

# 1 Introduction, Classes and Data Abstraction

- Basic characteristics of O-O languages
  - Everything is an object.
  - Object-orientation is a natural way of thinking about the world and of writing computer programs.
  - Objects are all around us—people, animals, plants, cars, planes, buildings, computers, etc.
  - All objects have attributes such as size, shape, color, weight, etc.
  - All objects exhibit various behaviors.
  - Humans learn about objects by studying their attributes and observing their behaviors.
  - Different objects can have many of the same attributes and exhibit similar behaviors.
  - Object-oriented programming (OOP) models real-world objects with software counterparts.
    - \* It takes advantage of class relationships where objects of a certain class, such as a class of vehicles, have the same characteristics.
    - \* It takes advantage of inheritance relationships, and even multiple inheritance relationships, where newly created classes are derived by inheriting characteristics of existing classes, yet contain unique characteristics of their own.
  - A program is a bunch of objects telling each other what to do, by sending messages.
  - Every object has a type (class).
  - All objects of the same type can receive the same messages.
- Objects
  - An object has an **interface**, determined by the class it's an instance of.
  - A class is an **abstract data type** (or **user-defined data type**).
  - What about built-in types?
    - \* Think of an *int*

\* What's its interface?

- The interface is the critical part, but the details (implementation) are important too
- Users use the interface (the "public part"), the implementation is hidden by "access control".
- C++ uses "access specifiers": **public**, **protected**, and **private** to determine who can use the attribute or function.
- Two Ways of Reusing Classes
  - **Composition:** One class has another as a "part".
  - **Inheritance:** One class is a specialized version of another
- **Polymorphism:** Different subclasses respond to the same message, possibly with different actions.
- Creating and Destroying Objects
  - We usually get this for free with built-in types like int or char, we just say
    - \* int i;
    - \* char c;
  - With user-defined types (the ones we make), we need to be explicit about what we want:
    - \* constructor function
    - \* destructor function
    - \* C++ has new and delete (similar to malloc and free in C)
    - \* This is a very important issue! What is a memory leak?
- A compiler typically does
  - preprocessing
  - first pass to make parse tree
  - second pass to generate code
- The result is an object module (.obj file).
- A linker produces an .exe file by

- Resolving references between compilation units (i.e., separate source files)
  - Adding code from libraries
  - Adding special startup code
  - Building the final executable file
- In C++, variables and functions must be both declared and defined. The rules:
    - A declaration tells the compiler that you intend to use a variable/function with a certain name.
    - A variable declaration specifies the type (int, float, etc.) so the compiler can check your usage. A variable declaration doesn't allocate space for the variable.
    - A function declaration specifies the function name, argument types, and return type, so the compiler can check your usage. A function declaration doesn't allocate space for the function code.
    - A variable definition causes memory to be allocated to hold its value. This can only be done (must be done) exactly once in the entire program. Why?
    - And so for functions.
- Libraries are collections of compiled function definitions.
    - Library header files (.h files, or files with no extension) are collections of (uncompiled e.g., ASCII) function declarations.
    - *#include*ing a header file is a fast and painless way of providing the declarations the compiler insists on.
    - The compiler is happy, since it has declarations from the .h file(s)
    - The linker is happy, because there is exactly one definition of a library function.
    - The linker resolves references to variables/functions that are spread across files.
- Survey of Programming Techniques (see Fig. 1)
    - Unstructured programming.
      - \* Simple sequence of command statements.

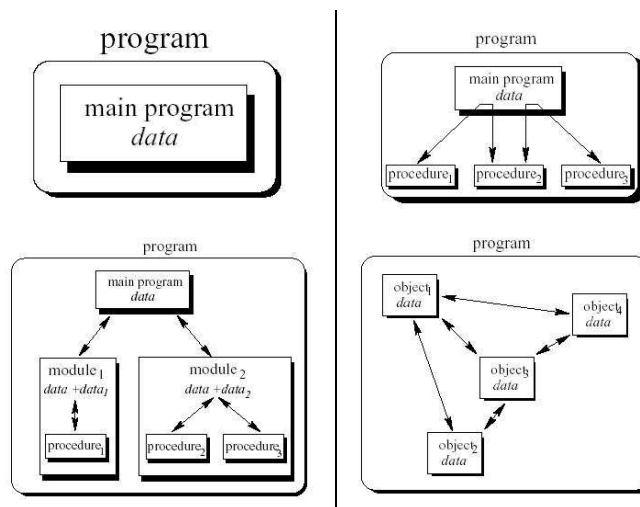


Figure 1: Survey of Programming Techniques; unstructured, procedural, modular, and object-oriented programming.

