traits::eq_int_type

```
static bool eq_int_type(const int_type& ch1, const int_type& ch2);
```

The static member function returns true if `ch1 == ch2`.

char_traits::find

```
static const E *find(const E *x, size_t n, const E& y);
```

The static member function determines the lowest `N` in the range `[0, n)` for which `eq(x[N], y)` is true. If successful, it returns `x + N`. Otherwise, it returns a null pointer.

char_traits::int_type

```
typedef T1 int_type;
```

The type is (typically) an integer type `T1` that describes an object that can represent any element of the controlled sequence as well as the value returned by `eof()`. It must be possible to type cast a value of type `E` to `int_type` then back to `E` without altering the original value. In addition, the expression `int_type('\0')` must yield the code that terminates a null-terminated strings for elements of type `E`. Also, the expression `int_type('\n')` must yield a suitable newline character of type `E`.

char_traits::length

```
static size_t length(const E *x);
```

The static member function returns the number of elements `N` in the sequence beginning at `x` up to but not including the element `x[N]` which compares equal to `E(0)`.

char_traits::lt

```
static bool lt(const E& x, const E& y);
```

The static member function returns true if `x` compares less than `y`.

char_traits::move

```
static E *move(E *x, const E *y, size_t n);
```

The static member function copies the sequence of `n` elements beginning at `y` to the array beginning at `x`, then returns `x`. The source and destination may overlap.

char_traits::not_eof

```
static int_type not_eof(const int_type& ch);
```

If `!eq_int_type( eof(), ch)`, the static member function returns `ch`. Otherwise, it returns a value other than `eof()`.

char_traits::off_type

```
typedef T3 off_type;
```

The type is a signed integer type `T3` that describes an object that can store a byte offset involved in various stream positioning operations. It is typically a synonym for `streamoff`, but in any case it has essentially the same properties as that type.

char_traits::pos_type

```
typedef T2 pos_type;
```

The type is an opaque type `T2` that describes an object that can store all the information needed to restore an

arbitrary file-position indicator within a stream. It is typically a synonym for `streampos`, but in any case it has essentially the same properties as that type.

char_traits::state_type

```
typedef T4 state_type;
```

The type is an opaque type `T4` that describes an object that can represent a conversion state. It is typically a synonym for `mbstate_t`, but in any case it has essentially the same properties as that type.

char_traits::to_char_type

```
static E to_char_type(const int_type& ch);
```

The static member function returns `ch`, represented as type `E`. A value of `ch` that cannot be so represented yields an unspecified result.

char_traits::to_int_type

```
static int_type to_int_type(const E& c);
```

The static member function returns `ch`, represented as type `int_type`. It should always be true that `to_char_type(to_int_type(c) == c` for any value of `c`.

char_traits<char>

```
class char_traits<char>;
```

The class is an explicit specialization of template class `char_traits` for elements of type *char*, (so that it can take advantage of library functions that manipulate objects of this type).

char_traits<wchar_t>

```
class char_traits<wchar_t>;
```

The class is an explicit specialization of template class `char_traits` for elements of type `wchar_t` (so that it can take advantage of library functions that manipulate objects of this type).

In this implementation, if `wchar_t` is not a unique type, then **`char_type`** is defined as an **encapsulated wchar_t**, so that `operator>>:` and `operator<<:` can be overloaded on `char_type&`.

getline

```
template<class E, class T, class A>
    basic_istream<E, T>& getline(
        basic_istream <E, T>& is,
        basic_string<E, T, A>& str);
template<class E, class T, class A>
    basic_istream<E, T>& getline(
```

```
        basic_istream <E, T>& is,
        basic_string<E, T, A>& str,
        E delim);
```

The first template function returns `getline(is, str, is.widen('\n'))`.

The second template function replaces the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. In order of testing, extraction stops:

1. at end of file
2. after the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence
3. after the function extracts `is.max_size()` elements, in which case the function calls `setstate(ios_base::failbit)`.

If the function extracts no elements, it calls `setstate(failbit)`. In any case, it returns `*this`.

operator+

```
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const basic_string<E, T, A>& lhs,
        E rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    basic_string<E, T, A> operator+(
        E lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator+` to concatenate two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).append(rhs)`.

operator!=

```
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
```

```
template<class E, class T, class A>
    bool operator!=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator!=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator!=` to compare two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) != 0`.

operator==

```
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator==(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator==(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator==` to compare two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) == 0`.

operator<

```
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator<` to compare two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) < 0`.

operator<<

```
template<class E, class T, class A>
    basic_ostream<E, T>& operator<<(
        basic_ostream <E, T>& os,
        const basic_string<E, T, A>& str);
```

The template function overloads `operator<<` to insert an object `str` of template class `basic_string` into the stream `os` The function effectively returns `os.write( str.c_str(), str.size())`.

operator<=

```
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator<=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator<=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator<=` to compare two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) <= 0`.

operator>

```
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator>` to compare two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) > 0`.

operator>=

```
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
template<class E, class T, class A>
    bool operator>=(
        const basic_string<E, T, A>& lhs,
        const E *rhs);
template<class E, class T, class A>
    bool operator>=(
        const E *lhs,
        const basic_string<E, T, A>& rhs);
```

The template functions each overloads `operator>=` to compare two objects of template class `basic_string`. All effectively return `basic_string<E, T, A>(lhs).compare(rhs) >= 0`.

operator>>

```
template<class E, class T, class A>
    basic_istream<E, T>& operator>>(
        basic_istream <E, T>& is,
        const basic_string<E, T, A>& str);
```

The template function overloads `operator>>` to replace the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. Extraction stops:

- at end of file
- after the function extracts `is.width()` elements, if that value is nonzero
- after the function extracts `is.max_size()` elements
- after the function extracts an element `c` for which `use_facet< ctype<E> >( getloc()). is( ctype<E>::space, c)` is true, in which case the character is put back

If the function extracts no elements, it calls `setstate(ios_base::failbit)`. In any case, it calls `width(0)` and returns `*this`.

string

```
typedef basic_string<char> string;
```

The type describes a specialization of template class `basic_string` specialized for elements of type *char*.

swap

```
template<class T, class A>
    void swap(
```

```
        const basic_string<E, T, A>& lhs,
        const basic_string<E, T, A>& rhs);
```

The template function executes `swap(lhs, rhs)`.

wstring

```
typedef basic_string<wchar_t> wstring;
```

The type describes a specialization of template class `basic_string` for elements of type `wchar_t`.

See also the **Table of Contents** and the **Index**.

<strstream>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    class strstreambuf;
    class istrstream;
    class ostrstream;
    class strstream;
    };
```

Include the iostreams standard header **<strstream>** to define several classes that support iostreams operations on sequences stored in an allocated array of *char* object. Such sequences are easily converted to and from C strings.

strstreambuf

```
class strstreambuf : public streambuf {
public:
    typedef E char_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit strstreambuf(streamsize n = 0);
    strstreambuf(void (*palloc)(size_t),
        void (*pfree)(void *));
    strstreambuf(char *gp, streamsize n,
        char *pp = 0);
    strstreambuf(signed char *gp, streamsize n,
        signed char *pp = 0);
    strstreambuf(unsigned char *gp, streamsize n,
        unsigned char *pp = 0);
    strstreambuf(const char *gp, streamsize n);
    strstreambuf(const signed char *gp, streamsize n);
    strstreambuf(const unsigned char *gp, streamsize n);
    void freeze(bool frz = true) const;
    char *str();
    streamsize pcount();
protected:
```

```
    virtual streampos seekoff(streamoff off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual streampos seekpos(streampos sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int underflow();
    virtual int pbackfail(int c = EOF);
    virtual int overflow(int c = EOF);
    };
```

The class describes a **stream buffer** that controls the transmission of elements to and from a sequence of elements stored in a *char* array object. Depending on how it is constructed, the object can be allocated, extended, and freed as necessary to accommodate changes in the sequence.

An object of class `strstreambuf` stores several bits of mode information as its **strstreambuf mode**. These bits indicate whether the controlled sequence:

- has been **allocated**, and hence needs to be freed eventually
- is **modifiable**
- is **extendable** by reallocating storage
- has been **frozen** and hence needs to be unfrozen before the object is destroyed, or freed (if allocated) by an agency other than the object

A controlled sequence that is frozen cannot be modified or extended, regardless of the state of these separate mode bits.

The object also stores pointers to two functions that control **strstreambuf allocation**. If these are null pointers, the object devises its own method of allocating and freeing storage for the controlled sequence.

basic_strstreambuf::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

strstreambuf::freeze

```
void freeze(bool frz = true) const;
```

If `frz` is true, the function alters the stored strstreambuf mode to make the controlled sequence frozen. Otherwise, it makes the controlled sequence not frozen.

basic_strstreambuf::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

basic_strstreambuf::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

strstreambuf::pcount

```
streamsize pcount();
```

The member function returns a count of the number of elements written to the controlled sequence. Specifically, if `pptr()` is a null pointer, the function returns zero. Otherwise, it returns `pptr() - pbase()`.

basic_strstreambuf::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

strstreambuf::strstreambuf

```
explicit strstreambuf(streamsize n = 0);
strstreambuf(void (*palloc)(size_t),
    void (*pfree)(void *));
strstreambuf(char *gp, streamsize n,
    char *pp = 0);
strstreambuf(signed char *gp, streamsize n,
    signed char *pp = 0);
strstreambuf(unsigned char *gp, streamsize n,
    unsigned char *pp = 0);
strstreambuf(const char *gp, streamsize n);
strstreambuf(const signed char *gp, streamsize n);
strstreambuf(const unsigned char *gp, streamsize n);
```

The first constructor stores a null pointer in all the pointers controlling the input buffer, the output buffer, and strstreambuf allocation. It sets the stored strstreambuf mode to make the controlled sequence modifiable and extendable.

The second constructor behaves much as the first, except that it stores `palloc` as the pointer to the function to call to allocate storage, and `pfree` as the pointer to the function to call to free that storage.

The three constructors:

```
strstreambuf(char *gp, streamsize n,
    char *pp = 0);
strstreambuf(signed char *gp, streamsize n,
    signed char *pp = 0);
```

```
strstreambuf(unsigned char *gp, streamsize n,
    unsigned char *pp = 0);
```

also behave much as the first, except that `gp` designates the array object used to hold the controlled sequence. (Hence, it must not be a null pointer.) The number of elements `N` in the array is determined as follows:

- If `(n > 0)`, then `N` is `n`.
- If `(n == 0)`, then `N` is `strlen((const char *)gp)`.
- If `(n < 0)`, then `N` is `INT_MAX`.

If `pp` is a null pointer, the function establishes just an input buffer, by executing:

```
setg(gp, gp, gp + N);
```

Otherwise, it establishes both input and output buffers, by executing:

```
setg(gp, gp, pp);
setp(pp, gp + N);
```

In this case, `pp` must be in the interval `[gp, gp + N]`.

Finally, the three constructors:

```
strstreambuf(const char *gp, streamsize n);
strstreambuf(const signed char *gp, streamsize n);
strstreambuf(const unsigned char *gp, streamsize n);
```

all behave the same as:

```
streambuf((char *)gp, n);
```

except that the stored mode makes the controlled sequence neither modifiable not extendable.

strstreambuf::overflow

```
virtual int overflow(int c = EOF);
```

If `c != EOF`, the protected virtual member function endeavors to insert the element `(char)c` into the output buffer. It can do so in various ways:

- If a write position is available, it can store the element into the write position and increment the next pointer for the output buffer.
- If the stored strstreambuf mode says the controlled sequence is modifiable, extendable, and not frozen, the function can make a write position available by allocating new for the output buffer. (Extending the output buffer this way also extends any associated input buffer.)

If the function cannot succeed, it returns `EOF`. Otherwise, if `c == EOF` it returns some value other than `EOF`. Otherwise, it returns `c`.

`strstreambuf::pbackfail`

```
virtual int pbackfail(int c = EOF);
```

The protected virtual member function endeavors to put back an element into the input buffer, then make it the current element (pointed to by the next pointer).

If `c == EOF`, the element to push back is effectively the one already in the stream before the current element. Otherwise, that element is replaced by `x = (char)c`. The function can put back an element in various ways:

- If a putback position is available, and the element stored there compares equal to `x`, it can simply decrement the next pointer for the input buffer.
- If a putback position is available, and if the strstreambuf mode says the controlled sequence is modifiable, the function can store `x` into the putback position and decrement the next pointer for the input buffer.

If the function cannot succeed, it returns `EOF`. Otherwise, if `c == EOF` it returns some value other than `EOF`. Otherwise, it returns `c`.

`strstreambuf::seekoff`

```
virtual streampos seekoff(streamoff off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `strstreambuf`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence.

The new position is determined as follows:

- If `way == ios_base::beg`, the new position is the beginning of the stream plus `off`.
- If `way == ios_base::cur`, the new position is the current stream position plus `off`.
- If `way == ios_base::end`, the new position is the end of the stream plus `off`.

If `which & ios_base::in` is nonzero and the input buffer exist, the function alters the next position to read in the input buffer. If `which & ios_base::out` is also nonzero, `way != ios_base::cur`, and the output buffer exists, the function also sets the next position to write to match the next position to read.

Otherwise, if `which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

If the function succeeds in altering the stream position(s), it returns the resultant stream position.

Otherwise, it fails and returns an invalid stream position.

strstreambuf::seekpos

```
virtual streampos seekpos(streampos sp,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

The protected virtual member function endeavors to alter the current positions for the controlled streams. For an object of class `strstreambuf`, a stream position consists purely of a stream offset. Offset zero designates the first element of the controlled sequence. The new position is determined by `sp`.

If `which & ios_base::in` is nonzero and the input buffer exists, the function alters the next position to read in the input buffer. (If `which & ios_base::out` is nonzero and the output buffer exists, the function also sets the next position to write to match the next position to read.) Otherwise, if `which & ios_base::out` is nonzero and the output buffer exists, the function alters the next position to write in the output buffer. Otherwise, the positioning operation fails. For a positioning operation to succeed, the resulting stream position must lie within the controlled sequence.

If the function succeeds in altering the stream position(s), it returns the resultant stream position. Otherwise, it fails and returns an invalid stream position.

strstreambuf::str

```
char *str();
```

The member function calls `freeze()`, then returns a pointer to the beginning of the controlled sequence. (Note that no terminating null element exists, unless you insert one explicitly.)

strstreambuf::underflow

```
virtual int underflow();
```

The protected virtual member function endeavors to extract the current element `c` from the input buffer, then advance the current stream position, and return the element as `(int)(unsigned char)c`. It can do so in only one way: If a read position is available, it takes `c` as the element stored in the read position and advances the next pointer for the input buffer.

If the function cannot succeed, it returns `EOF`. Otherwise, it returns the current element in the input stream, converted as described above.

istrstream

```
class istrstream : public istream {
public:
    typedef E char_type;
    typedef T::int_type int_type;
```

```
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    explicit istrstream(const char *s);
    explicit istrstream(char *s);
    istrstream(const char *s, streamsize n);
    istrstream(char *s, streamsize n);
    strstreambuf *rdbuf() const;
    char *str();
    };
```

The class describes an object that controls extraction of elements and encoded objects from a stream buffer of class `strstreambuf`. The object stores an ojbect of class `strstreambuf`.

basic_istrstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

basic_istrstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for `T::int_type`.

istrstream::istrstream

```
explicit istrstream(const char *s);
explicit istrstream(char *s);
istrstream(const char *s, streamsize n);
istrstream(char *s, streamsize n);
```

All the constructors initialize the base class by calling `istream(sb)`, where `sb` is the stored object of class `strstreambuf`. The first two constructors also initialize `sb` by calling `strstreambuf((const char *)s, 0)`. The remaining two constructors instead call `strstreambuf((const char *)s, n)`.

basic_istrstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for `T::off_type`.

basic_istrstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for `T::pos_type`.

istrstream::rdbuf

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `strstreambuf`.

istrstream::str

```
char *str();
```

The member function returns `rdbuf()-> str()`.

ostrstream

```
class ostrstream : public ostream {
public:
    typedef E char_type;
    typedef T::int_type int_type;
    typedef T::pos_type pos_type;
    typedef T::off_type off_type;
    ostrstream();
    ostrstream(char *s, streamsize n,
        ios_base::openmode mode = ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
    };
```

The class describes an object that controls insertion of elements and encoded objects into a stream buffer of class `strstreambuf`. The object stores an ojbect of class `strstreambuf`.

basic_ostrstream::char_type

```
typedef E char_type;
```

The type is a synonym for the template parameter `E`.

ostrstream::freeze

```
void freeze(bool frz = true)
```

The member function calls rdbuf()-> freeze(frz).

basic_ostrstream::int_type

```
typedef T::int_type int_type;
```

The type is a synonym for T::int_type.

basic_ostrstream::off_type

```
typedef T::off_type off_type;
```

The type is a synonym for T::off_type.

ostrstream::ostrstream

```
ostrstream();
ostrstream(char *s, streamsize n,
    ios_base::openmode mode = ios_base::out);
```

Both constructors initialize the base class by calling ostream(sb), where sb is the stored object of class strstreambuf. The first constructor also initializes sb by calling strstreambuf(). The second constructor initializes the base class one of two ways:

- If mode & ios_base::app == 0, then s must designate the first element of an array of n elements, and the constructor calls strstreambuf(s, n, s).
- Otherwise, s must designate the first element of an array of n elements that contains a C string whose first element is designated by s, and the constructor calls strstreambuf(s, n, s + strlen(s)).

ostrstream::pcount

```
streamsize pcount() const;
```

The member function returns rdbuf()-> pcount().

basic_ostrstream::pos_type

```
typedef T::pos_type pos_type;
```

The type is a synonym for T::pos_type.

ostrstream::rdbuf

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to strstreambuf.

ostrstream::str

```
char *str();
```

The member function returns rdbuf()-> str().

strstream

```
class strstream : public iostream {
public:
    strstream();
    strstream(char *s, streamsize n,
        ios_base::openmode mode = ios_base::in | ios_base::out);
    strstreambuf *rdbuf() const;
    void freeze(bool frz = true);
    char *str();
    streamsize pcount() const;
    };
```

The class describes an object that controls insertion and extraction of elements and encoded objects using a stream buffer of class strstreambuf. The object stores an ojbect of class strstreambuf.

strstream::freeze

```
void freeze(bool frz = true)
```

The member function calls rdbuf()-> freeze(zfrz).

strstream::pcount

```
streamsize pcount() const;
```

The member function returns rdbuf()-> pcount().

strstream::strstream

```
strstream();
strstream(char *s, streamsize n,
    ios_base::openmode mode = ios_base::in | ios_base::out);
```

Both constructors initialize the base class by calling `streambuf(sb)`, where `sb` is the stored object of class `strstreambuf`. The first constructor also initializes `sb` by calling `strstreambuf()`. The second constructor initializes the base class one of two ways:

- If `mode & ios_base::app == 0`, then `s` must designate the first element of an array of `n` elements, and the constructor calls `strstreambuf(s, n, s)`.
- Otherwise, `s` must designate the first element of an array of `n` elements that contains a C string whose first element is designated by `s`, and the constructor calls `strstreambuf(s, n, s + strlen(s))`.

strstream::rdbuf

```
strstreambuf *rdbuf() const
```

The member function returns the address of the stored stream buffer, of type pointer to `strstreambuf`.

strstream::str

```
char *str();
```

The member function returns `rdbuf()-> str()`.

See also the **Table of Contents** and the **Index**.

<typeinfo>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
    };
```

Include the standard header **<typeinfo>** to define several types associated with the type-identification operator **typeid**, which yields information about both static and dynamic types.

bad_cast

```
class bad_cast : public exception {
    };
```

The class describes an exception thrown to indicate that a **dynamic cast** expression, of the form:

`dynamic_cast<`*type*`>(`*expression*`)`

generated a null pointer to initialize a reference. The value returned by what() is implementation-defined. None of the member functions throw any exceptions.

bad_typeid

```
class bad_typeid : public exception {
    };
```

The class describes an exception thrown to indicate that a typeid operator encountered a null pointer. The value returned by what() is implementation-defined. None of the member functions throw any exceptions.

type_info

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
```

```
    bool before(const type_info& rhs) const;
    const char *name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
    };
```

The class describes type information generated within the program by the implementation. Objects of this class effectively store a pointer to a **name** for the type, and an encoded value suitable for comparing two types for equality or **collating order**. The names, encoded values, and collating order for types are all unspecified and may differ between program executions.

An expression of the form `typeid x` is the *only* way to construct a (temporary) `typeinfo` object. The class has only a private copy constructor. Since the assignment operator is also private, you cannot copy or assign objects of class `typeinfo` either.

type_info::operator!=

```
bool operator!=(const type_info& rhs) const;
```

The function returns `!(*this == rhs)`.

type_info::operator==

```
bool operator==(const type_info& rhs) const;
```

The function returns a nonzero value if `*this` and `rhs` represent the same type.

type_info::before

```
bool before(const type_info& rhs) const;
```

The function returns a nonzero value if `*this` precedes `rhs` in the collating order for types.

type_info::name

```
const char *name() const;
```

The function returns a C string which specifies the name of the type.

See also the **Table of Contents** and the **Index**.

<utility>


```
namespace std {
//  TEMPLATE CLASSES
template<class T, class U>
    struct pair;
//  TEMPLATE FUNCTIONS
template<class T, class U>
    pair<T, U> make_pair(const T& x, const U& y);
template<class T, class U>
    bool operator==(const pair<T, U>& x, const pair<T, U>& y);
template<class T, class U>
    bool operator!=(const pair<T, U>& x, const pair<T, U>& y);
template<class T, class U>
    bool operator<(const pair<T, U>& x, const pair<T, U>& y);
template<class T, class U>
    bool operator>(const pair<T, U>& x, const pair<T, U>& y);
template<class T, class U>
    bool operator<=(const pair<T, U>& x, const pair<T, U>& y);
template<class T, class U>
    bool operator>=(const pair<T, U>& x, const pair<T, U>& y);
namespace rel_ops {
    template<class T>
        bool operator!=(const T& x, const T& y);
    template<class T>
        bool operator<=(const T& x, const T& y);
    template<class T>
        bool operator>(const T& x, const T& y);
    template<class T>
        bool operator>=(const T& x, const T& y);
        };
    };
```

Include the STL standard header **<utility>** to define several templates of general use throughout the Standard Template Library.

If an implementation supports namespaces, four template operators are defined in the **rel_ops** namespace, nested within the `std` namespace. They define a **total ordering** on pairs of operands of the same type, given definitions of `operator==` and `operator<`. If you wish to make use of these

template operators, write the declaration:

```
using namespace std::rel_ops;
```

which promotes the template operators into the current namespace.

make_pair

```
template<class T, class U>
    pair<T, U> make_pair(const T& x, const U& y);
```

The template function returns pair<T, U>(x, y).

operator!=

```
template<class T>
    bool operator!=(const T& x, const T& y);
template<class T, class U>
    bool operator!=(const pair<T, U>& x, const pair<T, U>& y);
```

The template function returns !(x == y).

operator==

```
template<class T, class U>
    bool operator==(const pair<T, U>& x, const pair<T, U>& y);
```

The template function returns x.first == y.first && x.second == y.second.

operator<

```
template<class T, class U>
    bool operator<(const pair<T, U>& x, const pair<T, U>& y);
template<class T, class U>
    bool operator<(const pair<T, U>& x, const pair<T, U>& y);
```

The template function returns x.first < y.first || !(y.first < x.first && x.second < y.second.

operator<=

```
template<class T>
    bool operator<=(const T& x, const T& y);
template<class T, class U>
```

```
    bool operator<=(const pair<T, U>& x, const pair<T, U>& y);
```

The template function returns `!(y < x)`.

operator>

```
template<class T>
    bool operator>(const T& x, const T& y);
template<class T, class U>
    bool operator>(const pair<T, U>& x, const pair<T, U>& y);
```

The template function returns `y < x`.

operator>=

```
template<class T>
    bool operator>=(const T& x, const T& y);
template<class T, class U>
    bool operator>=(const pair<T, U>& x, const pair<T, U>& y);
```

The template function returns `!(x < y)`.

pair

```
template<class T, class U>
    struct pair {
    typedef T first_type;
    typedef U second_type
    T first;
    U second;
    pair();
    pair(const T& x, const U& y);
    template<class V, class W>
        pair(const pair<V, W>& pr);
    };
```

The template class stores a pair of objects, **`first`**, of type `T`, and **`second`**, of type `U`. The type definition **`first_type`**, is the same as the template parameter `T`, while **`second_type`**, is the same as the template parameter `U`.

The first (default) constructor initializes `first` to `T()` and `second` to `U()`. The second constructor initializes `first` to `x` and `second` to `y`. The third (template) constructor initializes `first` to `pr.first` and `second` to `pr.second`. `T` and `U` each need supply only a single-argument constructor and a destructor.

See also the **Table of Contents** and the **Index**.

<valarray>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

---

gslice · gslice_array · indirect_array · mask_array · slice · slice_array · valarray · valarray<bool>

abs · acos · asin · atan · atan2 · cos · cosh · exp · log · log10 · max · min · operator!= · operator% · operator& · operator&& · operator> · operator>> · operator>= · operator< · operator<< · operator<= · operator* · operator+ · operator- · operator/ · operator== · operator^ · operator| · operator|| · pow · sin · sinh · sqrt · tan · tanh

---

```
namespace std {
class slice;
class gslice;
template<class T>
    class valarray;
template<class T>
    class slice_array;
template<class T>
    class gslice_array;
template<class T>
    class mask_array;
template<class T>
    class indirect_array;
        // TEMPLATE FUNCTIONS
template<class T>
    valarray<T> operator*(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator*(const valarray<T> lhs,
        const T& rhs);
template<class T>
    valarray<T> operator*(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator/(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
```

```
    valarray<T> operator/(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator/(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator%(const valarray<T>& lhs,
        const vararray<T>& rhs);
template<class T>
    valarray<T> operator%(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator%(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator+(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator+(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator+(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator-(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator-(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator-(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator^(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator^(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator^(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator&(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator&(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator&(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator|(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator|(const valarray<T> lhs, const T& rhs);
```

```
template<class T>
    valarray<T> operator|(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator<<(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator<<(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator<<(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<T> operator>>(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator>>(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator>>(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator&&(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator&&(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator&&(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<biil> operator||(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator||(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator||(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator==(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator==(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator==(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator!=(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator!=(const valarray<T> lhs, const T& rhs);
template<class T>
```

```
    valarray<bool> operator!=(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator<(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator<(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator<(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator>=(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator>=(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator>=(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator>(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator>(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator>(const T& lhs, const valarray<T>& rhs);
template<class T>
    valarray<bool> operator<=(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator<=(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator<=(const T& lhs, const valarray<T>& rhs);
template<class T>
    T max(const valarray<T>& x);
template<class T>
    T min(const valarray<T>& x);
template<class T>
    valarray<T> abs(const valarray<T>& x);
template<class T>
    valarray<T> acos(const valarray<T>& x);
template<class T>
    valarray<T> asin(const valarray<T>& x);
template<class T>
    valarray<T> atan(const valarray<T>& x);
template<class T>
    valarray<T> atan2(const valarray<T>& x,
```

```
        const valarray<T>& y);
template<class T>
    valarray<T> atan2(const valarray<T> x, const T& y);
template<class T>
    valarray<T> atan2(const T& x, const valarray<T>& y);
template<class T>
    valarray<T> cos(const valarray<T>& x);
template<class T>
    valarray<T> cosh(const valarray<T>& x);
template<class T>
    valarray<T> exp(const valarray<T>& x);
template<class T>
    valarray<T> log(const valarray<T>& x);
template<class T>
    valarray&tt;T> log10(const valarray<T>& x);
template<class T>
    valarray<T> pow(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> pow(const valarray<T> x, const T& y);
template<class T>
    valarray<T> pow(const T& x, const valarray<T>& y);
template<class T>
    valarray<T> sin(const valarray<T>& x);
template<class T>
    valarray<T> sinh(const valarray<T>& x);
template<class T>
    valarray<T> sqrt(const valarray<T>& x);
template<class T>
    valarray<T> tan(const valarray<T>& x);
template<class T>
    valarray<T> tanh(const valarray<T>& x);
    };
```

Include the standard header **<valarray>** to define the template class valarray and numerous supporting template classes and functions. These template classes and functions are permitted unusual latitude, in the interest of improved performance. Specifically, any function returning `valarray<T>` may return an object of some other type `T'`. In that case, any function that accepts one or more arguments of type `valarray<T>` must have overloads that accept arbitrary combinations of those arguments, each replaced with an argument of type `T'`. (Put simply, the only way you can detect such a substitution is to go looking for it.)

abs

```
template<class T>
    valarray<T> abs(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the absolute value of x[I].

acos

```
template<class T>
    valarray<T> acos(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the arccosine of x[I].

asin

```
template<class T>
    valarray<T> asin(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the arcsine of x[I].

atan

```
template<class T>
    valarray<T> atan(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the arctangent of x[I].

atan2

```
template<class T>
    valarray<T> atan2(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> atan2(const valarray<T> x, const T& y);
template<class T>
    valarray<T> atan2(const T& x, const valarray<T>& y);
```

The first template function returns an object of class valarray<T>, each of whose elements I is the arctangent of x[I] / y[I]. The second template function stores in element I the arctangent of x[I]

`/ y`. The third template function stores in element `I` the arctangent of `x / y[I]`.

cos

```
template<class T>
    valarray<T> cos(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the cosine of `x[I]`.

cosh

```
template<class T>
    valarray<T> cosh(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic cosine of `x[I]`.

exp

```
template<class T>
    valarray<T> exp(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the exponential of `x[I]`.

gslice

```
class gslice {
public:
    gslice();
    gslice(size_t st,
        const valarray<size_t> len, const valarray<size_t> str);
    size_t start() const;
    const valarray<size_t> size() const;
    const valarray<size_t> stride() const;
    };
```

The class stores the parameters that characterize a `gslice_array` when an object of class `gslice` appears as a subscript for an object of class `valarray<T>`. The stored values include:

- a **starting index**
- a **length vector** of class `valarray<size_t>`

- a **stride vector** of class `valarray<size_t>`

The two vectors must have the same length.

gslice::gslice

```
gslice();
gslice(size_t st,
    const valarray<size_t> len, const valarray<size_t> str);
```

The default constructor stores zero for the starting index, and zero-length vectors for the length and stride vectors. The second constructor stores `st` for the starting index, `len` for the length vector, and `str` for the stride vector.

gslice::size

```
const valarray<size_t> size() const;
```

The member function returns the stored length vector.

gslice::start

```
size_t start() const;
```

The member function returns the stored starting index.

gslice::stride

```
const valarray<size_t> stride() const;
```

The member function returns the stored stride vector.

gslice_array

```
template<class T>
    class gslice_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
```

```
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
    void fill();
    };
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `gs` of class `gslice` which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `gslice_array<T>` object only by writing an expression of the form `x[gs]`. The member functions of class `gslice_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence is determined as follows. For a length vector `gs.size()` of length `N`, construct the index vector `valarray<size_t> idx(0, N)`. This designates the initial element of the sequence, whose index `k` within `x` is given by the mapping:

```
k = start;
for (size_t i = 0; i < gs.size()[i]; ++i)
    k += idx[i] * gs.stride()[i];
```

The successor to an index vector value is given by:

```
for (size_t i = N; 0 < i--; )
    if (++idx[i] < gs.size()[i])
        break;
    else
        idx[i] = 0;
```

For example:

```
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// x[gslice(3, len, str)] selects elements with indices
//   3, 5, 7, 10, 12, 14
```

indirect_array

```
template<class T>
    class indirect_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
```

```
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
    void fill();
    };
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `xa` of class `valarray<size_t>` which describes the sequence of elements to select from the `valarray<T>` object.

