More about OOP and ADTs Classes

Chapter 4

Chapter Contents

Chapter Objectives

- Contrast OOP with procedural programming
- Review classes in C++
- Study in detail a specific example of how a class is built
- Show how operators can be overloaded for new types
- Show how conditional compilation directives are used to avoid redundant declarations
- Discuss pointers to class objects – the `this` pointer, in particular

Contrast Procedural, Object Oriented Paradigms

Procedural

- *Action*-oriented — concentrates on the *verbs*
- Programmers:
  - Identify basic tasks to solve problem
  - Implement actions to do tasks as subprograms (procedures/functions/ subroutines)
  - Group subprograms into programs/modules/libraries,
  - together make up a complete system for solving the problem

Object-oriented

- Focuses on the *nouns* of problem specification
- Programmers:
  - Determine objects needed for problem
  - Determine how they should work together to solve the problem.
  - Create types called *classes* made up of
    - *data members*
    - *function members* to operate on the data.
  - Instances of a type (class) called *objects.*

Structs and Classes Similarities

- Essentially the same syntax
- Both are used to model objects with multiple attributes (characteristics)
  - represented as data members
  - also called fields … or …
  - instance or attribute variables).
- Thus, both are used to process non-homogeneous data sets.

Structs vs. Classes Differences

- No classes in C
- Members public by default
- Can be specified private

- Both structs and classes in C++
- Structs can have members declared private
- Class members are private by default
- Can be specified public

Advantages in C++ (structs and Classes)

- C++ structs and classes model objects which have:
  - Attributes represented as data members
  - Operations represented as functions (or methods)
- Leads to object oriented programming
  - Objects are self contained
  - "I can do it myself" mentality
  - They do not pass a parameter to an external function

Class Declaration

- Syntax

```
class ClassName
{
   public:
      Declarations of public members
   private:
      Declarations of private members
};
```

- See sample class declaration on page 147

Class Declaration outline

```
class Class_name
{
public: (BEHAVIOR)
        constructors
        destructor
        member functions
                accessors
                mutators
        public data
private: (STATE)
        helper functions
        data
};
```

Designing a Class

- Data members are normally placed in private section of a class
- Function members are usually placed in public section
- Typically, the public section is followed by the private section, although not required by compiler

