Introduction to Classes

Department of Software Engineering

National University of Computer & Emerging Sciences, Islamabad Campus

Why Objects?

At the end of the day...

computers just manipulate 0's and 1's

Figure by MIT OpenCourseWare.

*But **binary is hard (for humans)** to work with…*

Towards a higher level of abstraction

?

declarative languages (Haskell, ML, Prolog...)

OO languages (C++, Java, Python...)

procedural languages (C, Fortran, COBOL...)

assembly languages

binary code

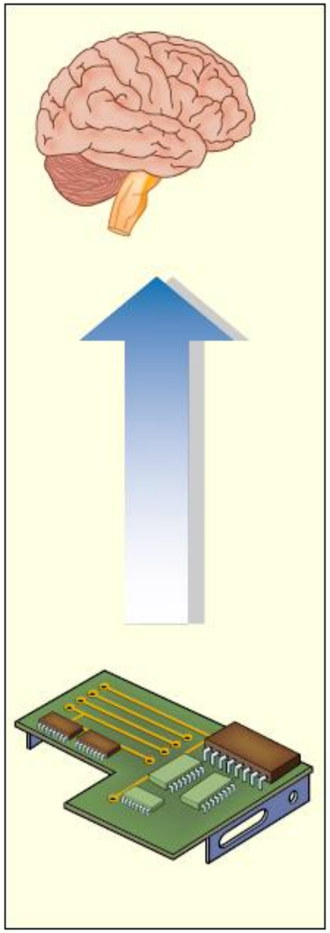
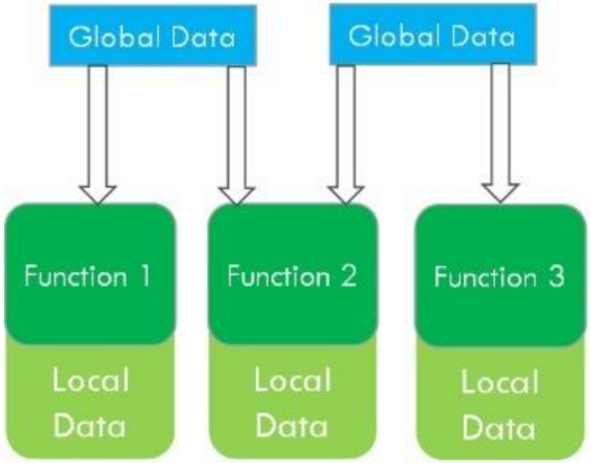


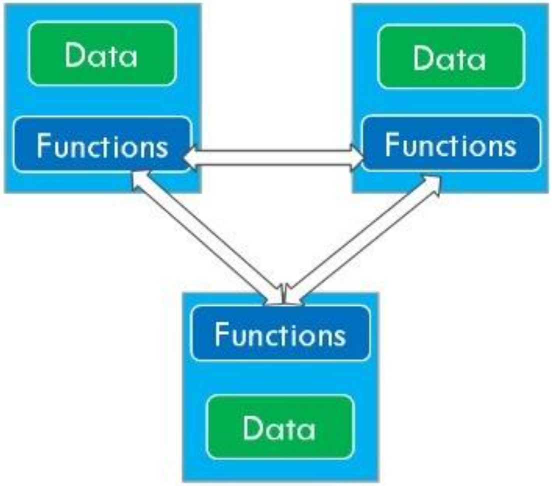
Figure by MIT OpenCourseWare.

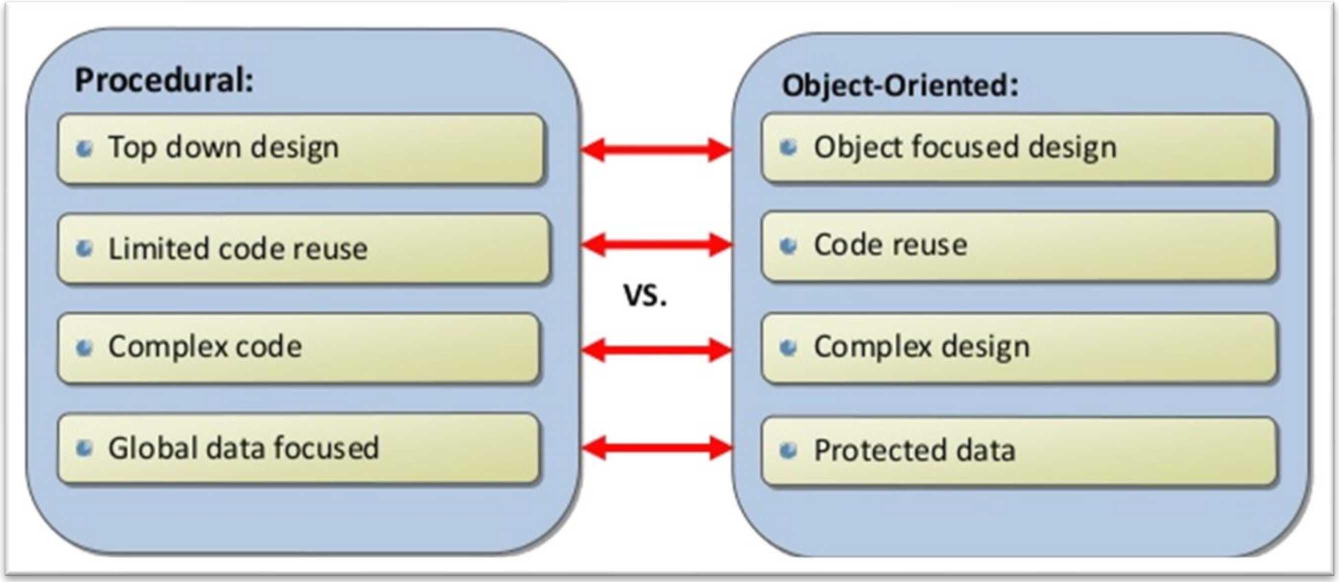
Procedural VS. Object-Oriented Programming

Procedural Oriented Programming



Object Oriented Programming





Procedural:		Object-Oriented:
Top down design	↔	Object focused design
Limited code reuse	↔	Code reuse
Complex code	↔ VS.	Complex design
Global data focused	↔	Protected data

Object-oriented Programming (OOP)

- **Object-oriented programming** approach **organizes programs** in a way that ***mirrors the real world***, in which all **objects** are associated with **both** **attributes** and **behaviors**

- **Object-oriented programming** involves **thinking in terms of objects**

- An **OOP program** can be **viewed** as a **collection of cooperating objects**

Classes

A class is like a cookie cutter; it defines the shape of objects

Objects are like cookies; they are instances of the class

Photograph courtesy of Guillaume Brialon on Flickr.

Classes in OOP

- **Classes are constructs/templates that define objects of the same type.**

- A **class** uses **Variables (data fields) to define state**

- A **class** uses **Functions to define behaviors**.

- Additionally, ***a class provides a*** **special type of function, known as constructors**:
  - **Invoked** to **construct objects** from the **class**.

