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.
  - Abstractions allow us to view screen images as objects such as people, planes, trees, etc. rather than as individual dots of color.
  - Abstractions allow us to think in terms of beaches rather than grains of sand, houses rather than bricks.
  - All objects have attributes such as size, shape, color, weight, etc.
  - All objects exhibit various behaviors. A baby cries, sleeps, crawls, walks; a car accelerates, brakes, turns, etc.
  - 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.
    - Comparisons can be made between babies and adults, and between humans and chimpanzees.
    - Cars, trucks, little red wagons, and roller blades have much in common.
  - 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.

- Each object has its own memory, and is made up of other objects.
- 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**).
  - Defining a class requires defining its interface.
  - What about built-in types?
    * Think of an *int*
    * What's its interface?
    * How do you "send it messages"?
    * How do you make (construct) one?

- 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 libraries have always been like this, sort of:
  - The library designer invents a useful struct.
  - Then she provides some useful functions for the struct.
  - The user creates an instance of the struct, then applies library functions to it.

- 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.

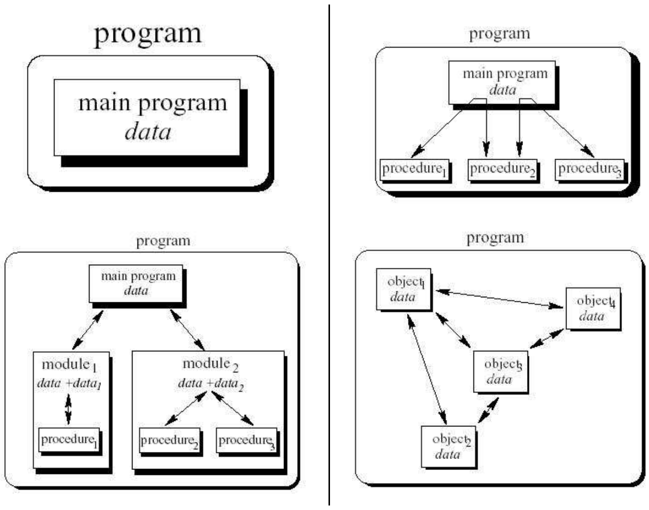


Figure 1: Survey of Programming Techniques; unstructured, procedural, modular, and object-oriented programming.

- 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.
    * 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 base 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 in to 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 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 panacea, 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 OPP 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. Altho 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 latter.

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. Encapsulation and weak coupling between objects means class definitions are more likely to be re-used in other applications. Objects as well as procedures (focus of C libraries) become likely candidates for re-use. The enforcement of a consistent interface to objects lessens code duplication.

- 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. The analysis/design/coding phases in development become more seamless since they can all deal in the same concepts.

- Ease of fabrication and maintenance (redesign and extension) facilitated by encapsulation, data abstraction which allow for very clean designs. When an object is going into disallowed states, only its methods need be investigated. This narrows down search for problems.

- C++ Objectives

  - extend C to allow for object-oriented programming
  - other improvements - some resulting in deprecation of some C facilities
  - remain compatible and comparable (syntax, performance, portability, design philosophy - don't pay for what you don't use, don't get stuck with things you don't need) with C
  - emphasize compile-time type checking

- C++ is multi-paradigm. It provides for the object-oriented approach but doesn't enforce its use. This makes it a good transition language and gives it flexibility when a particular situation doesn't fit the object-oriented philosophy.

- With this object-oriented approach, C++ overcomes certain shortcomings of C:

  - Lack of encapsulation means that if an object is getting trashed, it's difficult to find the code responsible. Many procedures may have had idiosyncratic interactions with the object.
  - Doesn't recognize relationships between types. Pointer casting necessary. In C++, pointer casting can just about always be dispensed with. Pointer casting is a kludge. Compiler can't check if you are doing it correctly. No type safety (see definition below).
  - Not easy to extend existing libraries; for example, make it so printf() can handle new types.
  - Except for FILEs, there are no well-developed objects (like stacks and lists) in the standard libraries.

- C's future is as a portable "universal" assembler, a back end for code generators.