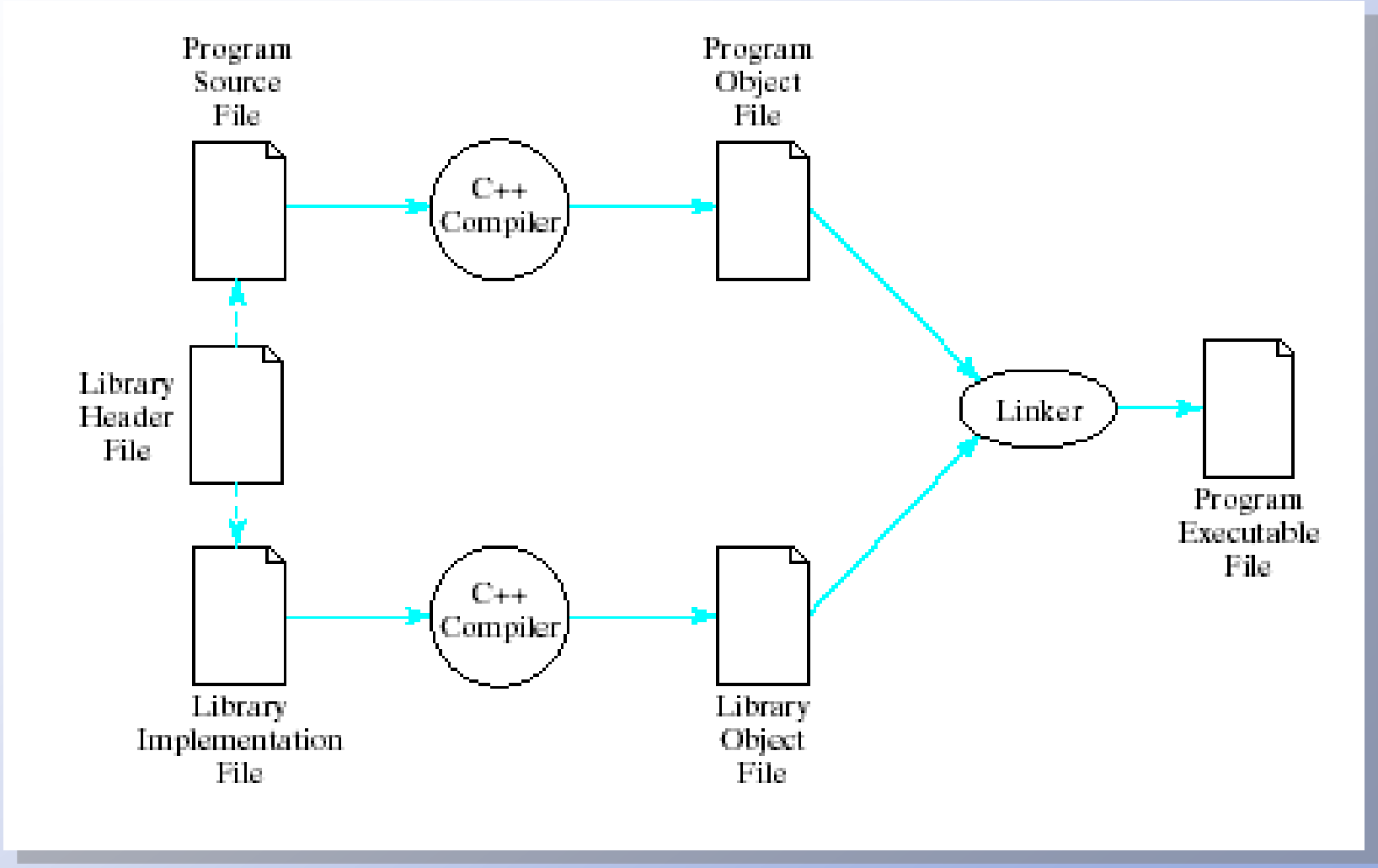
Class Libraries

- Class declarations placed in header file
  - Given `.h` extension
  - Contains data items and prototypes
- Implementation file
  - Same prefix name as header file
  - Given `.cpp` extension
- Programs which use this class library called client programs

Definition of Compilation and Linking

- Compilation, in which a source program is translated to an equivalent machine-language program, called an object program, which is stored in an object file.
- Linking, in which any calls to functions that are defined in a library are linked to their definitions, creating an executable program, which is stored in an executable file.

Translating a Library



Object

- An instance of a class is called a object.
- The type of an object is a class.
- An object can be accessed using the dot operator:
  - objectName.memberName

Example of User-Defined `Time` Class

(see code in CodeSampleChapter 04)

- See time class declaration and implementation (time.h and time.cpp)
  - Actions done to `Time` object, done by the object itself
- Note interface for `Time` class object,
  Data members private – inaccessible to users of the class. The allows information hiding to occur.

Constructors

(see code in CodeSampleChapter 04)

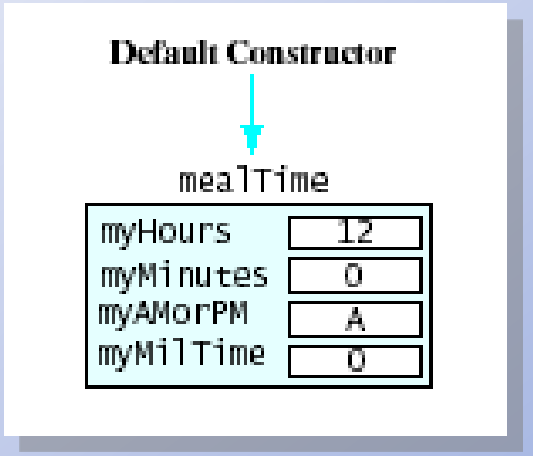
- Note constructor definition in **`Time.cpp`**
- example
- Syntax

```
ClassName::ClassName (parameter_list)
: member_initializer_list
{
   // body of constructor definition
}
```