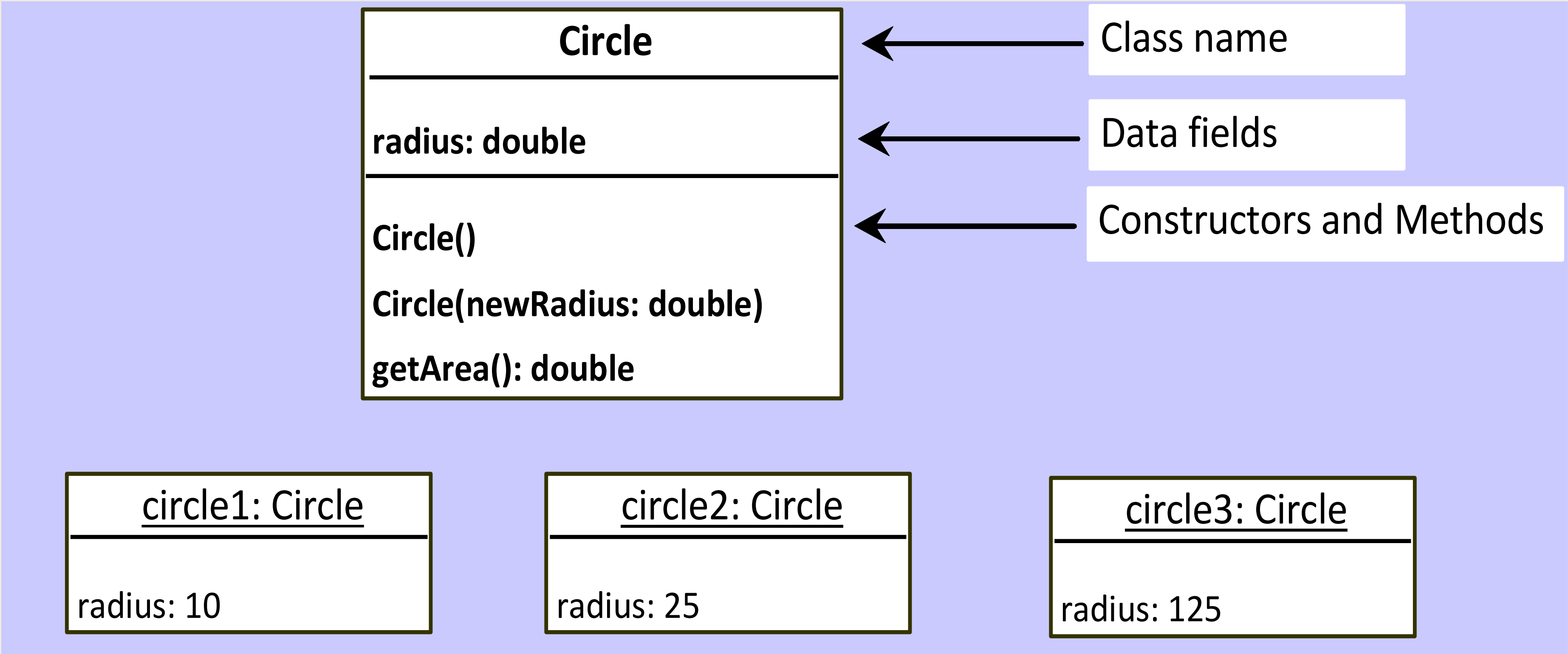
Objects in OOP

- **An object has a unique identity, state, and behaviors.**

- The **state** of an **object** consists of ***a set of data fields*** (also known as **properties**) **with their current values**.

- The **behavior** of an **object** is **defined** by **a set of functions**.

UML Diagram for Class and Object



Circle

radius: double

Circle()

Circle(newRadius: double)

getArea(): double

Class name

Data fields

Constructors and Methods

circle1: Circle

radius: 10

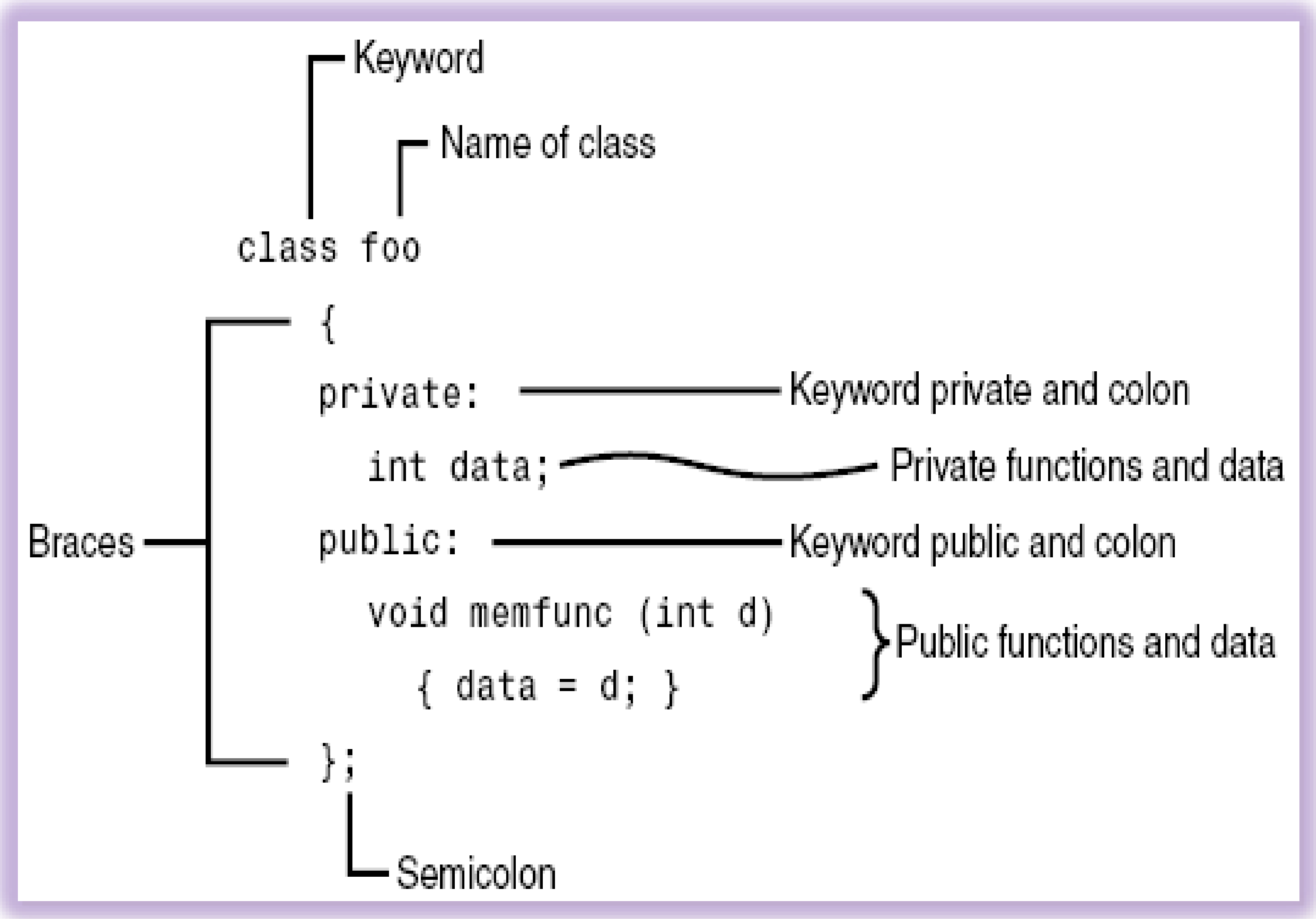
circle2: Circle

radius: 25

circle3: Circle

radius: 125

Class in C++ - Example



```
class foo
{
private:
   int data;
public:
   void memfunc (int d)
     { data = d; }
};
```

Keyword
Name of class
Braces
Keyword private and colon
Private functions and data
Keyword public and colon
Public functions and data
Semicolon

Class in C++ - Example

```
class Circle
{
public:
   // The radius of this circle
   double radius;

   // Construct a circle object
   Circle()
   {
      radius = 1;
   }

   // Construct a circle object
   Circle(double newRadius)
   {
      radius = newRadius;
   }

   // Return the area of this circle
   double getArea()
   {
      return radius * radius * 3.14159;
   }
};
```

Data field

Constructors

Function

Note:

the special syntax for constructor (no return type!)

Class is a Type

- You can use **primitive data types** to **define variables**.

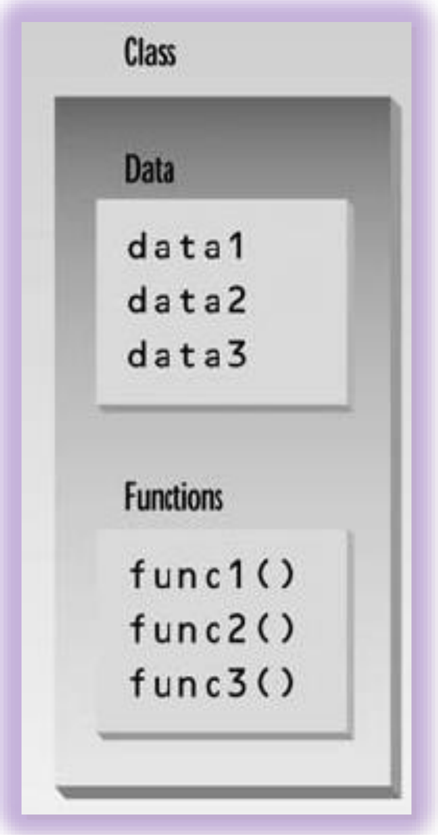
- You can also use **class names** to **declare object names**.