- 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.

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 OO 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 new.
class	A struct with both data and functions.
object	Memory allocated to a class/struct. Often allocated with new.
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 new).
destructor	Function-like code that is called when an object is deleted to free any resources (eg, memory) that is has pointers to.
inheritance	Defining a class (child) in terms of another class (parent). All of the public members of the public 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
  };
  ```

- Structure member naming
  - In same **struct**: must have unique names
  - In different **struct**s: can share name
- **struct** definition must end with semicolon
- 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;}
    ```

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

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

- 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

5 Implementing a Time Abstract Data Type with a class

- Classes
  - Model objects
    - Attributes (data members)

7

Outline

fig06_01.cpp
(1 of 3)

```cpp
// Fig. 6.1: fig06_01.cpp
// Create a structure, set its members, and print it.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setfill;
using std::setw;

// structure definition
struct Time {
   int hour;     // 0-23 (24-hour clock format)
   int minute;   // 0-59
   int second;   // 0-59

}; // end struct Time

void printUniversal( const Time & );  // prototype
void printStandard( const Time & );   // prototype

```

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

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

8

Outline

fig06_01.cpp
(2 of 3)

```cpp
int main()
{
   Time dinnerTime;      /

   dinnerTime.hour = 18;     // set hour member of dinnerTime
   dinnerTime.minute = 30;   // set minute member of dinnerTime
   dinnerTime.second = 0;    // set second member of dinnerTime

   cout << "Dinner will be held at ";
   printUniversal( dinnerTime );
   cout << " universal time,\nwhich is ";
   printStandard( dinnerTime );
   cout << " standard time.\n";

   dinnerTime.hour = 29;     // set hour to invalid value
   dinnerTime.minute = 73;   // set minute to invalid value

   cout << "\nTime with invalid values: ";
   printUniversal( dinnerTime );
   cout << endl;

   return 0;

} // end main

```

Use dot operator to initialize structure members.

Direct access to data allows assignment of bad values.

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

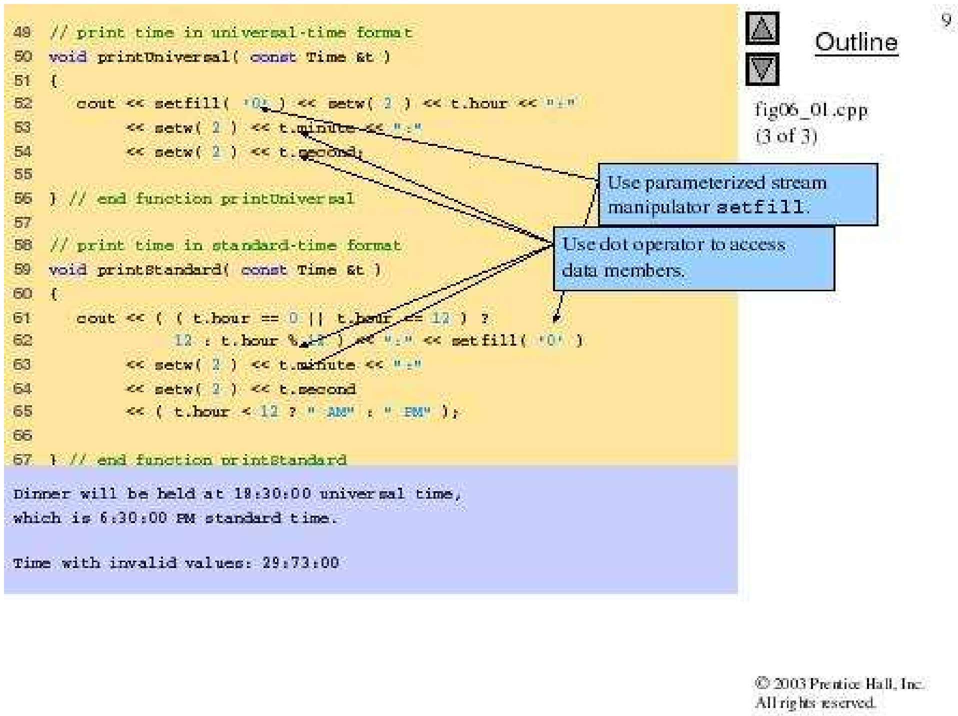


Figure 3: Creating a structure, setting its members and printing the structure (part 2 of 2).

- ∗ 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

