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

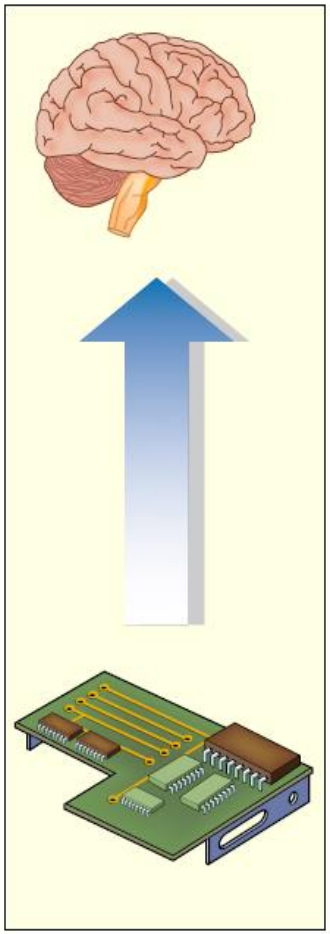
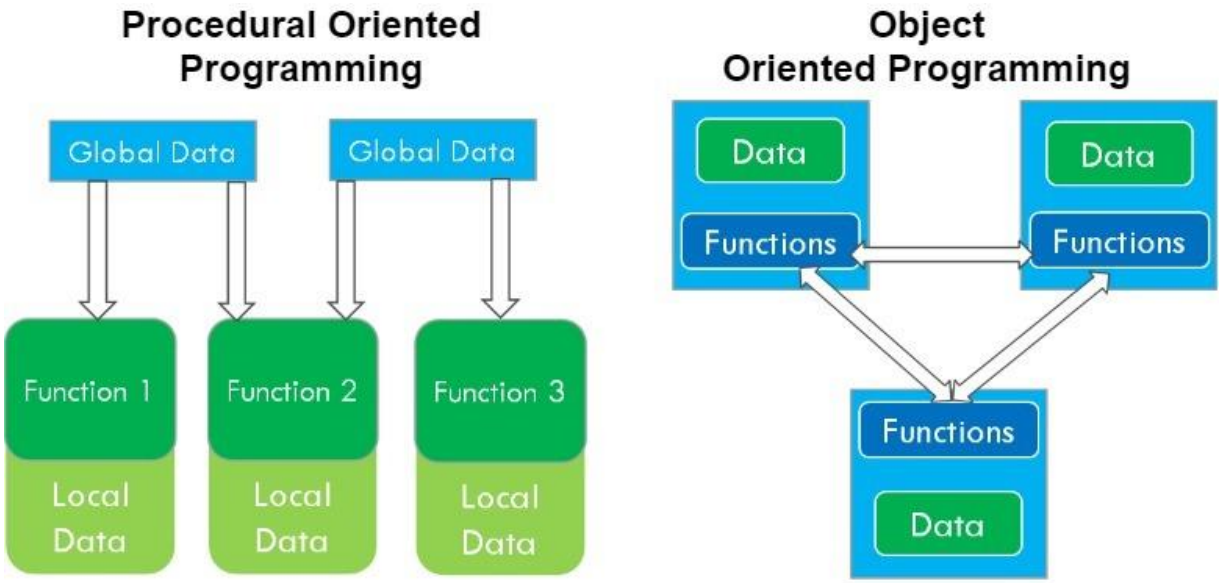
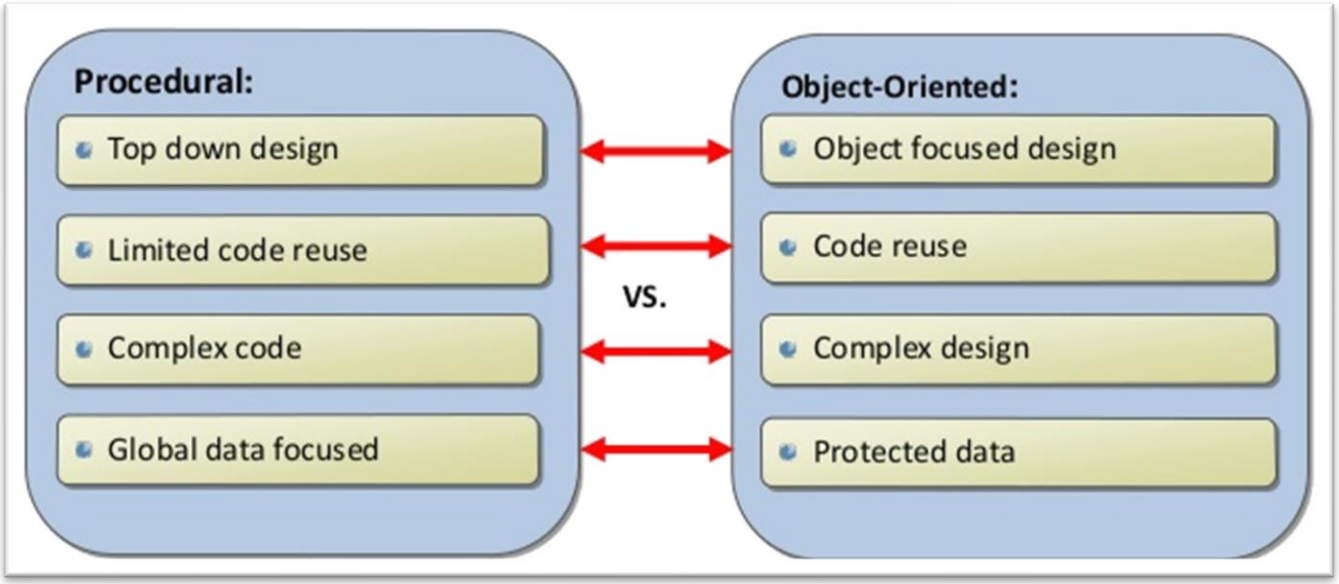