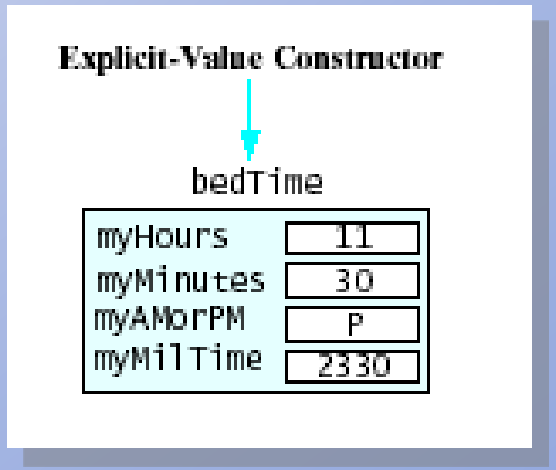
Constructors in time.cpp

(see code in CodeSampleChapter 04)

- Results of default constructor



- Results of explicit-value constructor



Overloading Functions

(see code in CodeSampleChapter 04)

- Note existence of multiple functions with the same name

```
Time();
Time(unsigned initHours,
     unsigned initMinutes,
     char initAMPM);
```

  - Known as overloading
- Compiler compares numbers and types of arguments of overloaded functions
  - Checks the "signature" of the functions

Default Arguments

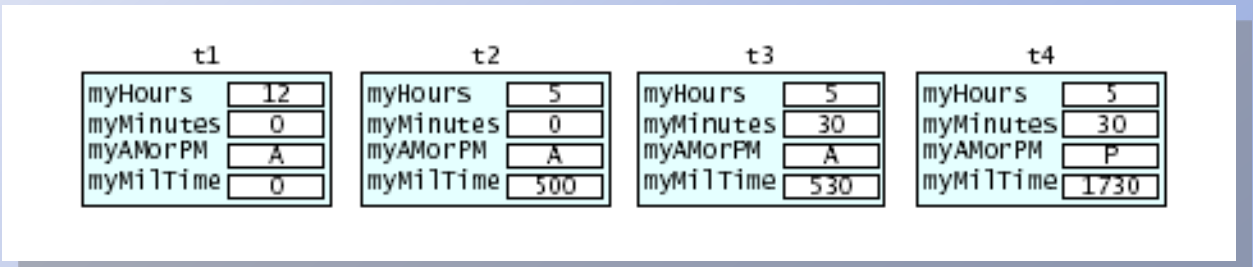
(see code in CodeSampleChapter 04)

- Possible to specify default values for constructor arguments

```
Time(unsigned initHours = 12,
     unsigned initMinutes = 0,
     char initAMPM = 'A');
```

- Consider

```
Time t1, t2(5), t3(6,30), t4(8,15,'P');
```

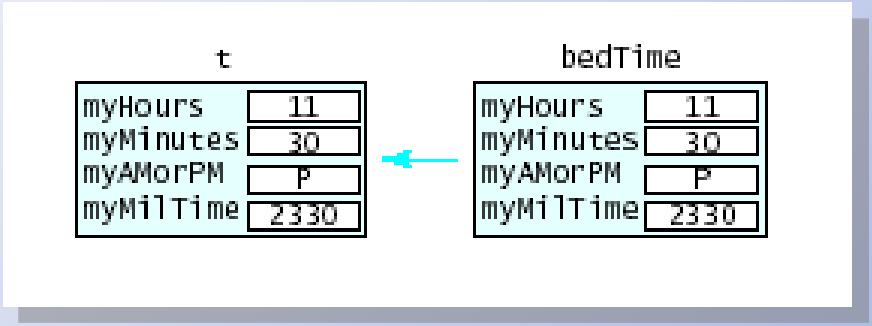


Copy Operations

(see code in CodeSampleChapter 04)