- \* Operates directly on global data.
  - \* Not good for large programs.
  - \* Repetitive statement segments are copied over.
  - \* The repetitive sequences extracted and named so that they can be called and values returned leads to the idea of procedures.
- Procedural programming.
- \* Combines returning sequences of statements into one function.
  - \* Procedure calls are used to invoke procedures.
  - \* Programs are now more structured.
  - \* Errors are easier to detect.
  - \* Combining procedures into modules is the next logical extension.
- Modular programming.
- \* Procedures with common functionality are grouped into modules.
  - \* Main program coordinates calls to procedures within modules.
  - \* Each module has its own data and isolated for other modules.
- Object-oriented programming.

- \* Data and the functions that operate on that data are combined into an object.
- \* Programming is not function based but object based.
- \* Objects are based on three basic ideas: Encapsulation, Inheritance and Polymorphism.

## 1.1 History: The Rise and Decline of Structured Programming

For many years (roughly 1970 to 1990), *structured programming* was the most common way to organize a program. This is characterized by a functional-decomposition style - breaking the algorithms into every smaller functions. This technique was a great improvement over the ad hoc programming which preceded it. However, as programs became larger, structured programming was not able to control the exponential increase in complexity.

### 1.1.1 The Problem - Complexity

Complexity measurements grow exponentially as the size of programs grow. One measurement is *coupling*, or much different elements (modules, data structures) interact with each other. The fewer the connections, the less complex a program is. Low coupling is highly desirable.

There have been several post-structured programming attempts to control complexity. One of these is to use software *components* - preconstructed software “parts” to avoid programming. And when you have to program, use *object-oriented programming* (OOP).

Bjarne Stroustrup of Bell Labs extended the C language to be capable of Object-Oriented Programming (OOP), and it became popular in the 1990’s as C++. There were several enhancements, but the central change was extending **struct** to allow it to contain functions and use inheritance. These extended structs were later renamed **classes**. A C++ standard was established in 1999, so there are variations in the exact dialect that is accepted by pre-standard compilers.

## 1.2 Object-Oriented Programming (OOP)

Object-Oriented Programming groups related data and functions together in a *class*, generally making data private and only some functions public. Restricting access decreases coupling and increases cohesion. While it is not a cure-all, it has proven to be very effective in reducing the complexity increase with large programs. For small programs may be difficult to see the

advantage of OOP over, eg, structured programming because there is little complexity regardless of how it's written. Many of the mechanics of OOP are easy to demonstrate; it is somewhat harder to create small, convincing examples.

OOP is often said to incorporate three techniques: inheritance, encapsulation, and polymorphism. Of these, you should first devote yourself to choosing the right classes (possibly difficult) and getting the encapsulation right (fairly easy). Inheritance and polymorphism are not even present in many programs, so you can ignore them at that start.

### 1.2.1 Encapsulation

*Encapsulation* is grouping data and functions together and keeping their implementation details private. Greatly restricting access to functions and data reduces *coupling*, which increases the ability to create large programs. Classes also encourage *coherence*, which means that a given class does one thing. By increasing coherence, a program becomes easier to understand, more simply organized, and this better organization is reflected in a further reduction in coupling.

### 1.2.2 Inheritance

*Inheritance* means that a new class can be defined in terms of an existing class. There are three common terminologies for the new class: the *derived* class, the *child* class, or the *subclass*. The original class is the *base* class, the *parent* class, or the *superclass*. The new child class inherits all capabilities of the parent class and adds its own fields and methods. Although inheritance is very important, especially in many libraries, is often not used in an application.

### 1.2.3 Polymorphism

*Polymorphism* is the ability of different functions to be invoked with the same name. There are two forms.

*Static polymorphism* is the common case of *overriding* a function by providing additional definitions with different numbers or types of parameters. The compiler matches the parameter list to the appropriate function.

*Dynamic polymorphism* is much different and relies on parent classes to define *virtual functions* which child classes may redefine. When this virtual member function is called for an object of the parent class, the execution dynamically chooses the appropriate function to call - the parent function if the object really is the parent type, or the child function if the object really is the child

type. This explanation is too brief to be useful without an example, but that will have to be written later.