Figure by MIT OpenCourseWare.

Procedural VS. Object-Oriented Programming





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**.

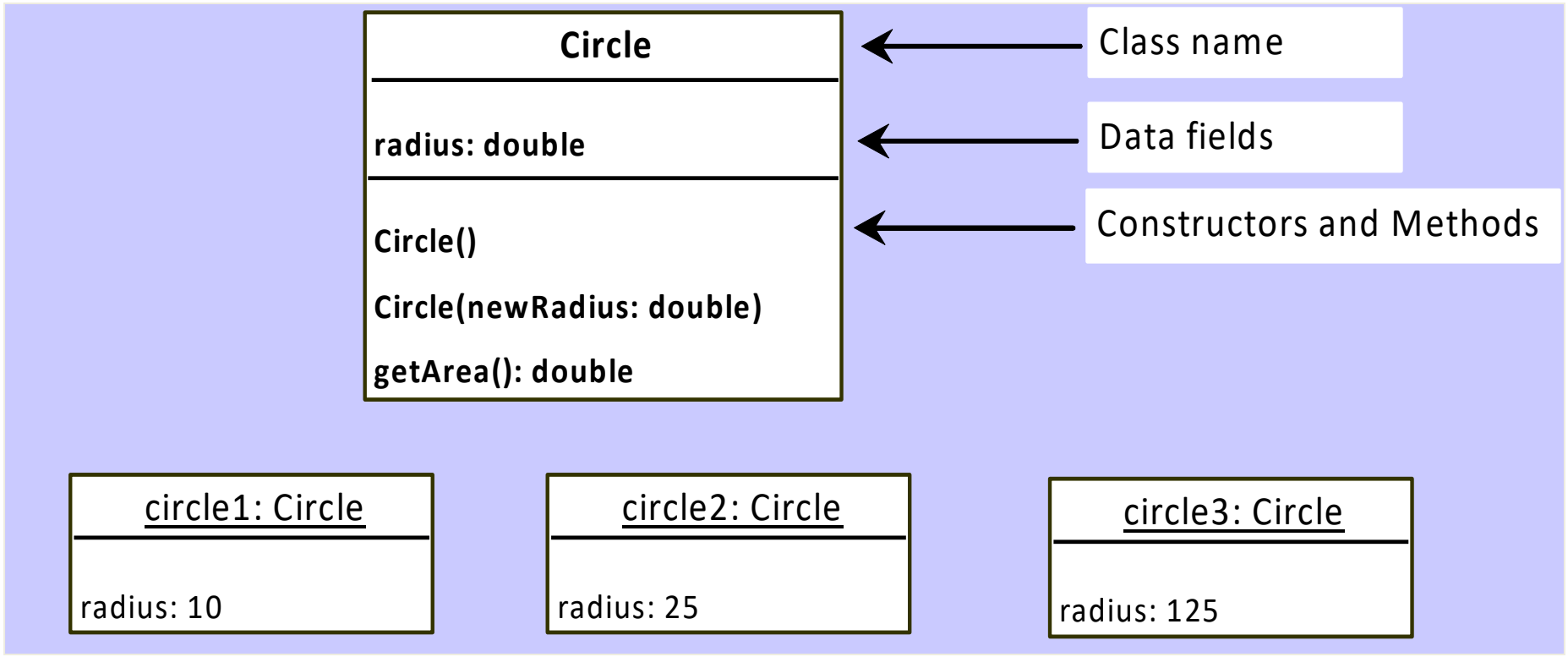
Objects in OOP