You construct an `indirect_array<T>` object only by writing an expression of the form `x[xa]`. The member functions of class `indirect_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of `xa.size()` elements, where element `i` becomes the index `xa[i]` within `x`. For example:

```
const size_t vi[] = {7, 5, 2, 3, 8};
// x[valarray<size_t>(vi, 5)] selects elements with indices
//   7, 5, 2, 3, 8
```

log

```
template<class T>
    valarray<T> log(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the natural logarithm of `x[I]`.

log10

```
template<class T>
    valarray<T> log10(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the base-10 logarithm of `x[I]`.

mask_array

```
template<class T>
    class mask_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
    void fill();
    };
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `ba` of class `valarray<bool>` which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `mask_array<T>` object only by writing an expression of the form `x[xa]`. The member functions of class `mask_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of at most `ba.size()` elements. An element `j` is included only if `ba[j]` is true. Thus, there are as many elements in the sequence as there are true elements in `ba`. If `i` is the index of the lowest true element in `ba`, then `x[i]` is element zero in the selected sequence. For example:

```
const bool vb[] = {false, false, true, true, false, true};
// x[valarray<bool>(vb, 56] selects eleeents with indices
//   2, 3, 5
```

max

```
template<class T>
    T max(const valarray<T>& x);
```

The template function returns the value of the largest element of `x`, by applying `operator<` between pairs of elements of class `T`.

min

```
template<class T>
    T min(const valarray<T>& x);
```

The template function returns the value of the smallest element of `x`, by applying `operator<` between pairs of elements of class `T`.

operator!=

```
template<class T>
    valarray<bool> operator!=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator!=(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator!=(const T& x, const valarray<T>& y);
```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] != y[I]`. The second template operator stores in element `I` `x[I] != y`. The third template operator stores in element `I` `x != y[I]`.

operator%

```
template<class T>
    valarray<T> operator%(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator%(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator%(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] % y[I]`. The second template operator stores in element `I` `x[I] % y`. The third template operator stores in element `I` `x % y[I]`.

operator&

```
template<class T>
    valarray<T> operator&(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator&(const valarray<T> lhs, const T& rhs);
template<class T>
```

```
    valarray<T> operator&(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I] & y[I]. The second template operator stores in element I x[I] & y. The third template operator stores in element I x & y[I].

operator&&

```
template<class T>
    valarray<bool> operator&&(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator&&(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator&&(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class valarray<bool>, each of whose elements I is x[I] && y[I]. The second template operator stores in element I x[I] && y. The third template operator stores in element I x && y[I].

operator>

```
template<class T>
    valarray<bool> operator>(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator>(const T& x, const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements I is x[I] > y[I]. The second template operator stores in element I x[I] > y. The third template operator stores in element I x > y[I].

operator>>

```
template<class T>
    valarray<T> operator>>(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator>>(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator>>(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I]

`>> y[I]`. The second template operator stores in element `I x[I] >> y`. The third template operator stores in element `I x >> y[I]`.

operator>=

```
template<class T>
    valarray<bool> operator>=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator>=(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator>=(const T& x, const valarray<T>& y);
```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] >= y[I]`. The second template operator stores in element `I x[I] >= y`. The third template operator stores in element `I x >= y[I]`.

operator<

```
template<class T>
    valarray<bool> operator<(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator<(const T& x, const valarray<T>& y);
```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] < y[I]`. The second template operator stores in element `I x[I] < y`. The third template operator stores in element `I x < y[I]`.

operator<<

```
template<class T>
    valarray<T> operator<<(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator<<(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator<<(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] << y[I]`. The second template operator stores in element `I x[I] << y`. The third template operator stores in element `I x << y[I]`.

operator<=

```
template<class T>
    valarray<bool> operator<=(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator<=(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator<=(const T& x, const valarray<T>& y);
```

The first template operator retrrns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] <= y[I]`. The second template operator stores in element `I` `x[I] <= y`. The third template operator stores in element `I` `x <= y[I]`.

operator*

```
template<class T>
    valarray<T> operator*(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator*(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator*(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] * y[I]`. The second template operator stores in element `I` `x[I] * y`. The third template operator stores in element `I` `x * y[I]`.

operator+

```
template<class T>
    valarray<T> operator+(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator+(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator+(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] + y[I]`. The second template operator stores in element `I` `x[I] + y`. The third template operator stores in element `I` `x + y[I]`.

operator-

```
template<class T>
    valarray<T> operator-(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator-(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator-(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I] - y[I]. The second template operator stores in element I x[I] - y. The third template operator stores in element I x - y[I].

operator/

```
template<class T>
    valarray<T> operator/(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator/(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator/(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class valarray<T>, each of whose elements I is x[I] / y[I]. The second template operator stores in element I x[I] / y. The third template operator stores in element I x / y[I].

operator==

```
template<class T>
    valarray<bool> operator==(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<bool> operator==(const valarray<T> x, const T& y);
template<class T>
    valarray<bool> operator==(const T& x  const valarray<T>& y);
```

The first template operator returns an object of class valarray<bool>, each of whose elements I is x[I] == y[I]. The second template operator stores in element I x[I] == y. The third template operator stores in element I x == y[I].

operator^

```
template<class T>
    valarray<T> operator^(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator^(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator^(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] ^ y[I]`. The second template operator stores in element `I` `x[I] ^ y`. The third template operator stores in element `I` `x ^ y[I]`.

operator|

```
template<class T>
    valarray<T> operator|(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<T> operator|(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<T> operator|(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<T>`, each of whose elements `I` is `x[I] | y[I]`. The second template operator stores in element `I` `x[I] | y`. The third template operator stores in element `I` `x | y[I]`.

operator||

```
template<class T>
    valarray<bool> operator||(const valarray<T>& lhs,
        const valarray<T>& rhs);
template<class T>
    valarray<bool> operator||(const valarray<T> lhs, const T& rhs);
template<class T>
    valarray<bool> operator||(const T& lhs, const valarray<T>& rhs);
```

The first template operator returns an object of class `valarray<bool>`, each of whose elements `I` is `x[I] || y[I]`. The second template operator stores in element `I` `x[I] || y`. The third template operator stores in element `I` `x || y[I]`.

pow

```
template<class T>
    valarray<T> pow(const valarray<T>& x,
        const valarray<T>& y);
template<class T>
    valarray<T> pow(const valarray<T> x, const T& y);
template<class T>
    valarray<T> pow(const T& x, const valarray<T>& y);
```

The first template function returns an object of class valarray<T>, each of whose elements I is x[I] raised to the y[I] power. The second template function stores in element I x[I] raised to the y power. The third template function stores in element I x raised to the y[I] power.

sin

```
template<class T>
    valarray<T> sin(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the sine of x[I].

sinh

```
template<class T>
    valarray<T> sinh(const valarray<T>& x);
```

The template function returns an object of class valarray<T>, each of whose elements I is the hyperbolic sine of x[I].

slice

```
class slice {
public:
    slice();
    slice(size_t st, size_t len, size_t str);
    size_t start() const;
    size_t size() const;
    size_t stride() const;
    };
```

The class stores the parameters that characterize a slice_array when an object of class slice appears as a subscript for an object of class valarray<T>. The stored values include:

- a **starting index**
- a **total length**
- a **stride**, or distance between subsequent indices

slice::slice

```
slice();
slice(size_t st,
        const valarray<size_t> len, const valarray<size_t> str);
```

The default constructor stores zeros for the starting index, total length, and stride. The second constructor stores `st` for the starting index, `len` for the total length, and `str` for the stride.

slice::size

```
size_t size() const;
```

The member function returns the stored total length.

slice::start

```
size_t start() const;
```

The member function returns the stored starting index.

slice::stride

```
size_t stride() const;
```

The member function returns the stored stride.

slice_array

```
template<class T>
    class slice_array {
public:
    typedef T value_type;
    void operator=(const valarray<T> x) const;
    void operator=(const T& x);
    void operator*=(const valarray<T> x) const;
    void operator/=(const valarray<T> x) const;
    void operator%=(const valarray<T> x) const;
    void operator+=(const valarray<T> x) const;
    void operator-=(const valarray<T> x) const;
    void operator^=(const valarray<T> x) const;
    void operator&=(const valarray<T> x) const;
```

```
    void operator|=(const valarray<T> x) const;
    void operator<<=(const valarray<T> x) const;
    void operator>>=(const valarray<T> x) const;
    void fill();
    };
```

The class describes an object that stores a reference to an object `x` of class `valarray<T>`, along with an object `sl` of class `slice` which describes the sequence of elements to select from the `valarray<T>` object.

You construct a `slice_array<T>` object only by writing an expression of the form `x[sl]`. The member functions of class `slice_array` then behave like the corresponding function signatures defined for `valarray<T>`, except that only the sequence of selected elements is affected.

The sequence consists of `sl.size()` elements, where element `i` becomes the index `sl.start() + i * sl.stride()` within `x`. For example:

```
// x[slice(2, 5, 3)] selects elements with indices
//   2, 5, 8, 11, 14
```

sqrt

```
template<class T>
    valarray<T> sqrt(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the square root of `x[I]`.

tan

```
template<class T>
    valarray<T> tan(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the tangent of `x[I]`.

tanh

```
template<class T>
    valarray<T> tanh(const valarray<T>& x);
```

The template function returns an object of class `valarray<T>`, each of whose elements `I` is the hyperbolic tangent of `x[I]`.

valarray

---

apply · cshift · fill · free · max · min · operator T * · operator! · operator%= · operator&= · operator>>= · operator<<= · operator*= · operator+ · operator+= · operator- · operator-= · operator/= · operator= · operator[] · operator^= · operator|= · operator~ · resize · shift · size · sum · valarray · value_type

---

```
template<class T>
    class valarray {
public:
    typedef T value_type;
    valarray();
    explicit valarray(size_t n);
    valarray(const T& val, size_t n));
    valarray(const T *p, size_t n);
    valarray(const slice_array<T>& sa);
    valarray(const gslice_array<T>& ga);
    valarray(const mask_array<T>& ma);
    valarray(const indirect_array<T>& ia);
    valarray<T>& operator=(const valarray<T>& va);
    valarray<T>& operator=(const T& x);
    valarray<T>& operator=(const slice_array<T>& sa);
    valarray<T>& operator=(const gslice_array<T>& ga);
    valarray<T>& operator=(const mask_array<T>& ma);
    valarray<T>& operator=(const indirect_array<T>& ia);
    T operator[](size_t n) const;
    T& operator[](size_t n);
    valarray<T> operator[](slice sa) const;
    slice_array<T> operator[](slice sa);
    valarray<T> operator[](const gslice& ga) const;
    gslice_array<T> operator[](const gslice& ga);
    valarray<T> operator[](const valarray<bool>& ba) const;
    mask_array<T> operator[](const valarray<bool>& ba);
    valarray<T> operator[](const valarray<size_t>& xa) const;
    indirect_array<T> operator[](const valarray<size_t>& xa);
    valarray<T> operator+();
    valarray<T> operator-();
    valarray<T> operator~();
```

```
valarray<bool> operator!();
valarray<T>& operator*=(const valarray<T>& x);
valarray<T>& operator*=(const T& x);
valarray<T>& operator/=(const valarray<T>& x);
valarray<T>& operator/=(const T& x);
valarray<T>& operator%=(const valarray<T>& x);
valarray<T>& operator%=(const T& x);
valarray<T>& operator+=(const valarray<T>& x);
valarray<T>& operator+=(const T& x);
valarray<T>& operator-=(const valarray<T>& x);
valarray<T>& operator-=(const T& x);
valarray<T>& operator^=(const valarray<T>& x);
valarray<T>& operator^=(const T& x);
valarray<T>& operator&=(const valarray<T>& x);
valarray<T>& operator&=(const T& x);
valarray<T>& operator|=(const valarray<T>& x);
valarray<T>& operator|=(const T& x);
valarray<T>& operator<<=(const valarray<T>& x);
valarray<T>& operator<<=(const T& x);
valarray<T>& operator>>=(const valarray<T>& x);
valarray<T>& operator>>=(const T& x);
operator T *();
operator const T *() const;
size_t size() const;
T sum() const;
T max() const;
T min() const;
valarray<T> shift(int n) const;
valarray<T> cshift(int n) const;
valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;
void fill(const T& val);
void free();
void resize(size_t n, const T& c = T());
};
```

The template class describes an object that controls a varying-length sequence of elements of **type T**. The sequence is stored as an **array of T**. It differs from template class `vector` in two important ways:

- It defines numerous arithmetic operations between corresponding elements of `valarray<T>` objects of the same type and length, such as `x = cos(y) + sin(z)`.
- It defines a variety of interesting ways to subscript a `valarray<T>` object, by overloading

operator[].

valarray::apply

```
valarray<T> apply(T fn(T)) const;
valarray<T> apply(T fn(const T&)) const;
```

The member function returns an object of class valarray<T>, of length size(), each of whose elements I is fn((*this)[I]).

valarray::cshift

```
valarray<T> cshift(int n) const;
```

The member function returns an object of class valarray<T>, of length size(), each of whose elements I is (*this)[(I + n) % size()]. Thus, if element zero is taken as the leftmost element, a positive value of n shifts the elements circularly left n places.

valarray::fill

```
void fill(const T& val);
```

The member function stores val in every element of *this.

valarray::free

```
void free();
```

The member function destroys all elements of *this, leaving an array of zero length.

valarray::size

```
size_t size() const;
```

The member function returns the number of elements in the array.

valarray::max

```
T max() const;
```

The member function returns the value of the largest element of *this, which must have nonzero length. If the length is greater than one, it compares values by applying operator< between pairs of corresponding elements of class T.

valarray::min

```
T min() const;
```

The member function returns the value of the smallest element of `*this`, which must have nonzero length. If the length is greater than one, it compares values by applying `operator<` between pairs of elements of class `T`.

valarray::operator T *

```
operator T *();
operator const T *() const;
```

Both member functions return a pointer to the first element of the controlled array, which must have at least one element.

valarray::operator!

```
valarray<bool> operator!();
```

The member operator returns an object of class `valarray<bool>`, of length `size()`, each of whose elements `I` is `!(*this)`.

valarray::operator%=

```
valarray<T>& operator%=(const valarray<T>& x);
valarray<T>& operator%=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] % x[I]`. It returns `*this`.

valarray::operator&=

```
valarray<T>& operator&=(const valarray<T>& x);
valarray<T>& operator&=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] & x[I]`. It returns `*this`.

valarray::operator>>=

```
valarray<T>& operator>>=(const valarray<T>& x);
valarray<T>& operator>>=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] >> x[I]`. It returns `*this`.

valarray::operator<<=

```
valarray<T>& operator<<=(const valarray<T>& x);
valarray<T>& operator<<=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] << x[I]`. It returns `*this`.

valarray::operator*=

```
valarray<T>& operator*=(const valarray<T>& x);
valarray<T>& operator*=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] * x[I]`. It returns `*this`.

valarray::operator+

```
valarray<T> operator+();
```

The member operator returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `(*this)[I]`.

valarray::operator+=

```
valarray<T>& operator+=(const valarray<T>& x);
valarray<T>& operator+=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] + x[I]`. It returns `*this`.

valarray::operator-

```
valarray<T> operator-();
```

The member operator returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `-(*this)[I]`.

valarray::operator-=

```
valarray<T>& operator-=(const valarray<T>& x);
valarray<T>& operator-=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] - x[I]`. It returns `*this`.

valarray::operator/=

```
valarray<T>& operator/=(const valarray<T>& x);
valarray<T>& operator/=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] / x[I]`. It returns `*this`.

valarray::operator=

```
valarray<T>& operator=(const valarray<T>& va);
    valarray<T>& operator=(const T& x);
valarray<T>& operator=(const slice_array<T>& sa);
valarray<T>& operator=(const gslice_array<T>& ga);
valarray<T>& operator=(const mask_array<T>& ma);
valarray<T>& operator=(const indirect_array<T>& ia);
```

The first member operator replaces the controlled sequence with a copy of the sequence controlled by `va`. The second member operator replaces each element of the controlled sequence with a copy of `x`. The remaining member operators replace those elements of the controlled sequence selected by their arguments, which are generated only by `operator[]`. If the value of a member in the replacement controlled sequence depends on a member in the initial controlled sequence, the result is undefined.

If the length of the controlled sequence changes, the result is generally undefined. In this

in this implementation, however, the effect is merely to invalidate any pointers or references to elements in the controlled sequence.

valarray::operator[]

```
T& operator[](size_t n);
slice_array<T> operator[](slice sa);
gslice_array<T> operator[](const gslice& ga);
mask_array<T> operator[](const valarray<bool>& ba);
indirect_array<T> operator[](const valarray<size_t>& xa);

T operator[](size_t n) const;
valarray<T> operator[](slice sa) const;
valarray<T> operator[](const gslice& ga) const;
valarray<T> operator[](const valarray<bool>& ba) const;
valarray<T> operator[](const valarray<size_t>& xa) const;
```

The member operator is overloaded to provide several ways to select sequences of elements from among those controlled by `*this`. The first group of five member operators work in conjunction with various overloads of `operator=` (and other assigning operators) to allow selective replacement (slicing) of the controlled sequence. The selected elements must exist.

The first member operator selects element `n`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
v0[3] = 'A';
// v0 == valarray<char>("abcAefghijklmnop", 16)
```

The second member operator selects those elements of the controlled sequence designated by `sa`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
v0[slice(2, 5, 3)] = v1;
// v0 == valarray<char>("abAdeBghCjkDmnEp", 16)
```

The third member operator selects those elements of the controlled sequence designated by `ga`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDEF", 6);
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
v0[gslice(3, len, str)] = v1;
// v0 == valarray<char>("abcAeBgCijDlEnFp", 16)
```

The fourth member operator selects those elements of the controlled sequence designated by `ma`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABC", 3);
const bool vb[] = {false, false, true, true, false, true};
v0[valarray<bool>(vb, 6)] = v1;
// v0 == valarray<char>("abABeCghijklmnop", 16)
```

The fifth member operator selects those elements of the controlled sequence designated by `ia`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
valarray<char> v1("ABCDE", 5);
const size_t vi[] = {7, 5, 2, 3, 8};
v0[valarray<size_t>(vi, 5)] = v1;
// v0 == valarray<char>("abCDeBgAEjklmnop", 16)
```

The second group of five member operators each construct an object that represents the value(s) selected. The selected elements must exist.

The sixth member operator returns the value of element `n`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
// v0[3] returns 'd'
```

The seventh member operator returns an object of class `valarray<T>` containing those elements of the controlled sequence designated by `sa`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
// v0[slice(2, 5, 3)] returns valarray<char>("cfilo", 5)
```

The eighth member operator selects those elements of the controlled sequence designated by `ga`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
const size_t lv[] = {2, 3};
const size_t dv[] = {7, 2};
const valarray<size_t> len(lv, 2), str(dv, 2);
// v0[gslice(3, len, str)] returns valarray<char>("dfhkmo", 6)
```

The ninth member operator selects those elements of the controlled sequence designated by `ma`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
const bool vb[] = {false, false, true, true, false, true};
// v0[valarray<bool>(vb, 6)] returns valarray<char>("cdf", 3)
```

The last member operator selects those elements of the controlled sequence designated by `ia`. For example:

```
valarray<char> v0("abcdefghijklmnop", 16);
const size_t vi[] = {7, 5, 2, 3, 8};
// v0[valarray<size_t>(vi, 5)] returns valarray<char>("hfcdi", 3)
```

valarray::operator^=

```
valarray<T>& operator^=(const valarray<T>& x);
valarray<T>& operator^=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] ^ x[I]`. It returns `*this`.

valarray::operator|=

```
valarray<T>& operator|=(const valarray<T>& x);
valarray<T>& operator|=(const T& x);
```

The member operator replaces each element `I` of `*this` with `(*this)[I] | x[I]`. It returns `*this`.

valarray::operator~

```
valarray<T> operator~();
```

The member operator returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is `~(*this)[I]`.

valarray::resize

```
void resize(size_t n, const T& c = T());
```

The member function ensures that `size()` henceforth returns `n`. If it must make the controlled sequence longer, it appends elements with value `c`. Any pointers or references to elements in the controlled sequence are invalidated.

valarray::shift

```
valarray<T> shift(int n) const;
```

The member function returns an object of class `valarray<T>`, of length `size()`, each of whose elements `I` is either `(*this)[I + n]`, if `I + n` is a valid subscript, or `T()`. Thus, if element zero is taken as the leftmost element, a positive value of `n` shifts the elements left `n` places, with zero fill.

valarray::sum

```
T sum() const;
```

The member function returns the sum of all elements of `*this`, which must have nonzero length. If the length is greater than one, it adds values to the sum by applying `operator+=` between pairs of elements of class `T`.

valarray::valarray

```
valarray();
explicit valarray(size_t n);
valarray(const T& val, size_t n));
valarray(const T *p, size_t n);
valarray(const slice_array<T>& sa);
valarray(const gslice_array<T>& ga);
```

```
valarray(const mask_array<T>& ma);
valarray(const indirect_array<T>& ia);
```

The first (default) constructor initializes the object to an empty array. The next three constructors each initialize the object to an array of `n` elements as follows:

- For `explicit valarray(size_t n)`, each element is initialized with the default constructor.
- For `valarray(const T& val, size_t n))`, each element is initialized with `val`.
- For `valarray(const T *p, size_t n)`, the element at position `I` is initialized with `p[I]`.

Each of the remaining constructors initializes the object to a `valarray<T>` object determined by the argument.

valarray::value_type

```
typedef T value_type;
```

The type is a synonym for the template parameter `T`.

valarray<bool>

```
class valarray<bool>;
```

In this implementation, if `bool` is not a distinct type, the specialization `valarray<bool>` should be referred to by the synonym `_Boolarray`.

See also the **Table of Contents** and the **Index**.

<vector>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class T, class A>
    class vector;
template<class A>
    class vector<bool, A>;
//    TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
template<class T, class A>
    void swap(
        const vector<T, A>& lhs,
        const vector<T, A>& rhs);
    };
```

Include the STL standard header **<vector>** to define the container template class `vector` and three supporting templates.

operator!=

```
template<class T, class A>
    bool operator!=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `vector`. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `vector`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator>(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
    void swap(
        const vector <T, A>& lhs,
        const vector <T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

vector

---

allocator_type · assign · at · back · begin · capacity · clear · const_iterator · const_reference · const_reverse_iterator · difference_type · empty · end · erase · front · get_allocator · insert · iterator · max_size · operator[] · pop_back · push_back · rbegin · reference · rend · reserve · resize · reverse_iterator · size · size_type · swap · value_type · vector

---

```
template<class T, class A = allocator<T> >
    class vector {
public:
    typedef A allocator_type;
    typedef A::size_type size_type;
```

```
typedef A::difference_type difference_type;
typedef A::reference reference;
typedef A::const_reference const_reference;
typedef A::value_type value_type;
typedef T0 iterator;
typedef T1 const_iterator;
typedef reverse_iterator<iterator, value_type,
    reference, A::pointer, difference_type>
        reverse_iterator;
typedef reverse_iterator<const_iterator, value_type,
    const_reference, A::const_pointer, difference_type>
        const_reverse_iterator;
explicit vector(const A& al = A());
explicit vector(size_type n, const T& v = T(), const A& al = A());
vector(const vector& x);
template<class InIt>
    vector(InIt first, InIt last, const A& al = A());
void reserve(size_type n);
size_type capacity() const;
iterator begin();
const_iterator begin() const;
iterator end();
iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
void resize(size_type n, T x = T());
size_type size() const;
size_type max_size() const;
bool empty() const;
A get_allocator() const;
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
const_reference operator[](size_type pos);
reference front();
const_reference front() const;
reference back();
const_reference back() const;
void push_back(const T& x);
```

```
    void pop_back();
    template<class InIt>
        void assign(InIt first, InIt last);
    template<class Size, class T2>
        void assign(Size n, const T2& x = T2());
    iterator insert(iterator it, const T& x = T());
    void insert(iterator it, size_type n, const T& x);
    template<class InIt>
        void insert(iterator it, InIt first, InIt last);
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(vector x);
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements of **type `T`**. The sequence is stored as an **array of `T`**.

The object allocates and frees storage for the sequence it controls through a protected object named **`allocator`**, of **class `A`**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned.

Vector reallocation occurs when a member function must grow the controlled sequence beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the controlled sequence become **invalid**.

vector::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter `A`.

vector::assign

```
template<class InIt>
    void assign(InIt first, InIt last);
template<class Size, class T2>
    void assign(Size n, const T2& x = T2());
```

The first member template function replaces the sequence controlled by `*this` with the sequence `[first, last)`. The second member template function replaces the sequence controlled by `*this` with a repetition of `n` elements of value `x`.

In this implementation, if a translator does not support member template functions, the templates are replaced by:

```
void assign(const_iterator first, const_iterator last);
void assign(size_type n, const T& x = T());
```

vector::at

```
const_reference at(size_type pos) const;
reference at(size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the function throws an object of class `out_of_range`.

vector::back

```
reference back();
const_reference back() const;
```

The member function returns a reference to the last element of the controlled sequence, which must be non-empty.

vector::begin

```
const_iterator begin() const;
iterator begin();
```

The member function returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

vector::capacity

```
size_type capacity() const;
```

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as `size()`.

vector::clear

```
void clear() const;
```

The member function calls `erase( begin(), end())`.

vector::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled

sequence. It is described here as a synonym for the unspecified type `T1`.

vector::const_reference

```
typedef A::const_reference const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence.

vector::const_reverse_iterator

```
typedef reverse_iterator<const_iterator, value_type,
    const_reference, A::const_pointer, difference_type>
        const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence.

vector::difference_type

```
typedef A::difference_type difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence.

vector::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

vector::end

```
const_iterator end() const;
iterator end();
```

The member function returns a random-access iterator that points just beyond the end of the sequence.

vector::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

Erasing `N` elements causes `N` destructor calls and an assignment for each of the elements between the

insertion point and the end of the sequence. No reallocation occurs, so iterators and references become invalid only from the first element erased through the end of the sequence.

vector::front

```
reference front();
const_reference front() const;
```

The member function returns a reference to the first element of the controlled sequence, which must be non-empty.

vector::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

vector::insert

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
template<class InIt>
    void insert(iterator it, InIt first, InIt last);
```

Each of the member functions inserts, before the element pointed to by `it` in the controlled sequence, a sequence specified by the remaining operands. The first member function inserts a single element with value `x` and returns an iterator that points to the newly inserted element. The second member function inserts a repetition of `n` elements of value `x`. The member template function inserts the sequence `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
void insert(iterator it, const_iterator first, const_iterator last);
```

When inserting a single element, the number of element copies is linear in the number of elements between the insertion point and the end of the sequence. When inserting a single element at the end of the sequence, the amortized number of element copies is constant. When inserting `N` elements, the number of element copies is linear in `N` plus the number of elements between the insertion point and the end of the sequence -- except when the template member is specialized for `InIt` an input iterator, which behaves like `N` single insertions.

If reallocation occurs, the size of the controlled sequence at least doubles, and all iterators and references become invalid. If no reallocation occurs, iterators become invalid only from the point of insertion through the end of the sequence.

vector::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

vector::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

vector::operator[]

```
const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
```

The member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the behavior is undefined.

vector::pop_back

```
void pop_back();
```

The member function removes the last element of the controlled sequence, which must be non-empty.

vector::push_back

```
void push_back(const T& x);
```

The member function inserts an element with value `x` at the end of the controlled sequence.

vector::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

vector::reference

```
typedef A::reference reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence.

vector::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

vector::reserve

```
void reserve(size_type n);
```

The member function ensures that `capacity()` henceforth returns at least `n`.

vector::resize

```
void resize(size_type n, T x = T());
```

The member function ensures that `size()` henceforth returns `n`. If it must make the controlled sequence longer, it appends elements with value `x`.

vector::reverse_iterator

```
typedef reverse_iterator<iterator, value_type,
    reference, A::pointer, difference_type>
        reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence.

vector::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

vector::size_type

```
typedef A::size_type size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence.

vector::swap

```
void swap(vector& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

vector::value_type

```
typedef A::value_type value_type;
```

The type is a synonym for the template parameter T.

vector::vector

```
explicit vector(const A& al = A());
explicit vector(size_type n, const T& v = T(), const A& al = A());
vector(const vector& x);
template<class InIt>
    vector(InIt first, InIt last, const A& al = A());
```

All constructors store the allocator object `al` (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence. The first constructor specifies an empty initial controlled sequence. The second constructor specifies a repetition of `n` elements of value `x`. The third constructor specifies a copy of the sequence controlled by `x`. The member template constructor specifies the sequence `[first, last)`.

In this implementation, if a translator does not support member template functions, the template is replaced by:

```
vector(const_iterator first, const_iterator last, const A& al = A());
```

If the member template constructor is specialized for forward iterators, the constructor copies at most `2 * N` elements to initialize a sequence of `N` elements. It reallocates the sequence at most `ceil(log2(N))` times. All other constructors copy `N` elements and perform no interim reallocation.

vector<bool, A>

```
template<class A = allocator<bool> >
    class vector<bool, A> {
    class reference;
    typedef bool const_reference;
    typedef T0 iterator;
    typedef T1 const_iterator;
    void flip();
    static void swap(reference x, reference y);
// rest same as template class vector
    };
```

The class is a partial specialization of template class `vector` for elements of type `bool`. It alters the definition of four member types (to optimize the packing and unpacking of elements) and adds two member functions. Its behavior is otherwise the same as for template class `vector`.

In this implementation, if partial specializations are not supported or if `bool` is not a distinct type, the class should be referred to by the synonym **`_Bvector`**.

vector<bool, A>::const_iterator

```
typedef T1 const_iterator;
```

The type describes an object that can serve as a constant random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T1`.

vector<bool, A>::const_reference

```
typedef bool const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence, in this case `bool`.

vector<bool, A>::flip

```
void flip();
```

The member function inverts the values of all the members of the controlled sequence.

vector<bool, A>::iterator

```
typedef T0 iterator;
```

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T0`.

vector<bool, A>::reference

```
class reference {
public:
    reference& operator=(const reference& x);
    reference& operator=(bool x);
    void flip();
    bool operator~() const;
    operator bool() const;
    };
```

The type describes an object that can serve as a reference to an element of the controlled sequence. Specifically, for two objects `x` and `y` of class `reference`:

- **`bool(x)`** yields the value of the element designated by `x`
- **`~x`** yields the inverted value of the element designated by `x`
- **`x.flip()`** inverts the value stored in `x`

- **`y = bool(x)`** and **`y = x`** both assign the value of the element designated by `x` to the element designated by `y`

It is unspecified how member functions of class `vector<bool, A>` construct objects of class `reference` that designate elements of a controlled sequence. The default constructor for class `reference` generates an object that refers to no such element.

`vector<bool, A>::swap`

```
void swap(reference x, reference y);
```

The static member function swaps the members of the controlled sequences designated by `x` and `y`.

See also the **Table of Contents** and the **Index**.