#### 1.2.4 Advantages of OOP

- Re-use of code. Linking of code to objects and explicit specification of relations between objects allows related objects to share code.
- Ease of comprehension. Structure of code and data structures in it can be set up to closely mimic the generic application concepts and processes. High-level code could make some sense even to a non-programmer.
- Ease of fabrication and maintenance (redesign and extension) facilitated by encapsulation, data abstraction which allow for very clean designs.
- C++ Objectives
  - extend C to allow for object-oriented programming
  - other improvements - some resulting in deprecation of some C facilities. While any C++ compiler should be able to compile a C program successfully with minor changes, several aspects of C programming are discarded in the transition to C++: new facilities are supplied for I/O, memory allocation and error handling; macros and pointer casts become obsolete for the most part.
  - remain compatible and comparable (syntax, performance, portability, design philosophy)
  - emphasize compile-time type checking
- C++ is multi-paradigm. It provides for the object-oriented approach but doesn't enforce its use.
- C's future is as a portable "universal" assembler, a back end for code generators.

#### 1.2.5 OOP Terminology

Along with each programming revolution comes a new set of terminology. There are some new OOP concepts, but many have a simple analog in pre-OOP practice.

OOP Term	Definition
method	Same as function, but the typical OOP notation is used for the call, ie, $f(x,y)$ is written $x.f(y)$ where $x$ is an object of class that contains this $f$ method.
send a message	Call a function (method).
instantiate	Allocate a class/struct object (ie, instance) with <code>new</code> .
class	A struct with both data and functions.
object	Memory allocated to a class/struct. Often allocated with <code>new</code> .
member	A field or function is a member of a class if it's defined in that class
constructor	Function-like code that initializes new objects (structs) when they instantiated (allocated with <code>new</code> ).
destructor	Function-like code that is called when an object is deleted to free any resources (eg, memory) that it has pointers to.
inheritance	Defining a class (child) in terms of another class (parent). All of the public members of the parent class are available in the child class.
polymorphism	Defining functions with the same name, but different parameters.
overload	A function is overloaded if there is more than one definition. See polymorphism.
override	Redefine a function from a parent class in a child class.
subclass	Same as child, derived, or inherited class.
superclass	Same as parent or base class.
attribute	Same as data member or member field.

### 1.2.6 Other Object-Oriented Languages

- Objective C
- CLOS (Common Lisp Object System)
- Ada 9X
- FORTRAN 90
- Smalltalk
- Modula-3
- Eiffel

## 2 Structure Definitions

- Structures, Aggregate data types built using elements of other types



```

struct Time{ //structure tag
int hour;   //structure members
int minute; //structure members
int second; //structure members
}; //struct definition must end with semicolon

```

- Structure member naming
  - In same **struct**: must have unique names
  - In different **structs**: can share name
- Self-referential structure
  - Structure member cannot be instance of enclosing **struct**
  - Structure member can be pointer to instance of enclosing **struct** (self-referential structure), Used for linked lists, queues, stacks and trees
- **struct** definition
  - Creates new data type used to declare variables
  - Structure variables declared like variables of other types
  - Examples:

```

Time timeObject;
Time timeArray[ 10 ];
Time *timePtr;
Time &timeRef = timeObject;

```

### 3 Accessing Structure Members

- Member access operators
  - Dot operator (.) for structure and class members
  - Arrow operator (– >) for structure and class members via pointer to object
  - Print member **hour** of **timeObject**:

```

cout << timeObject.hour;
OR
timePtr = &timeObject;
cout << timePtr->hour;}

```

```

1 // Fig. 6.1: fig06_01.cpp
2 // Create a structure, set its members, and print it.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setfill;
11 using std::setw;
12
13 // structure definition
14 struct Time {
15     int hour;    // 0-23 (24-hour clock format)
16     int minute; // 0-59
17     int second; // 0-59
18 }; // end struct Time
19
20 void printUniversal( const Time &); // prototype
21 void printStandard( const Time &); // prototype
22
23

```

Outline  
 fig06\_01.cpp  
 (1 of 3)

Define structure type **Time**  
 with three integer members.

Pass references to constant  
**Time** objects to eliminate  
 copying overhead.

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 2: Creating a structure, setting its members and printing the structure (part 1 of 2).