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

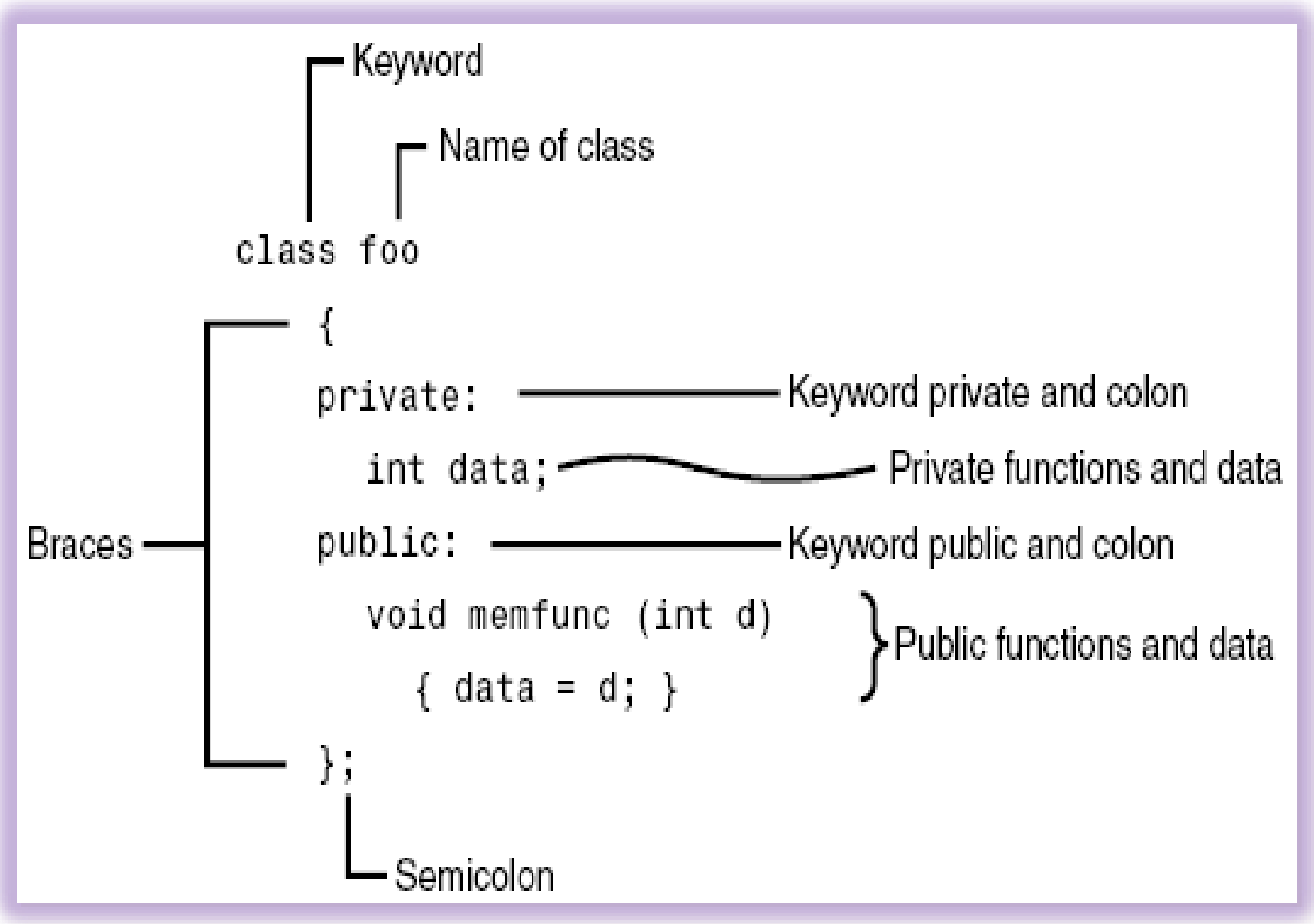
- The **state** of an **object** consists of ***a set of data fields*** (also known as **properties**) **with their current values**.