```
class Time {

public:
   Time();                        // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printUniversal();         // print universal-time format
   void printStandard();          // print standard-time format

private:
  int hour;     // 0 - 23 (24-hour clock format)
  int minute;   // 0 - 59
  int second;   // 0 - 59

 }; // 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

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

Outline

fig06_03.cpp
(1 of 5)

```cpp
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setfill;
using std::setw;

// Time abstract data type (ADT) definition
class Time {

public:
   Time();                        // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printUniversal();         // print universal-time format
   void printStandard();          // print standard-time format

```

Define class **Time**.

Outline

fig06_03.cpp
(2 of 5)

```cpp
private:
   int hour;     // 0 - 23 (24-hour clock format)
   int minute;   // 0 - 59
   int second;   // 0 - 59

}; // end class Time

// Time constructor initializes each data me
// ensures all Time objects start in a consi
Time::Time()
{
   hour = minute = second = 0;

} // end Time constructor

// set new Time value using universal time, perform validity
// checks on the data values and set invalid values to zero
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // end function setTime

```

Constructor initializes **private** data members to **0**.

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

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

```cpp
// print Time in universal format
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

// print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << setfill( '0' ) << setw( 2 ) << minute
        << ":" << setw( 2 ) << second
        << ( hour < 12 ? " AM" : " PM" );

} // end function printStandard

int main()
{
   Time t;  // instantiate object t of class Time

```

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

Declare variable **t** to be object of class **Time**.

Outline

fig06_03.cpp
(3 of 5)

17


```cpp
   // output Time object t's initial values
   cout << "The initial universal time is ";
   t.printUniversal();  // 00:00:00

   cout << "\nThe initial standard time is ";
   t.printStandard();   // 12:00:00 AM

   t.setTime( 13, 27, 6 );   // change time

   // output Time object t's new values
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal();   // 13:27:06

   cout << "\nStandard time after setTime is ";
   t.printStandard();   // 1:27:06 PM

   t.setTime( 99, 99, 99 );  // attempt invalid settings

   // output t's values after specifying invalid values
   cout << "\n\nAfter attempting invalid settings:"
        << "\nUniversal time: ";
   t.printUniversal();   // 00:00:00

