



COMPREHENSIVE EDITION

PROGRAMMING
AND PROBLEM
SOLVING WITH

C++

SIXTH EDITION

Nell Dale and Chip Weems

Chapter 12

Classes and Abstraction

Background image © Toncsi/Shutterstock, Inc.
Copyright © 2014 by Jones & Bartlett Learning, LLC, an Ascend Learning Company
www.jblearning.com

Chapter 12 Topics

- **Meaning of an Abstract Data Type**
- **Declaring and Using a `class` Data Type**
- **Using Separate Specification and Implementation Files**
- **Invoking `class` Member Functions in Client Code**
- **C++ `class` Constructors**

Abstraction

- **Abstraction** is the **separation** of the essential qualities of an object from the details of how it works or is composed
 - Focuses on **what, not how**
 - Necessary for managing large, complex software projects

Control Abstraction

- **Control abstraction** separates the logical properties of an action from its implementation:

`Search (list, item, length, where,
found);`

- The function call depends on the function's specification (description), not its implementation (algorithm)

Data Abstraction

- **Data abstraction** separates the logical properties of a data type from its implementation

LOGICAL PROPERTIES

What are the possible values?

What operations will be needed?

IMPLEMENTATION

How can this be done in C++?

How can data types be used?

Data Type

```
graph TD; A[Data Type] --> B[set of values (domain)]; A --> C[allowable operations on those values]
```

**set of values
(domain)**

**allowable operations
on those values**

FOR EXAMPLE, data type `int` has

domain

-32768 . . . 32767

operations

+, -, *, /, %, >>, <<

Abstract Data Type (ADT)

- An **abstract data type** is a data type whose properties (domain and operations) are specified (*what*) independently of any particular implementation (*how*)

For example . . .

ADT Specification Example

TYPE

Time

DOMAIN

Each Time value is a time in hours, minutes, and seconds.

OPERATIONS

Set the time

Print the time

Increment by one second

Compare 2 times for equality

Determine if one time is “less than” another

Another ADT Specification

TYPE

ComplexNumber

DOMAIN

Each value is an ordered pair of real numbers (a, b) representing $a + bi$

Another ADT Specification, cont...

OPERATIONS

Initialize the complex number

Write the complex number

Add

Subtract

Multiply

Divide

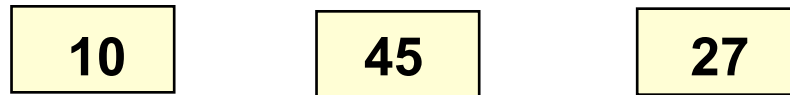
Determine the absolute value of a complex number

ADT Implementation

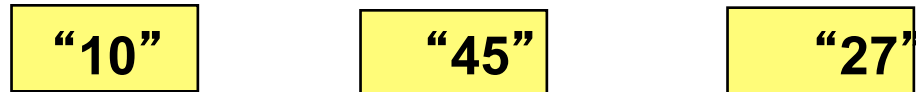
- **ADT implementation**
 - Choose a specific data representation for the abstract data using data types that already exist (built-in or programmer-defined)
 - Write functions for each allowable operation