- ***In this sense, a class is an abstract data-type or user-defined data type.***

Class Data Members and Member Functions

- The **data items** within a **class** are called ***data members*** or ***data fields*** or ***instance variables***

- ***Member functions*** are **functions** that are **included** within a **class**. Also known as ***instance functions***.



Object Creation - Instantiation

- **In C++,** you can **assign a name** when **creating an object**.

- A **constructor is invoked** when an **object is created**.

- The syntax to **create** an **object** using the ***no-arg constructor*** is:

  ClassName objectName;

- Defining **objects** in this way means ***creating them***. This is also **called *instantiating them***.

Object Member Access Operator

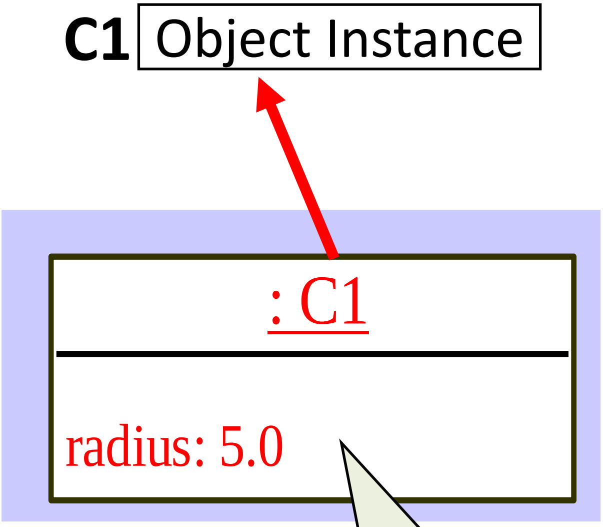
- After **object creation**, its **data** and **functions** can be **accessed** (invoked) using:
  - The **.** **operator**, also known as the ***object member access operator***.

- ***objectName.dataField*** references a **data field** in the object

- ***objectName.function( )*** **invokes** a **function on the object**

A Simple Program – *Object* Creation

```
class  Circle
{
     private:
          double radius;

     public:
     Circle()
     {       radius = 5.0;    }
     double   getArea( )
     {  return radius *   radius   *   3.14159; }
};
```



Allocate memory for radius

```
void main()
{
      Circle     C1;
     //C1.radius = 10;      can’t access private member outside the class
    cout<<“Area of circle = “<<C1.getArea( );
}
```

Local Classes

- **Local classes**: A local **class** is declared **within a function definition**.

- Scope is within the function
- Functions of local class must be inline
- A **local class cannot have static data members but can have static function.**

- Methods of local class can only access static members of the enclosing function.

Inline/Out-of-Line Member Functions

- **Inline functions:**
  - are **defined** **within the body of the class** definition.

- **Out-of-line functions**:
  - are **declared within the body of the class** definition and **defined outside**.

Inline/Out-of-Line Member Functions

- If a **member function** is **defined outside the class**
  - **Scope resolution operator** (::) and **class name** are needed
  - Defining a **function outside a class** **does not change** it being `public` or `private`

- **Binary scope resolution operator** (: :)
  - **Combines** the **class name** with the **member function name**
  - **Different classes** can have **member functions** with the **same name**

```
returnType ClassName::MemberFunctionName( ){
                …
           }
```

Member Functions
Separating Declaration from Implementation

```
class  Circle
{
    private:
        double radius;

    public:
    Circle(double   radius)
    {      this->radius = radius;    }
    double   getArea( ); // Not implemented yet
};
```

Class must define a no-argument constructor too….

```
double   Circle::getArea()
{  return  this->radius *  radius   *  3.14159; }
```

```
void main()
{
        Circle     C1(99.0);
        cout<<“Area of circle = “<<C1.getArea();
}
```

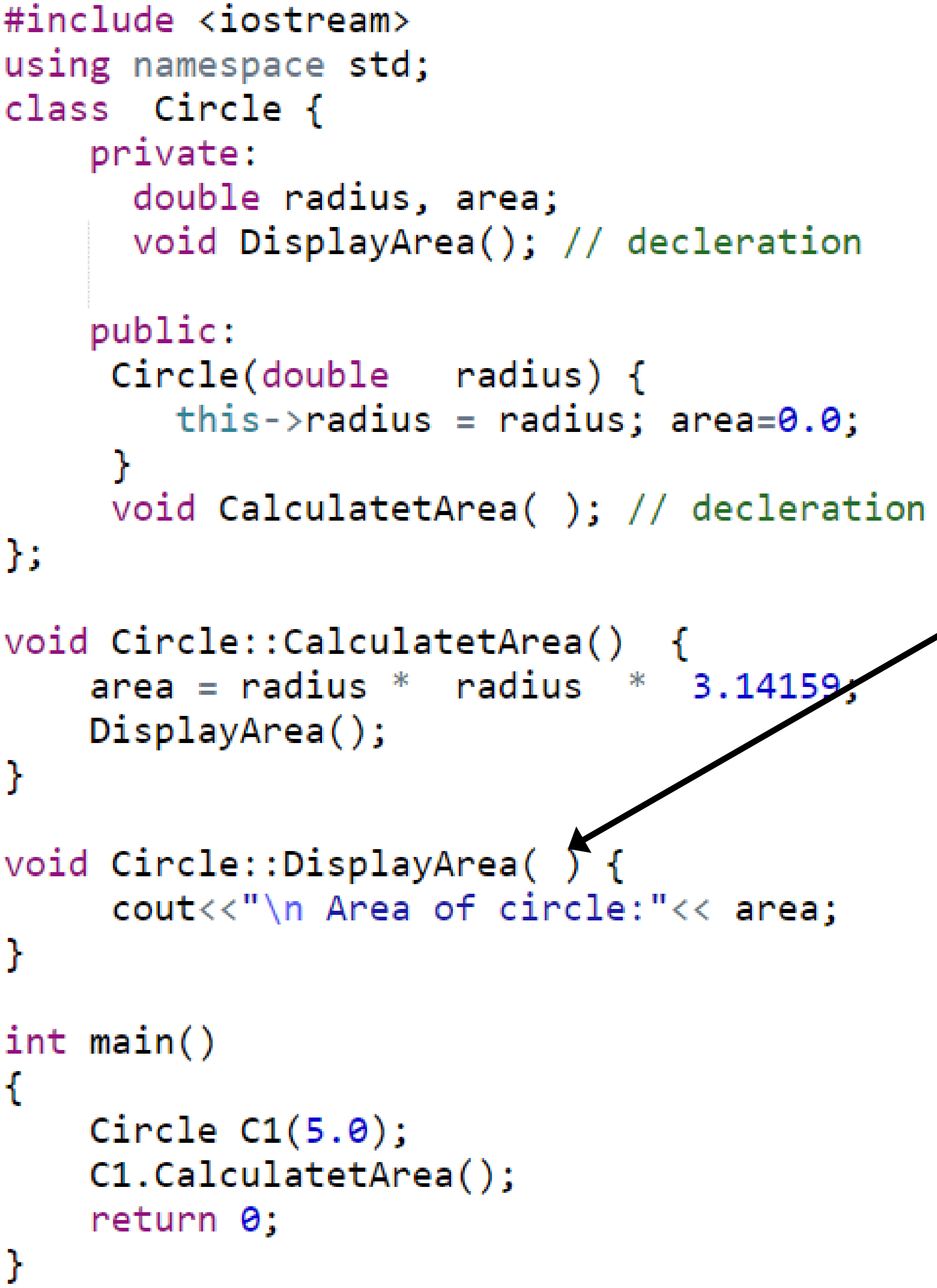
```cpp
// Fig. 6.3: fig06_03.cpp
// Time class.
#include <iostream>

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

// Time abstract data type (ADT) definition
class Time {
public:
   Time();                         // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();           // print military time format
   void printStandard();           // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military 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;
}
```

Note the `::` preceding the function names.

```cpp

// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print Time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}

```

Private Member Functions

- **Private Member Functions:**
  - Only **accessible** (callable) from **member functions** of the **class**
  - **No direct access possible** (with **object instance of the class**)
  - **Can be: inline / out-of-line**