```

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

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

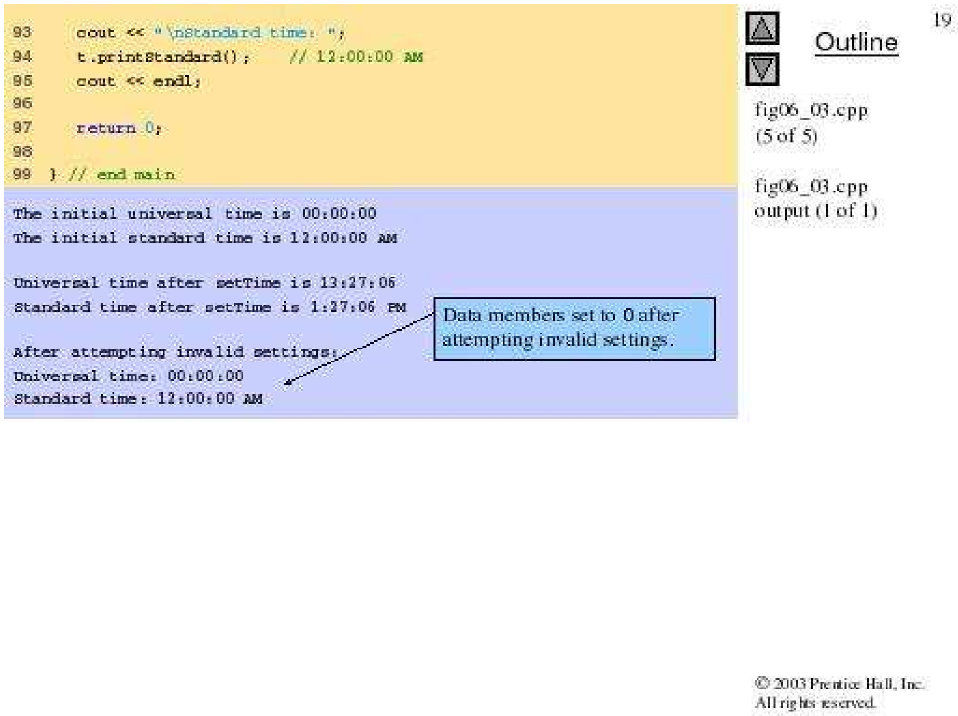


Figure 6: **Time** abstract data type implementation as a class, (part 3 of 3).

    * 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 **struct**s
  - Dot member selection operator (**.**)
    - Object
    - Reference to object
  - Arrow member selection operator ($->$)
    - pointers

7 Separating Interface from Implementation (see Figs 8-11)

- 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

Outline

fig06_04.cpp
(1 of 2)

```cpp
// Fig. 6.4: fig06_04.cpp
// Demonstrating the class member access operators . and ->
//
// CAUTION: IN FUTURE EXAMPLES WE AVOID PUBLIC DATA!
#include <iostream>

using std::cout;
using std::endl;

// class Count definition
class Count {

public:
   int x;

   void print()
   {
      cout << x << endl;
   }

}; // end class Count

```

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

Outline

fig06_04.cpp
(2 of 2)

fig06_04.cpp
output (1 of 1)

```cpp
int main()
{
   Count counter;                 // create counter object
   Count *counterPtr = &counter; // create pointer to counter
   Count &counterRef = counter;  /

   cout << "Assign 1 to x and print
   counter.x = 1;         // assign 1 t
   counter.print();       // call membe

   cout << "Assign 2 to x and print u
   counterRef.x = 2;      // assign 2 to
   counterRef.print();    // call member

   cout << "Assign 3 to x and print us
   counterPtr->x = 3;     // assign 3 to data member x
   counterPtr->print();   // call member function print

   return 0;

} // 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
```