<cassert>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <assert.h>
    };
```

Include the standard header **<cassert>** to effectively include the standard header <assert.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cctype>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <ctype.h>
    };
```

Include the standard header **<cctype>** to effectively include the standard header <ctype.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cerrno>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <errno.h>
    };
```

Include the standard header **<cerrno>** to effectively include the standard header <errno.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cfloat>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <float.h>
    };
```

Include the standard header **<cfloat>** to effectively include the standard header <float.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<ciso646>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <iso646.h>
    };
```

Include the standard header **<ciso646>** to effectively include the standard header <iso646.h> within the std namespace (for what it's worth).

See also the **Table of Contents** and the **Index**.

<climits>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <limits.h>
    };
```

Include the standard header **<climits>** to effectively include the standard header <limits.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<clocale>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <locale.h>
    };
```

Include the standard header **<clocale>** to effectively include the standard header <locale.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cmath>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <math.h>
    };
```

Include the standard header **<cmath>** to effectively include the standard header <math.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<csetjmp>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <setjmp.h>
    };
```

Include the standard header **<csetjmp>** to effectively include the standard header <setjmp.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<csignal>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <signal.h>
    };
```

Include the standard header **<csignal>** to effectively include the standard header <signal.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cstdarg>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <stdarg.h>
    };
```

Include the standard header **<cstdarg>** to effectively include the standard header <stdarg.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cstddef>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <stddef.h>
    };
```

Include the standard header **<cstddef>** to effectively include the standard header <stddef.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cstdio>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <stdio.h>
    };
```

Include the standard header **<cstdio>** to effectively include the standard header <stdio.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cstdlib>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <stdlib.h>
    };
```

Include the standard header **<cstdlib>** to effectively include the standard header <stdlib.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cstring>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <string.h>
    };
```

Include the standard header **<cstring>** to effectively include the standard header <string.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<ctime>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <time.h>
    };
```

Include the standard header **<ctime>** to effectively include the standard header <time.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cwchar>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <wchar.h>
    };
```

Include the standard header **<cwchar>** to effectively include the standard header <wchar.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<cwctype>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
#include <wctype.h>
    };
```

Include the standard header **<cwctype>** to effectively include the standard header <wctype.h> within the std namespace.

See also the **Table of Contents** and the **Index**.

<assert.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#undef assert
#if defined NDEBUG
#define assert(test) (void)0
#else
#define assert(test) <void expression>
#endif
```

Include the standard header **<assert.h>** to define the macro assert, which is useful for diagnosing logic errors in the program. You can eliminate the testing code produced by the macro assert without removing the macro references from the program by defining the macro **NDEBUG** in the program before you include <assert.h>. Each time the program includes this header, it redetermines the definition of the macro assert.

assert

```
#undef assert
#if defined NDEBUG
#define assert(test) (void)0
#else
#define assert(test) <void expression>
#endif
```

If the *int* expression test equals zero, the macro writes to stderr a diagnostic message that includes:

- the text of test
- the source filename (the predefined macro __FILE__)
- the source line number (the predefined macro __LINE__)

It then calls abort.

You can write the macro assert in the program in any side-effects context.

See also the **Table of Contents** and the **Index**.

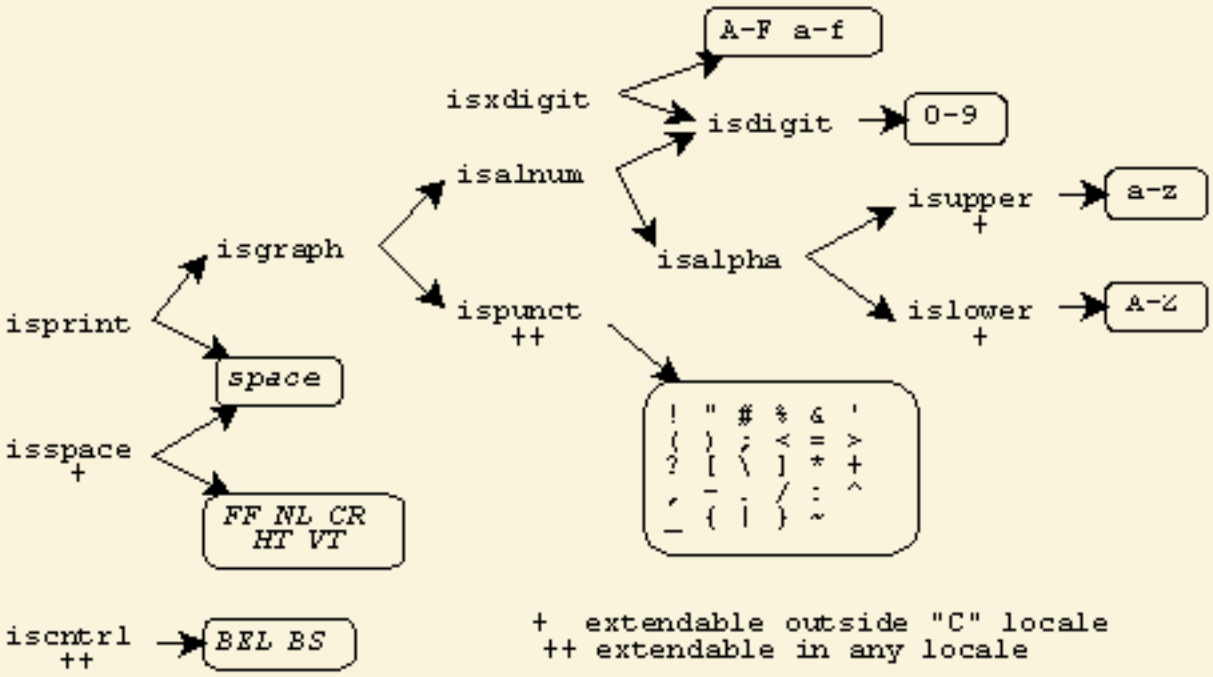
<ctype.h>

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int tolower(int c);
int toupper(int c);
```

Include the standard header **<ctype.h>** to declare several functions that are useful for classifying and mapping codes from the target character set. Every function that has a parameter of type *int* can accept the value of the macro EOF or any value representable as type *unsigned char*. Thus, the argument can be the value returned by any of the functions fgetc, fputc, getc, getchar, putc, putchar, tolower, toupper, and ungetc. You must not call these functions with other argument values.

Other library functions use these functions. The function scanf, for example, uses the function isspace to determine valid white space within an input field.

The **character classification** functions are strongly interrelated. Many are defined in terms of other functions. For characters in the basic C character set, here are the dependencies between these functions:



The diagram tells you that the function `isprint` returns nonzero for *`space`* or for any character for which the function `isgraph` returns nonzero. The function `isgraph`, in turn, returns nonzero for any character for which either the function `isalnum` or the function `ispunct` returns nonzero. The function `isdigit`, on the other hand, returns nonzero only for the digits `0-9`.

An implementation can define additional characters that return nonzero for some of these functions. Any character set can contain additional characters that return nonzero for:

- `ispunct` (provided the characters cause `isalnum` to return zero)
- `iscntrl` (provided the characters cause `isprint` to return zero)

The diagram indicates with `++` those functions that can define additional characters in any character set. Moreover, locales other than the `"C"` locale can define additional characters that return nonzero for:

- `isalpha`, `isupper`, and `islower` (provided the characters cause `iscntrl`, `isdigit`, `ispunct`, and `isspace` to return zero)
- `isspace` (provided the characters cause `isprint` to return zero)

The diagram indicates with `+` those functions that can define additional characters in locales other than the `"C"` locale.

Note that an implementation can define locales other than the `"C"` locale in which a character can cause `isalpha` (and hence `isalnum`) to return nonzero, yet still cause `isupper` and `islower` to return zero.

isalnum

```
int isalnum(int c);
```

The function returns nonzero if `c` is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
o 1 2 3 4 5 6 7 8 9
```

or any other locale-specific alphabetic character.

isalpha

```
int isalpha(int c);
```

The function returns nonzero if `c` is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

or any other locale-specific alphabetic character.

iscntrl

```
int iscntrl(int c);
```

The function returns nonzero if `c` is any of:

BEL BS CR FF HT NL VT

or any other implementation-defined control character.

isdigit

```
int isdigit(int c);
```

The function returns nonzero if `c` is any of:

```
0 1 2 3 4 5 6 7 8 9
```

isgraph

```
int isgraph(int c);
```

The function returns nonzero if `c` is any character for which either `isalnum` or `ispunct` returns nonzero.

islower

```
int islower(int c);
```

The function returns nonzero if `c` is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

or any other locale-specific lowercase character.

isprint

```
int isprint(int c);
```

The function returns nonzero if `c` is *space* or a character for which `isgraph` returns nonzero.

ispunct

```
int ispunct(int c);
```

The function returns nonzero if `c` is any of:

```
! " # % & ' ( ) ; <
= > ? [ \ ] * + , -
. / : ^ _ { | } ~
```

or any other implementation-defined punctuation character.

isspace

```
int isspace(int c);
```

The function returns nonzero if `c` is any of:

CR FF HT NL VT space

or any other locale-specific space character.

isupper

```
int isupper(int c);
```

The function returns nonzero if `c` is any of:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

or any other locale-specific uppercase character.

isxdigit

```
int isxdigit(int c);
```

The function returns nonzero if `c` is any of:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
```

tolower

```
int tolower(int c);
```

The function returns the corresponding lowercase letter if one exists and if `isupper(c)`; otherwise, it returns `c`.

toupper

```
int toupper(int c);
```

The function returns the corresponding uppercase letter if one exists and if `islower(c)`; otherwise, it returns `c`.

See also the **Table of Contents** and the **Index**.

<errno.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define EDOM <#if expression>
#define EILSEQ <#if expression>
#define ERANGE <#if expression>
#define errno <int modifiable lvalue>
```

Include the standard header **<errno.h>** to test the value stored in errno by certain library functions. At program startup, the value stored is zero. Library functions store only values greater than zero. Any library function can alter the value stored, but only those cases where a library function is explicitly required to store a value are documented here.

To test whether a library function stores a value in errno, the program should store the value zero there immediately before it calls the library function. An implementation can define additional macros in this standard header that you can test for equality with the value stored. All these additional macros have names that begin with E.

EDOM

```
#define EDOM <#if expression>
```

The macro yields the value stored in errno on a domain error.

EILSEQ

```
#define EILSEQ <#if expression>
```

The macro yields the value stored in errno on an invalid multibyte sequence.

ERANGE

```
#define ERANGE <#if expression>
```

The macro yields the value stored in errno on a range error.

errno

`#define` **`errno`** `<`*`int modifiable lvalue`*`>`

The macro designates an object that is assigned a value greater than zero on certain library errors.

See also the **Table of Contents** and the **Index**.

<float.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define DBL_DIG <integer rvalue >= 10>
#define DBL_EPSILON <double rvalue <= 10^(-9)>
#define DBL_MANT_DIG <integer rvalue>
#define DBL_MAX <double rvalue >= 10^37>
#define DBL_MAX_10_EXP <integer rvalue >= 37>
#define DBL_MAX_EXP <integer rvalue>
#define DBL_MIN <double rvalue <= 10^(-37)>
#define DBL_MIN_10_EXP <integer rvalue <= -37>
#define DBL_MIN_EXP <integer rvalue>
#define FLT_DIG <integer rvalue >= 6>
#define FLT_EPSILON <float rvalue <= 10^(-5)>
#define FLT_MANT_DIG <integer rvalue>
#define FLT_MAX <float rvalue >= 10^37>
#define FLT_MAX_10_EXP <integer rvalue >= 37>
#define FLT_MAX_EXP <integer rvalue>
#define FLT_MIN <float rvalue <= 10^(-37)>
#define FLT_MIN_10_EXP <integer rvalue <= -37>
#define FLT_MIN_EXP <integer rvalue>
#define FLT_RADIX <#if expression >= 2>
#define FLT_ROUNDS <integer rvalue>
#define LDBL_DIG <integer rvalue >= 10>
#define LDBL_EPSILON <long double rvalue <= 10^(-9)>
#define LDBL_MANT_DIG <integer rvalue>
#define LDBL_MAX <long double rvalue >= 10^37>
#define LDBL_MAX_10_EXP <integer rvalue >= 37>
#define LDBL_MAX_EXP <integer rvalue>
#define LDBL_MIN <long double rvalue <= 10^(-37)>
#define LDBL_MIN_10_EXP <integer rvalue <= -37>
#define LDBL_MIN_EXP <integer rvalue>
```

Include the standard header **<float.h>** to determine various properties of floating-point type representations. The standard header <float.h> is available even in a freestanding implementation.

You can test only the value of the macro FLT_RADIX in an *if* directive. (The macro expands to a *#if* expression.) All other macros defined in this header expand to expressions whose values can be

determined only when the program executes. (These macros are rvalue expressions.) Some target environments can change the rounding and error-reporting properties of floating-point type representations while the program is running.

DBL_DIG

`#define DBL_DIG <integer rvalue >= 10>`

The macro yields the precision in decimal digits for type *double*.

DBL_EPSILON

`#define DBL_EPSILON <double rvalue <= 10^(-9)>`

The macro yields the smallest *X* of type *double* such that 1.0 + *X* != 1.0.

DBL_MANT_DIG

`#define DBL_MANT_DIG <integer rvalue>`

The macro yields the number of mantissa digits, base FLT_RADIX, for type *double*.

DBL_MAX

`#define DBL_MAX <double rvalue >= 10^37>`

The macro yields the largest finite representable value of type *double*.

DBL_MAX_10_EXP

`#define DBL_MAX_10_EXP <integer rvalue >= 37>`

The macro yields the maximum integer *X*, such that 10^*X* is a finite representable value of type *double*.

DBL_MAX_EXP

`#define DBL_MAX_EXP <integer rvalue>`

The macro yields the maximum integer *X*, such that FLT_RADIX^(*X* - 1) is a finite representable value of type *double*.

DBL_MIN

`#define DBL_MIN <double rvalue <= 10^(-37)>`

The macro yields the smallest normalized, finite representable value of type *double*.

DBL_MIN_10_EXP

`#define DBL_MIN_10_EXP <integer rvalue <= -37>`

The macro yields the minimum integer *X* such that 10^*X* is a normalized, finite representable value of type *double*.

DBL_MIN_EXP

`#define DBL_MIN_EXP <integer rvalue>`

The macro yields the minimum integer *X* such that FLT_RADIX^(*X* - 1) is a normalized, finite representable value of type *double*.

FLT_DIG

`#define FLT_DIG <integer rvalue >= 6>`

The macro yields the precision in decimal digits for type *float*.

FLT_EPSILON

`#define FLT_EPSILON <float rvalue <= 10^(-5)>`

The macro yields the smallest *X* of type *float* such that 1.0 + *X* != 1.0.

FLT_MANT_DIG

`#define FLT_MANT_DIG <integer rvalue>`

The macro yields the number of mantissa digits, base FLT_RADIX, for type *float*.

FLT_MAX

`#define FLT_MAX <float rvalue >= 10^37>`

The macro yields the largest finite representable value of type *float*.

FLT_MAX_10_EXP

`#define` **`FLT_MAX_10_EXP`** *`<integer rvalue >= 37>`*

The macro yields the maximum integer *X*, such that 10^*X* is a finite representable value of type *float*.

FLT_MAX_EXP

`#define` **`FLT_MAX_EXP`** *`<integer rvalue>`*

The macro yields the maximum integer *X*, such that FLT_RADIX^(*X* - 1) is a finite representable value of type *float*.

FLT_MIN

`#define` **`FLT_MIN`** *`<float rvalue <= 10^(-37)>`*

The macro yields the smallest normalized, finite representable value of type *float*.

FLT_MIN_10_EXP

`#define` **`FLT_MIN_10_EXP`** *`<integer rvalue <= -37>`*

The macro yields the minimum integer *X*, such that 10^*X* is a normalized, finite representable value of type *float*.

FLT_MIN_EXP

`#define` **`FLT_MIN_EXP`** *`<integer rvalue>`*

The macro yields the minimum integer *X*, such that FLT_RADIX^(*X* - 1) is a normalized, finite representable value of type *float*.

FLT_RADIX

`#define` **`FLT_RADIX`** *`<#if expression >= 2>`*

The macro yields the radix of all floating-point representations.

FLT_ROUNDS

`#define` **`FLT_ROUNDS`** *`<integer rvalue>`*

The macro yields a value that describes the current rounding mode for floating-point operations. Note that the target environment can change the rounding mode while the program executes. How it does so, however, is not specified. The values are:

- -1 if the mode is indeterminate
- 0 if rounding is toward zero
- 1 if rounding is to nearest representable value
- 2 if rounding is toward +infinity
- 3 if rounding is toward -infinity

An implementation can define additional values for this macro.

LDBL_DIG

`#define` **`LDBL_DIG`** *`<integer rvalue >= 10>`*

The macro yields the precision in decimal digits for type *long double*.

LDBL_EPSILON

`#define` **`LDBL_EPSILON`** *`<long double rvalue <= 10^(-9)>`*

The macro yields the smallest *X* of type *long double* such that 1.0 + *X* != 1.0.

LDBL_MANT_DIG

`#define` **`LDBL_MANT_DIG`** *`<integer rvalue>`*

The macro yields the number of mantissa digits, base `FLT_RADIX`, for type *long double*.

LDBL_MAX

`#define` **`LDBL_MAX`** *`<long double rvalue >= 10^37>`*

The macro yields the largest finite representable value of type *long double*.

LDBL_MAX_10_EXP

`#define` **`LDBL_MAX_10_EXP`** *`<integer rvalue >= 37>`*

The macro yields the maximum integer *X*, such that 10^*X* is a finite representable value of type *long double*.

LDBL_MAX_EXP

`#define` **`LDBL_MAX_EXP`** *`<integer rvalue>`*

The macro yields the maximum integer *X*, such that FLT_RADIX^(*X* - 1) is a finite representable value of type *long double*.

LDBL_MIN

`#define` **`LDBL_MIN`** *`<long double rvalue <= 10^(-37)>`*

The macro yields the smallest normalized, finite representable value of type *long double*.

LDBL_MIN_10_EXP

`#define` **`LDBL_MIN_10_EXP`** *`<integer rvalue <= -37>`*

The macro yields the minimum integer *X*, such that 10^*X* is a normalized, finite representable value of type *long double*.

LDBL_MIN_EXP

`#define` **`LDBL_MIN_EXP`** *`<integer rvalue>`*

The macro yields the minimum integer *X*, such that FLT_RADIX^(*X* - 1) is a normalized, finite representable value of type *long double*.

See also the **Table of Contents** and the **Index**.

<iso646.h> [Added with Amendment 1]

```
#define and && [keyword in C++]
#define and_eq &= [keyword in C++]
#define bitand & [keyword in C++]
#define bitor | [keyword in C++]
#define compl ~ [keyword in C++]
#define not ! [keyword in C++]
#define not_eq != [keyword in C++]
#define or || [keyword in C++]
#define or_eq |= [keyword in C++]
#define xor ^ [keyword in C++]
#define xor_eq ^= [keyword in C++]
```

Include the standard header **<iso646.h>** to provide readable alternatives to certain operators or punctuators. The standard header <iso646.h> is available even in a freestanding implementation.

and

```
#define and && [keyword in C++]
```

The macro yields the operator &&.

and_eq

```
#define and_eq &= [keyword in C++]
```

The macro yields the operator &=.

bitand

```
#define bitand & [keyword in C++]
```

The macro yields the operator &.

bitor

`#define bitor | [keyword in C++]`

The macro yields the operator `|`.

compl

`#define compl ~ [keyword in C++]`

The macro yields the operator `~`.

not

`#define not ! [keyword in C++]`

The macro yields the operator `!`.

not_eq

`#define not_eq != [keyword in C++]`

The macro yields the operator `!=`.

or

`#define or || [keyword in C++]`

The macro yields the operator `||`.

or_eq

`#define or_eq |= [keyword in C++]`

The macro yields the operator `|=`.

xor

`#define xor ^ [keyword in C++]`

The macro yields the operator `^`.

xor_eq

`#define xor_eq ^= [keyword in C++]`

The macro yields the operator `^=`.

See also the **Table of Contents** and the **Index**.

<limits.h>

```
#define CHAR_BIT <#if expression >= 8>
#define CHAR_MAX <#if expression >= 127>
#define CHAR_MIN <#if expression <= 0>
#define INT_MAX <#if expression >= 32,767>
#define INT_MIN <#if expression <= -32,767>
#define LONG_MAX <#if expression >= 2,147,483,647>
#define LONG_MIN <#if expression <= -2,147,483,647>
#define MB_LEN_MAX <#if expression >= 1>
#define SCHAR_MAX <#if expression >= 127>
#define SCHAR_MIN <#if expression <= -127>
#define SHRT_MAX <#if expression >= 32,767>
#define SHRT_MIN <#if expression <= -32,767>
#define UCHAR_MAX <#if expression >= 255>
#define UINT_MAX <#if expression >= 65,535>
#define ULONG_MAX <#if expression >= 4,294,967,295>
#define USHRT_MAX <#if expression >= 65,535>
```

Include the standard header **<limits.h>** to determine various properties of the integer type representations. The standard header <limits.h> is available even in a freestanding implementation.

You can test the values of all these macros in an *if* directive. (The macros are *#if* expressions.)

CHAR_BIT

```
#define CHAR_BIT <#if expression >= 8>
```

The macro yields the maximum value for the number of bits used to represent an object of type *char*.

CHAR_MAX

```
#define CHAR_MAX <#if expression >= 127>
```

The macro yields the maximum value for type *char*. Its value is:

- SCHAR_MAX if *char* represents negative values
- UCHAR_MAX otherwise

CHAR_MIN

```
#define CHAR_MIN <#if expression <= 0>
```

The macro yields the minimum value for type *char*. Its value is:

- SCHAR_MIN if *char* represents negative values
- zero otherwise

INT_MAX

```
#define INT_MAX <#if expression >= 32,767>
```

The macro yields the maximum value for type *int*.

INT_MIN

```
#define INT_MIN <#if expression <= -32,767>
```

The macro yields the minimum value for type *int*.

LONG_MAX

```
#define LONG_MAX <#if expression >= 2,147,483,647>
```

The macro yields the maximum value for type *long*.

LONG_MIN

```
#define LONG_MIN <#if expression <= -2,147,483,647>
```

The macro yields the minimum value for type *long*.

MB_LEN_MAX

```
#define MB_LEN_MAX <#if expression >= 1>
```

The macro yields the maximum number of characters that constitute a multibyte character in any supported locale. Its value is >= MB_CUR_MAX.

SCHAR_MAX

`#define` **`SCHAR_MAX`** *`<#if expression >= 127>`*

The macro yields the maximum value for type *signed char*.

SCHAR_MIN

`#define` **`SCHAR_MIN`** *`<#if expression <= -127>`*

The macro yields the minimum value for type *signed char*.

SHRT_MAX

`#define` **`SHRT_MAX`** *`<#if expression >= 32,767>`*

The macro yields the maximum value for type *short*.

SHRT_MIN

`#define` **`SHRT_MIN`** *`<#if expression <= -32,767>`*

The macro yields the minimum value for type *short*.

UCHAR_MAX

`#define` **`UCHAR_MAX`** *`<#if expression >= 255>`*

The macro yields the maximum value for type *unsigned char*.

UINT_MAX

`#define` **`UINT_MAX`** *`<#if expression >= 65,535>`*

The macro yields the maximum value for type *unsigned int*.

ULONG_MAX

`#define` **`ULONG_MAX`** *`<#if expression >= 4,294,967,295>`*

The macro yields the maximum value for type *unsigned long*.

USHRT_MAX

`#define` **`USHRT_MAX`** *`<#if expression >= 65,535>`*

The macro yields the maximum value for type *unsigned short.*

See also the **Table of Contents** and the **Index**.

<locale.h>

```
#define LC_ALL <integer constant expression>
#define LC_COLLATE <integer constant expression>
#define LC_CTYPE <integer constant expression>
#define LC_MONETARY <integer constant expression>
#define LC_NUMERIC <integer constant expression>
#define LC_TIME <integer constant expression>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
struct lconv;
struct lconv *localeconv(void);
char *setlocale(int category, const char *locale);
```

Include the standard header **<locale.h>** to alter or access properties of the current **locale** -- a collection of culture-specific information. An implementation can define additional macros in this standard header with names that begin with LC_. You can use any of these macro names as the **locale category** argument (which selects a cohesive subset of a locale) to setlocale.

LC_ALL

```
#define LC_ALL <integer constant expression>
```

The macro yields the locale category argument value that affects all locale categories.

LC_COLLATE

```
#define LC_COLLATE <integer constant expression>
```

The macro yields the locale category argument value that affects the collation functions strcoll and strxfrm.

LC_CTYPE

```
#define LC_CTYPE <integer constant expression>
```

The macro yields the locale category argument value that affects character classification functions, wide-character classification functions, and various multibyte conversion functions.

LC_MONETARY

```
#define LC_MONETARY <integer constant expression>
```

The macro yields the locale category argument value that affects monetary information returned by `localeconv`.

LC_NUMERIC

```
#define LC_NUMERIC <integer constant expression>
```

The macro yields the locale category argument value that affects numeric information returned by `localeconv`, including the decimal point used by numeric conversion, read, and write functions.

LC_TIME

```
#define LC_TIME <integer constant expression>
```

The macro yields the locale category argument value that affects the time conversion function `strftime`.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

lconv

```
struct lconv {
    ELEMENT                  "C" LOCALE   LOCALE CATEGORY
    char *currency_symbol;   ""           LC_MONETARY
    char *decimal_point;     "."          LC_NUMERIC
    char *grouping;          ""           LC_NUMERIC
    char *int_curr_symbol;   ""           LC_MONETARY
    char *mon_decimal_point; ""           LC_MONETARY
    char *mon_grouping;      ""           LC_MONETARY
    char *mon_thousands_sep; ""           LC_MONETARY
    char *negative_sign;     ""           LC_MONETARY
    char *positive_sign;     ""           LC_MONETARY
    char *thousands_sep;     ""           LC_NUMERIC
    char frac_digits;        CHAR_MAX     LC_MONETARY
    char int_frac_digits;    CHAR_MAX     LC_MONETARY
```

```
    char n_cs_precedes;      CHAR_MAX     LC_MONETARY
    char n_sep_by_space;     CHAR_MAX     LC_MONETARY
    char n_sign_posn;        CHAR_MAX     LC_MONETARY
    char p_cs_precedes;      CHAR_MAX     LC_MONETARY
    char p_sep_by_space;     CHAR_MAX     LC_MONETARY
    char p_sign_posn;        CHAR_MAX     LC_MONETARY
    };
```

`struct lconv` contains members that describe how to format numeric and monetary values. Functions in the Standard C library use only the field `decimal_point`. The information is otherwise advisory:

- Members of type *pointer to char* all point to C strings.
- Members of type *char* have nonnegative values.
- A *char* value of `CHAR_MAX` indicates that a meaningful value is not available in the current locale.

The members shown above can occur in arbitrary order and can be interspersed with additional members. The comment following each member shows its value for the **"C" locale**, the locale in effect at program startup, followed by the locale category that can affect its value.

A description of each member follows, with an example in parentheses that would be suitable for a USA locale.

`currency_symbol` -- the local currency symbol (`"$"`)

`decimal_point` -- the decimal point for non-monetary values (`"."`)

`grouping` -- the sizes of digit groups for non-monetary values. Successive elements of the string describe groups going away from the decimal point:

- An element value of zero (the terminating null character) calls for the previous element value to be repeated indefinitely.
- An element value of `CHAR_MAX` ends any further grouping (and hence ends the string).

Thus, the array `{3, 2, CHAR_MAX}` calls for a group of three digits, then two, then whatever remains, as in `9876,54,321`, while `"\3"` calls for repeated groups of three digits, as in `987,654,321`. (`"\3"`)

`int_curr_symbol` -- the international currency symbol specified by ISO 4217 (`"USD "`)

`mon_decimal_point` -- the decimal point for monetary values (`"."`)

`mon_grouping` -- the sizes of digit groups for monetary values. Successive elements of the string describe groups going away from the decimal point. The encoding is the same as for `grouping`.

`mon_thousands_sep` -- the separator for digit groups to the left of the decimal point for monetary values (`","`)

`negative_sign` -- the negative sign for monetary values (`"-"`)

positive_sign -- the positive sign for monetary values ("+")

thousands_sep -- the separator for digit groups to the left of the decimal point for non-monetary values (",")

frac_digits -- the number of digits to display to the right of the decimal point for monetary values (2)

int_frac_digits -- the number of digits to display to the right of the decimal point for international monetary values (2)

n_cs_precedes -- whether the currency symbol precedes or follows the value for negative monetary values:

- A value of 0 indicates that the symbol follows the value.
- A value of 1 indicates that the symbol precedes the value. (1)

n_sep_by_space -- whether the currency symbol is separated by a space or by no space from the value for negative monetary values:

- A value of 0 indicates that no space separates symbol and value.
- A value of 1 indicates that a space separates symbol and value. (0)

n_sign_posn -- the format for negative monetary values:

- A value of 0 indicates that parentheses surround the value and the currency symbol.
- A value of 1 indicates that the negative sign precedes the value and the currency symbol.
- A value of 2 indicates that the negative sign follows the value and the currency_symbol.
- A value of 3 indicates that the negative sign immediately precedes the currency symbol.
- A value of 4 indicates that the negative sign immediately follows the currency_symbol. (4)

p_cs_precedes -- whether the currency symbol precedes or follows the value for positive monetary values:

- A value of 0 indicates that the symbol follows the value.
- A value of 1 indicates that the symbol precedes the value. (1)

p_sep_by_space -- whether the currency symbol is separated by a space or by no space from the value for positive monetary values:

- A value of 0 indicates that no space separates symbol and value.
- A value of 1 indicates that a space separates symbol and value. (0)

p_sign_posn -- the format for positive monetary values:

- A value of 0 indicates that parentheses surround the value and the currency symbol.
- A value of 1 indicates that the negative sign precedes the value and the currency symbol.
- A value of 2 indicates that the negative sign follows the value and the currency symbol.
- A value of 3 indicates that the negative sign immediately precedes the currency symbol.
- A value of 4 indicates that the negative sign immediately follows the currency symbol. (4)

localeconv

```
struct lconv *localeconv(void);
```

The function returns a pointer to a static-duration structure containing numeric formatting information for the current locale. You cannot alter values stored in the static-duration structure. The stored values can change on later calls to `localeconv` or on calls to `setlocale` that alter any of the categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`.

setlocale

```
char *setlocale(int category, const char *locale);
```

The function either returns a pointer to a static-duration string describing a new locale or returns a null pointer (if the new locale cannot be selected). The value of `category` selects one or more locale categories, each of which must match the value of one of the macros defined in this standard header with names that begin with `LC_`.

If `locale` is a null pointer, the locale remains unchanged. If `locale` points to the string `"C"`, the new locale is the `"C"` locale for the locale category specified. If `locale` points to the string `""`, the new locale is the **native locale** (a default locale presumably tailored for the local culture) for the locale category specified. `locale` can also point to a string returned on an earlier call to `setlocale` or to other strings that the implementation can define.

At program startup, the target environment calls `setlocale( LC_ALL, "C")` before it calls `main`.

See also the **Table of Contents** and the **Index**.