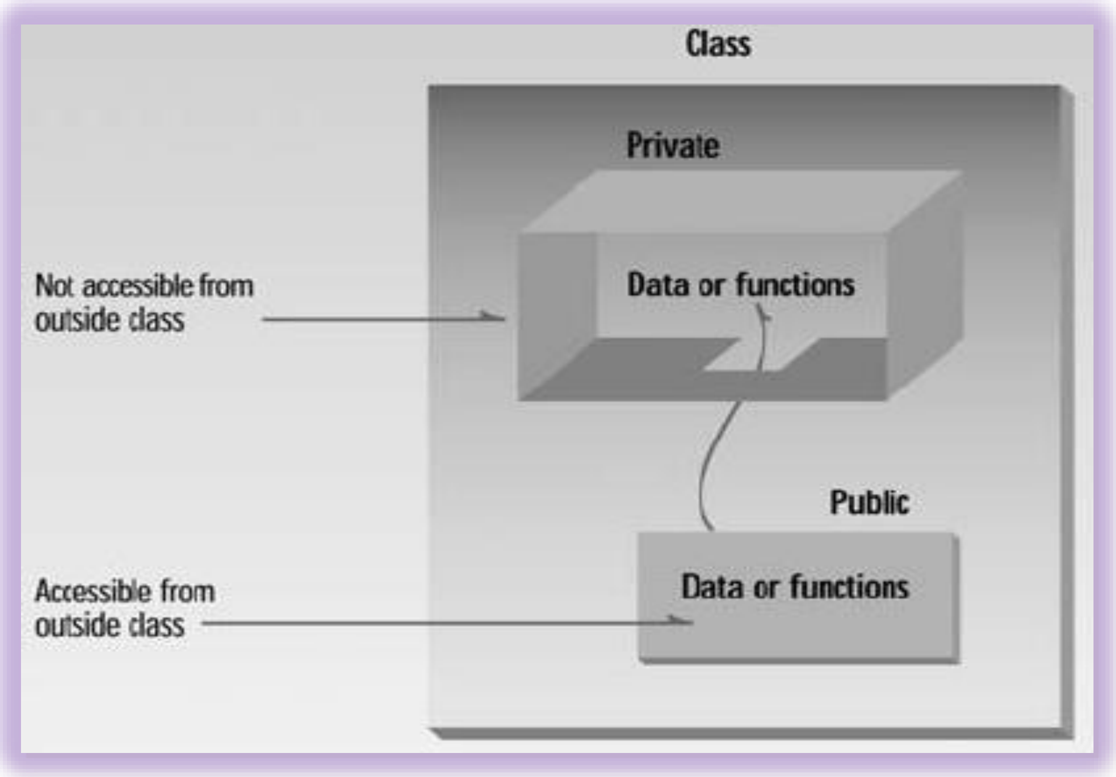
```cpp
#include <iostream>
using namespace std;
class  Circle {
    private:
      double radius, area;
      void DisplayArea(); // decleration

    public:
     Circle(double   radius) {
        this->radius = radius; area=0.0;
     }
     void CalculatetArea( ); // decleration
};

void Circle::CalculatetArea()  {
    area = radius *  radius  *  3.14159;
    DisplayArea();
}

void Circle::DisplayArea( ) {
     cout<<"\n Area of circle:"<< area;
}

int main()
{
    Circle C1(5.0);
    C1.CalculatetArea();
    return 0;
}
```

Private Member Functions

(out-of-line)

Access Modifiers/Specifier

- Access modifiers are used to set access levels for *variables, methods, and constructors*
- **private**, **public**, and **protected**
- In C++, ***default accessibility is private***


Class
Private
Data or functions
Not accessible from outside class
Public
Data or functions
Accessible from outside class

- The default constructor (provided by compiler) is always public.
- Programmer can specify a constructor to be private (no use) or public

- `public`
  - **Presents clients** with a view of the **services the class provides** (i.e., **interface**)
  - **Data** and **member functions** are **accessible** **(outside class)**
- `private`
  - **Default access mode**
  - **Data** **only accessible** to **member functions** and **`friends`**
  - **`private`** **members only accessible through** the **`public`** class interface using **`public`** **member functions**

Data Hiding - Data Field Encapsulation

- A **key feature** of **OOP** is **data hiding**
  - **data is concealed within a class so that it cannot be accessed mistakenly by functions outside the class**.

- **To prevent direct modification of class attributes** (outside the class), the **primary mechanism for hiding data** is to **put** it in a **class** and **make it private** using ***private*** keyword. **This is also known as *data field encapsulation*.**

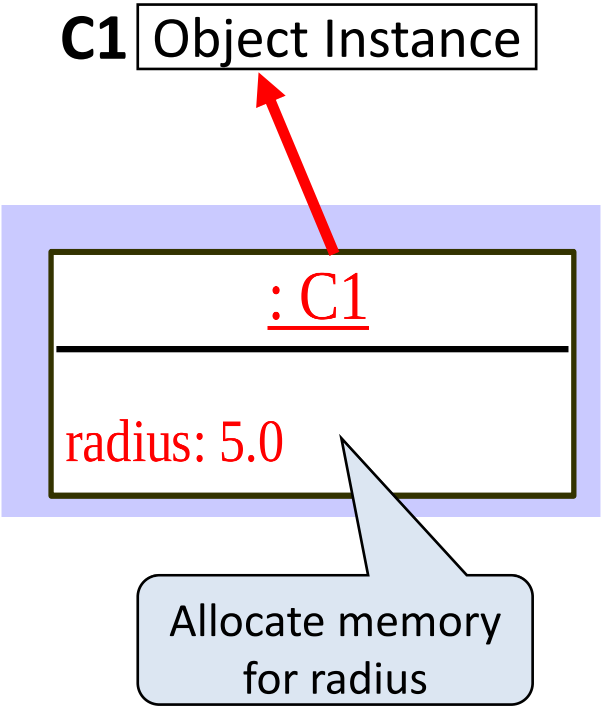
Hidden from Whom?

- ***Data hiding means hiding data from parts of the program that don’t need to access it.*** More specifically, **one class’s data** is **hidden** from **other classes**.

- **Data hiding** is **designed** to **protect well-intentioned programmers** from **mistakes**.

A Simple Program – **Accessing** ***Member Function***

```
class  Circle
{
     private:
          double radius;

     public:
     Circle()
     {      radius = 5.0;    }
     double   getArea()
     {  return radius *   radius   *   3.14159; }
};
```



C1 Object Instance

: C1

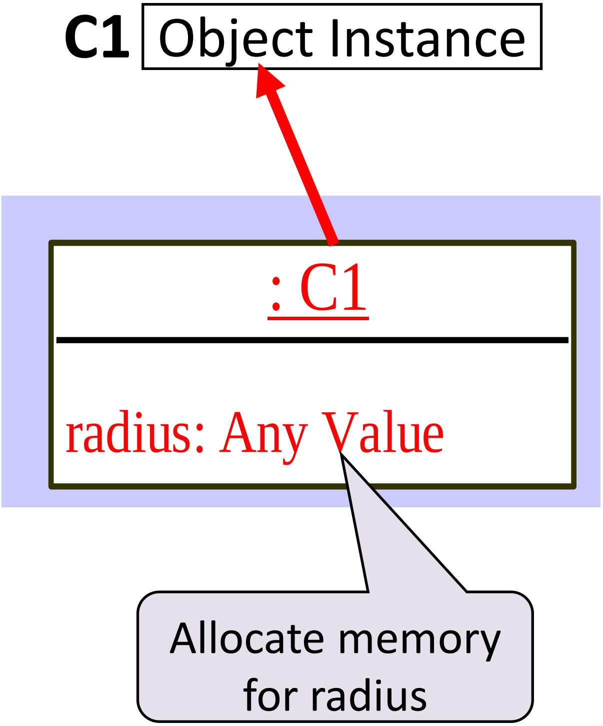
radius: 5.0

Allocate memory for radius

```
void main()
{
      Circle     C1;
     //C1.radius = 10;    can’t access private member outside the class
     cout<<“Area of circle = “<<C1.getArea();
}
```

A Simple Program – *Default Constructor*

```
class  Circle
{
     private:
          double radius;
     public:
      //Default Constructor
      Circle()
      {           }
      // No Constructor Here
     double   getArea()
     {  return radius *   radius   *   3.14159; }
};
```



```
void main()
{
    Circle     C1;
     //C1.radius = 10;      can’t access private member outside the class
    cout<<“Area of circle = “<<C1.getArea();
}
```