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

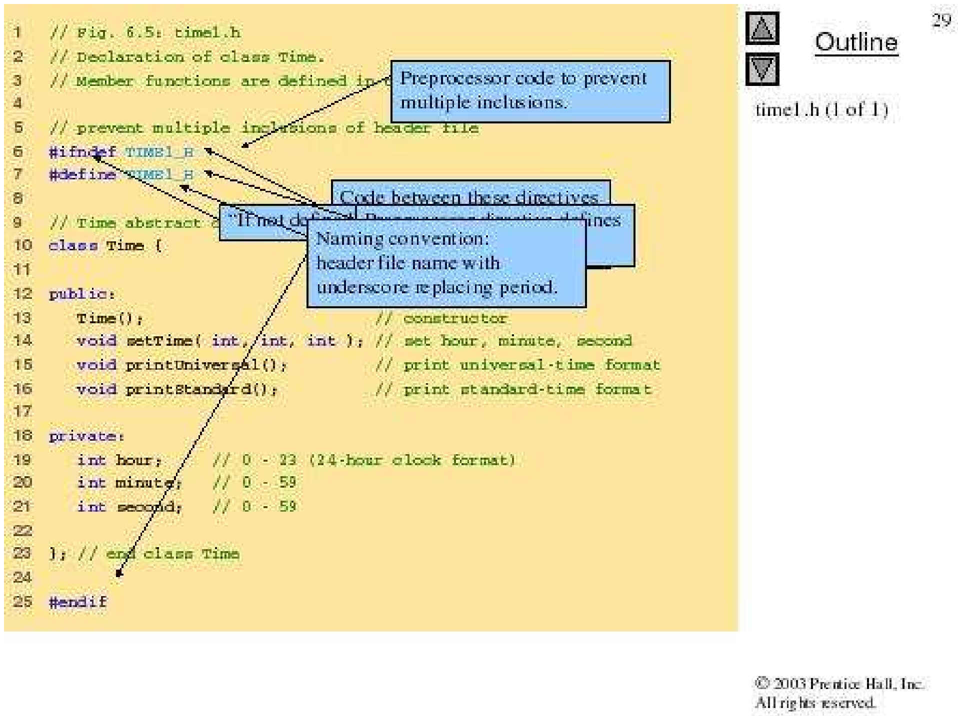


Figure 8: **Time** class definition

8 Controlling Access to Members (see Fig. 12)

- Access modes
  - **private**
    - Default access mode
    - Accessible to member functions and **friend**s
  - **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**

Outline

time1.cpp (1 of 3)

```
// Fig. 6.6: time1.cpp
// Member-function definitions for class Time.
#include <iostream>

using std::cout;

#include <iomanip>

using std::setfill;
using std::setw;

// include definition of class Time from time1.h
#include "time1.h"

// Time constructor initial
// Ensures all Time objects
Time::Time()
{
   hour = minute = second =

} // end Time constructor

```

Include header file time1.h.

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

Outline

time1.cpp (2 of 3)

```
// Set new Time value using universal time. Perform validity
// checks on the data values. Set invalid values to zero.
void Time::setTime( int h, int m, int s )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0;
   minute = ( m >= 0 && m < 60 ) ? m : 0;
   second = ( s >= 0 && s < 60 ) ? s : 0;

} // end function setTime

// print Time in universal format
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
        << setw( 2 ) << minute << ":"
        << setw( 2 ) << second;

} // end function printUniversal

```


Figure 9: **Time** class member-function definitions (part 1 of 2).

```cpp
// print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << setfill( '0' ) << setw( 2 ) << minute
        << ":" << setw( 2 ) << second
        << ( hour < 12 ? " AM" : " PM" );

} // end function printStandard
```

Outline

time1.cpp (3 of 3)

32

Figure 10: **Time** class member-function definitions (part 2 of 2).

- **struct** member access
  - Default **public**
  - Explicitly set to **private**, **public**, **protected**
- Access to class's **private** data
  - Controlled with access functions (accessor methods)
    - Get function; Read **private** data
    - Set function; Modify **private** data

33

Outline

fig06_07.cpp
(1 of 2)

```
// Fig. 6.7: fig06_07.cpp
// Program to test class Time.
// NOTE: This file must be compiled with time1.cpp.
#include <iostream>

using std::cout;
using std::endl;

// include definition of class Time
#include "time1.h"

int main()
{
   Time t;  // instantiate object t of class Time

   // output Time object t's initial values
   cout << "The initial universal time is ";
   t.printUniversal();   // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard();    // 12:00:00 AM

   t.setTime( 13, 27, 6 );   // change time

```

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

34

Outline

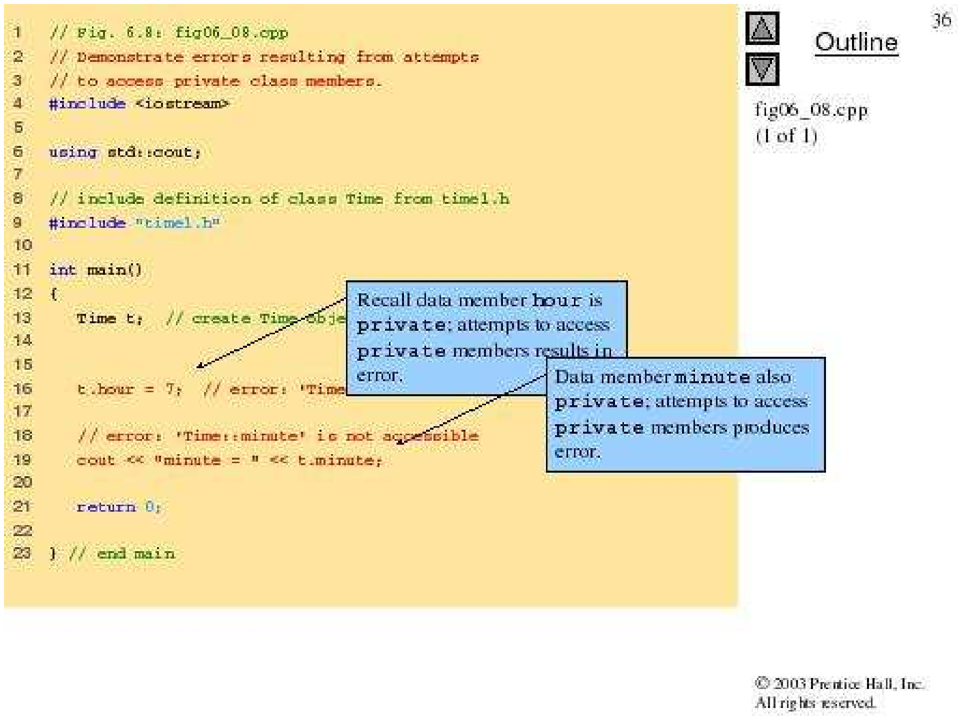
fig06_07.cpp
(2 of 2)