- **timePtr** -> **hour** same as ( **\*timePtr** ).**hour**  
 \* Parentheses required, \* lower precedence than .

## 4 Implementing a User-Defined Type *Time* with a *struct*

See Figs.2-3.

- Default: structures passed by value
  - Pass structure by reference; Avoid overhead of copying structure
- C-style structures
  - No *interface*; If implementation changes, all programs using that **struct** must change accordingly
  - Cannot print as unit; Must print/format member by member
  - Cannot compare in entirety; Must compare member by member

```

24 int main()
25 {
26     Time dinnerTime;
27
28     dinnerTime.hour = 18; // set hour member of dinnerTime
29     dinnerTime.minute = 30; // set minute member of dinnerTime
30     dinnerTime.second = 0; // set second member of dinnerTime
31
32     cout << "Dinner will be held at ";
33     printUniversal( dinnerTime );
34     cout << " universal time, \nwhich is ";
35     printStandard( dinnerTime );
36     cout << " standard time.\n";
37
38     dinnerTime.hour = 29; // set hour to invalid value
39     dinnerTime.minute = 73; // set minute to invalid value
40
41     cout << "\nTime with invalid values: ";
42     printUniversal( dinnerTime );
43     cout << endl;
44
45     return 0;
46
47 } // end main
48

```

Use dot operator to initialize structure members.

Direct access to data allows assignment of bad values.



[Outline](#)

fig06\_01.cpp  
(2 of 3)

8

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

49 // print time in universal-time format
50 void printUniversal( const Time &t )
51 {
52     cout << setfill( '0' ) << setw( 2 ) << t.hour << ":"
53         << setw( 2 ) << t.minute << ":"
54         << setw( 2 ) << t.second;
55 } // end function printUniversal
56
57 // print time in standard-time format
58 void printStandard( const Time &t )
59 {
60     cout << ( ( t.hour == 0 || t.hour == 12 ) ?
61             12 : t.hour % 12 ) << ":" << setfill( '0' )
62         << setw( 2 ) << t.minute << ":"
63         << setw( 2 ) << t.second
64         << ( t.hour < 12 ? " AM" : " PM" );
65 } // end function printStandard
66

```

Dinner will be held at 18:30:00 universal time,  
which is 6:30:00 PM standard time.

Time with invalid values: 29:73:00

Use parameterized stream manipulator `setfill`.

Use dot operator to access data members.



[Outline](#)

fig06\_01.cpp  
(3 of 3)

9

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 3: Creating a structure, setting its members and printing the structure (part 2 of 2).

## 5 Implementing a Time Abstract Data Type with a class

- Classes
  - Model objects
    - \* Attributes (data members)
    - \* Behaviors (member functions)
  - Defined using keyword **class**
  - Member functions
    - \* Methods
    - \* Invoked in response to messages
- Member access specifiers
  - **public:** Accessible wherever object of class in scope
  - **private:** Accessible only to member functions of class
  - **protected:**
- Constructor function
  - Special member function
    - \* Initializes data members
    - \* Same name as class
  - Called when object instantiated
  - Several constructors; Function overloading
  - No return type

```
1 class Time {
2
3 public:
4     Time();                // constructor
5     void setTime( int, int, int ); // set hour, minute, second
6     void printUniversal();    // print universal-time format
7     void printStandard();    // print standard-time format
8
9 private:
10    int hour;           // 0 - 23 (24-hour clock format)
```

```

11  int minute;    // 0 - 59
12  int second;   // 0 - 59
13
14  }; // end class Time

```

- Objects of class

- After class definition

- \* Class name new type specifier; C++ extensible language
    - \* Object, array, pointer and reference declarations

- Example:

```

Time sunset;
Time arrayofTimes[ 5 ];
Time *pointerToTime;
Time &dinnerTime = sunset;

```

- Member functions defined outside class

- Binary scope resolution operator (::)

- \* **Ties** member name to class name
    - \* Uniquely identify functions of particular class
    - \* Different classes can have member functions with same name

- Format for defining member functions

```

ReturnType ClassName::MemberFunctionName( ){
.
.
.
}

```

- Does not change whether function **public** or **private**

- Member functions defined inside class

- Do not need scope resolution operator, class name
  - Compiler attempts **inline**; Outside class, inline explicitly with keyword inline

- Destructors

- Same name as class; Preceded with tilde (~)

- No arguments
- Cannot be overloaded
- Performs *termination housekeeping*
- Advantages of using classes
  - Simplify programming
  - Interfaces; Hide implementation
  - Software reuse
    - \* Composition (aggregation); Class objects included as members of other classes
    - \* Inheritance; New classes derived from old