<math.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define HUGE_VAL <double rvalue>
double abs(double x); [C++ only]
float abs(float x); [C++ only]
long double abs(long double x); [C++ only]
double acos(double x);
float acos(float x); [C++ only]
long double acos(long double x); [C++ only]
float acosf(float x); [optional]
long double acosl(long double x); [optional]
double asin(double x);
float asin(float x); [C++ only]
long double asin(long double x); [C++ only]
float asinf(float x); [optional]
long double asinl(long double x); [optional]
double atan(double x);
float atan(float x); [C++ only]
long double atan(long double x); [C++ only]
float atanf(float x); [optional]
long double atanl(long double x); [optional]
double atan2(double y, double x);
float atan2(float y, float x); [C++ only]
long double atan2(long double y, long double x); [C++ only]
float atan2f(float y, float x); [optional]
long double atan2l(long double y, long double x); [optional]
double ceil(double x);
float ceil(float x); [C++ only]
long double ceil(long double x); [C++ only]
float ceilf(float x); [optional]
long double ceill(long double x); [optional]
double cos(double x);
float cos(float x); [C++ only]
long double cos(long double x); [C++ only]
float cosf(float x); [optional]
long double cosl(long double x); [optional]
```

```
double cosh(double x);
float cosh(float x); [C++ only]
long double cosh(long double x); [C++ only]
float coshf(float x); [optional]
long double coshl(long double x); [optional]
double exp(double x);
float exp(float x); [C++ only]
long double exp(long double x); [C++ only]
float expf(float x); [optional]
long double expl(long double x); [optional]
double fabs(double x);
float fabs(float x); [C++ only]
long double fabs(long double x); [C++ only]
float fabsf(float x); [optional]
long double fabsl(long double x); [optional]
double floor(double x);
float floor(float x); [C++ only]
long double floor(long double x); [C++ only]
float floorf(float x); [optional]
long double floorl(long double x); [optional]
double fmod(double x, double y);
float fmod(float x, float y); [C++ only]
long double fmod(long double x, long double y); [C++ only]
float fmodf(float x, float y); [optional]
long double fmodl(long double x, long double y); [optional]
double frexp(double x, int *pexp);
float frexp(float x, int *pexp); [C++ only]
long double frexp(long double x, int *pexp); [C++ only]
float frexpf(float x, int *pexp); [optional]
long double frexpl(long double x, int *pexp); [optional]
double ldexp(double x, int exp);
float ldexp(float x, int exp); [C++ only]
long double ldexp(long double x, int exp); [C++ only]
float ldexpf(float x, int exp); [optional]
long double ldexpl(long double x, int exp); [optional]
double log(double x);
float log(float x); [C++ only]
long double log(long double x); [C++ only]
float logf(float x); [optional]
long double logl(long double x); [optional]
```

```
double log10(double x);
float log10(float x); [C++ only]
long double log10(long double x); [C++ only]
float log10f(float x); [optional]
long double log10l(long double x); [optional]
double modf(double x, double *pint);
float modf(float x, float *pint); [C++ only]
long double modf(long double x, long double *pint); [C++ only]
float modff(float x, float *pint); [optional]
long double modfl(long double x, long double *pint); [optional]
double pow(double x, double y);
float pow(float x, float y); [C++ only]
long double pow(long double x, long double y); [C++ only]
double pow(double x, int y); [C++ only]
float pow(float x, int y); [C++ only]
long double pow(long double x, int y); [C++ only]
float powf(float x, float y); [optional]
long double powl(long double x, long double y); [optional]
double sin(double x);
float sin(float x); [C++ only]
long double sin(long double x); [C++ only]
float sinf(float x); [optional]
long double sinl(long double x); [optional]
double sinh(double x);
float sinh(float x); [C++ only]
long double sinh(long double x); [C++ only]
float sinhf(float x); [optional]
long double sinhl(long double x); [optional]
double sqrt(double x);
float sqrt(float x); [C++ only]
long double sqrt(long double x); [C++ only]
float sqrtf(float x); [optional]
long double sqrtl(long double x); [optional]
double tan(double x);
float tan(float x); [C++ only]
long double tan(long double x); [C++ only]
float tanf(float x); [optional]
long double tanl(long double x); [optional]
double tanh(double x);
float tanh(float x); [C++ only]
```

```
long double tanh(long double x); [C++ only]
float tanhf(float x); [optional]
long double tanhl(long double x); [optional]
```

Include the standard header **<math.h>** to declare several functions that perform common mathematical operations on floating-point values.

A **domain error** exception occurs when the function is not defined for its input argument value or values. A function reports a domain error by storing the value of EDOM in errno and returning a peculiar value defined for each implementation.

A **range error** exception occurs when the return value of the function is defined but cannot be represented. A function reports a range error by storing the value of ERANGE in errno and returning one of three values:

- HUGE_VAL -- if the value of a function returning *double* is positive and too large in magnitude to represent
- zero -- if the value of the function is too small to represent with a finite value
- -HUGE_VAL -- if the value of a function returning *double* is negative and too large in magnitude to represent

HUGE_VAL

```
#define HUGE_VAL <double rvalue>
```

The macro yields the value returned by some functions on a range error. The value can be a representation of infinity.

abs

```
double abs(double x); [C++ only]
float abs(float x); [C++ only]
long double abs(long double x); [C++ only]
```

The function returns the absolute value of x, |x|, the same as fabs.

acos, acosf, acosl

```
double acos(double x);
float acos(float x); [C++ only]
long double acos(long double x); [C++ only]
float acosf(float x); [optional]
long double acosl(long double x); [optional]
```

The function returns the angle whose cosine is x, in the range [0, pi] radians.

asin, asinf, asinl

```
double asin(double x);
float asin(float x); [C++ only]
long double asin(long double x); [C++ only]
float asinf(float x); [optional]
long double asinl(long double x); [optional]
```

The function returns the angle whose sine is x, in the range [-pi/2, +pi/2] radians.

atan, atanf, atanl

```
double atan(double x);
float atan(float x); [C++ only]
long double atan(long double x); [C++ only]
float atanf(float x); [optional]
long double atanl(long double x); [optional]
```

The function returns the angle whose tangent is x, in the range [-pi/2, +pi/2] radians.

atan2, atan2f, atan2l

```
double atan2(double y, double x);
float atan2(float y, float x); [C++ only]
long double atan2(long double y, long double x); [C++ only]
float atan2f(float y, float x); [optional]
long double atan2l(long double y, long double x); [optional]
```

The function returns the angle whose tangent is y/x, in the full angular range [-pi, +pi] radians.

ceil, ceilf, ceill

```
double ceil(double x);
float ceil(float x); [C++ only]
long double ceil(long double x); [C++ only]
float ceilf(float x); [optional]
long double ceill(long double x); [optional]
```

The function returns the smallest integer value not less than x.

cos, cosf, cosl

```
double cos(double x);
float cos(float x); [C++ only]
long double cos(long double x); [C++ only]
float cosf(float x); [optional]
long double cosl(long double x); [optional]
```

The function returns the cosine of x for x in radians. If x is large the value returned might not be meaningful, but the function reports no error.

cosh, coshf, coshl

```
double cosh(double x);
float cosh(float x); [C++ only]
long double cosh(long double x); [C++ only]
float coshf(float x); [optional]
long double coshl(long double x); [optional]
```

The function returns the hyperbolic cosine of x.

exp, expf, expl

```
double exp(double x);
float exp(float x); [C++ only]
long double exp(long double x); [C++ only]
float expf(float x); [optional]
long double expl(long double x); [optional]
```

The function returns the exponential of x, e^x.

fabs, fabsf, fabsl

```
double fabs(double x);
float fabs(float x); [C++ only]
long double fabs(long double x); [C++ only]
float fabsf(float x); [optional]
long double fabsl(long double x); [optional]
```

The function returns the absolute value of x, |x|, the same as abs.

floor, floorf, floorl

```
double floor(double x);
float floor(float x); [C++ only]
long double floor(long double x); [C++ only]
float floorf(float x); [optional]
long double floorl(long double x); [optional]
```

The function returns the largest integer value not greater than `x`.

fmod, fmodf, fmodl

```
double fmod(double x, double y);
float fmod(float x, float y); [C++ only]
long double fmod(long double x, long double y); [C++ only]
float fmodf(float x, float y); [optional]
long double fmodl(long double x, long double y); [optional]
```

The function returns the remainder of `x/y`, which is defined as follows:

- If `y` is zero, the function either reports a domain error or simply returns zero.
- Otherwise, if `0 <= x`, the value is `x - i*y` for some integer `i` such that:
  `0 <= i*|y| <= x < (i + 1)*|y|`
- Otherwise, `x < 0` and the value is `x - i*y` for some integer `i` such that:
  `i*|y| <= x < (i + 1)*|y| <= 0`

frexp, frexpf, frexpl

```
double frexp(double x, int *pexp);
float frexp(float x, int *pexp); [C++ only]
long double frexp(long double x, int *pexp); [C++ only]
float frexpf(float x, int *pexp); [optional]
long double frexpl(long double x, int *pexp); [optional]
```

The function determines a fraction `f` and base-2 integer `i` that represent the value of `x`. It returns the value `f` and stores the integer `i` in `*pexp`, such that `|f|` is in the interval [1/2, 1) or has the value 0, and `x` equals `f*2^i`. If `x` is zero, `*pexp` is also zero.

ldexp, ldexpf, ldexpl

```
double ldexp(double x, int exp);
float ldexp(float x, int exp); [C++ only]
long double ldexp(long double x, int exp); [C++ only]
float ldexpf(float x, int exp); [optional]
```

```
long double ldexpl(long double x, int exp); [optional]
```

The function returns `x*2^exp`.

log, logf, logl

```
double log(double x);
float log(float x); [C++ only]
long double log(long double x); [C++ only]
float logf(float x); [optional]
long double logl(long double x); [optional]
```

The function returns the natural logarithm of `x`.

log10, log10f, log10l

```
double log10(double x);
float log10(float x); [C++ only]
long double log10(long double x); [C++ only]
float log10f(float x); [optional]
long double log10l(long double x); [optional]
```

The function returns the base-10 logarithm of `x`.

modf, modff, modfl

```
double modf(double x, double *pint);
float modf(float x, float *pint); [C++ only]
long double modf(long double x, long double *pint); [C++ only]
float modff(float x, float *pint); [optional]
long double modfl(long double x, long double *pint); [optional]
```

The function determines an integer `i` plus a fraction `f` that represent the value of `x`. It returns the value `f` and stores the integer `i` in `*pint`, such that `f + i == x`, `|f|` is in the interval [0, 1), and both `f` and `i` have the same sign as `x`.

pow, powf, powl

```
double pow(double x, double y);
float pow(float x, float y); [C++ only]
long double pow(long double x, long double y); [C++ only]
double pow(double x, int y); [C++ only]
float pow(float x, int y); [C++ only]
long double pow(long double x, int y); [C++ only]
```

```
float powf(float x, float y); [optional]
long double powl(long double x, long double y); [optional]
```

The function returns x raised to the power y, x^y.

sin, sinf, sinl

```
double sin(double x);
float sin(float x); [C++ only]
long double sin(long double x); [C++ only]
float sinf(float x); [optional]
long double sinl(long double x); [optional]
```

The function returns the sine of x for x in radians. If x is large the value returned might not be meaningful, but the function reports no error.

sinh, sinhf, sinhl

```
double sinh(double x);
float sinh(float x); [C++ only]
long double sinh(long double x); [C++ only]
float sinhf(float x); [optional]
long double sinhl(long double x); [optional]
```

The function returns the hyperbolic sine of x.

sqrt, sqrtf, sqrtl

```
double sqrt(double x);
float sqrt(float x); [C++ only]
long double sqrt(long double x); [C++ only]
float sqrtf(float x); [optional]
long double sqrtl(long double x); [optional]
```

The function returns the square root of x, x^(1/2).

tan, tanf, tanl

```
double tan(double x);
float tan(float x); [C++ only]
long double tan(long double x); [C++ only]
float tanf(float x); [optional]
long double tanl(long double x); [optional]
```

The function returns the tangent of x for x in radians.If x is large the value returned might not be

meaningful, but the function reports no error.

tanh, tanhf, tanhl

```
double tanh(double x);
float tanh(float x); [C++ only]
long double tanh(long double x); [C++ only]
float tanhf(float x); [optional]
long double tanhl(long double x); [optional]
```

The function returns the hyperbolic tangent of `x`.

See also the **Table of Contents** and the **Index**.

<setjmp.h>

```
typedef a-type jmp_buf;
void longjmp(jmp_buf env, int val);
#define setjmp(jmp_buf env) <int rvalue>
```

Include the standard header **<setjmp.h>** to perform control transfers that bypass the normal function call and return protocol.

jmp_buf

```
typedef a-type jmp_buf;
```

The type is the array type *a-type* of an object that you declare to hold the context information stored by setjmp and accessed by longjmp.

longjmp

```
void longjmp(jmp_buf env, int val);
```

The function causes a second return from the execution of setjmp that stored the current context value in env. If val is nonzero, the return value is val; otherwise, it is 1.

The function that was active when setjmp stored the current context value must not have returned control to its caller. An object with dynamic duration that does not have a *volatile* type and whose stored value has changed since the current context value was stored will have a stored value that is indeterminate.

setjmp

```
#define setjmp(jmp_buf env) <int rvalue>
```

The macro stores the current context value in the array designated by env and returns zero. A later call to longjmp that accesses the same context value causes setjmp to again return, this time with a nonzero value. You can use the macro setjmp only in an expression that:

- has no operators
- has only the unary operator !
- has one of the relational or equality operators (==, !=, <, <=, >, or >=) with the other operand an

integer constant expression

You can write such an expression only as the *expression* part of a *do*, *expression*, *for*, *if*, *if-else*, *switch*, or *while* statement.

See also the **Table of Contents** and the **Index**.

<signal.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define SIGABRT <integer constant expression >= 0>
#define SIGFPE <integer constant expression >= 0>
#define SIGILL <integer constant expression >= 0>
#define SIGINT <integer constant expression >= 0>
#define SIGSEGV <integer constant expression >= 0>
#define SIGTERM <integer constant expression >= 0>
#define SIG_DFL <address constant expression>
#define SIG_ERR <address constant expression>
#define SIG_IGN <address constant expression>
int raise(int sig);
typedef i-type sig_atomic_t;
void (*signal(int sig, void (*func)(int)))(int);
```

Include the standard header **<signal.h>** to specify how the program handles **signals** while it executes. A signal can report some exceptional behavior within the program, such as division by zero. Or a signal can report some asynchronous event outside the program, such as someone striking an interactive attention key on a keyboard.

You can report any signal by calling raise. Each implementation defines what signals it generates (if any) and under what circumstances it generates them. An implementation can define signals other than the ones listed here. The standard header <signal.h> can define additional macros with names beginning with SIG to specify the values of additional signals. All such values are integer constant expressions >= 0.

You can specify a **signal handler** for each signal. A signal handler is a function that the target environment calls when the corresponding signal occurs. The target environment suspends execution of the program until the signal handler returns or calls longjmp. For maximum portability, an asynchronous signal handler should only:

- make calls (that succeed) to the function signal
- assign values to objects of type *volatile* sig_atomic_t
- return control to its caller

If the signal reports an error within the program (and the signal is not asynchronous), the signal handler can terminate by calling abort, exit, or longjmp.

SIGABRT

```
#define SIGABRT <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the abort signal.

SIGFPE

```
#define SIGFPE <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the arithmetic error signal, such as for division by zero or result out of range.

SIGILL

```
#define SIGILL <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the invalid execution signal, such as for a corrupted function image.

SIGINT

```
#define SIGINT <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the asynchronous interactive attention signal.

SIGSEGV

```
#define SIGSEGV <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the invalid storage access signal, such as for an erroneous lvalue expression.

SIGTERM

```
#define SIGTERM <integer constant expression >= 0>
```

The macro yields the `sig` argument value for the asynchronous termination request signal.

SIG_DFL

```
#define SIG_DFL <address constant expression>
```

The macro yields the `func` argument value to `signal` to specify default signal handling.

SIG_ERR

```
#define SIG_ERR <address constant expression>
```

The macro yields the `signal` return value to specify an erroneous call.

SIG_IGN

```
#define SIG_IGN <address constant expression>
```

The macro yields the `func` argument value to `signal` to specify that the target environment is to henceforth ignore the signal.

raise

```
int raise(int sig);
```

The function sends the signal `sig` and returns zero if the signal is successfully reported.

sig_atomic_t

```
typedef i-type sig_atomic_t;
```

The type is the integer type *`i-type`* for objects whose stored value is altered by an assigning operator as an **atomic operation** (an operation that never has its execution suspended while partially completed). You declare such objects to communicate between signal handlers and the rest of the program.

signal

```
void (*signal(int sig, void (*func)(int)))(int);
```

The function specifies the new handling for signal `sig` and returns the previous handling, if successful; otherwise, it returns `SIG_ERR`.

- If `func` is `SIG_DFL`, the target environment commences default handling (as defined by the implementation).
- If `func` is `SIG_IGN`, the target environment ignores subsequent reporting of the signal.

- Otherwise, `func` must be the address of a function returning *void* that the target environment calls with a single *int* argument. The target environment calls this function to handle the signal when it is next reported, with the value of the signal as its argument.

When the target environment calls a signal handler:

- The target environment can block further occurrences of the corresponding signal until the handler returns, calls `longjmp`, or calls `signal` for that signal.
- The target environment can perform default handling of further occurrences of the corresponding signal.
- For signal `SIGILL`, the target environment can leave handling unchanged for that signal.

See also the **Table of Contents** and the **Index**.

<stdarg.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define va_arg(va_list ap, T) <rvalue of type T>
#define va_end(va_list ap) <void expression>
typedef do-type va_list;
#define va_start(va_list ap, last-par) <void expression>
```

Include the standard header **<stdarg.h>** to access the unnamed additional arguments (arguments with no corresponding parameter declarations) in a function that accepts a **varying number of arguments**. To access the additional arguments:

- The program must first execute the macro va_start within the body of the function to initialize an object with context information.
- Subsequent execution of the macro va_arg, designating the same context information, yields the values of the additional arguments in order, beginning with the first unnamed argument. You can execute the macro va_arg from any function that can access the context information saved by the macro va_start.
- If you have executed the macro va_start in a function, you must execute the macro va_end in the same function, designating the same context information, before the function returns.

You can repeat this sequence (as needed) to access the arguments as often as you want.

You declare an object of type va_list to store context information. va_list can be an array type, which affects how the program shares context information with functions that it calls. (The address of the first element of an array is passed, rather than the object itself.)

For example, here is a function that concatenates an arbitrary number of strings onto the end of an existing string (assuming that the existing string is stored in an object large enough to hold the resulting string):

```
#include <stdarg.h>
void va_cat(char *s, ...)
    {
    char *t;
    va_list ap;

    va_start(ap, s);
    while (t = va_arg(ap, char *)) null pointer ends list
        {
        s += strlen(s);             skip to end
```

```
        strcpy(s, t);              and copy a string
        }
    va_end(ap);
    }
```

va_arg

`#define` **`va_arg`**`(va_list ap, T) <rvalue of type T>`

The macro yields the value of the next argument in order, specified by the context information designated by `ap`. The additional argument must be of object type *`T`* after applying the rules for promoting arguments in the absence of a function prototype.

va_end

`#define` **`va_end`**`(va_list ap) <void expression>`

The macro performs any cleanup necessary, after processing the context information designated by `ap`, so that the function can return.

va_list

`typedef` *`do-type`* **`va_list`**`;`

The type is the object type *`do-type`* that you declare to hold the context information initialized by `va_start` and used by `va_arg` to access additional unnamed arguments.

va_start

`#define` **`va_start`**`(va_list ap,` *`last-par`*`) <void expression>`

The macro stores initial context information in the object designated by `ap`. *`last-par`* is the name of the last parameter you declare. For example, *`last-par`* is `b` for the function declared as `int f(int a, int b, ...)`. The last parameter must not have `register` storage class, and it must have a type that is not changed by the translator. It cannot have:

- an array type
- a function type
- type *float*
- any integer type that changes when promoted
- a reference type **[C++ only]**

See also the **Table of Contents** and the **Index**.

<stddef.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define offsetof(s-type, mbr) %lt;size_t constant expression>
typedef si-type ptrdiff_t;
typedef ui-type size_t;
typedef i-type wchar_t; [keyword in C++]
```

Include the standard header **<stddef.h>** to define several types and macros that are of general use throughout the program. The standard header `<stddef.h>` is available even in a freestanding implementation.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

offsetof

```
#define offsetof(s-type, mbr) <size_t constant expression>
```

The macro yields the offset in bytes, of type `size_t`, of member `mbr` from the beginning of structure type *`s-type`*, where for `X` of type *`s-type`*, `&X.mbr` is an address constant expression.

ptrdiff_t

```
typedef si-type ptrdiff_t;
```

The type is the signed integer type *`si-type`* of an object that you declare to store the result of subtracting two pointers.

size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type *`ui-type`* of an object that you declare to store the result of the *sizeof* operator.

wchar_t

`typedef` *`i-type`* **`wchar_t; [keyword in C++]`**

The type is the integer type *`i-type`* of a wide-character constant, such as `L'X'`. You declare an object of type `wchar_t` to hold a wide character.

See also the **Table of Contents** and the **Index**.

<stdlib.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define EXIT_FAILURE <rvalue integer expression>
#define EXIT_SUCCESS <rvalue integer expression>
#define MB_CUR_MAX <rvalue integer expression >= 1>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define RAND_MAX <integer constant expression >= 32,767>
void abort(void);
int abs(int i);
long abs(long i); [C++ only]
int atexit(void (*func)(void));
double atof(const char *s);
int atoi(const char *s);
long atol(const char *s);
void *bsearch(const void *key, const void *base, size_t nelem, size_t
size, int (*cmp)(const void *ck, const void *ce));
void *calloc(size_t nelem, size_t size);
div_t div(int numer, int denom);
ldiv_t div(long numer, long denom); [C++ only]
typedef T div_t;
void exit(int status);
void free(void *ptr);
char *getenv(const char *name);
long labs(long i);
ldiv_t ldiv(long numer, long denom);
typedef T ldiv_t;
void *malloc(size_t size);
int mblen(const char *s, size_t n);
size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

```
void qsort(void *base, size_t nelem, size_t size, int (*cmp)(const
void *e1, const void *e2));
int rand(void);
void *realloc(void *ptr, size_t size);
typedef ui-type size_t;
void srand(unsigned int seed);
double strtod(const char *s, char **endptr);
long strtol(const char *s, char **endptr, int base);
unsigned long strtoul(const char *s, char **endptr, int base);
int system(const char *s);
typedef i-type wchar_t; [keyword in C++]
size_t wcstombs(char *s, const wchar_t *wcs, size_t n);
int wctomb(char *s, wchar_t wchar);
```

Include the standard header **<stdlib.h>** to declare an assortment of useful functions and to define the macros and types that help you use them.

EXIT_FAILURE

```
#define EXIT_FAILURE <rvalue integer expression>
```

The macro yields the value of the `status` argument to `exit` that reports unsuccessful termination.

EXIT_SUCCESS

```
#define EXIT_SUCCESS <rvalue integer expression>
```

The macro yields the value of the `status` argument to `exit` that reports successful termination.

MB_CUR_MAX

```
#define MB_CUR_MAX <rvalue integer expression >= 1>
```

The macro yields the maximum number of characters that constitute a multibyte character in the current locale. Its value is <= `MB_LEN_MAX`.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

RAND_MAX

```
#define RAND_MAX <integer constant expression >= 32,767>
```

The macro yields the maximum value returned by `rand`.

abort

```
void abort(void);
```

The function calls `raise(SIGABRT)`, which reports the abort signal, `SIGABRT`. Default handling for the abort signal is to cause abnormal program termination and report unsuccessful termination to the target environment. Whether or not the target environment flushes output streams, closes open files, or removes temporary files on abnormal termination is implementation defined. If you specify handling that causes `raise` to return control to `abort`, the function calls `exit(EXIT_FAILURE)`, to report unsuccessful termination with `EXIT_FAILURE`. `abort` never returns control to its caller.

abs

```
int abs(int i);
long abs(long i); [C++ only]
```

The function returns the absolute value of `i`, `|i|`. The version that accepts a *long* argument behaves the same as `labs`

atexit

```
int atexit(void (*func)(void));
```

The function registers the function whose address is `func` to be called by `exit` (or when `main` returns) and returns zero if successful. The functions are called in reverse order of registry. You can register at least 32 functions.

atof

```
double atof(const char *s);
```

The function converts the initial characters of the string `s` to an equivalent value `x` of type *double* and then returns `x`. The conversion is the same as for `strtod(s, 0)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

atoi

```
int atoi(const char *s);
```

The function converts the initial characters of the string `s` to an equivalent value `x` of type *int* and then returns `x`. The conversion is the same as for `(int)strtol(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

atol

```
long atol(const char *s);
```

The function converts the initial characters of the string `s` to an equivalent value `x` of type *long* and then returns `x`. The conversion is the same as for `strtol(s, 0, 10)`, except that a value is not necessarily stored in `errno` if a conversion error occurs.

bsearch

```
void *bsearch(const void *key, const void *base, size_t nelem, size_t
size, int (*cmp)(const void *ck, const void *ce));
```

The function searches an array of ordered values and returns the address of an array element that equals the search key `key` (if one exists); otherwise, it returns a null pointer. The array consists of `nelem` elements, each of `size` bytes, beginning with the element whose address is `base`.

`bsearch` calls the comparison function whose address is `cmp` to compare the search key with elements of the array. The comparison function must return:

- a negative value if the search key `ck` is less than the array element `ce`
- zero if the two are equal
- a positive value if the search key is greater than the array element

`bsearch` assumes that the array elements are in ascending order according to the same comparison rules that are used by the comparison function.

calloc

```
void *calloc(size_t nelem, size_t size);
```

The function allocates an array object containing `nelem` elements each of size `size`, stores zeros in all bytes of the array, and returns the address of the first element of the array if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size in bytes is not greater than `size`.

div

```
div_t div(int numer, int denom);
ldiv_t div(long numer, long denom); [C++ only]
```

The function divides `numer` by `denom` and returns both quotient and remainder in the structure `div_t` (or `ldiv_t`) result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure member `x.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

div_t

```
typedef struct {
    int quot, rem;
    } div_t;
```

The type is the structure type returned by the function `div`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type *int*. The members shown above can occur in either order.

exit

```
void exit(int status);
```

The function calls all functions registered by `atexit`, closes all files, and returns control to the target environment. If `status` is zero or `EXIT_SUCCESS`, the program reports successful termination. If `status` is `EXIT_FAILURE`, the program reports unsuccessful termination. An implementation can define additional values for `status`.

free

```
void free(void *ptr);
```

If `ptr` is not a null pointer, the function deallocates the object whose address is `ptr`; otherwise, it does nothing. You can deallocate only objects that you first allocate by calling `calloc`, `malloc`, or `realloc`.

getenv

```
char *getenv(const char *name);
```

The function searches an **environment list**, which each implementation defines, for an entry whose name matches the string `name`. If the function finds a match, it returns a pointer to a static-duration object that

holds the definition associated with the target environment name. Otherwise, it returns a null pointer. Do not alter the value stored in the object. If you call `getenv` again, the value stored in the object can change. No target environment names are required of all environments.

labs

```
long labs(long i);
```

The function returns the absolute value of `i`, `|i|`, the same as `abs`.

ldiv

```
ldiv_t ldiv(long numer, long denom);
```

The function divides `numer` by `denom` and returns both quotient and remainder in the structure `ldiv_t` result `x`, if the quotient can be represented. The structure member `x.quot` is the algebraic quotient truncated toward zero. The structure member `x.rem` is the remainder, such that `numer == x.quot*denom + x.rem`.

ldiv_t

```
typedef struct {
    long quot, rem;
    } ldiv_t;
```

The type is the structure type returned by the function `ldiv`. The structure contains members that represent the quotient (`quot`) and remainder (`rem`) of a signed integer division with operands of type *long*. The members shown above can occur in either order.

malloc

```
void *malloc(size_t size);
```

The function allocates an object of size `size`, and returns the address of the object if successful; otherwise, it returns a null pointer. The values stored in the object are indeterminate. You can safely convert the return value to an object pointer of any type whose size is not greater than `size`.

mblen

```
int mblen(const char *s, size_t n);
```

If `s` is not a null pointer, the function returns the number of bytes in the multibyte string `s` that constitute the next multibyte character, or it returns -1 if the next `n` (or the remaining) bytes do not constitute a valid multibyte character. `mblen` does not include the terminating null in the count of bytes. The

function can use a conversion state stored in an internal static-duration object to determine how to interpret the multibyte string.

If `s` is a null pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its internal static-duration object and returns nonzero; otherwise, it returns zero.

mbstowcs

```
size_t mbstowcs(wchar_t *wcs, const char *s, size_t n);
```

The function stores a wide character string, in successive elements of the array whose first element has the address `wcs`, by converting, in turn, each of the multibyte characters in the multibyte string `s`. The string begins in the initial conversion state. The function converts each character as if by calling `mbtowc` (except that the internal conversion state stored for that function is unaffected). It stores at most `n` wide characters, stopping after it stores a null wide character. It returns the number of wide characters it stores, not counting the null wide character, if all conversions are successful; otherwise, it returns -1.

mbtowc

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

If `s` is not a null pointer, the function determines `x`, the number of bytes in the multibyte string `s` that constitute the next multibyte character. (`x` cannot be greater than `MB_CUR_MAX`.) If `pwc` is not a null pointer, the function converts the next multibyte character to its corresponding wide-character value and stores that value in `*pwc`. It then returns `x`, or it returns -1 if the next `n` or the remaining bytes do not constitute a valid multibyte character. `mbtowc` does not include the terminating null in the count of bytes. The function can use a conversion state stored in an internal static-duration object to determine how to interpret the multibyte string.

If `s` is a null pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its internal static-duration object and returns nonzero; otherwise, it returns zero.

qsort

```
void qsort(void *base, size_t nelem, size_t size, int (*cmp)(const
void *e1, const void *e2));
```

The function sorts, in place, an array consisting of `nelem` elements, each of `size` bytes, beginning with the element whose address is `base`. It calls the comparison function whose address is `cmp` to compare pairs of elements. The comparison function must return a negative value if `e1` is less than `e2`, zero if the two are equal, or a positive value if `e1` is greater than `e2`. Two array elements that are equal can appear in the sorted array in either order.

rand

```
int rand(void);
```

The function computes a pseudo-random number `x` based on a seed value stored in an internal static-duration object, alters the stored seed value, and returns `x`. `x` is in the interval [0, RAND_MAX].

realloc

```
void *realloc(void *ptr, size_t size);
```

The function allocates an object of size `size`, possibly obtaining initial stored values from the object whose address is `ptr`. It returns the address of the new object if successful; otherwise, it returns a null pointer. You can safely convert the return value to an object pointer of any type whose size is not greater than `size`.

If `ptr` is not a null pointer, it must be the address of an existing object that you first allocate by calling `calloc`, `malloc`, or `realloc`. If the existing object is not larger than the newly allocated object, `realloc` copies the entire existing object to the initial part of the allocated object. (The values stored in the remainder of the object are indeterminate.) Otherwise, the function copies only the initial part of the existing object that fits in the allocated object. If `realloc` succeeds in allocating a new object, it deallocates the existing object. Otherwise, the existing object is left unchanged.

If `ptr` is a null pointer, the function does not store initial values in the newly created object.

size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type *`ui-type`* of an object that you declare to store the result of the *sizeof* operator.

srand