Several Possible Representations of ADT Time

3 int variables



3 strings



3-element int array



**Choice of representation depends on time,
space, and algorithms needed to
implement operations**

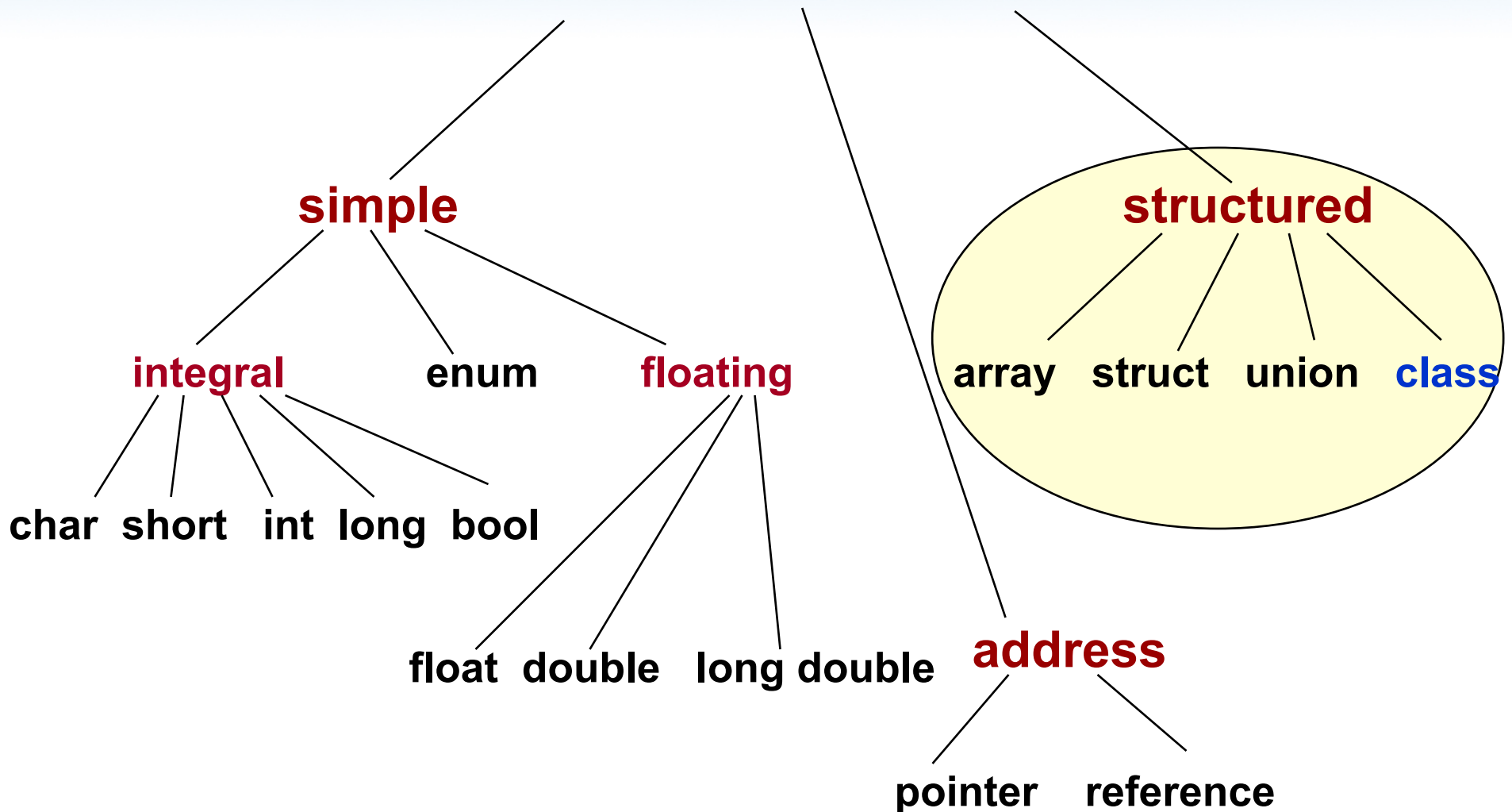
Some Possible Representations of ADT `ComplexNumber` struct with 2 float members

-16.2	5.8
.real	.imag

2-element float array

-16.2	5.8
--------------	------------

C++ Data Types



class Time Specification

// Specification file (Time.h)

```
class Time      // Declares a class data type  
{              // does not allocate memory
```


class Time Specification

```
public :    // Five public function members
```

```
    void Set (int  hours , int  mins , int  
secs);
```

```
    void Increment ();
```

```
    void Write ()  const;
```

```
    bool Equal (Time  otherTime)  const;
```

```
    bool LessThan (Time  otherTime)  const;
```

```
private :  // Three private data members
```

```
    int  hrs;
```

```
    int  mins;
```

```
    int  secs;
```

```
};
```

C++ classType

- Facilitates **re-use** of C++ code for an ADT
- Software that uses the class is called a **client**
- Variables of the class type are called **class objects** or **class instances**
- Client code uses class's public member functions to manipulate class objects

Client Code Using Time

```
// Includes specification of the class
#include "time.h"

using namespace std;

int main ()
{
```

Client Code Using Time

```
Time  currentTime; // Declares two objects of Time
Time  endTime;
bool  done  =  false;

currentTime.Set (5, 30, 0);
endTime.Set (18, 30, 0);
while (! done)
{ . . .

    currentTime.Increment ();
    if (currentTime.Equal (endTime))
        done  =  true;
};
}
```

class type Declaration

- **The class declaration creates a data type and names the members of the class**
- **It **does not allocate memory** for any variables of that type!**
- **Client code still needs to declare class variables**

Remember ...

- Two kinds of class members:

1) **data members** and 2) **function members**

- Class members are private by default

Remember as well . . .