```

1 // Fig. 6.3: fig06_03.cpp
2 // Time class.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9
10 using std::setfill;
11 using std::setw;
12
13 // Time abstract data type (ADT) definition
14 class Time {
15
16 public:
17     Time(); // constructor
18     void setTime( int, int, int ); // set hour, minute, second
19     void printUniversal(); // print universal-time format
20     void printStandard(); // print standard-time format
21

```

15

Outline

fig06\_03.cpp  
(1 of 5)

Define class Time.

Figure 4: **Time** abstract data type implementation as a class, (part 1 of 3).

```

22 private:
23     int hour;    // 0 - 23 (24-hour clock format)
24     int minute; // 0 - 59
25     int second; // 0 - 59
26
27 }; // end class Time
28
29 // Time constructor initializes each data member
30 // ensures all Time objects start in a consistent state
31 Time::Time()
32 {
33     hour = minute = second = 0;
34 }
35 // end Time constructor
36
37 // set new Time value using universal time, perform validity
38 // checks on the data values and set invalid values to zero
39 void Time::setTime( int h, int m, int s )
40 {
41     hour = ( h >= 0 && h < 24 ) ? h : 0;
42     minute = ( m >= 0 && m < 60 ) ? m : 0;
43     second = ( s >= 0 && s < 60 ) ? s : 0;
44 }
45 // end function setTime
46

```

Constructor initializes private data members to 0.

public member function checks parameter values for validity before setting private data members.

```

47 // print Time in universal format
48 void Time::printUniversal()
49 {
50     cout << setfill( '0' ) << setw( 2 ) << hour << ":" <<
51         << setw( 2 ) << minute << ":" <<
52         << setw( 2 ) << second;
53 }
54 // end function printUniversal
55
56 // print Time in standard format
57 void Time::printStandard()
58 {
59     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
60         << ":" << setfill( '0' ) << setw( 2 ) << minute
61         << ":" << setw( 2 ) << second
62         << ( hour < 12 ? " AM" : " PM" );
63 }
64 // end function printStandard
65
66 int main()
67 {
68     Time t; // instantiate object t of class Time
69

```

No arguments (implicitly "know" purpose is to print data members); member function calls more concise.

Declare variable t to be object of class Time.

Figure 5: **Time** abstract data type implementation as a class, (part 2 of 3).

```

70 // output Time object t's initial values
71 cout << "The initial universal time is ";
72 t.printUniversal(); // 00:00:00
73
74 cout << "\nThe initial standard time is ";
75 t.printStandard(); // 12:00:00 AM
76
77 t.setTime( 13, 27, 6 ); // change time
78
79 // output Time object t's new values
80 cout << "\n\nUniversal time after setTime is ";
81 t.printUniversal(); // 13:27:06
82
83 cout << "\n\nStandard time after setTime is ";
84 t.printStandard(); // 1:27:06 PM
85
86 t.setTime( 99, 99, 99 ); // attempt invalid settings
87
88 // output t's values after specifying invalid values
89 cout << "\n\nAfter attempting invalid settings:"
90 << "\n\nUniversal time: ";
91 t.printUniversal(); // 00:00:00
92

```

Invoke public member functions to print time.

Set data members using public member function.

Attempt to set data members to invalid values using public member function.



Outline

fig06\_03.cpp  
(4 of 5)

18

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

93 cout << "\nstandard time: ";
94 t.printStandard(); // 12:00:00 AM
95 cout << endl;
96
97 return 0;
98
99 } // end main

```

```

The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM

```

Data members set to 0 after attempting invalid settings.



Outline

fig06\_03.cpp  
(5 of 5)

fig06\_03.cpp  
output (1 of 1)

19

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 6: **Time** abstract data type implementation as a class, (part 3 of 3).