```
void srand(unsigned int seed);
```

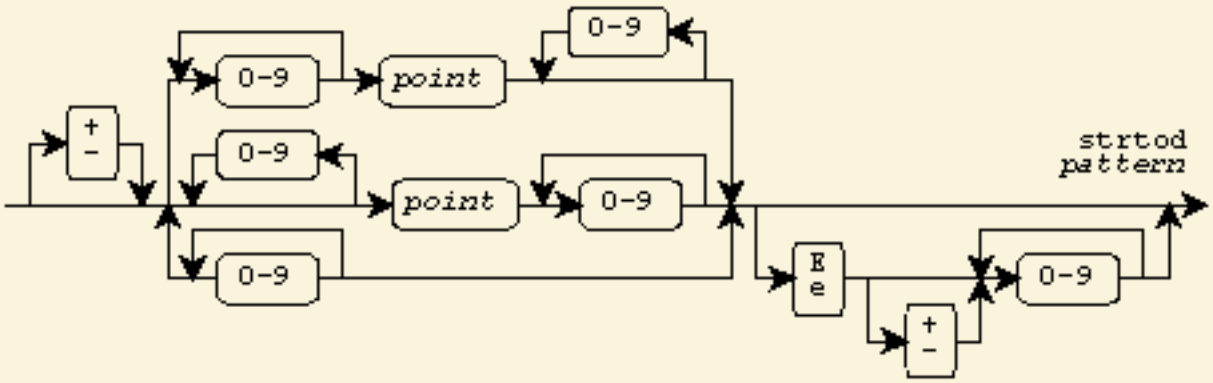
The function stores the seed value `seed` in a static-duration object that `rand` uses to compute a pseudo-random number. From a given seed value, that function always generates the same sequence of return values. The program behaves as if the target environment calls `srand(1)` at program startup.

strtod

```
double strtod(const char *s, char **endptr);
```

The function converts the initial characters of the string `s` to an equivalent value `x` of type *double*. If `endptr` is not a null pointer, the function stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`.

The initial characters of the string `s` must consist of zero or more characters for which `isspace` returns nonzero, followed by the longest sequence of one or more characters that match the pattern:



Here, a *`point`* is the decimal-point character for the current locale. (It is the dot (`.`) in the `"C"` locale.) If the string `s` matches this pattern, its equivalent value is the decimal integer represented by any digits to the left of the *`point`*, plus the decimal fraction represented by any digits to the right of the *`point`*, times 10 raised to the signed decimal integer power that follows an optional `e` or `E`. A leading minus sign negates the value. In locales other than the `"C"` locale, `strtod` can define additional patterns as well.

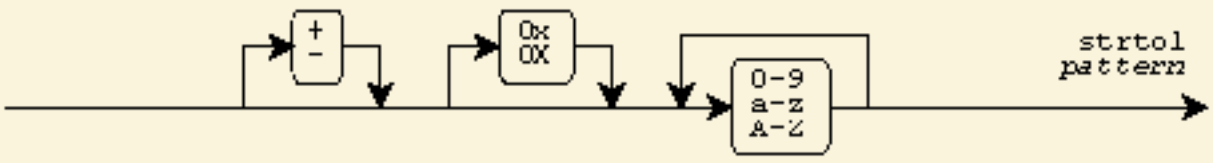
If the string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `strtod` behaves exactly as the functions declared in `<math.h>`.

strtol

```
long strtol(const char *s, char **endptr, int base);
```

The function converts the initial characters of the string `s` to an equivalent value `x` of type *long*. If `endptr` is not a null pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`.

The initial characters of the string `s` must consist of zero or more characters for which `isspace` returns nonzero, followed by the longest sequence of one or more characters that match the pattern:



The function accepts the sequences `0x` or `0X` only when `base` equals zero or 16. The letters `a-z` or `A-Z`

represent digits in the range [10, 36). If `base` is in the range [2, 36], the function accepts only digits with values less than `base`. If `base` == 0, then a leading `0x` or `0X` (after any sign) indicates a hexadecimal (base 16) integer, a leading `0` indicates an octal (base 8) integer, and any other valid pattern indicates a decimal (base 10) integer.

If the string `s` matches this pattern, its equivalent value is the signed integer of the appropriate base represented by the digits that match the pattern. (A leading minus sign negates the value.) In locales other than the `"C"` locale, `strtol` can define additional patterns as well.

If the string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large to represent as type *long*, `strtol` stores the value of `ERANGE` in `errno` and returns either `LONG_MAX`, if `x` is positive, or `LONG_MIN`, if `x` is negative.

strtoul

```
unsigned long strtoul(const char *s, char **endptr, int base);
```

The function converts the initial characters of the string `s` to an equivalent value `x` of type *unsigned long*. If `endptr` is not a null pointer, it stores a pointer to the unconverted remainder of the string in `*endptr`. The function then returns `x`.

`strtoul` converts strings exactly as does `strtol`, but reports a range error only if the equivalent value is too large to represent as type *unsigned long*. In this case, `strtoul` stores the value of `ERANGE` in `errno` and returns `ULONG_MAX`.

system

```
int system(const char *s);
```

If `s` is not a null pointer, the function passes the string `s` to be executed by a **command processor**, supplied by the target environment, and returns the status reported by the command processor. If `s` is a null pointer, the function returns nonzero only if the target environment supplies a command processor. Each implementation defines what strings its command processor accepts.

wchar_t

```
typedef i-type wchar_t; [keyword in C++]
```

The type is the integer type *`i-type`* of a wide-character constant, such as `L'X'`. You declare an object of type `wchar_t` to hold a wide character.

wcstombs

```
size_t wcstombs(char *s, const wchar_t *wcs, size_t n);
```

The function stores a multibyte string, in successive elements of the array whose first element has the address `s`, by converting in turn each of the wide characters in the string `wcs`. The multibyte string begins in the initial conversion state. The function converts each wide character as if by calling `wctomb` (except that the conversion state stored for that function is unaffected). It stores no more than `n` bytes, stopping after it stores a null byte. It returns the number of bytes it stores, not counting the null byte, if all conversions are successful; otherwise, it returns -1.

wctomb

```
int wctomb(char *s, wchar_t wchar);
```

If `s` is not a null pointer, the function determines `x`, the number of bytes needed to represent the multibyte character corresponding to the wide character `wchar`. `x` cannot exceed `MB_CUR_MAX`. The function converts `wchar` to its corresponding multibyte character, which it stores in successive elements of the array whose first element has the address `s`. It then returns `x`, or it returns -1 if `wchar` does not correspond to a valid multibyte character. `wctomb` includes the terminating null byte in the count of bytes. The function can use a conversion state stored in a static-duration object to determine how to interpret the multibyte character string.

If `s` is a null pointer and if multibyte characters have a state-dependent encoding in the current locale, the function stores the initial conversion state in its static-duration object and returns nonzero; otherwise, it returns zero.

See also the **Table of Contents** and the **Index**.

<string.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

NULL · memchr · memcmp · memcpy · memmove · memset · size_t · strcat · strchr · strcmp · strcoll · strcpy · strcspn · strerror · strlen · strncat · strncmp · strncpy · strpbrk · strrchr · strspn · strstr · strtok · strxfrm

---

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
void *memchr(const void *s, int c, size_t n); [not in C++]
const void *memchr(const void *s, int c, size_t n); [C++ only]
void *memchr(void *s, int c, size_t n); [C++ only]
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
void *memset(void *s, int c, size_t n);
typedef ui-type size_t;
char *strcat(char *s1, const char *s2);
char *strchr(const char *s, int c); [not in C++]
const char *strchr(const char *s, int c); [C++ only]
char *strchr(char *s, int c); [C++ only]
int strcmp(const char *s1, const char *s2);
int strcoll(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
size_t strcspn(const char *s1, const char *s2);
char *strerror(int errcode);
size_t strlen(const char *s);
char *strncat(char *s1, const char *s2, size_t n);
int strncmp(const char *s1, const char *s2, size_t n);
char *strncpy(char *s1, const char *s2, size_t n);
char *strpbrk(const char *s1, const char *s2); [not in C++]
const char *strpbrk(const char *s1, const char *s2); [C++ only]
char *strpbrk(char *s1, const char *s2); [C++ only]
char *strrchr(const char *s, int c); [not in C++]
const char *strrchr(const char *s, int c); [C++ only]
char *strrchr(char *s, int c); [C++ only]
size_t strspn(const char *s1, const char *s2);
```

```
char *strstr(const char *s1, const char *s2); [not in C++]
const char *strstr(const char *s1, const char *s2); [C++ only]
char *strstr(char *s1, const char *s2); [C++ only]
char *strtok(char *s1, const char *s2);
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Include the standard header **<string.h>** to declare a number of functions that help you manipulate C strings and other arrays of characters.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

memchr

```
void *memchr(const void *s, int c, size_t n); [not in C++]
const void *memchr(const void *s, int c, size_t n); [C++ only]
void *memchr(void *s, int c, size_t n); [C++ only]
```

The function searches for the first element of an array of *unsigned char*, beginning at the address `s` with size `n`, that equals `(unsigned char)c`. If successful, it returns the address of the matching element; otherwise, it returns a null pointer.

memcmp

```
int memcmp(const void *s1, const void *s2, size_t n);
```

The function compares successive elements from two arrays of *unsigned char*, beginning at the addresses `s1` and `s2` (both of size `n`), until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a positive number.
- Otherwise, the function returns a negative number.

memcpy

```
void *memcpy(void *s1, const void *s2, size_t n);
```

The function copies the array of *char* beginning at the address `s2` to the array of *char* beginning at the address `s1` (both of size `n`). It returns `s1`. The elements of the arrays can be accessed and stored in any order.

memmove

```
void *memmove(void *s1, const void *s2, size_t n);
```

The function copies the array of *char* beginning at `s2` to the array of *char* beginning at `s1` (both of size `n`). It returns `s1`. If the arrays overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

memset

```
void *memset(void *s, int c, size_t n);
```

The function stores `(unsigned char)c` in each of the elements of the array of *unsigned char* beginning at `s`, with size `n`. It returns `s`.

size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type *`ui-type`* of an object that you declare to store the result of the *sizeof* operator.

strcat

```
char *strcat(char *s1, const char *s2);
```

The function copies the string `s2`, including its terminating null character, to successive elements of the array of *char* that stores the string `s1`, beginning with the element that stores the terminating null character of `s1`. It returns `s1`.

strchr

```
char *strchr(const char *s, int c); [not in C++]
const char *strchr(const char *s, int c); [C++ only]
char *strchr(char *s, int c); [C++ only]
```

The function searches for the first element of the string `s` that equals `(char)c`. It considers the terminating null character as part of the string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

strcmp

```
int strcmp(const char *s1, const char *s2);
```

The function compares successive elements from two strings, `s1` and `s2`, until it finds elements that are not equal.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2` (both taken as *unsigned char*), the function returns a positive number.
- Otherwise, the function returns a negative number.

strcoll

```
int strcoll(const char *s1, const char *s2);
```

The function compares two strings, `s1` and `s2`, using a comparison rule that depends on the current locale. If `s1` compares greater than `s2` by this rule, the function returns a positive number. If the two strings compare equal, it returns zero. Otherwise, it returns a negative number.

strcpy

```
char *strcpy(char *s1, const char *s2);
```

The function copies the string `s2`, including its terminating null character, to successive elements of the array of *char* whose first element has the address `s1`. It returns `s1`.

strcspn

```
size_t strcspn(const char *s1, const char *s2);
```

The function searches for the first element `s1[i]` in the string `s1` that equals *any one* of the elements of the string `s2` and returns `i`. Each terminating null character is considered part of its string.

strerror

```
char *strerror(int errcode);
```

The function returns a pointer to an internal static-duration object containing the message string corresponding to the error code `errcode`. The program must not alter any of the values stored in this object. A later call to `strerror` can alter the value stored in this object.

strlen

```
size_t strlen(const char *s);
```

The function returns the number of characters in the string `s`, *not* including its terminating null character.

strncat

```
char *strncat(char *s1, const char *s2, size_t n);
```

The function copies the string `s2`, *not* including its terminating null character, to successive elements of the array of *char* that stores the string `s1`, beginning with the element that stores the terminating null character of `s1`. The function copies no more than `n` characters from `s2`. It then stores a null character, in the next element to be altered in `s1`, and returns `s1`.

strncmp

```
int strncmp(const char *s1, const char *s2, size_t n);
```

The function compares successive elements from two strings, `s1` and `s2`, until it finds elements that are not equal or until it has compared the first `n` elements of the two strings.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2` (both taken as *unsigned char*), the function returns a positive number.
- Otherwise, it returns a negative number.

strncpy

```
char *strncpy(char *s1, const char *s2, size_t n);
```

The function copies the string `s2`, *not* including its terminating null character, to successive elements of the array of *char* whose first element has the address `s1`. It copies no more than `n` characters from `s2`. The function then stores zero or more null characters in the next elements to be altered in `s1` until it stores a total of `n` characters. It returns `s1`.

strpbrk

```
char *strpbrk(const char *s1, const char *s2); [not in C++]
const char *strpbrk(const char *s1, const char *s2); [C++ only]
char *strpbrk(char *s1, const char *s2); [C++ only]
```

The function searches for the first element `s1[i]` in the string `s1` that equals *any one* of the elements of the string `s2`. It considers each terminating null character as part of its string. If `s1[i]` is not the

terminating null character, the function returns `&s1[i]`; otherwise, it returns a null pointer.

strrchr

```
char *strrchr(const char *s, int c); [not in C++]
const char *strrchr(const char *s, int c); [C++ only]
char *strrchr(char *s, int c); [C++ only]
```

The function searches for the last element of the string `s` that equals `(char)c`. It considers the terminating null character as part of the string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

strspn

```
size_t strspn(const char *s1, const char *s2);
```

The function searches for the first element `s1[i]` in the string `s1` that equals *none* of the elements of the string `s2` and returns `i`. It considers the terminating null character as part of the string `s1` only.

strstr

```
char *strstr(const char *s1, const char *s2); [not in C++]
const char *strstr(const char *s1, const char *s2); [C++ only]
char *strstr(char *s1, const char *s2); [C++ only]
```

The function searches for the first sequence of elements in the string `s1` that matches the sequence of elements in the string `s2`, *not* including its terminating null character. If successful, the function returns the address of the matching first element; otherwise, it returns a null pointer.

strtok

```
char *strtok(char *s1, const char *s2);
```

If `s1` is not a null pointer, the function begins a search of the string `s1`. Otherwise, it begins a search of the string whose address was last stored in an internal static-duration object on an earlier call to the function, as described below. The search proceeds as follows:

1. The function searches the string for `begin`, the address of the first element that equals *none* of the elements of the string `s2` (a set of token separators). It considers the terminating null character as part of the search string only.
2. If the search does not find an element, the function stores the address of the terminating null character in the internal static-duration object (so that a subsequent search beginning with that address will fail) and returns a null pointer. Otherwise, the function searches from `begin` for `end`, the address of the first element that equals *any one* of the elements of the string `s2`. It again considers the terminating null character as part of the search string only.

3. If the search does not find an element, the function stores the address of the terminating null character in the internal static-duration object. Otherwise, it stores a null character in the element whose address is `end`. Then it stores the address of the next element after `end` in the internal static-duration object (so that a subsequent search beginning with that address will continue with the remaining elements of the string) and returns `begin`.

strxfrm

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

The function stores a string in the array of *char* whose first element has the address `s1`. It stores no more than `n` characters, *including* the terminating null character, and returns the number of characters needed to represent the entire string, *not* including the terminating null character. If the value returned is `n` or greater, the values stored in the array are indeterminate. (If `n` is zero, `s1` can be a null pointer.)

`strxfrm` generates the string it stores from the string `s2` by using a transformation rule that depends on the current locale. For example, if `x` is a transformation of `s1` and `y` is a transformation of `s2`, then `strcmp(x, y)` returns the same value as `strcoll(s1, s2)`.

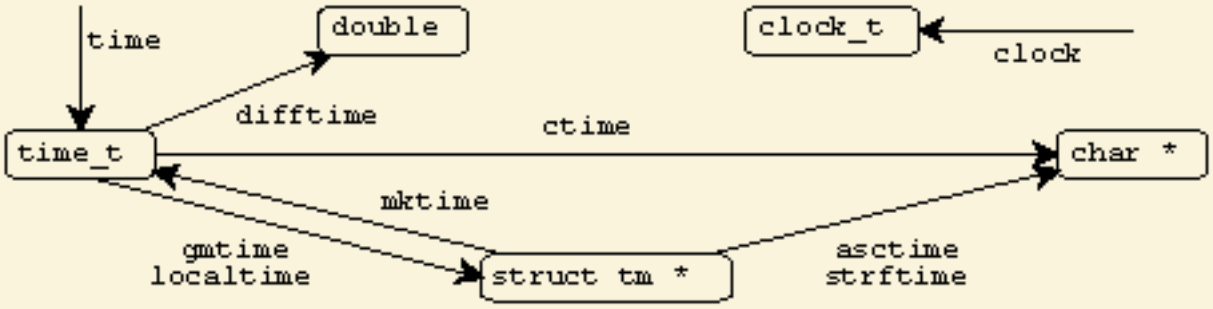
See also the **Table of Contents** and the **Index**.

<time.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#define CLOCKS_PER_SEC <integer constant expression > 0>
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
char *asctime(const struct tm *tptr);
clock_t clock(void);
typedef a-type clock_t;
char *ctime(const time_t *tod);
double difftime(time_t t1, time_t t0);
struct tm *gmtime(const time_t *tod);
struct tm *localtime(const time_t *tod);
time_t mktime(struct tm *tptr);
typedef ui-type size_t;
size_t strftime(char *s, size_t n, const char *format, const struct tm
*tptr);
time_t time(time_t *tod);
typedef a-type time_t;
struct tm;
```

Include the standard header **<time.h>** to declare several functions that help you manipulate times. The following diagram summarizes the functions and the object types that they convert between:



The functions share two static-duration objects that hold values computed by the functions:

- a **time string** of type array of *char*
- a **time structure** of type `struct tm`

A call to one of these functions can alter the value that was stored earlier in a static-duration object by another of these functions.

CLOCKS_PER_SEC

```
#define CLOCKS_PER_SEC <integer constant expression > 0>
```

The macro yields the number of clock ticks, returned by `clock`, in one second.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

asctime

```
char *asctime(const struct tm *tptr);
```

The function stores in the static-duration time string a 26-character English-language representation of the time encoded in `*tptr`. It returns the address of the static-duration time string. The text representation takes the form:

```
Sun Dec  2 06:55:15 1979\n\0
```

clock

```
clock_t clock(void);
```

The function returns the number of clock ticks of elapsed processor time, counting from a time related to program startup, or it returns -1 if the target environment cannot measure elapsed processor time.

clock_t

```
typedef a-type clock_t;
```

The type is the arithmetic type *`a-type`* of an object that you declare to hold the value returned by `clock`, representing elapsed processor time.

ctime

```
char *ctime(const time_t *tod);
```

The function converts the calendar time in `*tod` to a text representation of the local time in the static-duration time string. It returns the address of that string. It is equivalent to `asctime(localtime(tod))`.

difftime

```
double difftime(time_t t1, time_t t0);
```

The function returns the difference `t1 - t0`, in seconds, between the calendar time `t0` and the calendar time `t1`.

gmtime

```
struct tm *gmtime(const time_t *tod);
```

The function stores in the static-duration time structure an encoding of the calendar time in `*tod`, expressed as **Universal Time Coordinated**, or UTC. (UTC was formerly Greenwich Mean Time, or GMT). It returns the address of that structure.

localtime

```
struct tm *localtime(const time_t *tod);
```

The function stores in the static-duration time structure an encoding of the calendar time in `*tod`, expressed as local time. It returns the address of that structure.

mktime

```
time_t mktime(struct tm *tptr);
```

The function alters the values stored in `*tptr` to represent an equivalent encoded local time, but with the values of all members within their normal ranges. It then determines the values `tptr->wday` and `tptr->yday` from the values of the other members. It returns the calendar time equivalent to the encoded time, or it returns a value of -1 if the calendar time cannot be represented.

size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type *`ui-type`* of an object that you declare to store the result of the *sizeof* operator.

strftime

```
size_t strftime(char *s, size_t n, const char *format, const struct tm
*tptr);
```

The function generates formatted text, under the control of the format `format` and the values stored in the time structure `*tptr`. It stores each generated character in successive locations of the array object of size `n` whose first element has the address `s`. The function then stores a null character in the next location of the array. It returns `x`, the number of characters generated, if `x < n`; otherwise, it returns zero, and the values stored in the array are indeterminate.

For each multibyte character other than `%` in the format, the function stores that multibyte character in the array object. Each occurrence of `%` followed by another character in the format is a **conversion specifier**. For each conversion specifier, the function stores a replacement character sequence.

The following table lists all conversion specifiers defined for `strftime`. Example replacement character sequences in parentheses follow each conversion specifier. All examples are for the `"C"` locale, using the date and time Sunday, 2 December 1979 at 06:55:15 AM EST.

```
%a    abbreviated weekday name (Sun)
%A    full weekday name (Sunday)
%b    abbreviated month name (Dec)
%B    full month name (December)
%c    date and time (Dec  2 06:55:15 1979)
%d    day of the month (02)
%H    hour of the 24-hour day (06)
%I    hour of the 12-hour day (06)
%j    day of the year, from 001 (335)
%m    month of the year, from 01 (12)
%M    minutes after the hour (55)
%p    AM/PM indicator (AM)
%S    seconds after the minute (15)
%U    Sunday week of the year, from 00 (48)
%w    day of the week, from 0 for Sunday (6)
%W    Monday week of the year, from 00 (47)
%x    date (Dec  2 1979)
%X    time (06:55:15)
%y    year of the century, from 00 (79)
%Y    year (1979)
%Z    time zone name, if any (EST)
%%    percent character %
```

The current locale category `LC_TIME` can affect these replacement character sequences.

time

```
time_t time(time_t *tod);
```

If `tod` is not a null pointer, the function stores the current calendar time in `*tod`. The function returns the current calendar time, if the target environment can determine it; otherwise, it returns -1.

time_t

```
typedef a-type time_t;
```

The type is the arithmetic type *a-type* of an object that you declare to hold the value returned by `time`. The value represents calendar time.

tm

```
struct tm {
    int tm_sec;         seconds after the minute (from 0)
    int tm_min;         minutes after the hour (from 0)
    int tm_hour;        hour of the day (from 0)
    int tm_mday;        day of the month (from 1)
    int tm_mon;         month of the year (from 0)
    int tm_year;        years since 1900 (from 0)
    int tm_wday;        days since Sunday (from 0)
    int tm_yday;        day of the year (from 0)
    int tm_isdst;       Daylight Saving Time flag
    };
```

`struct tm` contains members that describe various properties of the calendar time. The members shown above can occur in any order, interspersed with additional members. The comment following each member briefly describes its meaning.

The member `tm_isdst` contains:

- a positive value if **Daylight Saving Time** is in effect
- zero if Daylight Saving Time is not in effect
- a negative value if the status of Daylight Saving Time is not known (so the target environment should attempt to determine its status)

See also the **Table of Contents** and the **Index**.

<wchar.h> [Added with Amendment 1]

---

btowc · fgetwc · fgetws · fputwc · fputws · fwide · fwprintf · fwscanf · getwc · getwchar · mbrlen · mbrtowc · mbsinit · mbsrtowcs · mbstate_t · NULL · putwc · putwchar · size_t · swprintf · swscanf · tm · ungetwc · vfwprintf · vswprintf · vwprintf · WCHAR_MAX · WCHAR_MIN · wchar_t · wcrtomb · wcscat · wcschr · wcscmp · wcscoll · wcscpy · wcscspn · wcsftime · wcslen · wcsncat · wcsncmp · wcsncpy · wcspbrk · wcsrchr · wcsrtombs · wcsspn · wcsstr · wcstod · wcstok · wcstol · wcstoul · wcsxfrm · wctob · WEOF · wint_t · wmemchr · wmemcmp · wmemcpy · wmemmove · wmemset · wprintf · wscanf

---

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
#define WCHAR_MAX <#if expression >= 127>
#define WCHAR_MIN <#if expression <= 0>
#define WEOF <wint_t constant expression>
wint_t btowc(int c);
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t *s, FILE *stream);
int fwide(FILE *stream, int mode);
int fwprintf(FILE *stream, const wchar_t *format, ...);
int fwscanf(FILE *stream, const wchar_t *format, ...);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
int mbsinit(const mbstate_t *ps);
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t
*ps);
typedef o-type mbstate_t;
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
```

```
typedef ui-type size_t;
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
int swscanf(const wchar_t *s, const wchar_t *format, ...);
struct tm;
wint_t ungetwc(wint_t c, FILE *stream);
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list
arg);
int vwprintf(const wchar_t *format, va_list arg);
typedef i-type wchar_t; [keyword in C++]
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t *format,
const struct tm *timeptr);
size_t wcslen(const wchar_t *s);
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t n);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t
*ps);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
double wcstod(const wchar_t *nptr, wchar_t **endptr);
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t **ptr);
long wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
unsigned long wcstoul(const wchar_t *nptr, wchar_t **endptr, int
base);
size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);
int wctob(wint_t c);
typedef i_type wint_t;
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [not in C++]
const wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [C++
only]
wchar_t *wmemchr(wchar_t *s, wchar_t c, size_t n); [C++ only]
```

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
int wprintf(const wchar_t *format, ...);
int wscanf(const wchar_t *format, ...);
```

Include the standard header **<wchar.h>** so that you can perform input and output operations on wide streams or manipulate wide strings.

NULL

```
#define NULL <either 0, 0L, or (void *)0> [0 in C++]
```

The macro yields a null pointer constant that is usable as an address constant expression.

WCHAR_MAX

```
#define WCHAR_MAX <#if expression >= 127>
```

The macro yields the maximum value for type `wchar_t`.

WCHAR_MIN

```
#define WCHAR_MIN <#if expression <= 0>
```

The macro yields the minimum value for type `wchar_t`.

WEOF

```
#define WEOF <wint_t constant expression>
```

The macro yields the return value, of type `wint_t`, used to signal the end of a wide stream or to report an error condition.

btowc

```
wint_t btowc(int c);
```

The function returns `WEOF` if `c` equals `EOF`. Otherwise, it converts `(unsigned char)c` as a one-byte multibyte character beginning in the initial conversoon state, as if by calling `mbrtowc`. If the conversion succeeds, the function returns the wide-character conversion. Otherwise, it returns `WEOF`.

fgetwc

```
wint_t fgetwc(FILE *stream);
```

The function reads the next wide character `c` (if present) from the input stream `stream`, advances the file-position indicator (if defined), and returns `(wint_t)c`. If the function sets either the end-of-file indicator or the error indicator, it returns WEOF.

fgetws

```
wchar_t *fgetws(wchar_t *s, int n, FILE *stream);
```

The function reads wide characters from the input stream `stream` and stores them in successive elements of the array beginning at `s` and continuing until it stores `n - 1` wide characters, stores an *NL* wide character, or sets the end-of-file or error indicators. If `fgetws` stores any wide characters, it concludes by storing a null wide character in the next element of the array. It returns `s` if it stores any wide characters and it has not set the error indicator for the stream; otherwise, it returns a null pointer. If it sets the error indicator, the array contents are indeterminate.

fputwc

```
wint_t fputwc(wchar_t c, FILE *stream);
```

The function writes the wide character `c` to the output stream `stream`, advances the file-position indicator (if defined), and returns `(wint_t)c`. If the function sets the error indicator for the stream, it returns WEOF.

fputws

```
int fputws(const wchar_t *s, FILE *stream);
```

The function accesses wide characters from the string `s` and writes them to the output stream `stream`. The function does not write the terminating null wide character. It returns a nonnegative value if it has not set the error indicator; otherwise, it returns WEOF.

fwide

```
int fwide(FILE *stream, int mode);
```

The function determines the orientation of the stream `stream`. If `mode` is greater than zero, it first attempts to make the stream wide oriented. If `mode` is less than zero, it first attempts to make the stream byte oriented. In any event, the function returns:

- a value greater than zero if the stream is left wide oriented

- zero if the stream is left unbound
- a value less than zero if the stream is left byte oriented

In no event will the function alter the orientation of a stream once it has been oriented.

fwprintf

```
int fwprintf(FILE *stream, const wchar_t *format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stream`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

fwscanf

```
int fwscanf(FILE *stream, const wchar_t *format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned character from the stream `stream`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicator for the stream.

getwc

```
wint_t getwc(FILE *stream);
```

The function has the same effect as `fgetwc(stream)` except that a macro version of `getwc` can evaluate `stream` more than once.

getwchar

```
wint_t getwchar(void);
```

The function has the same effect as `fgetwc(stdin)`.

mbrlen

```
size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
```

The function is equivalent to the call:

```
mbrtowc(0, s, n, ps != 0 ? ps : &internal)
```

where `internal` is an object of type `mbstate_t` internal to the `mbrlen` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

The function returns:

- `(size_t)-2` if, after converting all `n` characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` if the function detects an encoding error before completing the next multibyte character, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- zero, if the next completed character is a null character, in which case the resulting conversion state is the initial conversion state
- `x,` the number of bytes needed to complete the next muitibyte character, in which case the resulting conversion state indicates that `x` bytes have been converted

Thus, `mbrlen` effectively returns the number of bytes that would be consumed in successfully converting a multibyte character to a wide character (without storing the converted wide character), or an error code if the conversion cannot succeed.

mbrtowc

```
size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

The function determines the number of bytes in a multibyte string that completes the next multibyte character, if possible.

If `ps` is not a null pointer, the conversion state for the multibyte string is assumed to be `*ps`. Otherwise, it is assumed to be `&internal`, where `internal` is an object of type `mbstate_t` internal to the `mbrtowc` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If `s` is not a null pointer, the function determines `x,` the number of bytes in the multibyte string `s` that complete or contribute to the next multibyte character. (`x` cannot be greater than `n`.) Otherwise, the function effectively returns `mbrtowc(0, "", 1, ps)`, ignoring `pwc` and `n`. (The function thus returns zero only if the conversion state indicates that no incomplete multibyte character is pending from a previous call to `mbrlen`, `mbrtowc`, or `mbsrtowcs` for the same string and conversion state.)

If `pwc` is not a null pointer, the function converts a completed multibyte character to its corresponding wide-character value and stores that value in `*pwc`.

The function returns:

- `(size_t)-2` if, after converting all `n` characters, the resulting conversion state indicates an incomplete multibyte character
- `(size_t)-1` if the function detects an encoding error before completing the next multibyte

character, in which case the function stores the value EILSEQ in errno and leaves the resulting conversion state undefined

- zero, if the next completed character is a null character, in which case the resulting conversion state is the initial conversion state
- x, the number of bytes needed to complete the next muitibyte character, in which case the resulting conversion state indicates that x bytes have been converted

mbsinit

```
int mbsinit(const mbstate_t *ps);
```

The function returns a nonzero value if ps is a null pointer or if *ps designates an initial conversion state. Otherwise, it returns zero.

mbsrtowcs

```
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t
*ps);
```

The function converts the multibyte string beginning at *src to a sequence of wide characters as if by repeated calls of the form:

```
x = mbrtowc(dst, *src, n, ps != 0 ? ps : &internal)
```

where n is some value > 0 and internal is an object of type mbstate_t internal to the mbsrtowcs function. At program startup, internal is initialized to the initial conversion state. No other library function alters the value stored in internal.

If dst is not a null pointer, the mbsrtowcs function stores at most len wide characters by calls to mbrtowc. The function effectively increments dst by one and *src by x after each call to mbrtowc that stores a converted wide character. After a call that returns zero, mbsrtowcs stores a null wide character at dst and stores a null pointer at *src.

If dst is a null pointer, len is effectively assigned a large value.

The function returns:

- (size_t)-1, if a call to mbrtowc returns (size_t)-1, indicating that it has detected an encoding error before completing the next multibyte character
- the number of multibyte characters successfully converted, not including the terminating null character

mbstate_t

```
typedef o-type mbstate_t;
```

The type is an object type *o-type* that can represent a conversion state for any of the functions `mbrlen`, `mbrtowc`, `mbsrtowcs`, `wcrtomb`, or `wcsrtombs`. A definition of the form:

```
mbstate_t mbst = {0};
```

ensures that `mbst` represents the initial conversion state. Note, however, that other values stored in an object of type `mbstate_t` can also represent this state. To test safely for this state, use the function `mbsinit`.

putwc

```
wint_t putwc(wchar_t c, FILE *stream);
```

The function has the same effect as `fputwc(c, stream)` except that a macro version of `putwc` can evaluate `stream` more than once.

putwchar

```
wint_t putwchar(wchar_t c);
```

The function has the same effect as `fputwc(c, stdout)`.

size_t

```
typedef ui-type size_t;
```

The type is the unsigned integer type *`ui-type`* of an object that you declare to store the result of the *sizeof* operator.

swprintf

```
int swprintf(wchar_t *s, size_t n, const wchar_t *format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a null wide character in the next location of the array. It returns the number of wide characters generated -- not including the null wide character.

swscanf

```
int swscanf(const wchar_t *s, const wchar_t *format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It accesses each scanned character from successive locations of the array object whose first element has the address `s`. It returns the number of items matched and assigned, or it returns `EOF` if the function does not store values before it accesses a null wide character from the array.

tm

```
struct tm;
```

`struct tm` contains members that describe various properties of the calendar time. The declaration in this header leaves `struct tm` an incomplete type. Include the header `<time.h>` to complete the type.

ungetwc

```
wint_t ungetwc(wint_t c, FILE *stream);
```

If `c` is not equal to `WEOF`, the function stores `(wchar_t)c` in the object whose address is `stream` and clears the end-of-file indicator. If `c` equals `WEOF` or the store cannot occur, the function returns `WEOF`; otherwise, it returns `(wchar_t)c`. A subsequent library function call that reads a wide character from the stream `stream` obtains this stored value, which is then forgotten.

Thus, you can effectively push back a wide character to a stream after reading a wide character.

vfwprintf

```
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stream`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

vswprintf

```
int vswprintf(wchar_t *s, size_t n, const wchar_t *format, va_list arg);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and stores each generated wide character in successive locations of the array object whose first element has the address `s`. The function concludes by storing a null wide character in the next location of the array. It returns the number of characters generated -- not including the null wide character.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

vwprintf

```
int vwprintf(const wchar_t *format, va_list arg);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stdout`. It returns the number of characters generated, or a negative value if the function sets the error indicator for the stream.