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

UML Diagram for Class and Object



Class in C++ - Example



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**.

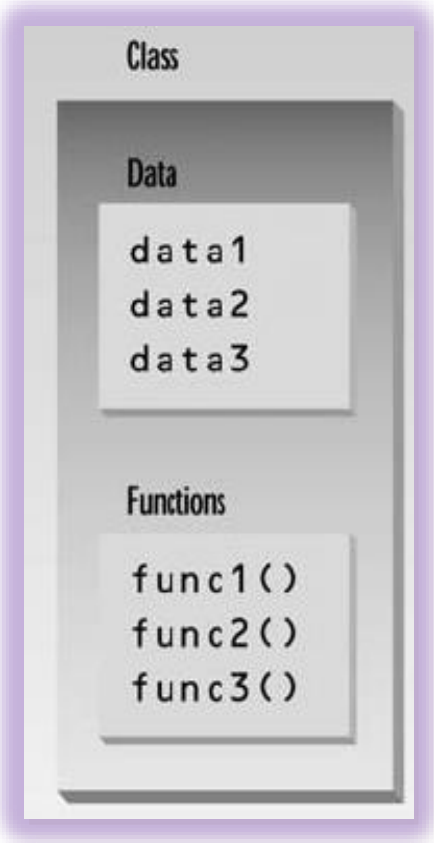
- You can also use **class names** to **declare object names**.

- ***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

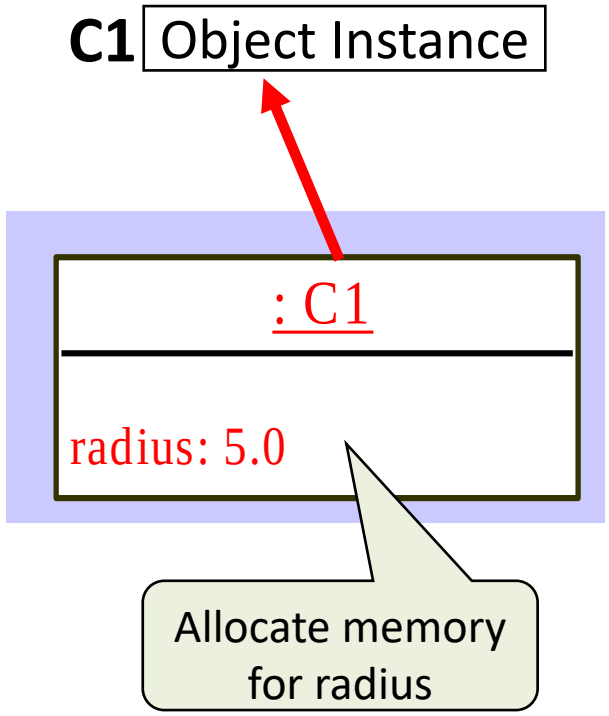
- After **object creation**, its **data** and **functions** can be **accessed** (invoked) using:
  - The **.** **operator**, also known as the ***object member access operator***.

- ***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; }
};
```




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

Class must define a no-argument constructor too….