## 6 Class Scope and Accessing Class Members

- Class scope
  - Data members, member functions
  - Within class scope
    - \* Class members; Immediately accessible by all member functions, Referenced by name
  - Outside class scope
    - \* Referenced through handles; Object name, reference to object, pointer to object
- File scope
  - Nonmember functions
- Function scope
  - Variables declared in member function
  - Only known to function
  - Variables with same name as class-scope variables
    - \* Class-scope variable *hidden*; Access with scope resolution operator (`::`)  
`ClassName::classVariableName`
  - Variables only known to function they are defined in
  - Variables are destroyed after function completion
- Operators to access class members
  - Identical to those for **structs**
  - Dot member selection operator (`.`)
    - \* Object
    - \* Reference to object
  - Arrow member selection operator (`->`)
    - \* pointers


```


1 // Fig. 6.4: fig06_04.cpp
2 // Demonstrating the class member access operators . and ->
3 //
4 // CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
5 #include <iostream>
6
7 using std::cout;
8 using std::endl;
9
10 // class Count definition
11 class Count {
12
13 public:
14     int x;
15
16     void print()
17     {
18         cout << x << endl;
19     }
20
21 }; // end class Count
22

```

Data member `x` public to illustrate class member access operators; typically data members private.

25

 [Outline](#)

 `fig06_04.cpp`  
(1 of 2)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

23 int main()
24 {
25     Count counter; // create counter object
26     Count *counterPtr = &counter; // create pointer to counter
27     Count &counterRef = counter;
28
29     cout << "Assign 1 to x and print using the object's name: ";
30     counter.x = 1; // assign 1 to x
31     counter.print(); // call member function print
32
33     cout << "Assign 2 to x and print using a reference: ";
34     counterRef.x = 2; // assign 2 to x
35     counterRef.print(); // call member function print
36
37     cout << "Assign 3 to x and print using a pointer: ";
38     counterPtr->x = 3; // assign 3 to data member x
39     counterPtr->print(); // call member function print
40
41     return 0;
42
43 } // end main

```


Use dot member selection operator for counter object.


Use dot member selection operator for counterRef reference to object.

Use arrow member selection operator for counterPtr pointer to object.

Assign 1 to x and print using the object's name: 1  
Assign 2 to x and print using a reference: 2  
Assign 3 to x and print using a pointer: 3

26

 [Outline](#)

 `fig06_04.cpp`  
(2 of 2)

`fig06_04.cpp`  
output (1 of 1)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 7: Demonstrating the class member access operators. `.` and `->`

## 7 Separating Interface from Implementation

(see Figs. 8-10)

- Separating interface from implementation
  - Advantage; Easier to modify programs
  - Disadvantage
    - \* Header files
    - \* Portions of implementation; Inline member functions
    - \* Hints about other implementation; private members
    - \* Can hide more with proxy class
- Header files
  - Class definitions and function prototypes
  - Included in each file using class; **#include**
  - File extension **.h**
- Source-code files
  - Member function definitions
  - Same base name; Convention
  - Compiled and linked

·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·  
·

```

1 // Fig. 6.5: time1.h
2 // Declaration of class Time.
3 // Member functions are defined in time1.cpp
4
5 // prevent multiple inclusions of header file
6 #ifndef TIME1_H
7 #define TIME1_H
8
9 // Time abstract class
10 class Time {
11
12 public:
13     Time(); // constructor
14     void setTime( int, int, int ); // set hour, minute, second
15     void printUniversal(); // print universal-time format
16     void printStandard(); // print standard-time format
17
18 private:
19     int hour; // 0 - 23 (24-hour clock format)
20     int minute; // 0 - 59
21     int second; // 0 - 59
22
23 }; // end class Time
24
25 #endif

```

Preprocessor code to prevent multiple inclusions.

Code between these directives

If not defined, then... defines

Naming convention: header file name with underscore replacing period.

```

1 // Fig. 6.6: time1.cpp
2 // Member-function definitions for class Time.
3 #include <iostream>
4
5 using std::cout;
6
7 #include <iomanip>
8
9 using std::setfill;
10 using std::setw;
11
12 // include definition of class Time from time1.h
13 #include "time1.h"
14
15 // Time constructor initial
16 // Ensures all Time objects
17 Time::Time()
18 {
19     hour = minute = second = 0;
20 } // end Time constructor
21
22

```

Include header file time1.h.

Name of header file enclosed in quotes; angle brackets cause preprocessor to assume header part of C++ Standard Library.

Figure 8: **Time** class definition and **Time** class member-function definitions (part 1 of 2).