The function accesses additional arguments by using the context information designated by `ap`. The program must execute the macro `va_start` before it calls the function, and then execute the macro `va_end` after the function returns.

wchar_t

```
typedef i-type wchar_t; [keyword in C++]
```

The type is the integer type *`i-type`* of a wide-character constant, such as `L'X'`. You declare an object of type `wchar_t` to hold a wide character.

wcrtomb

```
size_t wcrtomb(char *s, wchar_t wc, mbstate_t *ps);
```

The function determines the number of bytes needed to represent the wide character `wc` as a multibyte character, if possible. (Not all values representable as type `wchar_t` are necessarily valid wide-character codes.)

If `ps` is not a null pointer, the conversion state for the multibyte string is assumed to be `*ps`. Otherwise, it is assumed to be `&internal`, where `internal` is an object of type `mbstate_t` internal to the

`wcrtomb` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If `s` is not a null pointer and `wc` is a valid wide-character code, the function determines `x`, the number of bytes needed to represent `wc` as a multibyte character, and stores the converted bytes in the array of *char* beginning at `s`. (`x` cannot be greater than `MB_CUR_MAX`.) If `wc` is a null wide character, the function stores any shift sequence needed to restore the initial shift state. followed by a null byte. The resulting conversion state is the initial conversion state.

If `s` is a null pointer, the function effectively returns `wcrtomb(buf, L'\0', ps)`, where `buf` is a buffer internal to the function. (The function thus returns the number of bytes needed to restore the initial conversion state and to terminate the multibyte string pending from a previous call to `wcrtomb` or `wcsrtombs` for the same string and conversion state.)

The function returns:

- `(size_t)-1` if `wc` is an invalid wide-character code, in which case the function stores the value `EILSEQ` in `errno` and leaves the resulting conversion state undefined
- `x`, the number of bytes needed to complete the next muitibyte character, in which case the resulting conversion state indicates that `x` bytes have been generated

wcscat

```
wchar_t *wcscat(wchar_t *s1, const wchar_t *s2);
```

The function copies the wide string `s2`, including its terminating null wide character, to successive elements of the array that stores the wide string `s1`, beginning with the element that stores the terminating null wide character of `s1`. It returns `s1`.

wcschr

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

The function searches for the first element of the wide string `s` that equals `c`. It considers the terminating null wide character as part of the wide string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

wcscmp

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

The function compares successive elements from two wide strings, `s1` and `s2`, until it finds elements that are not equal.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a

positive number.

- Otherwise, the function returns a negative number.

wcscoll

```
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

The function compares two wide strings, `s1` and `s2`, using a comparison rule that depends on the current locale. If `s1` compares greater than `s2` by this rule, the function returns a positive number. If the two wide strings compare equal, it returns zero. Otherwise, it returns a negative number.

wcscpy

```
wchar_t *wcscpy(wchar_t *s1, const wchar_t *s2);
```

The function copies the wide string `s2`, including its terminating null wide character, to successive elements of the array whose first element has the address `s1`. It returns `s1`.

wcscspn

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

The function searches for the first element `s1[i]` in the wide string `s1` that equals *any one* of the elements of the wide string `s2` and returns `i.` Each terminating null wide character is considered part of its wide string.

wcsftime

```
size_t wcsftime(wchar_t *s, size_t maxsize, const wchar_t *format,
const struct tm *timeptr);
```

The function generates formatted text, under the control of the format `format` and the values stored in the time structure `*tptr`. It stores each generated wide character in successive locations of the array object of size `n` whose first element has the address `s`. The function then stores a null wide character in the next location of the array. It returns `x`, the number of wide characters generated, if `x < n`; otherwise, it returns zero, and the values stored in the array are indeterminate.

For each wide character other than % in the format, the function stores that wide character in the array object. Each occurrence of % followed by another character in the format is a **conversion specifier.** For each conversion specifier, the function stores a replacement wide character sequence. Conversion specifiers are the same as for the function `strftime`. The current locale category `LC_TIME` can affect these replacement character sequences.

wcslen

```
size_t wcslen(const wchar_t *s);
```

The function returns the number of wide characters in the wide string `s`, *not* including its terminating null wide character.

wcsncat

```
wchar_t *wcsncat(wchar_t *s1, const wchar_t *s2, size_t n);
```

The function copies the wide string `s2`, *not* including its terminating null wide character, to successive elements of the array that stores the wide string `s1`, beginning with the element that stores the terminating null wide character of `s1`. The function copies no more than `n` wide characters from `s2`. It then stores a null wide character, in the next element to be altered in `s1`, and returns `s1`.

wcsncmp

```
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

The function compares successive elements from two wide strings, `s1` and `s2`, until it finds elements that are not equal or until it has compared the first `n` elements of the two wide strings.

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a positive number.
- Otherwise, it returns a negative number.

wcsncpy

```
wchar_t *wcsncpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

The function copies the wide string `s2`, *not* including its terminating null wide character, to successive elements of the array whose first element has the address `s1`. It copies no more than `n` wide characters from `s2`. The function then stores zero or more null wide characters in the next elements to be altered in `s1` until it stores a total of `n` wide characters. It returns `s1`.

wcspbrk

```
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
```

The function searches for the first element `s1[i]` in the wide string `s1` that equals *any one* of the elements of the wide string `s2`. It considers each terminating null wide character as part of its wide string. If `s1[i]` is not the terminating null wide character, the function returns `&s1[i]`; otherwise, it

returns a null pointer.

wcsrchr

```
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

The function searches for the last element of the wide string `s` that equals `c`. It considers the terminating null wide character as part of the wide string. If successful, the function returns the address of the matching element; otherwise, it returns a null pointer.

wcsrtombs

```
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);
```

The function converts the wide-character string beginning at `*src` to a sequence of multibyte characters as if by repeated calls of the form:

```
x = wcrtomb(dst ? dst : buf, *src, ps != 0 ? ps : &internal)
```

where `buf` is an array of type *char* and `internal` is an object of type `mbstate_t`, both internal to the `wcsrtombs` function. At program startup, `internal` is initialized to the initial conversion state. No other library function alters the value stored in `internal`.

If `dst` is not a null pointer, the `wcsrtombs` function stores at most `len` bytes by calls to `wcrtomb`. The function effectively increments `dst` by `x` and `*src` by one after each call to `wcrtomb` that stores a *complete* converted multibyte character in the remaining space available. After a call that stores a complete null multibyte character at `dst` (including any shift sequence needed to restore the initial shift state), the function stores a null pointer at `*src`.

If `dst` is a null pointer, `len` is effectively assigned a large value.

The function returns:

- `(size_t)-1`, if a call to `wcrtomb` returns `(size_t)-1`, indicating that it has detected an invalid wide-character code
- the number of bytes successfully converted, not including the terminating null byte

wcsspn

```
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

The function searches for the first element `s1[i]` in the wide string `s1` that equals *none* of the elements of the wide string `s2` and returns `i`. It considers the terminating null wide character as part of the wide string `s1` only.

wcsstr

```
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

The function searches for the first sequence of elements in the wide string `s1` that matches the sequence of elements in the wide string `s2`, *not* including its terminating null wide character. If successful, the function returns the address of the matching first element; otherwise, it returns a null pointer.

wcstod

```
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

The function converts the initial wide characters of the wide string `s` to an equivalent value `x` of type *double*. If `endptr` is not a null pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtod`, where each wide character `wc` is converted as if by calling `wctob(wc)`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtod` for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If a range error occurs, `wcstod` behaves exactly as the functions declared in `<math.h>`.

wcstok

```
wchar_t *wcstok(wchar_t *s1, const wchar_t *s2, wchar_t **ptr);
```

If `s1` is not a null pointer, the function begins a search of the wide string `s1`. Otherwise, it begins a search of the wide string whose address was last stored in `*ptr` on an earlier call to the function, as described below. The search proceeds as follows:

1. The function searches the wide string for `begin`, the address of the first element that equals *none* of the elements of the wide string `s2` (a set of token separators). It considers the terminating null character as part of the search wide string only.
2. If the search does not find an element, the function stores the address of the terminating null wide character in `*ptr` (so that a subsequent search beginning with that address will fail) and returns a null pointer. Otherwise, the function searches from `begin` for `end`, the address of the first element that equals *any one* of the elements of the wide string `s2`. It again considers the terminating null wide character as part of the search string only.
3. If the search does not find an element, the function stores the address of the terminating null wide character in `*ptr`. Otherwise, it stores a null wide character in the element whose address is `end`. Then it stores the address of the next element after `end` in `*ptr` (so that a subsequent search beginning with that address will continue with the remaining elements of the string) and returns `begin`.

wcstol

```
long wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

The function converts the initial wide characters of the wide string `s` to an equivalent value `x` of type *long*. If `endptr` is not a null pointer, the function stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

The initial wide characters of the wide string `s` must match the same pattern as recognized by the function `strtol`, with the same `base` argument, where each wide character `wc` is converted as if by calling `wctob(wc)`.

If the wide string `s` matches this pattern, its equivalent value is the value returned by `strtol`, with the same `base` argument, for the converted sequence. If the wide string `s` does not match a valid pattern, the value stored in `*endptr` is `s`, and `x` is zero. If the equivalent value is too large in magnitude to represent as type *long*, `wcstol` stores the value of `ERANGE` in `errno` and returns either `LONG_MAX` if `x` is positive or `LONG_MIN` if `x` is negative.

wcstoul

```
unsigned long wcstoul(const wchar_t *nptr, wchar_t **endptr, int
base);
```

The function converts the initial wide characters of the wide string `s` to an equivalent value `x` of type *unsigned long*. If `endptr` is not a null pointer, it stores a pointer to the unconverted remainder of the wide string in `*endptr`. The function then returns `x`.

`wcstoul` converts strings exactly as does `wcstol`, but checks only if the equivalent value is too large to represent as type *unsigned long*. In this case, `wcstoul` stores the value of `ERANGE` in `errno` and returns `ULONG_MAX`.

wcsxfrm

```
size_t wcsxfrm(wchar_t *s1, const wchar_t *s2, size_t n);
```

The function stores a wide string in the array whose first element has the address `s1`. It stores no more than `n` wide characters, *including* the terminating null wide character, and returns the number of wide characters needed to represent the entire wide string, *not* including the terminating null wide character. If the value returned is `n` or greater, the values stored in the array are indeterminate. (If `n` is zero, `s1` can be a null pointer.)

`wcsxfrm` generates the wide string it stores from the wide string `s2` by using a transformation rule that depends on the current locale. For example, if `x` is a transformation of `s1` and `y` is a transformation of `s2`, then `wcscmp(x, y)` returns the same value as `wcscoll(s1, s2)`.

wctob

```
int wctob(wint_t c);
```

The function determines whether `c` can be represented as a one-byte multibyte character `x`, beginning in the initial shift state. (It effectively calls `wcrtomb` to make the conversion.) If so, the function returns `x`. Otherwise, it returns `WEOF`.

wint_t

```
typedef i_type wint_t;
```

The type is the integer type *`i_type`* that can represent all values of type `wchar_t` as well as the value of the macro `WEOF`, and that doesn't change when promoted.

wmemchr

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [not in C++]
const wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n); [C++ only]
wchar_t *wmemchr(wchar_t *s, wchar_t c, size_t n); [C++ only]
```

The function searches for the first element of an array beginning at the address `s` with size `n`, that equals `c`. If successful, it returns the address of the matching element; otherwise, it returns a null pointer.

wmemcmp

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

The function compares successive elements from two arrays beginning at the addresses `s1` and `s2` (both of size `n`), until it finds elements that are not equal:

- If all elements are equal, the function returns zero.
- If the differing element from `s1` is greater than the element from `s2`, the function returns a positive number.
- Otherwise, the function returns a negative number.

wmemcpy

```
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

The function copies the array beginning at the address `s2` to the array beginning at the address `s1` (both of size `n`). It returns `s1`. The elements of the arrays can be accessed and stored in any order.

wmemmove

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

The function copies the array beginning at `s2` to the array beginning at `s1` (both of size `n`). It returns `s1`. If the arrays overlap, the function accesses each of the element values from `s2` before it stores a new value in that element, so the copy is not corrupted.

wmemset

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

The function stores `c` in each of the elements of the array beginning at `s`, with size `n`. It returns `s`.

wprintf

```
int wprintf(const wchar_t *format, ...);
```

The function generates formatted text, under the control of the format `format` and any additional arguments, and writes each generated wide character to the stream `stdout`. It returns the number of wide characters generated, or it returns a negative value if the function sets the error indicator for the stream.

wscanf

```
int wscanf(const wchar_t *format, ...);
```

The function scans formatted text, under the control of the format `format` and any additional arguments. It obtains each scanned wide character from the stream `stdin`. It returns the number of input items matched and assigned, or it returns `EOF` if the function does not store values before it sets the end-of-file or error indicators for the stream.

See also the **Table of Contents** and the **Index**.

<wctype.h> [Added with Amendment 1]

```
int iswalnum(wint_t c);
int iswalpha(wint_t c);
int iswcntrl(wint_t c);
int iswctype(wint_t c, wctype_t category);
int iswdigit(wint_t c);
int iswgraph(wint_t c);
int iswlower(wint_t c);
int iswprint(wint_t c);
int iswpunct(wint_t c);
int iswspace(wint_t c);
int iswupper(wint_t c);
int iswxdigit(wint_t c);
wint_t towctrans(wint_t c, wctrans_t category);
wint_t towlower(wint_t c);
wint_t towupper(wint_t c);
wctrans_t wctrans(const char *property);
typedef s_type wctrans_t;
wctype_t wctype(const char *property);
typedef s_type wctype_t;
typedef i_type wint_t;
```

Include the standard header **<wctype.h>** to declare several functions that are useful for classifying and mapping codes from the target wide-character set.

Every function that has a parameter of type `wint_t` can accept the value of the macro `WEOF` or any valid wide-character code (of type `wchar_t`). Thus, the argument can be the value returned by any of the functions: `btowc`, `fgetwc`, `fputwc`, `getwc`, `getwchar`, `putwc`, `putwchar`, `towctrans`, `towlower`, `towupper`, or `ungetwc`. You must not call these functions with other wide-character argument values.

The **wide-character classification** functions are strongly related to the (byte) character classification functions. Each function `is`*XXX* has a corresponding wide-character classification function `isw`*XXX*. Moreover, the wide-character classification functions are interrelated much the same way as their corresponding byte functions, with two added provisos:

- The function `iswprint`, unlike `isprint`, can return a nonzero value for additional space characters besides the wide-character equivalent of *`space`* (`L' '`). Any such additional characters return a nonzero value for `iswspace` and return zero for `iswgraph` or `iswpunct`.
- The characters in each wide-character class are a superset of the characters in the corresponding byte class. If the call `is`*`XXX`*`(c)` returns a nonzero value, then the corresponding call `isw`*`XXX`*`(btowc(c))` also returns a nonzero value.

An implementation can define additional characters that return nonzero for some of these functions. Any character set can contain additional characters that return nonzero for:

- `iswpunct` (provided the characters cause `iswalnum` to return zero)
- `iswcntrl` (provided the characters cause `iswprint` to return zero)

Moreover, a locale other than the `"C"` locale can define additional characters for:

- `iswalpha`, `iswupper`, and `iswlower` (provided the characters cause `iswcntrl`, `iswdigit`, `iswpunct`, and `iswspace` to return zero)
- `iswspace` (provided the characters cause `iswpunct` to return zero)

Note that the last rule differs slightly from the corresponding rule for the function `isspace`, as indicated above. Note also that an implementation can define a locale other than the `"C"` locale in which a character can cause `iswalpha` (and hence `iswalnum`) to return nonzero, yet still cause `iswupper` and `iswlower` to return zero.

WEOF

```
#define WEOF <wint_t constant expression>
```

The macro yields the return value, of type `wint_t`, used to signal the end of a wide stream or to report an error condition.

iswalnum

```
int iswalnum(wint_t c);
```

The function returns nonzero if `c` is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
```

or any other locale-specific alphabetic character.

iswalpha

```
int iswalpha(wint_t c);
```

The function returns nonzero if `c` is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

or any other locale-specific alphabetic character.

iswcntrl

```
int iswcntrl(wint_t c);
```

The function returns nonzero if `c` is any of:

BEL BS CR FF HT NL VT

or any other implementation-defined control character.

iswctype

```
int iswctype(wint_t c, wctype_t category);
```

The function returns nonzero if `c` is any character in the category `category`. The value of `category` must have been returned by an earlier successful call to `wctype`.

iswdigit

```
int iswdigit(wint_t c);
```

The function returns nonzero if `c` is any of:

```
0 1 2 3 4 5 6 7 8 9
```

iswgraph

```
int iswgraph(wint_t c);
```

The function returns nonzero if `c` is any character for which either `iswalnum` or `iswpunct` returns nonzero.

iswlower

```
int iswlower(wint_t c);
```

The function returns nonzero if `c` is any of:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

or any other locale-specific lowercase character.

iswprint

```
int iswprint(wint_t c);
```

The function returns nonzero if `c` is *space*, a character for which `iswgraph` returns nonzero, or an implementation-defined subset of the characters for which `iswspace` returns nonzero.

iswpunct

```
int iswpunct(wint_t c);
```

The function returns nonzero if `c` is any of:

```
! " # % & ' ( ) ; <
= > ? [ \ ] * + , -
. / : ^ _ { | } ~
```

or any other implementation-defined punctuation character.

iswspace

```
int iswspace(wint_t c);
```

The function returns nonzero if `c` is any of:

CR FF HT NL VT space

or any other locale-specific space character.

iswupper

```
int iswupper(wint_t c);
```

The function returns nonzero if `c` is any of:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

or any other locale-specific uppercase character.

iswxdigit

```
int iswxdigit(wint_t c);
```

The function returns nonzero if `c` is any of

`0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

towctrans

```
wint_t towctrans(wint_t c, wctrans_t category);
```

The function returns the transformation of the character `c`, using the transform in the category `category`. The value of `category` must have been returned by an earlier successful call to `wctrans`.

towlower

```
wint_t towlower(wint_t c);
```

The function returns the corresponding lowercase letter if one exists and if `iswupper(c)`; otherwise, it returns `c`.

towupper

```
wint_t towupper(wint_t c);
```

The function returns the corresponding uppercase letter if one exists and if `iswlower(c)`; otherwise, it returns `c`.

wctrans

```
wctrans_t wctrans(const char *property);
```

The function determines a mapping from one set of wide-character codes to another. If the `LC_CTYPE` category of the current locale does not define a mapping whose name matches the property string `property`, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to `towctrans`.

The following pairs of calls have the same behavior in all locales (but an implementation can define additional mappings even in the "C" locale):

```
towlower(c) same as towctrans(c, wctrans("tolower"))
towupper(c) same as towctrans(c, wctrans("toupper"))
```

wctrans_t

```
typedef s_type wctrans_t;
```

The type is the scalar type *s-type* that can represent locale-specific character mappings, as specified by the return value of wctrans.

wctype

```
wctype_t wctype(const char *property);
```

```
wctrans_t wctrans(const char *property);
```

The function determines a classification rule for wide-character codes. If the LC_CTYPE category of the current locale does not define a classification rule whose name matches the property string `property`, the function returns zero. Otherwise, it returns a nonzero value suitable for use as the second argument to a subsequent call to towctrans.

The following pairs of calls have the same behavior in all locales (but an implementation can define additional classification rules even in the "C" locale):

```
iswalnum(c) same as iswctype(c, wctype("alnum"))
iswalpha(c) same as iswctype(c, wctype("alpha"))
iswcntrl(c) same as iswctype(c, wctype("cntrl"))
iswdigit(c) same as iswctype(c, wctype("digit"))
iswgraph(c) same as iswctype(c, wctype("graph"))
iswlower(c) same as iswctype(c, wctype("lower"))
iswprint(c) same as iswctype(c, wctype("print"))
iswpunct(c) same as iswctype(c, wctype("punct"))
iswspace(c) same as iswctype(c, wctype("space"))
iswupper(c) same as iswctype(c, wctype("upper"))
iswxdigit(c) same as iswctype(c, wctype("xdigit"))
```

wctype_t

```
typedef s_type wctype_t;
```

The type is the scalar type *s-type* that can represent locale-specific character classifications, as specified by the return value of wctype.

wint_t

```
typedef i_type wint_t;
```

The type is the integer type *i_type* that can represent all values of type wchar_t as well as the value of the macro WEOF, and that doesn't change when promoted.

See also the **Table of Contents** and the **Index**.

<fstream.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#include <fstream>
using namespace std;
```

Include the traditional header **<fstream.h>** to effectively include the standard header <fstream> and hoist its names outside the std namespace.

In this implementation, *all* names are hoisted, to provide a more traditional library environment.

See also the **Table of Contents** and the **Index**.

<iomanip.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#include <iomanip>
using namespace std;
```

Include the traditional header **<iomanip.h>** to effectively include the standard header <iomanip> and hoist its names outside the std namespace.

In this implementation, *all* names are hoisted, to provide a more traditional library environment.

See also the **Table of Contents** and the **Index**.

<iostream.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#include <iostream>
using namespace std;
```

Include the traditional header **<iostream.h>** to effectively include the standard header <iostream> and hoist its names outside the std namespace.

In this implementation, *all* names are hoisted, to provide a more traditional library environment. Moreover, **<iostream.h>** does *not* declare the wide oriented stream objects wcin, wcout, wcerr, and wclog.

See also the **Table of Contents** and the **Index**.

<new.h>

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
#include <new>
using namespace std;
```

Include the traditional header **<new.h>** to effectively include the standard header <new> and hoist its names outside the std namespace.

In this implementation, *all* names are hoisted, to provide a more traditional library environment.

See also the **Table of Contents** and the **Index**.

<stl.h>

```
#include <stl>
using namespace std;
```

Include the traditional header **<stl.h>** to effectively include all the standard headers that constitute the Standard Template Library (STL) and hoist their names outside the `std` namespace. The header also redefines the STL container template classes to match their more traditional definitions.

`<stl.h>` is an unsupported header supplied to aid the migration of existing code that uses STL to standard-conforming form.

A few special conventions are introduced into this document specifically for this particular **implementation** of the Standard Template library. Because the draft C++ Standard is still changing, not all implementations support all the features described here. Hence, this implementation introduces macros, or alternative declarations, where necessary to provide reasonable substitutes for the capabilities required by the current draft C++ Standard.

See also the **Table of Contents** and the **Index**.

Characters

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

Characters play a central role in Standard C. You represent a C program as one or more **source files**. The translator reads a source file as a text stream consisting of characters that you can read when you display the stream on a terminal screen or produce hard copy with a printer. You often manipulate text when a C program executes. The program might produce a text stream that people can read, or it might read a text stream entered by someone typing at a keyboard or from a file modified using a text editor. This document describes the characters that you use to write C source files and that you manipulate as streams when executing C programs.

Character Sets

When you write a program, you express C source files as text lines containing characters from the **source character set**. When a program executes in the **target environment**, it uses characters from the **target character set**. These character sets are related, but need not have the same encoding or all the same members.

Every character set contains a distinct code value for each character in the **basic C character set**. A character set can also contain additional characters with other code values. For example:

- The **character constant** `'x'` becomes the value of the code for the character corresponding to `x` in the target character set.
- The **string literal** `"xyz"` becomes a sequence of character constants stored in successive bytes of memory, followed by a byte containing the value zero:
  `{'x', 'y', 'z', '\0'}`

A string literal is one way to specify a **null-terminated string**, an array of zero or more bytes followed by a byte containing the value zero.

Visible graphic characters in the basic C character set:

Form	**Members**
letter	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

```
digit        0 1 2 3 4 5 6 7 8 9

underscore   _

punctuation  ! " # % & ' ( ) * + , - . / :
             ; < = > ? [ \ ] ^ { | } ~
```

Additional graphic characters in the basic C character set:

Character	Meaning
space	**leave blank space**
BEL	**signal an alert (BELl)**
BS	**go back one position (BackSpace)**
FF	**go to top of page (Form Feed)**
NL	**go to start of next line (NewLine)**
CR	**go to start of this line (Carriage Return)**
HT	**go to next Horizontal Tab stop**
VT	**go to next Vertical Tab stop**

The code value zero is reserved for the **null character** which is always in the target character set. Code values for the basic C character set are positive when stored in an object of type *char*. Code values for the digits are contiguous, with increasing value. For example, `'0' + 5` equals `'5'`. Code values for any two letters are *not* necessarily contiguous.

Character Sets and Locales

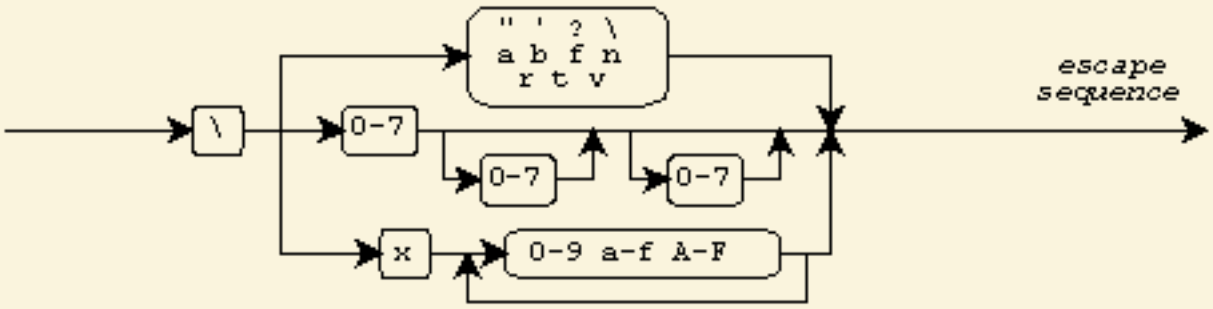
An implementation can support multiple locales, each with a different character set. A locale summarizes conventions peculiar to a given culture, such as how to format dates or how to sort names. To change locales and, therefore, target character sets while the program is running, use the function `setlocale`. The translator encodes character constants and string literals for the `"C"` locale, which is the locale in effect at program startup.

Escape Sequences

Within character constants and string literals, you can write a variety of **escape sequences**. Each escape sequence determines the code value for a single character. You use escape sequences to represent character codes:

- you cannot otherwise write (such as `\n`)
- that can be difficult to read properly (such as `\t`)
- that might change value in different target character sets (such as `\a`)
- that must not change in value among different target environments (such as `\0`)

An escape sequence takes the form:



Mnemonic escape sequences help you remember the characters they represent:

Character	Escape Sequence
"	\"
'	\'
?	\?
\	\\
BEL	\a
BS	\b
FF	\f
NL	\n
CR	\r
HT	\t
VT	\v

Numeric Escape Sequences

You can also write **numeric escape sequences** using either octal or hexadecimal digits. An **octal escape sequence** takes one of the forms:

> `\`*d* **or** `\`*dd* **or** `\`*ddd*

The escape sequence yields a code value that is the numeric value of the 1-, 2-, or 3-digit octal number following the backslash (`\`). Each *d* can be any digit in the range `0-7`.

A **hexadecimal escape sequence** takes one of the forms:

> `\x`*h* **or** `\x`*hh* **or ...**

The escape sequence yields a code value that is the numeric value of the arbitrary-length hexadecimal number following the backslash (`\`). Each *h* can be any decimal digit `0-9`, or any of the letters `a-f` or `A-F`. The letters represent the digit values 10-15, where either `a` or `A` has the value 10.

A numeric escape sequence terminates with the first character that does not fit the digit pattern. Here are some examples:

- You can write the null character as `'\0'`.
- You can write a newline character (*NL*) within a string literal by writing:
  `"hi\n"` **which becomes the array**

```
{'h', 'i', '\n', 0}
```

- You can write a string literal that begins with a specific numeric value:
  `"\3abc"` **which becomes the array**
  `{3, 'a', 'b', 'c', 0}`
- You can write a string literal that contains the hexadecimal escape sequence `\xF` followed by the digit `3` by writing two string literals:
  `"\xF" "3"` **which becomes the array**
  `{0xF, '3', 0}`

Trigraphs

A **trigraph** is a sequence of three characters that begins with two question marks (`??`). You use trigraphs to write C source files with a character set that does not contain convenient graphic representations for some punctuation characters. (The resultant C source file is not necessarily more readable, but it is unambiguous.)

The list of all **defined trigraphs** is:

Character	Trigraph
[	??(
\	??/
]	??)
^	??'
{	??<
|	??!
}	??>
~	??-
#	??=

These are the only trigraphs. The translator does not alter any other sequence that begins with two question marks.

For example, the expression statements:

```
printf("Case ??=3 is done??/n");
printf("You said what????/n");
```

are equivalent to:

```
printf("Case #3 is done\n");
printf("You said what??\n");
```

The translator replaces each trigraph with its equivalent single character representation in an early phase of translation. You can always treat a trigraph as a single source character.

Multibyte Characters

A source character set or target character set can also contain **multibyte characters** (sequences of one or more bytes). Each sequence represents a single character in the **extended character set**. You use multibyte characters to represent large sets of characters, such as Kanji. A multibyte character can be a one-byte sequence that is a character from the basic C character set, an additional one-byte sequence that is implementation defined, or an additional sequence of two or more bytes that is implementation defined.

Any multibyte encoding that contains sequences of two or more bytes depends, for its interpretation between bytes, on a **conversion state** determined by bytes earlier in the sequence of characters. In the **initial conversion state** if the byte immediately following matches one of the characters in the basic C character set, the byte must represent that character.

For example, the **EUC encoding** is a superset of ASCII. A byte value in the interval [0xA1, 0xFE] is the first of a two-byte sequence (whose second byte value is in the interval [0x80, 0xFF]). All other byte values are one-byte sequences. Since all members of the basic C character set have byte values in the range [0x00, 0x7F] in ASCII, EUC meets the requirements for a multibyte encoding in Standard C. Such a sequence is *not* in the initial conversion state immediately after a byte value in the interval [0xA1, 0xFe]. It is ill-formed if a second byte value is not in the interval [0x80, 0xFF].

Multibyte characters can also have a **state-dependent encoding**. How you interpret a byte in such an encoding depends on a conversion state that involves both a **parse state**, as before, and a **shift state**, determined by bytes earlier in the sequence of characters. The **initial shift state**, at the beginning of a new multibyte character, is also the initial conversion state. A subsequent **shift sequence** can determine an **alternate shift state**, after which all byte sequences (including one-byte sequences) can have a different interpretation. A byte containing the value zero, however, always represents the null character. It cannot occur as any of the bytes of another multibyte character.

For example, the **JIS encoding** is another superset of ASCII. In the initial shift state, each byte represents a single character, except for two three-byte shift sequences:

- The three-byte sequence `"\x1B$B"` shifts to two-byte mode. Subsequently, two successive bytes (both with values in the range [0x21, 0x7E]) constitute a single multibyte character.
- The three-byte sequence `"\x1B(B"` shifts back to the initial shift state.

JIS also meets the requirements for a multibyte encoding in Standard C. Such a sequence is *not* in the initial conversion state when partway through a three-byte shift sequence or when in two-byte mode.

(Amendment 1 adds the type `mbstate_t`, which describes an object that can store a conversion state. It also relaxes the above rules for generalized multibyte characters, which describe the encoding rules for a broad range of wide streams.)

You can write multibyte characters in C source text as part of a comment, a character constant, a string literal, or a filename in an *include* directive. How such characters print is implementation defined. Each sequence of multibyte characters that you write must begin and end in the initial shift state. The program can also include multibyte characters in null-terminated C strings used by several library functions,

including the format strings for `printf` and `scanf`. Each such character string must begin and end in the initial shift state.

Wide-Character Encoding

Each character in the extended character set also has an integer representation, called a **wide-character encoding**. Each extended character has a unique wide-character value. The value zero always corresponds to the **null wide character**. The type definition `wchar_t` specifies the integer type that represents wide characters.

You write a **wide-character constant** as `L'mbc'`, where `mbc` represents a single multibyte character. You write a **wide-character string literal** as `L"mbs"`, where `mbs` represents a sequence of zero or more multibyte characters. The wide-character string literal `L"xyz"` becomes a sequence of wide-character constants stored in successive bytes of memory, followed by a null wide character:
`{L'x', L'y', L'z', L'\0'}`

The following library functions help you convert between the multibyte and wide-character representations of extended characters: `btowc`, `mblen`, `mbrlen`, `mbrtowc`, `mbsrtowcs`, `mbstowcs`, `mbtowc`, `wcrtomb`, `wcsrtombs`, `wcstombs`, `wctob`, and `wctomb`.

The macro `MB_LEN_MAX` specifies the length of the longest possible multibyte sequence required to represent a single character defined by the implementation across supported locales. And the macro `MB_CUR_MAX` specifies the length of the longest possible multibyte sequence required to represent a single character defined for the current locale.

For example, the string literal `"hello"` becomes an array of six *char*:

```
{'h', 'e', 'l', 'l', 'o', 0}
```

while the wide-character string literal `L"hello"` becomes an array of six integers of type `wchar_t`:

```
{L'h', L'e', L'l', L'l', L'o', 0}
```

See also the **Table of Contents** and the **Index**.

Formatted Output

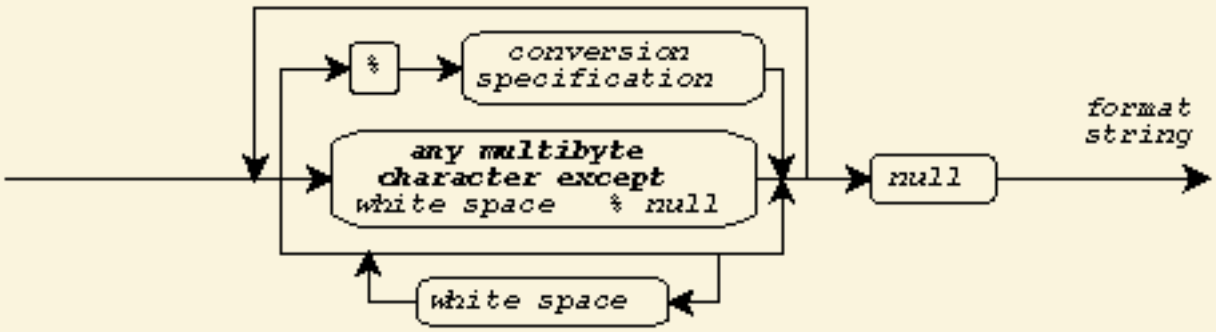
Print Formats · Print Functions · Print Conversion Specifiers

Several library functions help you convert data values from encoded internal representations to text sequences that are generally readable by people. You provide a format string as the value of the `format` argument to each of these functions, hence the term **formatted output**. The functions fall into two categories:

- The **byte print functions** (declared in `<stdio.h>`) convert internal representations to sequences of type *char*, and help you compose such sequences for display: fprintf, printf, sprintf, vfprintf, vprintf, and vsprintf. For these function, a format string is a multibyte string that begins and ends in the initial shift state.
- The **wide print functions** (declared in `<wchar.h>` and hence added with **Amendment 1**) convert internal representations to sequences of type `wchar_t`, and help you compose such sequences for display: fwprintf, swprintf, wprintf, vfwprintf, vswprintf, and vwprintf. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character `wc` from a format string or a stream is compared to a specific (byte) character `c` as if by evaluating the expression `wctob(wc) == c`.

Print Formats

A **format string** has the same syntax for both the print functions and the scan functions:



A format string consists of zero or more **conversion specifications** interspersed with literal text and **white space**. White space is a sequence of one or more characters `c` for which the call `isspace(c)` returns nonzero. (The characters defined as white space can change when you change the `LC_CTYPE` locale category.) For the print functions, a conversion specification is one of the print conversion specifications described below.

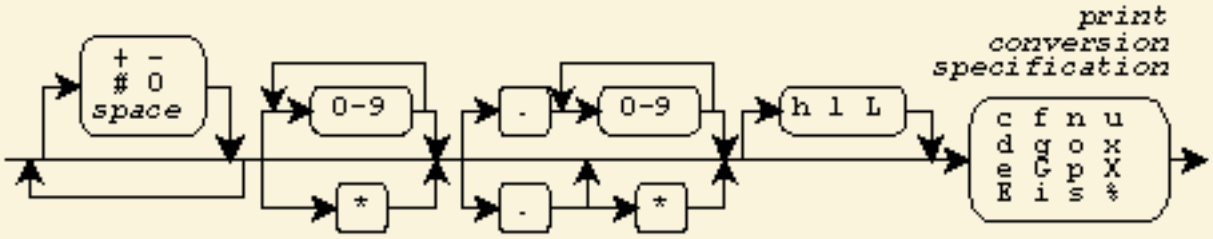
A print function scans the format string once from beginning to end to determine what conversions to perform. Every print function accepts a varying number of arguments, either directly or under control of an argument of type `va_list`. Some print conversion specifications in the format string use the next argument in the list. A print function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows:

- **integer conversions** are the **conversion specifiers** that end in `d`, `i`, `o`, `u`, `x`, or `X`
- **floating-point conversions** are the conversion specifiers that end in `e`, `E`, `f`, `g`, or `G`

Print Functions

For the print functions, literal text or white space in a format string generates characters that match the characters in the format string. A **print conversion specification** typically generates characters by converting the next argument value to a corresponding text sequence. A print conversion specification has the format:



Following the percent character (`%`) in the format string, you can write zero or more **format flags**:

- `-` -- to left-justify a conversion
- `+` -- to generate a plus sign for signed values that are positive
- *`space`* -- to generate a *`space`* for signed values that have neither a plus nor a minus sign
- `#` -- to prefix `0` on an `o` conversion, to prefix `0x` on an `x` conversion, to prefix `0X` on an `X` conversion, or to generate a decimal point and fraction digits that are otherwise suppressed on a floating-point conversion
- `0` -- to pad a conversion with leading zeros after any sign or prefix, in the absence of a minus (`-`) format flag or a specified precision

Following any format flags, you can write a **field width** that specifies the minimum number of characters to generate for the conversion. Unless altered by a format flag, the default behavior is to pad a short conversion on the left with *`space`* characters. If you write an asterisk (`*`) instead of a decimal number for a field width, then a print function takes the value of the next argument (which must be of type *int*) as the field width. If the argument value is negative, it supplies a `-` format flag and its magnitude is the field width.

Following any field width, you can write a dot (`.`) followed by a **precision** that specifies one of the following: the minimum number of digits to generate on an integer conversion; the number of fraction digits to generate on an `e`, `E`, or `f` conversion; the maximum number of significant digits to generate on a `g` or `G` conversion; or the maximum number of characters to generate from a C string on an `s` conversion.

If you write an ***** instead of a decimal number for a precision, a print function takes the value of the next argument (which must be of type *int*) as the precision. If the argument value is negative, the default precision applies. If you do not write either an * or a decimal number following the dot, the precision is zero.

Print Conversion Specifiers

Following any precision, you must write a one-character **print conversion specifier**, possibly preceded by a one-character qualifier. Each combination determines the type required of the next argument (if any) and how the library functions alter the argument value before converting it to a text sequence. The integer and floating-point conversions also determine what base to use for the text representation. If a conversion specifier requires a precision *p* and you do not provide one in the format, then the conversion specifier chooses a default value for the precision. The following table lists all defined combinations and their properties.

Conversion Specifier	Argument Type	Converted Value	Default Base	Precision
%c	int x	(unsigned char)x		
%lc	wint_t x	wchar_t a[2] = {x}		
%d	int x	(int)x	10	1
%hd	int x	(short)x	10	1
%ld	long x	(long)x	10	1
%e	double x	(double)x	10	6
%Le	long double x	(long double)x	10	6
%E	double x	(double)x	10	6
%E	long double x	(long double)x	10	6
%f	double x	(double)x	10	6
%Lf	long double x	(long double)x	10	6
%g	double x	(double)x	10	6
%Lg	long double x	(long double)x	10	6
%G	double x	(double)x	10	6
%LG	long double x	(long double)x	10	6
%i	int x	(int)x	10	1
%hi	int x	(short)x	10	1
%li	long x	(long)x	10	1
%n	int *x			
%hn	short *x			
%ln	long *x			
%o	int x	(unsigned int)x	8	1
%ho	int x	(unsigned short)x	8	1
%lo	long x	(unsigned long)x	8	1
%p	void *x	(void *)x		
%s	char x[]	x[0]...		**large**
%ls	wchar_t x[]	x[0]...		**large**

%u	int x	(unsigned int)x	10	1
%hu	int x	(unsigned short)x	10	1
%lu	long x	(unsigned long)x	10	1
%x	int x	(unsigned int)x	16	1
%hx	int x	(unsigned short)x	16	1
%lx	long x	(unsigned long)x	16	1
%X	int x	(unsigned int)x	16	1
%hX	int x	(unsigned short)x	16	1
%lX	long x	(unsigned long)x	16	1
%%	**none**	'%'		

The print conversion specifier determines any behavior not summarized in this table. In the following descriptions, *p* is the precision. Examples follow each of the print conversion specifiers. A single conversion can generate up to 509 characters.

You write **%c** to generate a single character from the converted value. For a wide stream, conversion of the character x occurs as if by calling btowc(x).

```
printf("%c", 'a')                    generates a
printf("<%3c|%-3c>", 'a', 'b')       generates <  a|b  >
wprintf(L"%c", 'a')                  generates (wide) btowc(a)
```

You write **%lc** to generate a single character from the converted value. Conversion of the character x occurs as if it is followed by a null character in an array of two elements of type wchar_t converted by the conversion specification ls.

```
printf("%lc", L'a')                  generates a
wprintf(L"lc", L'a')                 generates (wide) L'a'
```

You write **%d**, **%i**, **%o**, **%u**, **%x**, or **%X** to generate a possibly signed integer representation. %d or %i specifies signed decimal representation, %o unsigned octal, %u unsigned decimal, %x unsigned hexadecimal using the digits 0-9 and a-f, and %X unsigned hexadecimal using the digits 0-9 and A-F. The conversion generates at least *p* digits to represent the converted value. If *p* is zero, a converted value of zero generates no digits.

```
printf("%d %o %x", 31, 31, 31)       generates 31 37 1f
printf("%hu", 0xffff)                generates 65535
printf("%#X %+d", 31, 31)            generates 0X1F +31
```

You write **%e** or **%E** to generate a signed fractional representation with an exponent. The generated text takes the form ±*d.dddE±dd* , where ± is either a plus or minus sign, *d* is a decimal digit, the dot (.) is the decimal point for the current locale, and *E* is either e (for %e conversion) or E (for %E conversion). The generated text has one integer digit, a decimal point if *p* is nonzero or if you specify the # format flag, *p* fraction digits, and at least two exponent digits. The result is rounded. The value zero has a zero exponent.

```
printf("%e", 31.4)                    generates 3.140000e+01
printf("%.2E", 31.4)                  generates 3.14E+01
```

You write **%f** to generate a signed fractional representation with no exponent. The generated text takes the form ±*d.ddd*, where ± is either a plus or minus sign, *d* is a decimal digit, and the dot (.) is the decimal point for the current locale. The generated text has at least one integer digit, a decimal point if *p* is nonzero or if you specify the # format flag, and *p* fraction digits. The result is rounded.

```
printf("%f", 31.4)                    generates 31.400000
printf("%.0f %#.0f", 31.0, 31.0)      generates 31 31.
```

You write **%g** or **%G** to generate a signed fractional representation with or without an exponent, as appropriate. For %g conversion, the generated text takes the same form as either %e or %f conversion. For %G conversion, it takes the same form as either %E or %f conversion. The precision *p* specifies the number of significant digits generated. (If *p* is zero, it is changed to 1.) If %e conversion would yield an exponent in the range [-4, *p*), then %f conversion occurs instead. The generated text has no trailing zeros in any fraction and has a decimal point only if there are nonzero fraction digits, unless you specify the # format flag.

```
printf("%.6g", 31.4)                  generates 31.4
printf("%.1g", 31.4)                  generates 3.14e+01
```

You write **%n** to store the number of characters generated (up to this point in the format) in the object of type *int* whose address is the value of the next successive argument.

```
printf("abc%n", &x)                   stores 3
```

You write **%p** to generate an external representation of a *pointer to void*. The conversion is implementation defined.

```
printf("%p", (void *)&x)              generates, e.g. F4C0
```

You write **%s** to generate a sequence of characters from the values stored in the argument C string. For a wide stream, conversion occurs as if by repeatedly calling mbrtowc, beginning in the initial conversion state. The conversion generates no more than *p* characters, up to but not including the terminating null character.

```
printf("%s", "hello")                 generates hello
printf("%.2s", "hello")               generates he
wprintf(L"%s", "hello")               generates (wide) hello
```

You write **%ls** to generate a sequence of characters from the values stored in the argument wide-character string. For a byte stream, conversion occurs as if by repeatedly calling wcrtomb, beginning in the initial conversion state, so long as complete multibyte characters can be generated. The conversion generates no more than *p* characters, up to but not including the terminating null character.

```
printf("%ls", L"hello")              generates hello
wprintf(L"%.2s", L"hello")           generates (wide) he
```

You write **%%** to generate the percent character (%).

```
printf("%%")                         generates %
```

See also the **Table of Contents** and the **Index**.

Formatted Input

Scan Formats · **Scan Functions** · **Scan Conversion Specifiers**

Several library functions help you convert data values from text sequences that are generally readable by people to encoded internal representations. You provide a format string as the value of the `format` argument to each of these functions, hence the term **formatted input**. The functions fall into two categories:

- The **byte scan functions** (declared in `<stdio.h>`) convert sequences of type *char* to internal representations, and help you scan such sequences that you read: fscanf, scanf, and sscanf. For these function, a format string is a multibyte string that begins and ends in the initial shift state.
- The **wide scan functions** (declared in `<wchar.h>` and hence added with **Amendment 1**) convert sequences of type `wchar_t`, to internal representations, and help you scan such sequences that you read: fwscanf, wscanf and swscanf. For these functions, a format string is a wide-character string. In the descriptions that follow, a wide character `wc` from a format string or a stream is compared to a specific (byte) character `c` as if by evaluating the expression `wctob(wc) == c`.

Scan Formats

A format string has the same general **syntax** for the scan functions as for the print functions: zero or more **conversion specifications**, interspersed with literal text and **white space**. For the scan functions, however, a conversion specification is one of the scan conversion specifications described below.

A scan function scans the format string once from beginning to end to determine what conversions to perform. Every scan function accepts a varying number of arguments, either directly or under control of an argument of type `va_list`. Some scan conversion specifications in the format string use the next argument in the list. A scan function uses each successive argument no more than once. Trailing arguments can be left unused.

In the description that follows, the integer conversions and floating-point conversions are the same as for the print functions.

Scan Functions

For the scan functions, literal text in a format string must match the next characters to scan in the input text. White space in a format string must match the longest possible sequence of the next zero or more white-space characters in the input. Except for the scan conversion specifier `%n` (which consumes no input), each **scan conversion specification** determines a pattern that one or more of the next characters in the input must match. And except for the scan conversion specifiers `c`, `n`, and `[`, every match begins by skipping any white space characters in the input.

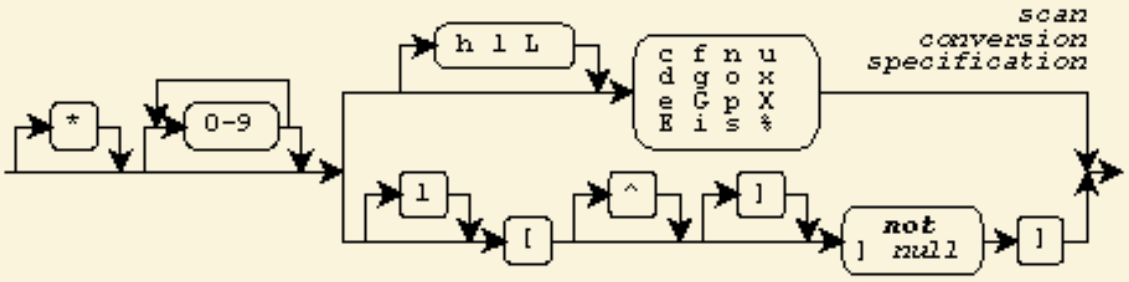
A scan function returns when:

- it reaches the terminating null in the format string
- it cannot obtain additional input characters to scan (**input failure**)
- a conversion fails (**matching failure**)

A scan function returns EOF if an input failure occurs before any conversion. Otherwise it returns the number of converted values stored. If one or more characters form a valid prefix but the conversion fails, the valid prefix is consumed before the scan function returns. Thus:

```
scanf("%i", &i)                    consumes 0X from the field 0XZ
scanf("%f", &f)                    consumes 3.2E from the field 3.2EZ
```

A scan conversion specification typically converts the matched input characters to a corresponding encoded value. The next argument value must be the address of an object. The conversion converts the encoded representation (as necessary) and stores its value in the object. A scan conversion specification has the format:



Following the percent character (**%**) in the format string, you can write an asterisk (*****) to indicate that the conversion should not store the converted value in an object.

Following any `*`, you can write a nonzero **field width** that specifies the maximum number of input characters to match for the conversion (not counting any white space that the pattern can first skip).

Scan Conversion Specifiers

Following any field width, you must write a one-character **scan conversion specifier**, either a one-character code or a scan set, possibly preceded by a one-character qualifier. Each combination determines the type required of the next argument (if any) and how the scan functions interpret the text sequence and converts it to an encoded value. The integer and floating-point conversions also determine what base to assume for the text representation. (The base is the `base` argument to the functions `strtol` and `strtoul`.) The following table lists all defined combinations and their properties.

Conversion Specifier	Argument Type	Conversion Function	Base
%c	char x[]		
%lc	wchar_t x[]		
%d	int *x	strtol	10
%hd	short *x	strtol	10
%ld	long *x	strtol	10
%e	float *x	strtod	10
%le	double *x	strtod	10
%Le	long double *x	strtod	10
%E	float *x	strtod	10

```
  %lE       double *x           strtod     10
  %LE       long double *x      strtod     10
   %f       float *x            strtod     10
  %lf       double *x           strtod     10
  %Lf       long double *x      strtod     10
   %g       float *x            strtod     10
  %lg       double *x           strtod     10
  %Lg       long double *x      strtod     10
   %G       float *x            strtod     10
  %lG       double *x           strtod     10
  %LG       long double *x      strtod     10
   %i       int *x              strtol      0
  %hi       short *x            strtol      0
  %li       long *x             strtol      0
   %n       int *x
  %hn       short *x
  %ln       long *x
   %o       unsigned int *x     strtoul     8
  %ho       unsigned short *x   strtoul     8
  %lo       unsigned long *x    strtoul     8
   %p       void **x
   %s       char x[]
  %ls       wchar_t x[]
   %u       unsigned int *x     strtoul    10
  %hu       unsigned short *x   strtoul    10
  %lu       unsigned long *x    strtoul    10
   %x       unsigned int *x     strtoul    16
  %hx       unsigned short *x   strtoul    16
  %lx       unsigned long *x    strtoul    16
   %X       unsigned int *x     strtoul    16
  %hX       unsigned short *x   strtoul    16
  %lX       unsigned long *x    strtoul    16
 %[...]     char x[]
%l[...]     wchar_t x[]
   %%       none
```

The scan conversion specifier (or scan set) determines any behavior not summarized in this table. In the following descriptions, examples follow each of the scan conversion specifiers. In each example, the function `sscanf` matches the **bold** characters.

You write **%c** to store the matched input characters in an array object. If you specify no field width *w*, then *w* has the value one. The match does not skip leading white space. Any sequence of *w* characters matches the conversion pattern. For a wide stream, conversion occurs as if by repeatedly calling `wcrtomb`, beginning in the initial conversion state.

```
sscanf("129E-2", "%c", &c)              stores '1'
sscanf("129E-2", "%2c", &c[0])          stores '1', '2'
swscanf(L"129E-2", L"%c", &c)           stores '1'
```

You write **%lc** to store the matched input characters in an array object, with elements of type wchar_t. If you specify no field width *w*, then *w* has the value one. The match does not skip leading white space. Any sequence of *w* characters matches the conversion pattern. For a byte stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "%lc", &c)              stores L'1'
sscanf("129E-2", "%2lc", &c)             stores L'1', L'2'
swscanf(L"129E-2", L"%lc", &c)           stores L'1'
```

You write **%d**, **%i**, **%o**, **%u**, **%x**, or **%X** to convert the matched input characters as a signed integer and store the result in an integer object.

```
sscanf("129E-2", "%o%d%x", &i, &j, &k)   stores 10, 9, 14
```

You write **%e**, **%E**, **%f**, **%g**, or **%G** to convert the matched input characters as a signed fraction, with an optional exponent, and store the result in a floating-point object.

```
sscanf("129E-2", "%e", &f)               stores 1.29
```

You write **%n** to store the number of characters matched (up to this point in the format) in an integer object. The match does not skip leading white space and does not match any input characters.

```
sscanf("129E-2", "12%n", &i)             stores 2
```

You write **%p** to convert the matched input characters as an external representation of a *pointer to void* and store the result in an object of type *pointer to void*. The input characters must match the form generated by the %p print conversion specification.

```
sscanf("129E-2", "%p", &p)               stores, e.g. 0x129E
```

You write **%s** to store the matched input characters in an array object, followed by a terminating null character. If you do not specify a field width *w*, then *w* has a large value. Any sequence of up to *w* non white-space characters matches the conversion pattern. For a wide stream, conversion occurs as if by repeatedly calling `wcrtomb` beginning in the initial conversion state.

```
sscanf("129E-2", "%s", &s[0])            stores "129E-2"
swscanf(L"129E-2", L"%s", &s[0])         stores "129E-2"
```

You write **%ls** to store the matched input characters in an array object, with elements of type wchar_t, followed by a terminating null wide character. If you do not specify a field width *w*, then *w* has a large value. Any sequence of up to *w* non white-space characters matches the conversion pattern. For a byte stream, conversion occurs as if by repeatedly calling `mbrtowc`, beginning in the initial conversion state.

```
sscanf("129E-2", "%ls", &s[0])           stores L"129E-2"
swscanf(L"129E-2", L"%ls", &s[0])        stores L"129E-2"
```

You write **%[** to store the matched input characters in an array object, followed by a terminating null character. If you do not specify a field width *w*, then *w* has a large value. The match does not skip leading white space. A

sequence of up to *w* characters matches the conversion pattern in the **scan set** that follows. To complete the scan set, you follow the left bracket (**[**) in the conversion specification with a sequence of zero or more **match** characters, terminated by a right bracket (**]**).

If you do not write a caret (**^**) immediately after the **[**, then each input character must match *one* of the match characters. Otherwise, each input character must not match *any* of the match characters, which begin with the character following the **^**. If you write a **]** immediately after the **[** or **[^**, then the **]** is the first match character, not the terminating **]**. If you write a minus (**-**) as other than the first or last match character, an implementation can give it special meaning. It usually indicates a range of characters, in conjunction with the characters immediately preceding or following, as in **0-9** for all the digits.) You cannot specify a null match character.

For a wide stream, conversion occurs as if by repeatedly calling **wcrtomb**, beginning in the initial conversion state.

```
sscanf("12E-2", "[54321]", &s[0])        stores "12"
swscanf(L"12E-2", L"[54321]", &s[0])     stores "12"
```

You write **%l[** to store the matched input characters in an array object, with elements of type wchar_t, followed by a terminating null wide character. If you do not specify a field width *w*, then *w* has a large value. The match does not skip leading white space. A sequence of up to *w* characters matches the conversion pattern in the scan set that follows.

For a byte stream, conversion occurs as if by repeatedly calling **mbrtowc**, beginning in the initial conversion state.

```
sscanf("12E-2", "l[54321]", &s[0])       stores L"12"
swscanf(L"12E-2", L"l[54321]", &s[0])    stores L"12"
```

You write **%%** to match the percent character (%). The function does not store a value.

```
sscanf("%  0XA", "%% %i")                stores 10
```

See also the **Table of Contents** and the **Index**.

STL Conventions

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

The Standard Template Library, or STL, establishes uniform standards for the application of iterators to STL containers or other sequences that you define, by STL algorithms or other functions that you define. This document summarizes many of the conventions used widely throughout the Standard Template Library.

Iterator Conventions

The STL facilities make widespread use of **iterators**, to mediate between the various algorithms and the sequences upon which they act. For brevity in the remainder of this document, the name of an iterator type (or its prefix) indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- **`OutIt`** -- An **output iterator** `X` can only have a value `V` stored indirect on it, after which it *must* be incremented before the next store, as in `(*X++ = V)`, `(*X = V, ++X)`, or `(*X = V, X++)`.
- **`InIt`** -- An **input iterator** `X` can represent a singular value that indicates end-of-sequence. If such an iterator does not compare equal to its end-of-sequence value, it can have a value `V` accessed indirect on it any number of times, as in `(V = *X)`. To progress to the next value, or end-of-sequence, you increment it, as in `++X`, `X++`, or `(V = *X++)`. Once you increment *any* copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- **`FwdIt`** -- A **forward iterator** `X` can take the place of an output iterator (for writing) or an input iterator (for reading). You can, however, read (via `V = *X`) what you just wrote (via `*X = V`) through a forward iterator. And you can make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- **`BidIt`** -- A **bidirectional iterator** `X` can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in `--X`, `X--`, or `(V = *X--)`.
- **`RanIt`** -- A **random-access iterator** `X` can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For `N` an integer object, you can write `X[N]`, `X + N`, `X - N`, and `N + X`.

Note that an object pointer can take the place of a random-access iterator, or any other for that matter.

The hierarchy of iterator categories can be summarize by showing three sequences. For write-only access

to a sequence, you can use any of:

```
output iterator ->
    forward iterator ->
    bidirectional iterator ->
    random-access iterator
```

The right arrow means ``can be replaced by." So any algorithm that calls for an output iterator should work nicely with a forward iterator, for example, but *not* the other way around.

For read-only access to a sequence, you can use any of:

```
input iterator ->
    forward iterator ->
    bidirectional iterator ->
    random-access iterator
```

An input iterator is the weakest of all categories, in this case.

Finally, for read/write access to a sequence, you can use any of:

```
forward iterator ->
    bidirectional iterator ->
    random-access iterator
```

Remember that an object pointer can always serve as a random-access iterator. Hence, it can serve as any category of iterator, so long as it supports the proper read/write access to the sequence it designates.

This ``algebra" of iterators is fundamental to practically everything else in the Standard Template Library. It is important to understand the promises, and limitations, of each iterator category to see how iterators are used by containers and algorithms in STL.

Algorithm Conventions

The descriptions of the algorithm template functions employ several shorthand phrases:

- The phrase ``**in the range `[A, B)`**" means the sequence of zero or more discrete values beginning with `A` up to but not including `B`. A range is valid only if `B` is **reachable** from `A` -- you can store `A` in an object `N` (`N = A`), increment the object zero or more times (`++N`), and have the object compare equal to `B` after a finite number of increments (`N == B`).
- The phrase ``**each `N` in the range `[A, B)`**" means that `N` begins with the value `A` and is incremented zero or more times until it equals the value `B`. The case `N == B` is *not* in the range.
- The phrase ``**the lowest value of `N` in the range `[A, B)` such that X**" means that the condition X is determined for each `N` in the range `[A, B)` until the condition X is met.
- The phrase ``**the highest value of `N` in the range `[A, B)` such that X**" usually means that X is determined for each `N` in the range `[A, B)`. The function stores in `K` a copy of `N` each time the condition X is met. If any such store occurs, the function replaces the final value of `N` (which equals `B`) with the value of `K`. For a bidirectional or random-access iterator, however, it can also

mean that N begins with the highest value in the range and is decremented over the range until the condition X is met.

- Expressions such as **X - Y**, where X and Y can be iterators other than random-access iterators, are intended in the mathematical sense. The function does not necessarily evaluate operator- if it must determine such a value. The same is also true for expressions such as **X + N** and **X - N**, where N is an integer type.

Several algorithms make use of a predicate that must impose a **strict weak ordering** on pairs of elements from a sequence. For the predicate pr(X, Y):

- ``strict'' means that pr(X, X) is false
- ``weak'' means that X and Y have an **equivalent ordering** if !pr(X, Y) && !pr(Y, X) (X == Y need not be defined)
- ``ordering'' means that pr(X, Y) && pr(Y, Z) implies pr(X, Z)

Some of these algorithms implicitly use the predicate X < Y. Other predicates that typically satisfy the ``strict weak ordering'' requirement are X > Y, less(X, Y), and greater(X, Y). Note, however, that predicates such as X <= Y and X >= Y do *not* satisfy this requirement.