fig06_07.cpp
output (1 of 1)

```
   // output Time object t's new values
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal();   // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard();    // 1:27:06 PM

   t.setTime( 99, 99, 99 );  // attempt invalid settings

   // output t's values after specifying invalid values
   cout << "\n\nAfter attempting invalid settings:"
        << "\nUniversal time: ";
   t.printUniversal();   // 00:00:00
   cout << "\nStandard time: ";
   t.printStandard();    // 12:00:00 AM
   cout << endl;

   return 0;

```

```
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
```


Figure 11: Program to test class **Time**.



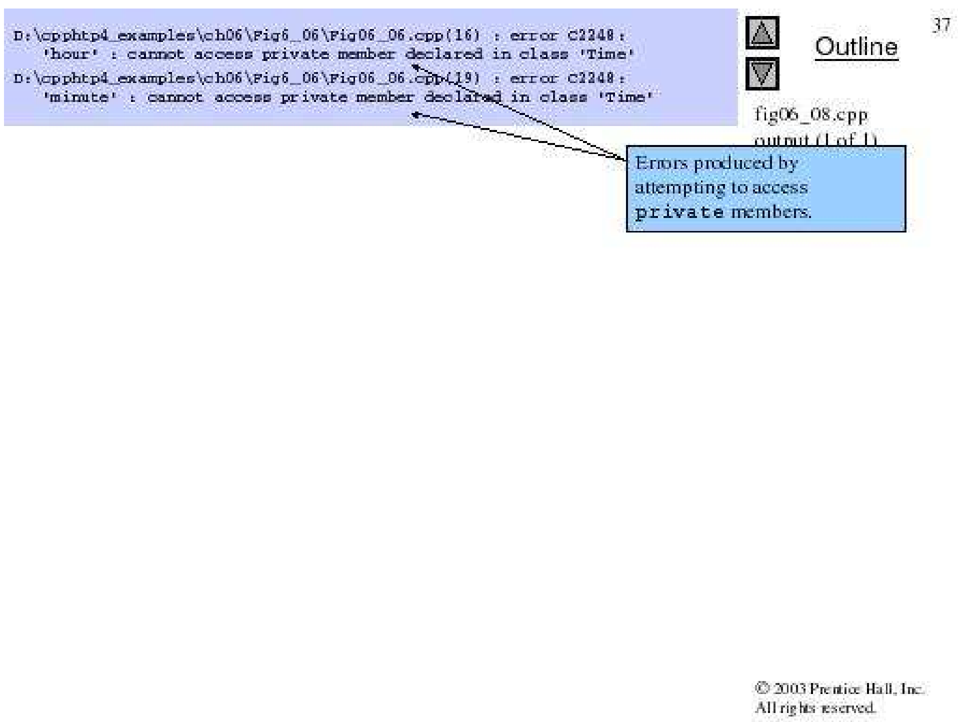


Figure 12: **private** members of a class are not accessible outside the class.