```
23 // Set new Time value using universal time. Perform validity
24 // checks on the data values. Set invalid values to zero.
25 void Time::setTime( int h, int m, int s )
26 {
27     hour = ( h >= 0 && h < 24 ) ? h : 0;
28     minute = ( m >= 0 && m < 60 ) ? m : 0;
29     second = ( s >= 0 && s < 60 ) ? s : 0;
30
31 } // end function setTime
32
33 // print Time in universal format
34 void Time::printUniversal()
35 {
36     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
37         << setw( 2 ) << minute << ":"
38         << setw( 2 ) << second;
39
40 } // end function printUniversal
41
```



[Outline](#)

31

time1.cpp (2 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```
42 // print Time in standard format
43 void Time::printStandard()
44 {
45     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
46         << ":" << setfill( '0' ) << setw( 2 ) << minute
47         << ":" << setw( 2 ) << second
48         << ( hour < 12 ? " AM" : " PM" );
49
50 } // end function printStandard
```



[Outline](#)

32

time1.cpp (3 of 3)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 9: **Time** class member-function definitions (part 2 of 2).


```

1 // Fig. 6.7: fig06_07.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with time1.cpp.
4 #include <iostream>
5
6 using std::cout;
7 using std::endl;
8
9 // include definition of class Time
10 #include "time1.h"
11
12 int main()
13 {
14     Time t; // instantiate object t of class Time
15
16     // output Time object t's initial values
17     cout << "The initial universal time is ";
18     t.printUniversal(); // 00:00:00
19     cout << "\nThe initial standard time is ";
20     t.printStandard(); // 12:00:00 AM
21
22     t.setTime( 13, 27, 6 ); // change time
23

```

Include header file `time1.h` to ensure correct creation/manipulation and determine size of `Time` class object.

33

 [Outline](#)




fig06\_07.cpp  
(1 of 2)

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

24 // output Time object t's new values
25 cout << "\n\nUniversal time after setTime is ";
26 t.printUniversal(); // 13:27:06
27 cout << "\n\nStandard time after setTime is ";
28 t.printStandard(); // 1:27:06 PM
29
30 t.setTime( 99, 99, 99 ); // attempt invalid settings
31
32 // output t's values after specifying invalid values
33 cout << "\n\nAfter attempting invalid settings:"
34     << "\n\nUniversal time: ";
35 t.printUniversal(); // 00:00:00
36 cout << "\n\nStandard time: ";
37 t.printStandard(); // 12:00:00 AM
38 cout << endl;
39
40 return 0;
41

```

```

The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

```

34

 [Outline](#)



fig06\_07.cpp  
(2 of 2)

fig06\_07.cpp  
output (1 of 1)

© 2003 Prentice Hall, Inc.  
All rights reserved.

Figure 10: Program to test class `Time`.

## 8 Controlling Access to Members

- Access modes
  - **private**
    - \* Default access mode
    - \* Accessible to member functions and **friends**
  - **public**
    - \* Accessible to any function in program with handle to class object
    - \* **protected** ; (discuss later)
- Class member access
  - Default **private**
  - Explicitly set to **private**, **public**, **protected**
- Access to class's **private** data
  - Controlled with access functions (accessory methods)
    - \* Get function; Read **private** data
    - \* Set function; Modify **private** data

36

```

1 // Fig. 6.8: fig06_08.cpp
2 // Demonstrate errors resulting from attempts
3 // to access private class members.
4 #include <iostream>
5
6 using std::cout;
7
8 // include definition of class Time from time1.h
9 #include "time1.h"
10
11 int main()
12 {
13     Time t; // create Time object
14
15     t.hour = 7; // error: 'Time'
16
17     // error: 'Time::minute' is not accessible
18     cout << "minute = " << t.minute;
19
20     return 0;
21 } // end main

```

Recall data member **hour** is **private**; attempts to access **private** members results in error.

Data member **minute** also **private**; attempts to access **private** members produces error.

Outline

fig06\_08.cpp  
(1 of 1)

37

© 2003 Prentice Hall, Inc.  
All rights reserved.

```

D:\cpphtp4_examples\ch06\Fig6_06\Fig06_06.cpp(16) : error C2248 :
'hour' : cannot access private member declared in class 'Time'
D:\cpphtp4_examples\ch06\Fig6_06\Fig06_06.cpp(19) : error C2248 :
'minute' : cannot access private member declared in class 'Time'

```

Errors produced by attempting to access **private** members.

Outline

fig06\_08.cpp  
output (1 of 1)

Figure 11: **private** members of a class are not accessible outside the class.