- **Data members are generally private**
- **Function members are generally declared public**
- **Private class members can be accessed only by the class member functions (and friend functions), not by client code**

Aggregate class Operations

- **Built-in operations valid on class objects are:**
 - Member selection using dot (.) operator ,
 - Assignment to another class variable using (=),
 - Pass to a function as argument (by value or by reference),

Aggregate class Operations

- **Built-in operations valid on class objects also:**
 - **Return as value of a function**

Other operations can be defined as class member functions

Separate Specification and Implementation

```
// Specification file "time.h"  
// Specifies the data and function members  
class Time  
{  
    public:  
        . . .  
  
    private:  
        . . .  
};
```

Separate Specification and Implementation

```
// Implementation file "time.cpp"  
// Implements the Time member functions  
{  
    ...  
}
```

Implementation File for Time

```
// Implementation file "time.cpp"  
// Implements the Time member functions.  
// Also must appear in client code:  
#include "time.h"  
#include <iostream>  
. . .  
bool Time::Equal(/* in */ Time otherTime) const
```

Implementation File for Time

```
// Postcondition: Return value == true,  
//               if this time equals otherTime,  
//               otherwise == false  
{  
    return ((hrs == otherTime.hrs)  
            && (mins == otherTime.mins)  
            && (secs == otherTime.secs));  
}
```

. . .

Should be familiar ...

- The member selection operator (.) selects either data members or function members
- Header files **iostream** and **fstream** declare the **istream**, **ostream**, **ifstream**, and **ofstream** I/O classes

Should be familiar...

- Both **cin** and **cout** are class objects and **get** and **ignore** are function members:

```
cin.get (someChar);  
cin.ignore (100, '\n');
```

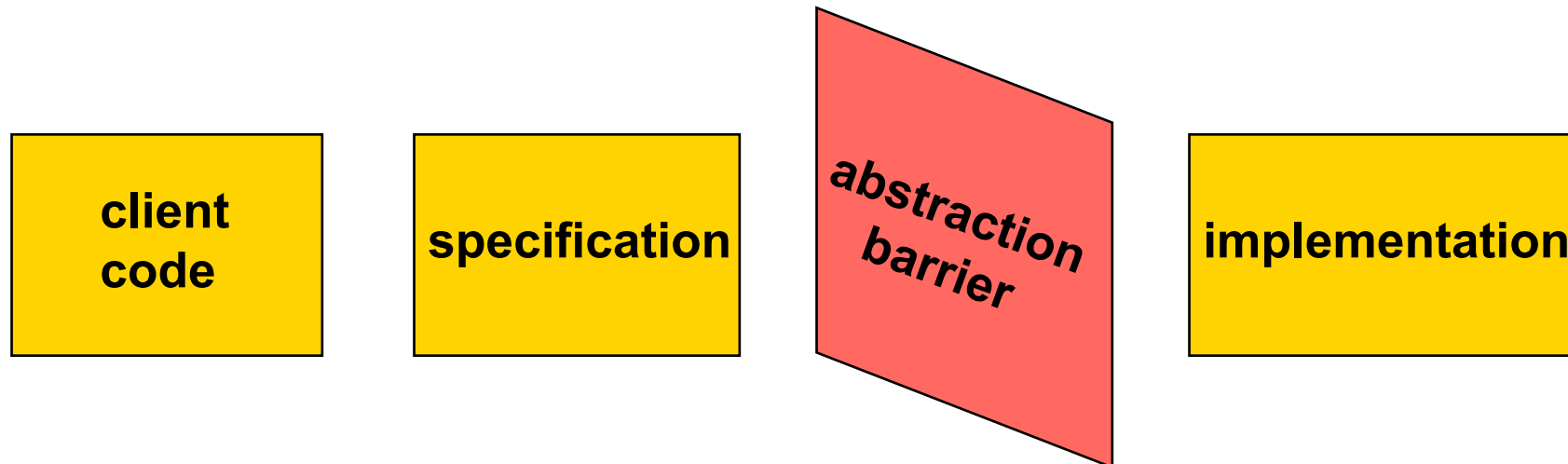
- These statements declare myInfile as an instance of class ifstream and invoke function member open :

```
ifstream myInfile;  
myInfile.open ("mydata.dat");
```

Information Hiding

Information hiding - Class implementation details are hidden from the client's view

Public functions of a class provide the **interface** between the client code and the class objects



Selection and Resolution

- **C++ programs typically use several class types**
- **Different classes can have member functions with the same identifier, like Write()**

Selection and Resolution

- Member **selection operator** is used to determine the object to whom member function `Write()` is applied