Object Construction with Arguments

- The syntax to declare **an object using a constructor with arguments** is:

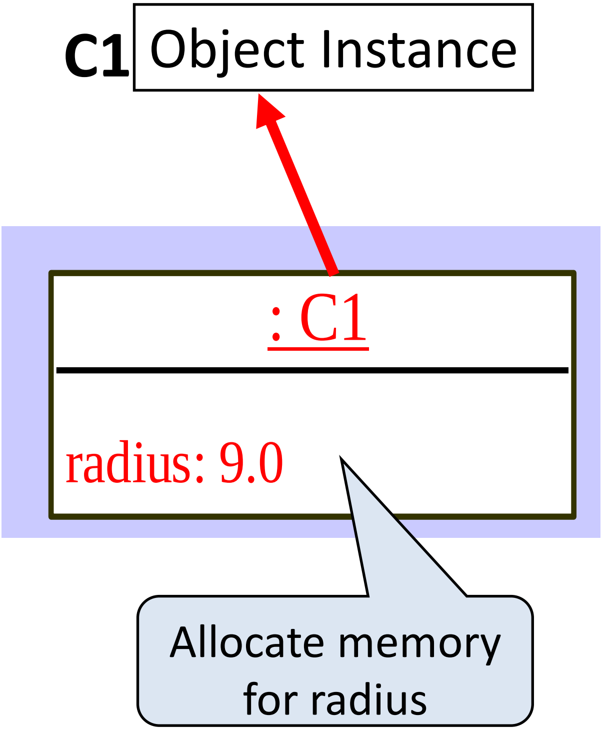
ClassName objectName(arguments);

- For example, the **following declaration creates an object** named ***circle1*** by invoking the *Circle* class's **constructor** with a specified radius ***5.5***.

```
Circle   circle1(5.5);
```

A Simple Program – *Constructor with Arguments*

```
class  Circle
{
     private:
          double radius;
public:
     Circle( ) {}
     Circle(double rad)
     {       radius = rad;    }
     double   getArea()
     {  return radius *   radius   *   3.14159; }
};
```



C1 Object Instance

: C1

radius: 9.0

Allocate memory for radius

```
void main()
{
      Circle     C1(9.0);
      //C1.radius = 10;      can’t access private member outside the class
     cout<<“Area of circle = “<<C1.getArea();
}
```

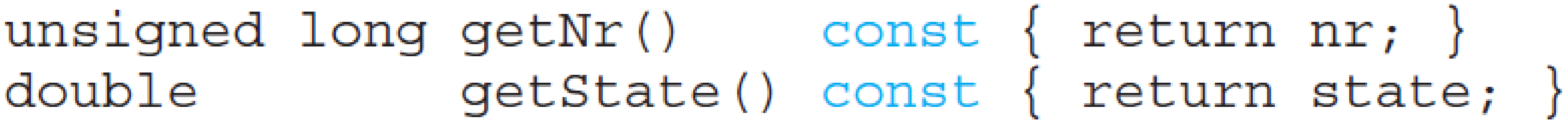
Output of the following Program?

```
class  Circle
{
     private:
          double radius;

     public:
     //Circle( ) {  }

     Circle(double rad)
     {       radius = rad;     }
     double   getArea()
     {  return radius *   radius   *   3.14159; }
};
```

```
void main()
{
       Circle     C1;
       cout<<“Area of circle = “<<C1.getArea();
}
```

const Member Functions

- **const Member Functions: Read-only functions cannot modify object’s data members**



```
unsigned long getNr()     const { return nr; }
double        getState() const { return state; }
```

Constant Functions

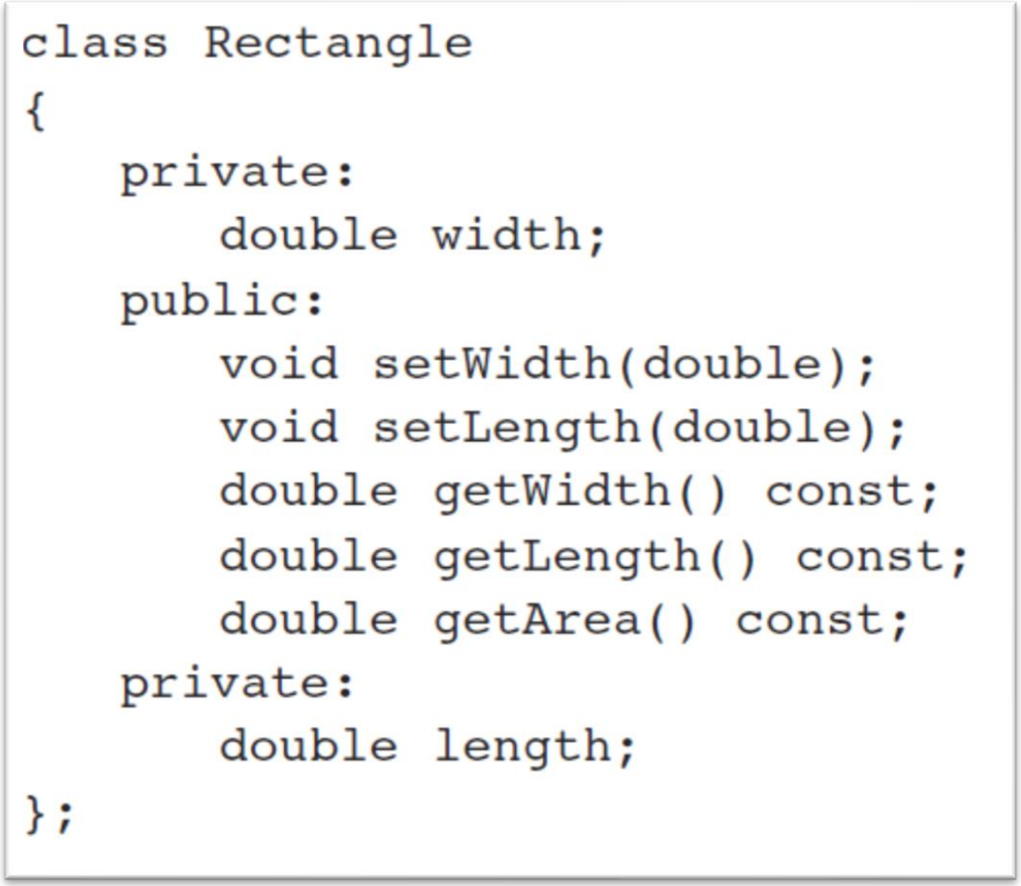
```
class  Circle
{
    private:
        double radius;
    public:
    Circle ()
    {   radius = 1;      }
    Circle(double rad)
    {      radius = rad;    }
    double   getArea() const
    {  return radius *  radius  *  3.14159; }
};
```

```
void main()
{
      Circle      C2(8.0);
      Circle     C1;
      cout<<“Area of circle = “<<C1.getArea();
}
```

const member function cannot update/change object’s data

Accessors and Mutators(Getters & Setters)

- **Accessors: member function** only **reads/gets value** from a **class's member variable** but **does not change it.**
- **Mutators: member function** that **stores a value** in **member variable**



```
class Rectangle
{
    private:
        double width;
    public:
        void setWidth(double);
        void setLength(double);
        double getWidth() const;
        double getLength() const;
        double getArea() const;
    private:
        double length;
};
```

const Objects

- const Object: Read-only objects
  - Object data members can only be read, NO write/update of data member allowed
  - Requires all member functions be const (except constructors and destructors)
  - const object must be initialized (using constructors) at the time of object creation

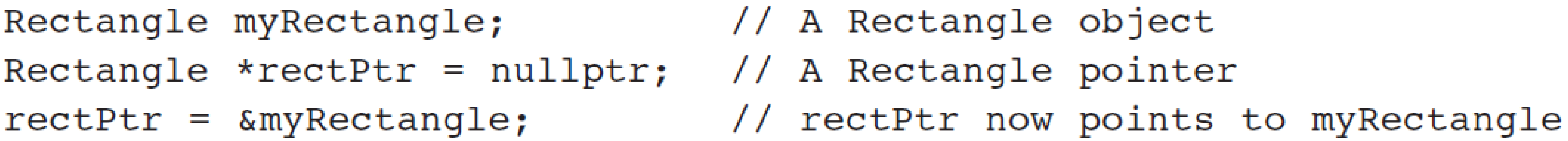
```
const Account inv("YMCA, FL", 5555, 5000.0);
```

const Objects

- **const property of an object goes into effect after the constructor finishes executing and ends before the class's destructor executes**
  - So the constructor and destructor *can modify the object*