A sequence of elements designated by iterators in the range [first, last) is ``**a sequence ordered by operator<**'' if, for each N in the range [0, last - first) and for each M in the range (N, last - first) the predicate !(*(first + M) < *(first + N)) is true. (Note that the elements are sorted in *ascending* order.) The predicate function operator<, or any replacement for it, must not alter either of its operands. Moreover, it must impose a strict weak ordering on the operands it compares.

A sequence of elements designated by iterators in the range [first, last) is ``**a heap ordered by operator<**'' if, for each N in the range [1, last - first) the predicate !(*first < *(first + N)) is true. (The first element is the largest.) Its internal structure is otherwise known only to the template functions make_heap, pop_heap, and push_heap. As with an ordered sequence, the predicate function operator<, or any replacement for it, must not alter either of its operands, and it must impose a strict weak ordering on the operands it compares.

See also the **Table of Contents** and the **Index**.

Containers

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

```
namespace std {
template<class T, class A>
    class Cont;
//    TEMPLATE FUNCTIONS
template<class T, class A>
    bool operator==(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
template<class T, class A>
    bool operator!=(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
template<class T, class A>
    bool operator<(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
template<class T, class A>
    bool operator>(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
template<class T, class A>
    bool operator<=(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
template<class T, class A>
    bool operator>=(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
template<class T, class A>
    void swap(
        const Cont<T, A>& lhs,
        const Cont<T, A>& rhs);
    };
```

A **container** is an STL template class that manages a sequence of elements. Such elements can be of any object type that supplies a default constructor, a destructor, and an assignment operator. This document describes the properties required of all such containers, in terms of a generic template class `Cont`. An actual container template class may have additional template parameters. It will certainly have additional

member functions.

The STL template container classes are:

- deque
- list
- map
- multimap
- multiset
- set
- vector

(The Standard C++ library template class `basic_string` also meets the requirements for a template container class.)

Cont

---

allocator_type · **begin** · **clear** · **const_iterator** · **const_reference** · **const_reverse_iterator** · **difference_type** · **empty** · **end** · **erase** · **get_allocator** · **iterator** · **max_size** · **rbegin** · **reference** · **rend** · **reverse_iterator** · **size** · **size_type** · **swap** · **value_type**

```
template<class T, class A = allocator<T> >
    class Cont {
public:
    typedef A allocator_type;
    typedef T0 size_type;
    typedef T1 difference_type;
    typedef T2 reference;
    typedef T3 const_reference;
    typedef T4 value_type;
    typedef T5 iterator;
    typedef T6 const_iterator;
    typedef T7 reverse_iterator;
    typedef T8 const_reverse_iterator;
    iterator begin();
    const_iterator begin() const;
    iterator end();
    iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin() const;
```

```
    reverse_iterator rend();
    const_reverse_iterator rend() const;
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    A get_allocator() const;
    iterator erase(iterator it);
    iterator erase(iterator first, iterator last);
    void clear();
    void swap(Cont x);
protected:
    A allocator;
    };
```

The template class describes an object that controls a varying-length sequence of elements, typically of **type T**. The sequence is stored in different ways, depending on the actual container.

The object allocates and frees storage for the sequence it controls through a protected object named **allocator**, of **class A**. Such an allocator object must have the same external interface as an object of template class `allocator`. Note that `allocator` is *not* copied when the object is assigned. All constructors store an allocator argument (or, for the copy constructor, `x.get_allocator()`) in `allocator` and initialize the controlled sequence.

Cont::allocator_type

```
typedef A allocator_type;
```

The type is a synonym for the template parameter A.

Cont::begin

```
const_iterator begin() const;
iterator begin();
```

The member function returns an iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

Cont::clear

```
void clear() const;
```

The member function calls `erase( begin(), end())`.

Cont::const_iterator

```
typedef T6 const_iterator;
```

The type describes an object that can serve as a constant iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T6`.

Cont::const_reference

```
typedef T3 const_reference;
```

The type describes an object that can serve as a constant reference to an element of the controlled sequence. It is described here as a synonym for the unspecified type `T3` (typically `A::const_reference`).

Cont::const_reverse_iterator

```
typedef T8 const_reverse_iterator;
```

The type describes an object that can serve as a constant reverse iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T8` (typically `reverse_iterator` or `reverse_bidirectional_iterator`).

Cont::difference_type

```
typedef T1 difference_type;
```

The signed integer type describes an object that can represent the difference between the addresses of any two elements in the controlled sequence. It is described here as a synonym for the unspecified type `T1` (typically `A::difference_type`).

Cont::empty

```
bool empty() const;
```

The member function returns true for an empty controlled sequence.

Cont::end

```
const_iterator end() const;
iterator end();
```

The member function returns an iterator that points just beyond the end of the sequence.

Cont::erase

```
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

The first member function removes the element of the controlled sequence pointed to by `it`. The second member function removes the elements of the controlled sequence in the range `[first, last)`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

Cont::get_allocator

```
A get_allocator() const;
```

The member function returns `allocator`.

Cont::iterator

```
typedef T5 iterator;
```

The type describes an object that can serve as an iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T5`.

Cont::max_size

```
size_type max_size() const;
```

The member function returns the length of the longest sequence that the object can control.

Cont::rbegin

```
const_reverse_iterator rbegin() const;
reverse_iterator rbegin();
```

The member function returns a reverse iterator that points just beyond the end of the controlled sequence. Hence, it designates the beginning of the reverse sequence.

Cont::reference

```
typedef T2 reference;
```

The type describes an object that can serve as a reference to an element of the controlled sequence. It is described here as a synonym for the unspecified type `T2` (typically `A::reference`).

Cont::rend

```
const_reverse_iterator rend() const;
reverse_iterator rend();
```

The member function returns a reverse iterator that points at the first element of the sequence (or just beyond the end of an empty sequence). Hence, it designates the end of the reverse sequence.

Cont::reverse_iterator

```
typedef T7 reverse_iterator;
```

The type describes an object that can serve as a reverse iterator for the controlled sequence. It is described here as a synonym for the unspecified type `T7` (typically `reverse_iterator` or `reverse_bidirectional_iterator`).

The required pointer type is described here as the unspecified type `T7` (typically `A::pointer`).

Cont::size

```
size_type size() const;
```

The member function returns the length of the controlled sequence.

Cont::size_type

```
typedef T0 size_type;
```

The unsigned integer type describes an object that can represent the length of any controlled sequence. It is described here as a synonym for the unspecified type `T0` (typically `A::size_type`).

Cont::swap

```
void swap(Cont& str);
```

The member function swaps the controlled sequences between `*this` and `str`. If `allocator == str.allocator`, it does so in constant time. Otherwise, it performs a number of element assignments and constructor calls proportional to the number of elements in the two controlled sequences.

Cont::value_type

```
typedef T4 value_type;
```

The type is a synonym for the template parameter `T`. It is described here as a synonym for the unspecified type `T4` (typically `A::value_type`).

operator!=

```
template<class T, class A>
    bool operator!=(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function returns `!(lhs == rhs)`.

operator==

```
template<class T, class A>
    bool operator==(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function overloads `operator==` to compare two objects of template class `Cont`. The function returns `lhs.size() == rhs.size() && equal(lhs. begin(), lhs. end(), rhs.begin())`.

operator<

```
template<class T, class A>
    bool operator<(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function overloads `operator<` to compare two objects of template class `Cont`. The function returns `lexicographical_compare(lhs. begin(), lhs. end(), rhs.begin(), rhs.end())`.

operator<=

```
template<class T, class A>
    bool operator<=(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function returns `!(rhs < lhs)`.

operator>

```
template<class T, class A>
    bool operator*gt;(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function returns `rhs < lhs`.

operator>=

```
template<class T, class A>
    bool operator>=(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function returns `!(lhs < rhs)`.

swap

```
template<class T, class A>
    void swap(
        const Cont <T, A>& lhs,
        const Cont <T, A>& rhs);
```

The template function executes `lhs.swap(rhs)`.

See also the **Table of Contents** and the **Index**.

Copyright Notice

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

References

- **ANSI Standard X3.159-1989** (New York NY: American National Standards Institute, 1989). The original C Standard, developed by the ANSI-authorized committee X3J11. The Rationale that accompanies the C Standard explains many of the decisions that went into it, if you can get your hands on a copy.
- **ISO/IEC Standard 9899:1990** (Geneva: International Standards Organization, 1990). The official C Standard around the world. Aside from formatting details and section numbering, the ISO C Standard is identical to the ANSI C Standard.
- **ISO/IEC Amendment 1 to Standard 9899:1990** (Geneva: International Standards Organization, 1995). The first (and only) amendment to the C Standard. It provides substantial support for

manipulating large character sets.

- **ISO/IEC Standard 14882:199X** (Geneva: International Standards Organization, 199X). Once adopted, the official C++ Standard around the world. Currently still in **draft** form, this document reflects changes through **October 1996**.
- ***** P.J. Plauger, **The Standard C Library** (Englewood Cliffs NJ: Prentice Hall, 1992). Contains a complete implementation of the Standard C library, as well as text from the library portion of the C Standard and guidance in using the Standard C library.
- ***** P.J. Plauger, **The Draft Standard C++ Library** (Englewood Cliffs NJ: Prentice Hall, 1995). Contains a complete implementation of the draft Standard C++ library as of early 1994.

Bug Reports

The author welcomes reports of any errors or omissions. Please send them to:


```
P.J. Plauger
Dinkumware, Ltd.
398 Main Street
Concord MA  01742-2321
    USA

+1-978-371-2773
+1-978-371-9014 fax

pjp@plauger.com
```

See also the **Table of Contents** and the **Index**.

Dinkum C/C++ Library Reference Index

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

basic_ios::setstate · basic_ios::tie · basic_iostream · basic_ios::widen · basic_istream · basic_istream::basic_istream · basic_istream::char_type · basic_istream::gcount · basic_istream::get · basic_istream::getline · basic_istream::ignore · basic_istream::int_type · basic_istream::ipfx · basic_istream::isfx · basic_istream::off_type · basic_istream::operator>> · basic_istream::peek · basic_istream::pos_type · basic_istream::putback · basic_istream::read · basic_istream::readsome · basic_istream::seekg · basic_istream::sentry · basic_istream::sync · basic_istream::tellg · basic_istream::traits_type · basic_istream::unget · basic_istringstream · basic_istringstream::basic_istringstream · basic_istringstream::char_type · basic_istringstream::int_type · basic_istringstream::off_type · basic_istringstream::pos_type · basic_istringstream::rdbuf · basic_istringstream::str · basic_istringstream::traits_type · basic_istrstream::char_type · basic_istrstream::int_type · basic_istrstream::off_type · basic_istrstream::pos_type · basic_ostream · basic_ostream::basic_ostream · basic_ostream::char_type · basic_ostream::flush · basic_ostream::int_type · basic_ostream::off_type · basic_ostream::operator<< · basic_ostream::opfx · basic_ostream::osfx · basic_ostream::pos_type · basic_ostream::put · basic_ostream::seekp · basic_ostream::sentry · basic_ostream::tellp · basic_ostream::traits_type · basic_ostream::write · basic_ostringstream · basic_ostringstream::basic_ostringstream · basic_ostringstream::char_type · basic_ostringstream::int_type · basic_ostringstream::off_type · basic_ostringstream::pos_type · basic_ostringstream::rdbuf · basic_ostringstream::str · basic_ostringstream::traits_type · basic_ostrstream::char_type · basic_ostrstream::int_type · basic_ostrstream::off_type · basic_ostrstream::pos_type · basic_streambuf · basic_streambuf::basic_streambuf · basic_streambuf::char_type · basic_streambuf::eback · basic_streambuf::egptr · basic_streambuf::epptr · basic_streambuf::gbump · basic_streambuf::getloc · basic_streambuf::gptr · basic_streambuf::imbue · basic_streambuf::in_avail · basic_streambuf::int_type · basic_streambuf::off_type · basic_streambuf::overflow · basic_streambuf::pbackfail · basic_streambuf::pbase · basic_streambuf::pbump · basic_streambuf::pos_type · basic_streambuf::pptr ·

basic_streambuf::pubimbue · basic_streambuf::pubseekoff · basic_streambuf::pubseekpos · basic_streambuf::pubsetbuf · basic_streambuf::pubsync · basic_streambuf::sbumpc · basic_streambuf::seekoff · basic_streambuf::seekpos · basic_streambuf::setbuf · basic_streambuf::setg · basic_streambuf::setp · basic_streambuf::sgetc · basic_streambuf::sgetn · basic_streambuf::showmanyc · basic_streambuf::snextc · basic_streambuf::sputbackc · basic_streambuf::sputc · basic_streambuf::sputn · basic_streambuf::sungetc · basic_streambuf::sync · basic_streambuf::traits_type · basic_streambuf::uflow · basic_streambuf::underflow · basic_streambuf::xsgetn · basic_streambuf::xsputn · basic_string · basic_string::allocator · basic_string::allocator_type · basic_string::append · basic_string::assign · basic_string::at · basic_string::basic_string · basic_string::begin · basic_stringbuf · basic_stringbuf::basic_stringbuf · basic_stringbuf::char_type · basic_stringbuf::int_type · basic_stringbuf::off_type · basic_stringbuf::overflow · basic_stringbuf::pbackfail · basic_stringbuf::pos_type · basic_stringbuf::seekoff · basic_stringbuf::seekpos · basic_stringbuf::str · basic_stringbuf::traits_type · basic_stringbuf::underflow · basic_string::capacity · basic_string::char_type · basic_string::compare · basic_string::const_iterator · basic_string::const_pointer · basic_string::const_reference · basic_string::const_reverse_iterator · basic_string::copy · basic_string::c_str · basic_string::data · basic_string::difference_type · basic_string::empty · basic_string::end · basic_string::erase · basic_string::find · basic_string::find_first_not_of · basic_string::find_first_of · basic_string::find_last_not_of · basic_string::find_last_of · basic_string::get_allocator · basic_string::insert · basic_string::iterator · basic_string::length · basic_string::max_size · basic_string::npos · basic_string::operator+= · basic_string::operator= · basic_string::operator[] · basic_string::pointer · basic_string::rbegin · basic_string::reference · basic_string::rend · basic_string::replace · basic_string::reserve · basic_string::resize · basic_string::reverse_iterator · basic_string::rfind · basic_string::size · basic_string::size_type · basic_stringstream · basic_stringstream::basic_stringstream ·

C

D

E

F

G

H

I

indirect_array::operator-= · indirect_array::operator/= · #indirect_array::operator= · indirect_array::operator= · indirect_array::operator^= · indirect_array::operator|= · indirect_array::operator&= · indirect_array::operator>>= · indirect_array::operator<<= · indirect_array::value_type · InIt · inner_product · input buffer · input failure · input_iterator_tag · inserter · insert_iterator · insert_iterator::container · insert_iterator::container_type · insert_iterator::insert_iterator · insert_iterator::iter · insert_iterator::operator* · insert_iterator::operator++ · insert_iterator::operator= · insert_iterator::value_type · int_curr_symbol · integer conversions · integer input field · integer output field · interactive files · internal · int_frac_digits · INT_MAX · INT_MIN · invalid list iterators · invalid vector iterators · invalid_argument · _IOFBF · _IOLBF · <iomanip> · <iomanip.h> · _IONBF · ios · ios_base · ios_base::adjustfield · ios_base::app · ios_base::ate · ios_base::badbit · ios_base::basefield · ios_base::beg · ios_base::binary · ios_base::boolalpha · ios_base::copyfmt_event · ios_base::cur · ios_base::dec · ios_base::end · ios_base::eofbit · ios_base::erase_event · ios_base::event · ios_base::event_callback · ios_base::failbit · ios_base::failure · ios_base::fixed · ios_base::flags · ios_base::floatfield · ios_base::fmtflags · ios_base::getloc · ios_base::goodbit · ios_base::hex · ios_base::imbue · ios_base::imbue_event · ios_base::in · ios_base::Init · ios_base::internal · ios_base::ios_base · ios_base::iostate · ios_base::iword · ios_base::left · ios_base::oct · ios_base::openmode · ios_base::operator= · ios_base::out · ios_base::precision · ios_base::pword · ios_base::register_callback · ios_base::right · ios_base::scientific · ios_base::seekdir · ios_base::setf · ios_base::showbase · ios_base::showpoint · ios_base::showpos · ios_base::skipws · ios_base::sync_with_stdio · ios_base::trunc · ios_base::unitbuf · ios_base::unsetf · ios_base::uppercase · ios_base::width · ios_base::xalloc · <iosfwd> · <ios> · iostream · <iostream> · <iostream.h> · iostreams · isalnum · isalpha · iscntrl · isdigit · isgraph · islower · <iso646.h> · isprint · ispunct · isspace · istream · istreambuf_iterator · istreambuf_iterator::char_type · istreambuf_iterator::equal · istreambuf_iterator::int_type · istreambuf_iterator::istreambuf_iterator · istreambuf_iterator::istream_type · istreambuf_iterator::operator* · istreambuf_iterator::operator++ · istreambuf_iterator::operator-> ·

J

K

L

M

moneypunct::do_positive_sign · moneypunct::do_thousands_sep · moneypunct::frac_digits · moneypunct::grouping · moneypunct::id · moneypunct::intl · moneypunct::moneypunct · moneypunct::negative_sign · moneypunct::neg_format · moneypunct::pos_format · moneypunct::positive_sign · moneypunct::string_type · moneypunct::thousands_sep · money_put · money_put::char_type · money_put::do_put · money_put::id · money_put::iter_type · money_put::money_put · money_put::put · money_put::string_type · mon_grouping · month input field · mon_thousands_sep · multibyte string · multimap · multimap::allocator · multimap::allocator_type · multimap::begin · multimap::clear · multimap::const_iterator · multimap::const_reference · multimap::const_reverse_iterator · multimap::count · multimap::difference_type · multimap::empty · multimap::end · multimap::equal_range · multimap::erase · multimap::find · multimap::get_allocator · multimap::insert · multimap::iterator · multimap::key_comp · multimap::key_compare · multimap::key_type · multimap::lower_bound · multimap::max_size · multimap::multimap · multimap::rbegin · multimap::reference · multimap::referent_type · multimap::rend · multimap::reverse_iterator · multimap::size · multimap::size_type · multimap::swap · multimap::upper_bound · multimap::value_comp · multimap::value_compare · multimap::value_compare::comp · multimap::value_type · multiset · multiset::allocator · multiset::allocator_type · multiset::begin · multiset::clear · multiset::const_iterator · multiset::const_reference · multiset::const_reverse_iterator · multiset::count · multiset::difference_type · multiset::empty · multiset::end · multiset::equal_range · multiset::erase · multiset::find · multiset::get_allocator · multiset::insert · multiset::iterator · multiset::key_comp · multiset::key_compare · multiset::key_type · multiset::lower_bound · multiset::max_size · multiset::multiset · multiset::rbegin · multiset::reference · multiset::rend · multiset::reverse_iterator · multiset::size · multiset::size_type · multiset::swap · multiset::upper_bound · multiset::value_comp · multiset::value_compare · multiset::value_type

N

%n · %n · names for types · namespaces · native locale · n_cs_precedes · negative_sign · new expression · new[] expression · new handler · <new> · new_handler · <new.h> · no buffering · noboolalpha · noshowbase · noshowpoint · noshowpos · noskipws · not · not_eq · nothrow · nothrow_t · nounitbuf · nouppercase · n_sep_by_space · n_sign_posn · NULL · NULL · NULL · NULL · NULL · NULL · NULL · null

string · <numeric> · numeric_limits · numeric_limits::denorm_min · numeric_limits::digits · numeric_limits::digits10 · numeric_limits::epsilon · numeric_limits::has_denorm · numeric_limits::has_denorm_loss · numeric_limits::has_infinity · numeric_limits::has_quiet_NaN · numeric_limits::has_signaling_NaN · numeric_limits::infinity · numeric_limits::is_bounded · numeric_limits::is_exact · numeric_limits::is_iec559 · numeric_limits::is_integer · numeric_limits::is_modulo · numeric_limits::is_signed · numeric_limits::is_specialized · numeric_limits::max · numeric_limits::max_exponent · numeric_limits::max_exponent10 · numeric_limits::min · numeric_limits::min_exponent · numeric_limits::min_exponent10 · numeric_limits::quiet_NaN · numeric_limits::radix · numeric_limits::round_error · numeric_limits::round_style · numeric_limits::signaling_NaN · numeric_limits::tinyness_before · numeric_limits::traps · num_get · num_get::char_type · num_get::do_get · num_get::get · num_get::id · num_get::iter_type · num_get::num_get · numpunct · numpunct_byname · numpunct::char_type · numpunct::decimal_point · numpunct::do_decimal_point · numpunct::do_falsename · numpunct::do_grouping · numpunct::do_thousands_sep · numpunct::do_truename · numpunct::falsename · numpunct::grouping · numpunct::id · numpunct::numpunct · numpunct::string_type · numpunct::thousands_sep · numpunct::truename · num_put · num_put::char_type · num_put::do_put · num_put::id · num_put::iter_type · num_put::num_put · num_put::put

O

%o · %o · oct · offsetof · opening mode · operand sequence · operator!= · operator+ · operator- · operator== · operator!= · operator== · operator!= · operator== · operator!= · operator== · operator!= · operator== · operator!= · operator== · operator!= · operator+ · operator== · operator!= · operator== · operator!= · operator% · operator* · operator+ · operator- · operator/ · operator== · operator^ · operator| · operator|| · operator!= · operator== · operator delete · operator delete[] · operator delete · operator delete[] · operator new · operator new[] · operator new · operator new[] · operator& · operator&& · operator> · operator>= · operator> · operator>= · operator> · operator>= · operator> · operator>= · operator>= · operator> · operator>= · operator> · operator>= · operator> · operator>= · operator> · operator>= · operator> · operator>= · operator> · operator>= · operator>> · operator>> ·

P

Q

R

S

set::empty · set::end · set::equal_range · set::erase · setfill · set::find · set::get_allocator · <set> · set::insert · setiosflags · set::iterator · setjmp · <setjmp.h> · set::key_comp · set::key_compare · set::key_type · setlocale · set::lower_bound · set::max_size · set_new_handler · setprecision · set::rbegin · set::reference · set::rend · set::reverse_iterator · set::set · set::size · set::size_type · set::swap · set::upper_bound · set::value_comp · set::value_compare · set::value_type · setvbuf · setw · shared memory · showbase · showpoint · showpos · SHRT_MAX · SHRT_MIN · SIGABRT · sig_atomic_t · SIG_DFL · SIG_ERR · SIGFPE · SIG_IGN · SIGILL · SIGINT · signal · signal handler · <signal.h> · signaling NaN · signals · SIGSEGV · SIGTERM · sin · sin · sinf · sinh · sinh · sinhf · sinhl · sinl · size_t · size_t · size_t · size_t · size_t · size_t · skipws · slice · slice_array · slice_array::fill · slice_array::operator%= · slice_array::operator*= · slice_array::operator+= · slice_array::operator-= · slice_array::operator/= · #slice_array::operator= · slice_array::operator= · slice_array::operator^= · slice_array::operator|= · slice_array::operator&= · slice_array::operator>>= · slice_array::operator<<= · slice_array::value_type · slice::size · slice::slice · slice::start · slice::stride · sprintf · sqrt · sqrt · sqrtf · sqrtl · srand · sscanf · <sstream> · stack · stack::allocator_type · stack::c · stack::empty · stack::get_allocator · <stack> · stack::pop · stack::push · stack::size · stack::size_type · stack::stack · stack::top · stack::value_type · Standard C++ headers · Standard C Library · Standard C++ Library · standard error · standard header · standard headers · standard input · standard output · standard streams · Standard Template Library · std · std namespace · <stdarg.h> · <stddef.h> · stderr · <stdexcept> · stdin · stdio sync flag · <stdio.h> · <stdlib.h> · stdout · STL · STL Conventions · <stl.h> · strcat · strchr · strcmp · strcoll · strcpy · strcspn · stream · stream buffer · stream buffer · stream buffer · stream buffer pointer · stream state information · Stream States · streambuf · <streambuf> · streamoff · streampos · streamsize · strerror · strftime · strict weak ordering · string · stringbuf · stringbuf mode · <string> · <string.h> · strings · stringstream · strlen · strncat · strncmp · strncpy · strpbrk · strrchr · strspn · strstr · strstream · strstreambuf · strstreambuf allocation · strstreambuf mode · strstreambuf::freeze · strstreambuf::overflow · strstreambuf::pbackfail · strstreambuf::pcount · strstreambuf::seekoff

· strstreambuf::seekpos · strstreambuf::str · strstreambuf::strstreambuf · strstreambuf::underflow · strstream::freeze · <strstream> · strstream::pcount · strstream::rdbuf · strstream::str · strstream::strstream · strtod · strtok · strtol · strtoul · strxfrm · swap · swap · swap · swap · swap · swap · swprintf · swscanf · system

T

Table of Contents · tan · tan · tanf · tanh · tanh · tanhf · tanhl · tanl · Text and Binary Streams · text lines · text stream · thousands_sep · tie pointer · time · time input field · time string · time structure · time_base · time_base::dateorder · time_base::dmy · time_base::mdy · time_base::no_order · time_base::ydm · time_base::ymd · time_get · time_get_byname · time_get::char_type · time_get::date_order · time_get::do_date_order · time_get::do_get_date · time_get::do_get_month · time_get::do_get_time · time_get::do_get_weekday · time_get::do_get_year · time_get::get_date · time_get::get_month · time_get::get_time · time_get::get_weekday · time_get::get_year · time_get::id · time_get::iter_type · time_get::time_get · <time.h> · time_put · time_put_byname · time_put::char_type · time_put::do_put · time_put::id · time_put::iter_type · time_put::put · time_put::time_put · time_t · tm · tm · tmpfile · TMP_MAX · tmpnam · tolower · total ordering · toupper · towctrans · towlower · towupper · translation unit · transparent locale · typeid · type_info · type_info::before · <typeinfo> · type_info::name · type_info::operator!= · type_info::operator==

U

%u · %u · UCHAR_MAX · UINT_MAX · ULONG_MAX · unbound stream · undef directive · underflow_error · unformatted input functions · unformatted output functions · ungetc · ungetwc · uninitialized fill · uninitialized_copy · uninitialized_fill_n · unitbuf · Universal Time Coordinated · uppercase · _USE · _USEFAC · use_facet · USHRT_MAX · Using Standard C++ Headers · Using Standard C Headers · <utility>

V

va_arg · va_end · valarray · valarray::apply · valarray::cshift · valarray::fill · valarray::free · <valarray> · valarray<bool> · valarray::max · valarray::min · valarray::operator! · valarray::operator%= · valarray::operator*= · valarray::operator+ · valarray::operator+= · valarray::operator- · valarray::operator-= · valarray::operator/= · valarray::operator= · valarray::operator[] · valarray::operator^= · valarray::operator|= · valarray::operator~ · valarray::operator T * · valarray::operator&= · valarray::operator>>=

See also the **Table of Contents**.

Copyright Notice

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

References

- **ANSI Standard X3.159-1989** (New York NY: American National Standards Institute, 1989). The original C Standard, developed by the ANSI-authorized committee X3J11. The Rationale that accompanies the C Standard explains many of the decisions that went into it, if you can get your hands on a copy.
- **ISO/IEC Standard 9899:1990** (Geneva: International Standards Organization, 1990). The official C Standard around the world. Aside from formatting details and section numbering, the ISO C Standard is identical to the ANSI C Standard.
- **ISO/IEC Amendment 1 to Standard 9899:1990** (Geneva: International Standards Organization, 1995). The first (and only) amendment to the C Standard. It provides substantial support for

manipulating large character sets.

- P.J. Plauger, **The Standard C Library** (Englewood Cliffs NJ: Prentice Hall, 1992). Contains a complete implementation of the Standard C library, as well as text from the library portion of the C Standard and guidance in using the Standard C library.
- * P.J. Plauger and Jim Brodie, **Standard C: A Programmer's Reference** (Redmond WA: Microsoft Press, 1989). The first complete but succinct reference to the entire C Standard. It covers both the language and the library.
- * P.J. Plauger and Jim Brodie, **ANSI and ISO Standard C: Programmer's Reference** (Redmond WA: Microsoft Press, 1992). An update to the above book.
- * P.J. Plauger and Jim Brodie, **Standard C** (Englewood Cliffs NJ: PTR Prentice Hall, 1996). An update to the above two books and the principal source book for this material. It includes a complete description of Amendment 1.

Bug Reports

The authors welcome reports of any errors or omissions. Please send them to:


```
P.J. Plauger
Dinkumware, Ltd.
398 Main Street
Concord MA  01742-2321
    USA

+1-978-371-2773
+1-978-371-9014 fax

pjp@plauger.com
```

See also the **Table of Contents** and the **Index**.

Hewlett-Packard Notice

This material is derived in part from software and documentation bearing the following restrictions:

See also the **Table of Contents** and the **Index**.

Functions

Dinkumware, Ltd.
Genuine Software
www.dinkumware.com
398 Main Street, Concord, MA 01742
1 888 4-DINKUM Fax: 978 371-9014

You write functions to specify all the actions that a program performs when it executes. The type of a function tells you the type of result it returns (if any). It can also tell you the types of any arguments that the function expects when you call it from within an expression.

This document describes briefly just those aspect of functions most relevant to the use of the Standard C library:

Argument promotion occurs when the type of the function fails to provide any information about an argument. Promotion occurs if the function declaration is not a function prototype or if the argument is one of the unnamed arguments in a varying number of arguments. In this instance, the argument must be an rvalue expression. Hence:

- An integer argument type is promoted.
- An lvalue of type *array of T* becomes an rvalue of type *pointer to T*.
- A function designator of type *function returning T* becomes an rvalue of type *pointer to function returning T*.
- An argument of type *float* is converted to *double*.

A ***do* statement** executes a statement one or more times, while its test-context expression has a nonzero value:

```
do
    statement
    while (test);
```

An ***expression* statement** evaluates an expression in a side-effects context:

```
printf("hello\n");          call a function
y = m * x + b;              store a value
++count;                    alter a stored value
```

A ***for* statement** executes a statement zero or more times, while the optional test-context expression *test* has a nonzero value. You can also write two expressions, *se-1* and *se-2*, in a *for* statement that are each in a side-effects context:

```
for (se-1; test; se-2)
    statement
```

An ***if* statement** executes a statement only if the test-context expression has a nonzero value:

```
if (test)
    statement
```

An ***if-else* statement** executes one of two statements, depending on whether the test-context expression has a nonzero value:

```
if (test)
    statement-1
else
    statement-2
```

A ***return* statement** terminates execution of the function and transfers control to the expression that called the function. If you write the optional rvalue expression within the *return* statement, the result must be assignment-compatible with the type returned by the function. The program converts the value of the expression to the type returned and returns it as the value of the function call:

```
return expression;
```

An expression that occurs in a **side-effects context** specifies no value and designates no object or function. Hence, it can have type *void.* You typically evaluate such an expression for its **side effects** -- any change in the state of the program that occurs when evaluating an expression. Side effects occur when the program stores a value in an object, accesses a value from an object of *volatile* qualified type, or alters the state of a file.

A ***switch* statement** jumps to a place within a controlled statement, depending on the value of an integer expression:

```
switch (expr)
    {
case val-1:
    stat-1;
    break;
case val-2:
    stat-2;             falls through to next
default:
    stat-n
}
```

In a **test-context expression** the value of an expression causes control to flow one way within the statement if the computed value is nonzero or another way if the computed value is zero. You can write only an expression that has a scalar rvalue result, because only scalars can be compared with zero.

A ***while* statement** executes a statement zero or more times, while the test-context expression has a nonzero value:

```
while (test)
    statement
```

See also the **Table of Contents** and the **Index**.