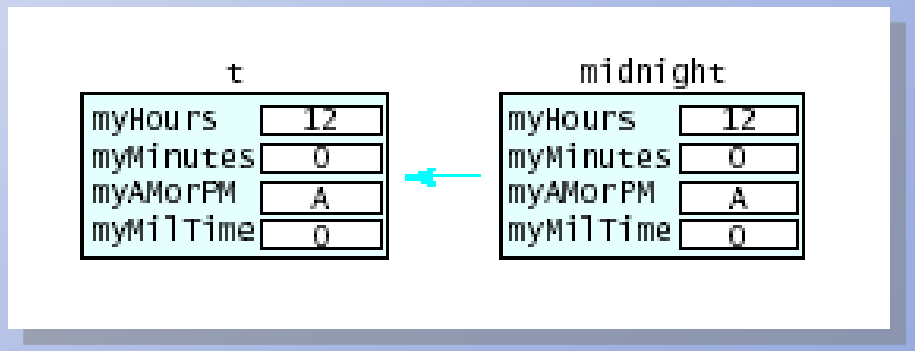
- During initialization

```
Time t = bedTime
```



- During Assignment

```
t = midnight;
```



Other Class Operations

(see code in CodeSampleChapter 04)

- Accessors and Mutators
  - See "get" and "set" functions
- Overloading operators
  - Same symbol can be used more than one way
  - Note declaration for I/O operators << and >>
  - Note definition of overloaded I/O operators

Friend Functions

(see code in CodeSampleChapter 04)

- Note use of two functions used for output
  - **`display()`** and **`operator<<()`**
- Possible to specify operator<<() as a "friend" function
  - Thus given "permission" to access private data elements
- Declaration in .h file (but not inside class)

```
friend ostream & operator<<(
    ostream & out, const Time & t)
```

Friend Functions

(see code in CodeSampleChapter 04)

- Definition in .cpp file

```
ostream & operator<<(
     ostream & out, const Time & t)
{  out << t.myHours<<":"
  <<(t.myMinutes< 10? "0" : "")
  <<t.myMinutes
  << ' '<<t.myAMorPM<<
  return out;
}
```

- Author prefers not to use friend function
- Violates principle of information hiding

- Note - a friend function not member function
  - not qualified with class name and `::`
  - receives class object on which it operates as a parameter

Other Operations

(see code in CodeSampleChapter 04)

- Advance Operation
  - `Time` object receives a number of hours and minutes
  - Advances itself by adding to `myHours`, `myMinutes`
- Relational Operators
  - `Time` object compares itself with another
  - Determines if it is less than the other

Redundant Declarations

(see code in CodeSampleChapter 04)

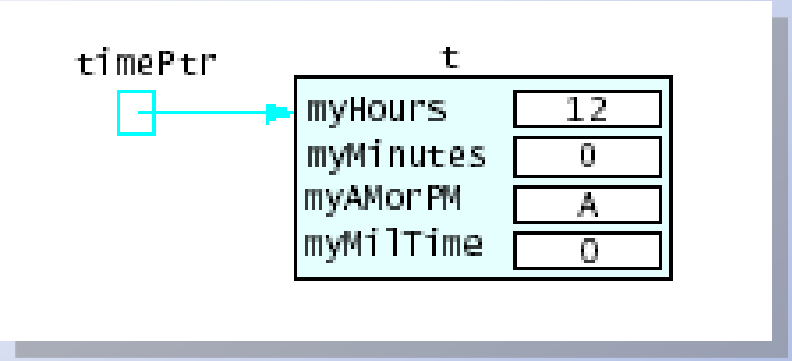
- Note use of #include "Time.h" in
  - Time.cpp
  - Client program
- Causes "redeclaration" errors at compile time
- Solution is to use conditional compilation
  - Use #ifndef and #define and #endif compiler directives

Pointers to Class Objects

(see code in CodeSampleChapter 04)

- Possible to declare pointers to class objects

```
Time * timePtr = &t;
```



- Access with

```
timePtr->getMilTime()   or
(*timePtr).getMilTime()
```

The `this` Pointer

(see code in CodeSampleChapter 04)

- Every class has a keyword, `this`
  - a pointer whose value is the address of the object
  - Value of `*this` would be the object itself