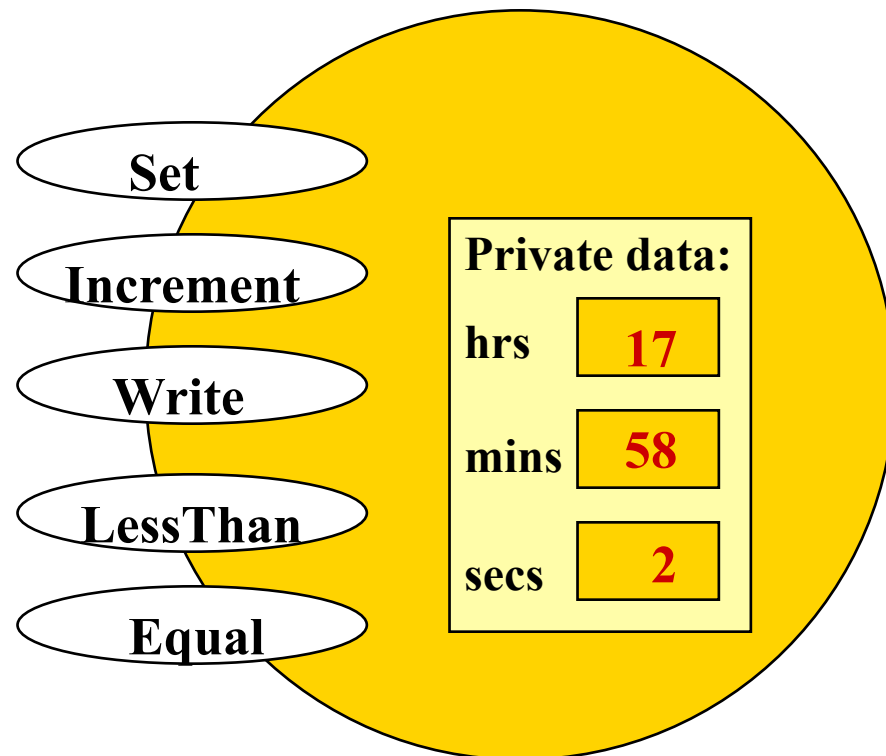
```
currentTime.Write(); // Class Time  
numberZ.Write();    // Class ComplexNumber
```

- In the implementation file, the **scope resolution operator** is used in the heading before the function member's name to specify its class

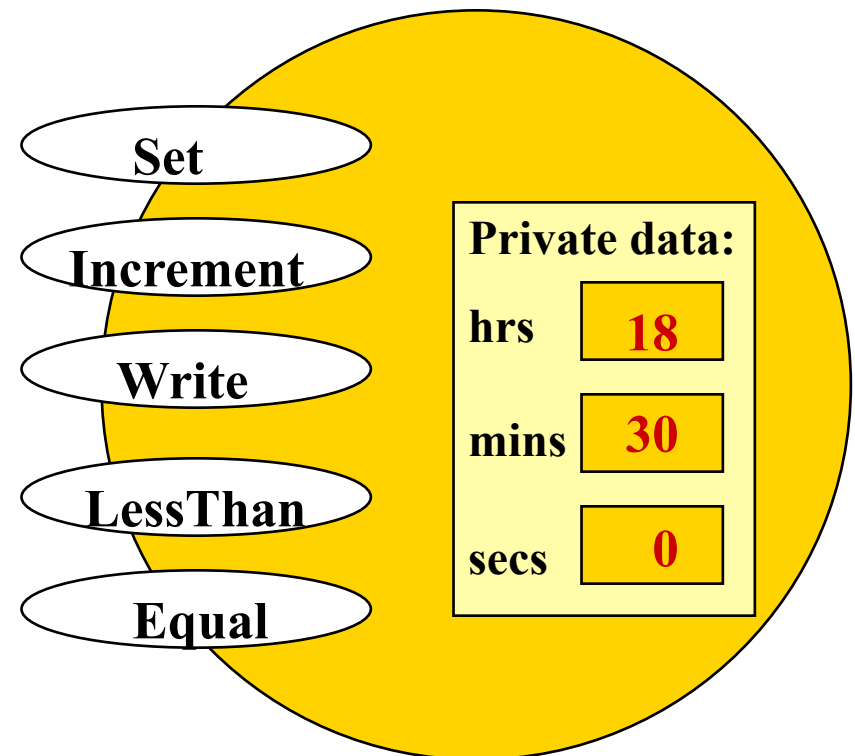
```
void Time::Write () const  
{  
    . . .  
}
```