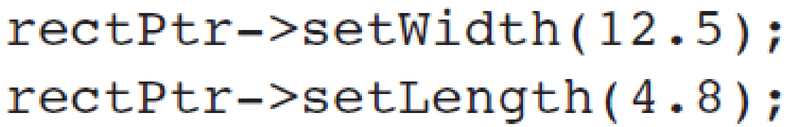
Pointers to Objects

- You can also **define pointers** to **class objects**



```
Rectangle myRectangle;          // A Rectangle object
Rectangle *rectPtr = nullptr;   // A Rectangle pointer
rectPtr = &myRectangle;         // rectPtr now points to myRectangle
```

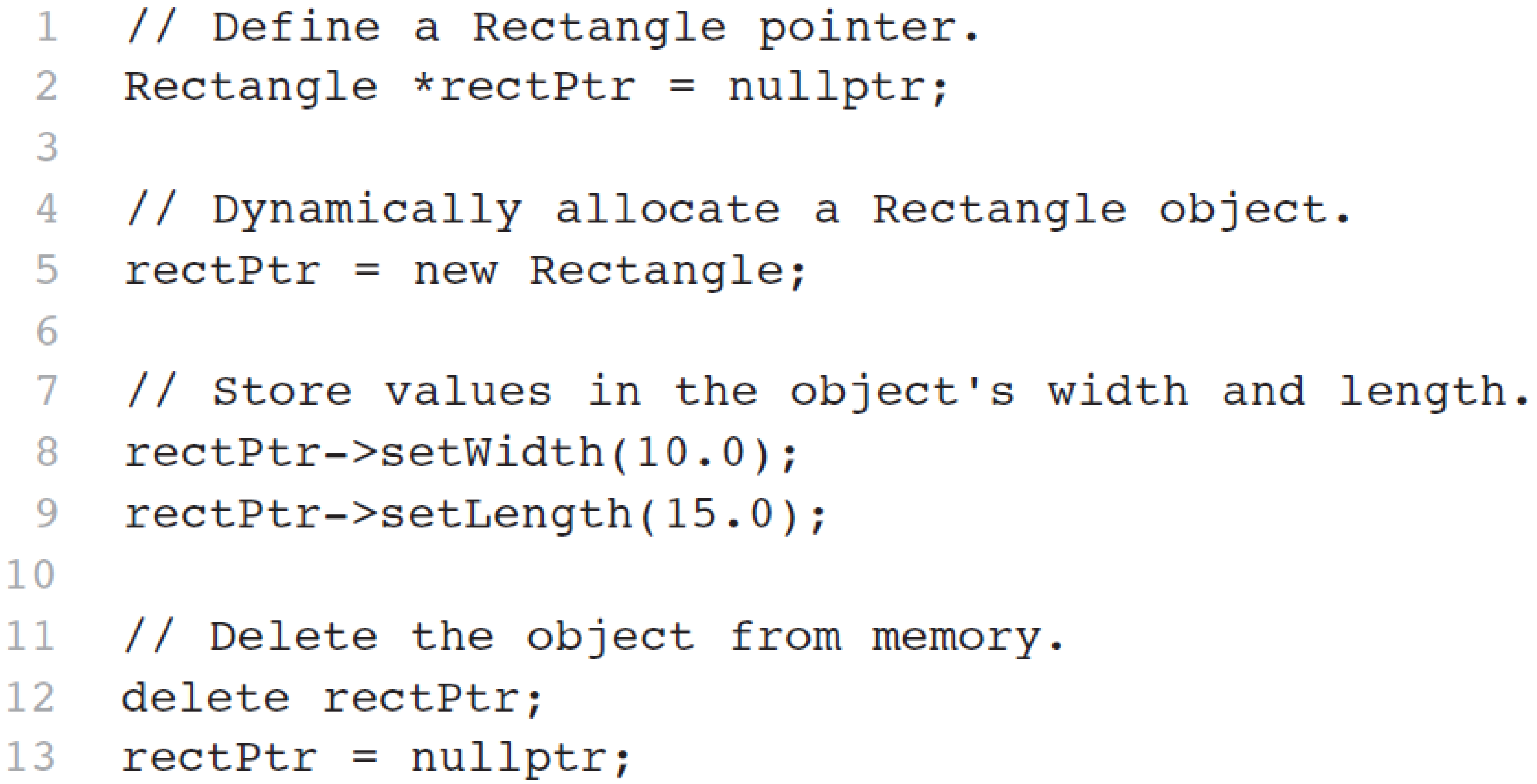
- You **can use * and . operators OR -> to access members:**



```
rectPtr->setWidth(12.5);
rectPtr->setLength(4.8);
```

Pointers to Objects

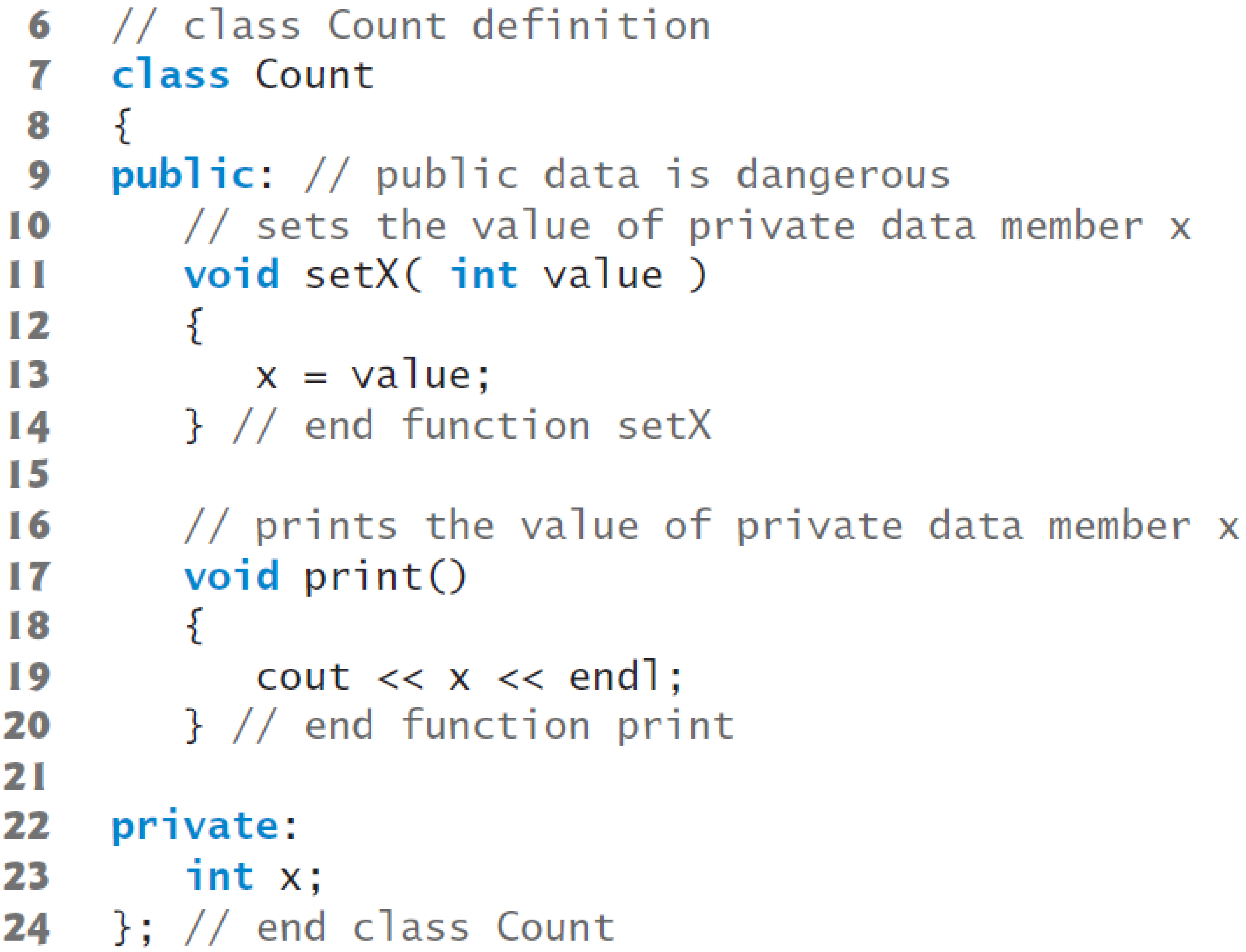
- **Dynamic Object Creation**



```cpp
// Define a Rectangle pointer.
Rectangle *rectPtr = nullptr;

// Dynamically allocate a Rectangle object.
rectPtr = new Rectangle;

// Store values in the object's width and length.
rectPtr->setWidth(10.0);
rectPtr->setLength(15.0);

// Delete the object from memory.
delete rectPtr;
rectPtr = nullptr;
```

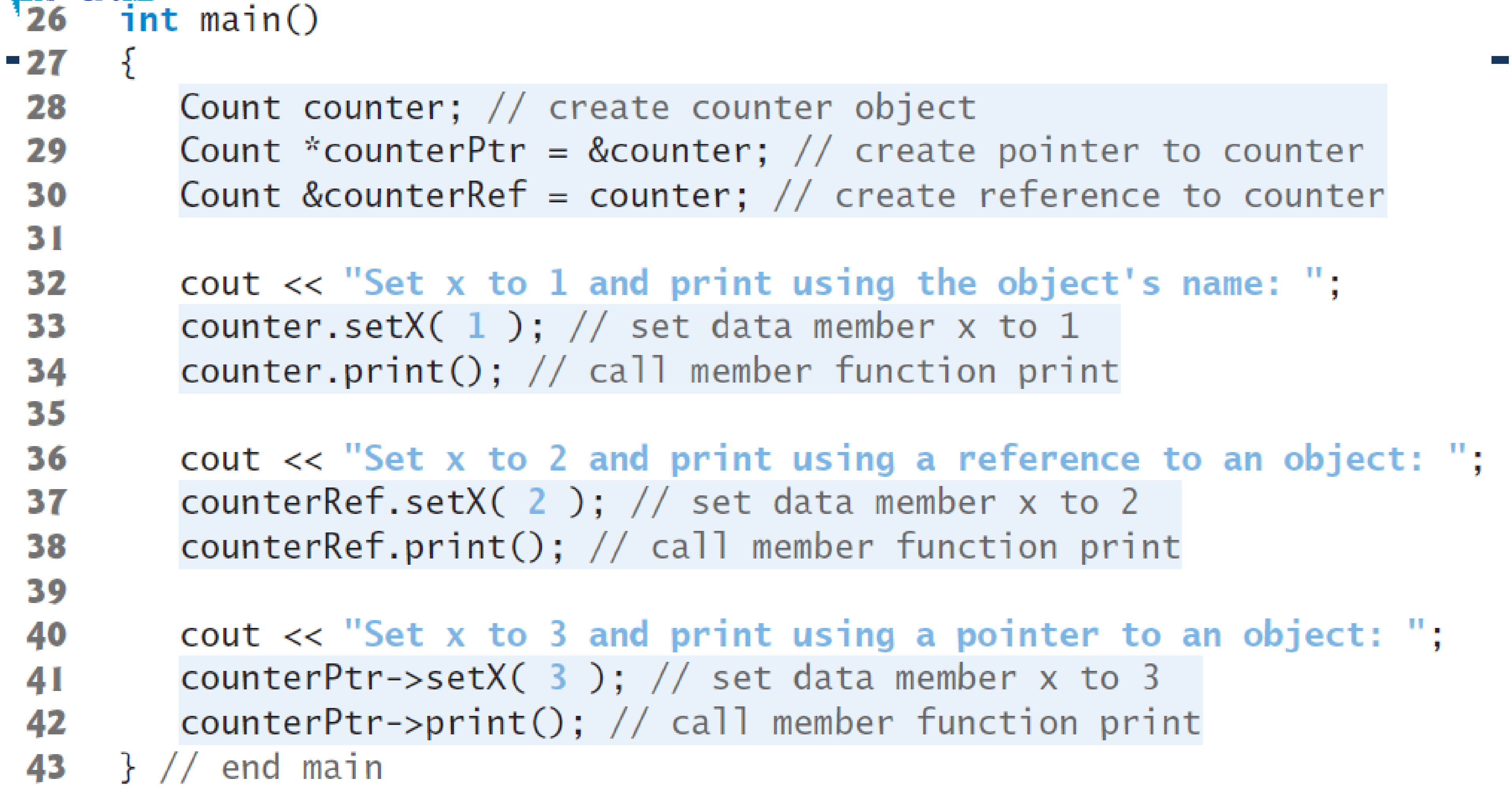
Reference to Objects

- **Reference** is an **alias** to an **existing object**

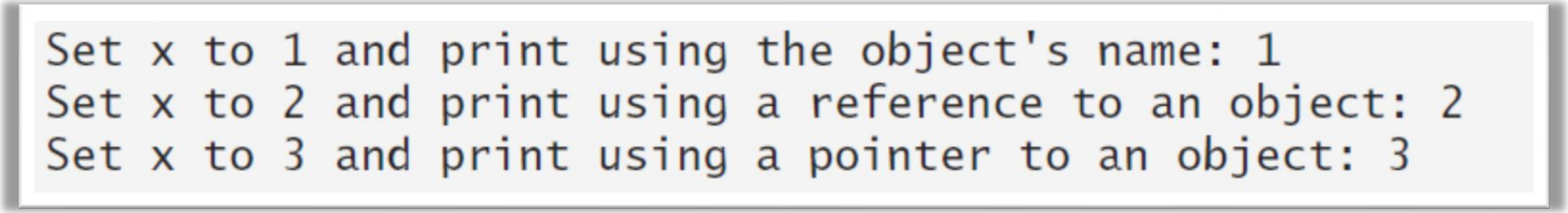


```
// class Count definition
class Count
{
public: // public data is dangerous
    // sets the value of private data member x
    void setX( int value )
    {
        x = value;
    } // end function setX

    // prints the value of private data member x
    void print()
    {
        cout << x << endl;
    } // end function print

private:
    int x;
}; // end class Count
```

Reference to Objects



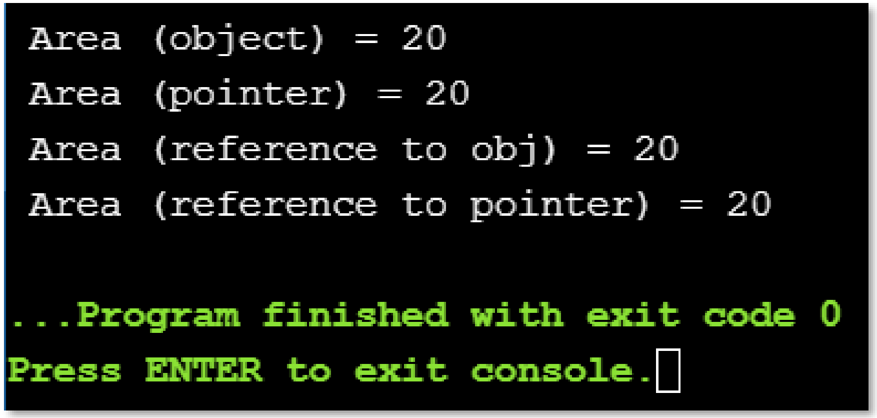
```
int main()
{
    Count counter; // create counter object
    Count *counterPtr = &counter; // create pointer to counter
    Count &counterRef = counter; // create reference to counter

    cout << "Set x to 1 and print using the object's name: ";
    counter.setX( 1 ); // set data member x to 1
    counter.print(); // call member function print

    cout << "Set x to 2 and print using a reference to an object: ";
    counterRef.setX( 2 ); // set data member x to 2
    counterRef.print(); // call member function print

    cout << "Set x to 3 and print using a pointer to an object: ";
    counterPtr->setX( 3 ); // set data member x to 3
    counterPtr->print(); // call member function print
} // end main
```