Time Class Instance Diagrams

currentTime



endTime



Use of `const` with Member Functions

- When a member function does not modify the private data members:
- Use `const` in both the function prototype (in specification file) and the heading of the function definition (in implementation file)

Example Using `const` with a Member Function

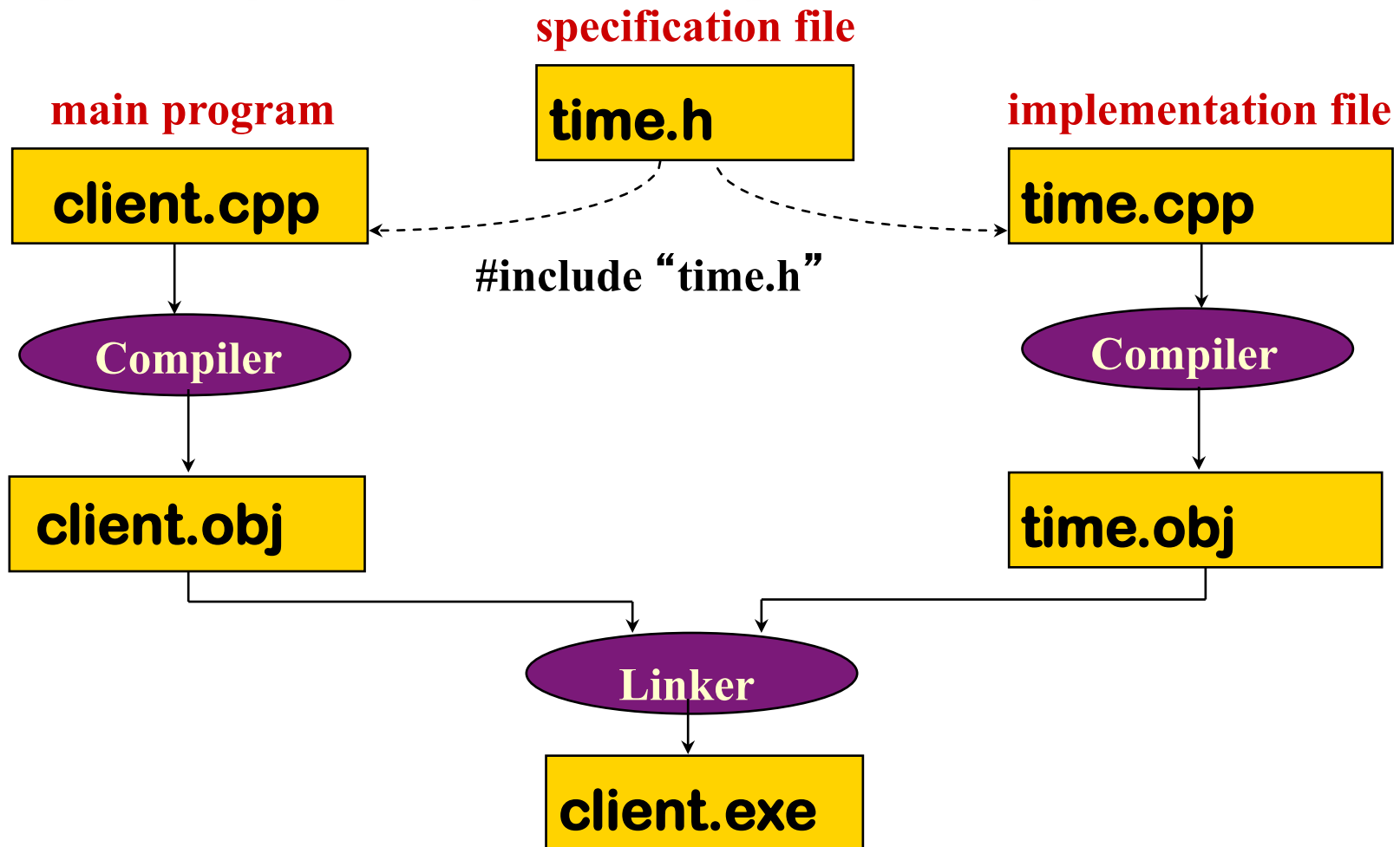
```
void Time::Write ()    const
```

```
// Postcondition: Time has been output in form  
//  HH:MM:SS
```


Example Using `const` with a Member Function, cont...

```
{  
    if (hrs < 10)  
        cout << '0';  
    cout << hrs << ':';  
    if (mins < 10)  
        cout << '0';  
    cout << mins << ':';  
    if (secs < 10)  
        cout << '0';  
    cout << secs;  
}
```