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

Reference and Pointers to Objects

```
class Rectangle
{
   private:
    int w;  int h;

   public:
     Rectangle () {}
    void SetWidth(int ww) { w=ww; }
    void SetHeight(int hh) { h=hh;}
    int getArea() { return w*h; }
};

int main() {
  Rectangle r1;
  Rectangle *ptr = &r1;
  Rectangle &ref = r1;
  Rectangle* &ref2 = ptr;

  r1.SetHeight(5);
  r1.SetWidth(4);
  cout<<"\n Area (object) = "<<r1.getArea();
  cout<<"\n Area (pointer) = "<<ptr->getArea();
  cout<<"\n Area (reference to obj) = "<<ref.getArea();
  cout<<"\n Area (reference to pointer) = "<<ref2->getArea();
  return 0;
}
```



```
 Area (object) = 20
 Area (pointer) = 20
 Area (reference to obj) = 20
 Area (reference to pointer) = 20

...Program finished with exit code 0
Press ENTER to exit console.
```

Constructors and Destructors

Interface vs Implementation

- **Separating interface** from **implementation**
  - **Makes it easier to modify programs**
  - **Header files**
    - **Contains class definitions** and **function prototypes**
  - **Source-code files**
    - **Contains member function definitions**

```
// Fig. 6.5: time1.h
// Declaration of the Time class.
// Member functions are defined in time1.cpp

// prevent multiple inclusions of header file
#ifndef TIME1_H
#define TIME1_H

// Time abstract data type definition
class Time {
public:
   Time();                          // constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

#endif
```

Dot (.) replaced with underscore (_) in file name.

If **`time1.h`** (**`TIME1_H`**) is not defined (**`#ifndef`**) then it is loaded (**`#define TIME1_H`**). If **`TIME1_H`** *is* already defined, then everything up to **`#endif`** is ignored.
This prevents loading a header file multiple times.

```cpp
// Fig. 6.5: time1.cpp
// Member function definitions for Time class.
#include <iostream>

using std::cout;

#include "time1.h"

// Time constructor initializes each data member to zero.
// Ensures all Time objects start in a consistent state.
Time::Time() { hour = minute = second = 0; }

// Set a new Time value using military 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;
}

// Print Time in military format
void Time::printMilitary()
{
   cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}

// Print time in standard format
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}
```

Source file uses **#include** to load the header file

Source file contains function definitions

Constructors

- A ***constructor is a special function used to create an object***. **Constructor** has **exactly** the **same name** as the **defining class**

- **Constructors** can be ***overloaded*** (i.e., **multiple constructors** with **different signatures**) [**Purpose: making it easy to construct objects with different initial data values**).

- A **class** may be **declared without constructors**. In this case, **a no-argument constructor with an empty body** is **implicitly declared** in the class known as **default constructor**

- ***Note:*** **Default constructor is provided automatically *only if no constructors are explicitly declared* in the class.**

Constructors' Properties

- ***must have the same name as the class*** itself.
- ***do not have a return type***—***<u>not even void</u>***.
- ***play the role of initializing objects***.

Constructors and Destructors

- **Constructor** is a **function in every class** which is **called** when **class creates its object**
  - Basically it **helps in initializing data members** of the class
  - A class may have **multiple constructors**

- **Destructors** is a **function in every class** which is **called** when the **object of a class is destroyed**
  - The main purpose of destructor is to **remove dynamic memories etc.**

Initializing Objects

Notice that default settings for the three member variables are set in constructor prototype. No names are needed; the defaults are applied in the order the variables are declared.

```
// Time abstract data type definition
class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor
   void setTime( int, int, int ); // set hour, minute, second
   void printMilitary();          // print military time format
   void printStandard();          // print standard time format
private:
   int hour;     // 0 - 23
   int minute;   // 0 - 59
   int second;   // 0 - 59
};

#endif
```

Default Parameters in Constructor

Same constructor, used in overloaded style

```

int main()
{
   Time t1,             // all arguments defaulted
        t2(2),          // minute and second defaulted
        t3(21, 34),     // second defaulted
        t4(12, 25, 42), // all values specified
        t5(27, 74, 99); // all bad values specified

   cout << "Constructed with:\n"
        << "all arguments defaulted:\n   ";
   t1.printMilitary();
   cout << "\n   ";
   t1.printStandard();

   cout << "\nhour specified; minute and second defaulted:"
        << "\n   ";
   t2.printMilitary();
   cout << "\n   ";
   t2.printStandard();

   cout << "\nhour and minute specified; second defaulted:"
        << "\n   ";
   t3.printMilitary();
```

```cpp
   cout << "\n   ";
   t3.printStandard();

   cout << "\nhour, minute, and second specified:"
        << "\n   ";
   t4.printMilitary();
   cout << "\n   ";
   t4.printStandard();

   cout << "\nall invalid values specified:"
        << "\n   ";
   t5.printMilitary();
   cout << "\n   ";
   t5.printStandard();
   cout << endl;

   return 0;
}
```

```
 OUTPUT
Constructed with:
all arguments defaulted:
   00:00
   12:00:00 AM
hour specified; minute and second defaulted:
   02:00
   2:00:00 AM
hour and minute specified; second defaulted:
   21:34
   9:34:00 PM
hour, minute, and second specified:
   12:25
   12:25:42 PM
all invalid values specified:
   00:00
   12:00:00 AM
```

When only **`hour`** is specified, **`minute`** and **`second`** are set to their default values of **`0`**.

Using Destructors

- **Destructors**
  - Are **member function** of **class**
  - **Perform termination housekeeping** before the system reclaims the object' s memory
  - Name is **tilde** (**~**) **followed by** the **class name** (i.e., **`~Time`**)
  - Receives **no parameters**, **returns no value**
  - **One destructor per class (no overloading)**
  - Destructors **cannot be declared *const***, ***static***
  - A **destructor can be declared *virtual*** or ***pure virtual***

When Constructors and Destructors Are Called

Constructors and destructors called automatically

- **Order depends on scope of objects**

1. Global scope objects
   - Constructors **called before any other function** (including `main`)
   - Destructors called when `main` **terminates** (*or* `exit` ***function called***)
   - Destructors not called if **program terminates** with `abort`

2. Automatic local objects
   - Constructors **called when** objects are defined
   - Destructors **called when** objects leave scope
   - **Destructors not called** if the program ends with `exit` or `abort`

```cpp
class CreateAndDestroy {
public:
   CreateAndDestroy( int );  // constructor
   ~CreateAndDestroy();      // destructor
private:
   int data;
};

#endif
```

```

CreateAndDestroy::CreateAndDestroy( int value )
{
   data = value;
   cout << "Object " << data << "   constructor";
}

CreateAndDestroy::~CreateAndDestroy()
   { cout << "Object " << data << "   destructor " << endl; }
```

Constructor and Destructor changed to print when they are called.

```cpp

void create( void );   // prototype

CreateAndDestroy first( 1 );  // global object

int main()
{
   cout << "   (global created before main)" << endl;

   CreateAndDestroy second( 2 );         // local object
   cout << "   (local automatic in main)" << endl;

   static CreateAndDestroy third( 3 );   // local object
   cout << "   (local static in main)" << endl;

   create();  // call function to create objects

   CreateAndDestroy fourth( 4 );         // local object
   cout << "   (local automatic in main)" << endl;
   return 0;
}
```

```cpp

// Function to create objects
void create( void )
{
   CreateAndDestroy fifth( 5 );
   cout << "   (local automatic in create)" << endl;

   static CreateAndDestroy sixth( 6 );
   cout << "   (local static in create)" << endl;

   CreateAndDestroy seventh( 7 );
   cout << "   (local automatic in create)" << endl;
}
```

```
OUTPUT
Object 1   constructor   (global created before main)
Object 2   constructor   (local automatic in main)
Object 3   constructor   (local static in main)
Object 5   constructor   (local automatic in create)
Object 6   constructor   (local static in create)
Object 7   constructor   (local automatic in create)
Object 7   destructor
Object 5   destructor
Object 4   constructor   (local automatic in main)
Object 4   destructor
Object 2   destructor
Object 6   destructor
Object 3   destructor
Object 1   destructor
```

Notice how the order of the constructor and destructor call depends on the types of variables (automatic, global and `static`) they are associated with.

Destructor Example

```
void f1()
{
   Employee *c = new Employee[3];
   c[0].var1 = 322;
   c[1].var1 = 5
   c[2].var1 = 9;

}
```