Separate Compilation and Linking of Files



Avoiding Multiple Inclusion of Header Files

- Often several program files use the same header file containing typedef statements, constants, or class type declarations
- But, it is a **compile-time error to define the same identifier twice within the same namespace**

Avoiding Multiple Inclusion of Header Files

- This preprocessor directive syntax is used to avoid the compilation error that would otherwise occur from multiple uses of `#include` for the same header file

```
#ifndef Preprocessor_Identifier  
#define Preprocessor_Identifier  
.  
.  
.  
#endif
```

Example Using Preprocessor Directive

#ifndef

For compilation the class declaration in File time.h will be included only once

```
// time .h
// Specification file

#ifndef TIME_H
#define TIME_H

class Time
{
    public:
        . . .

    private:
        . . .
};
#endif
```

```
// time .cpp
// IMPLEMENTATION FILE

#include "time.h"

. . .
```

```
// client.cpp
// Appointment program

#include "time.h"

int main (void)
{
    . . .
}
```

Class Constructors

- A **class constructor** – a member function whose purpose is to initialize the private data members of a class object
- The name of a constructor is always the name of the class, and there is no return type for the constructor

Class Constructors

- A class may have several constructors with different parameter lists
- A constructor with no parameters is the **default** constructor
- A constructor is **implicitly invoked** when a class object is declared
- If there are parameters, their values are listed in parentheses in the declaration

Specification of Time Class Constructors

```
class Time      // Time.h
{
public :    // 7 function members
    void Set(int hours, int minutes, int seconds);
    void Increment();
    void Write() const;
    bool Equal(Time otherTime) const;
    bool LessThan(Time otherTime) const;
```


Specification of Time Class Constructors

```
// Parameterized constructor  
Time (int initHrs, int initMins, int initSecs);  
// Default constructor  
Time();  
private :           // 3 data members  
    int hrs;  
    int mins;  
    int secs;  
};
```

Implementation of Time Default Constructor

```
Time::Time ()  
// Default Constructor  
// Postcondition:  
// hrs == 0 && mins == 0 && secs == 0  
{  
    hrs = 0;  
    mins = 0;  
    secs = 0;  
}
```

Parameterized Constructor

```
Time::Time( /* in */    int    initHrs,  
            /* in */    int    initMins,  
            /* in */    int    initSecs)  
  
//  Constructor  
//  Precondition:  
//      0 <= initHrs <= 23 && 0 <= initMins <= 59  
//      0 <= initSecs <= 59  
//  Postcondition:  
//      hrs == initHrs && mins == initMins  
//      && secs == initSecs
```

Parameterized Constructor

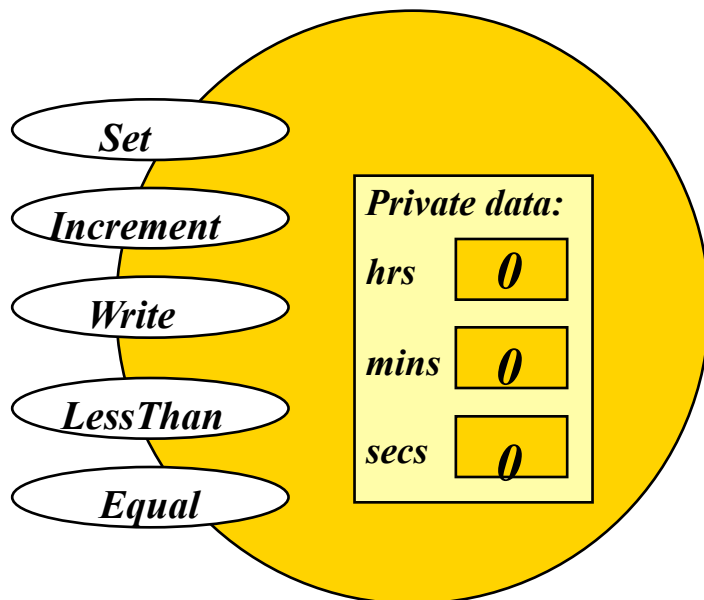
```
{  
    hrs    =    initHrs;  
    mins   =    initMins;  
    secs   =    initSecs;  
}
```

Automatic invocation of constructors occurs

```
Time departureTime; // Default constructor invoked
```

```
Time movieTime (19, 30, 0); // Parameterized constructor
```

departureTime



movieTime