```
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;
}
```

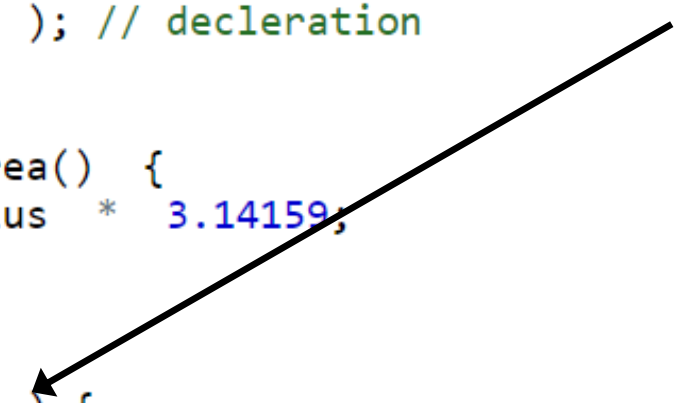
Note the :: preceding the function names.

```
33
34 // Print Time in military format
35 void Time::printMilitary()
36 {
37    cout << ( hour < 10 ? "0" : "" ) << hour << ":"
38         << ( minute < 10 ? "0" : "" ) << minute;
39 }
40
41 // Print Time in standard format
42 void Time::printStandard()
43 {
44    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
45         << ":" << ( minute < 10 ? "0" : "" ) << minute
46         << ":" << ( second < 10 ? "0" : "" ) << second
47         << ( hour < 12 ? " AM" : " PM" );
48 }
49
```

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;
}
```

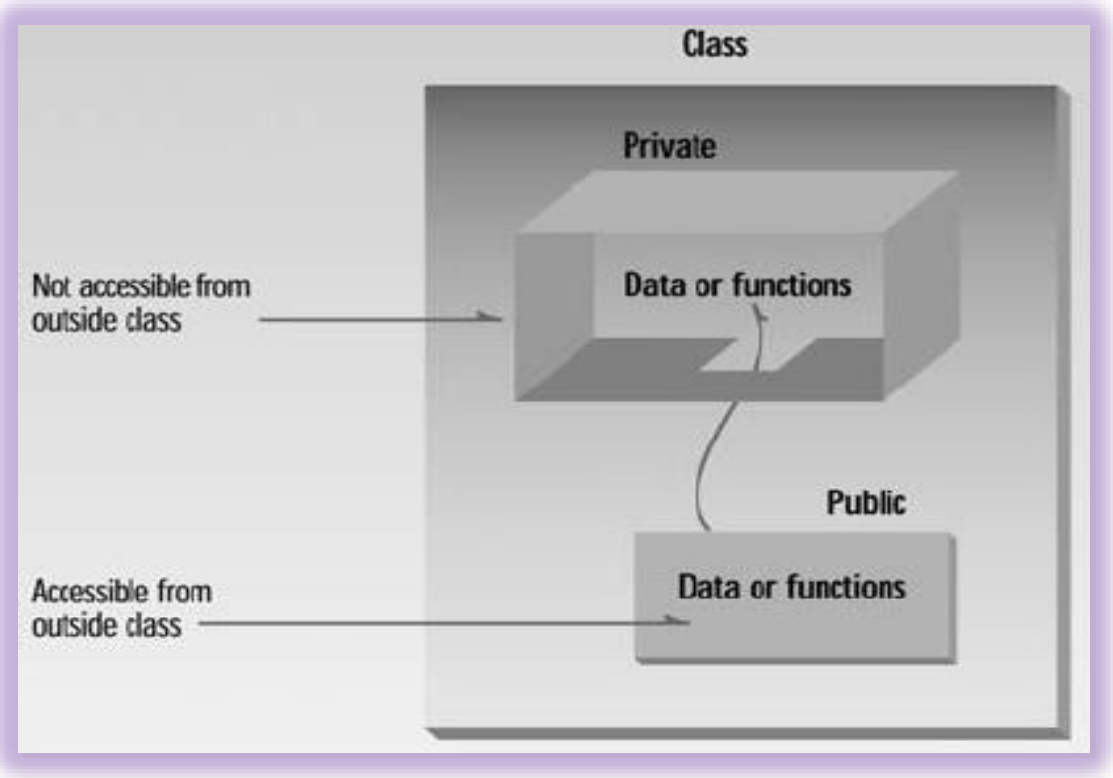


Private Member Functions

(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***



- **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*.**

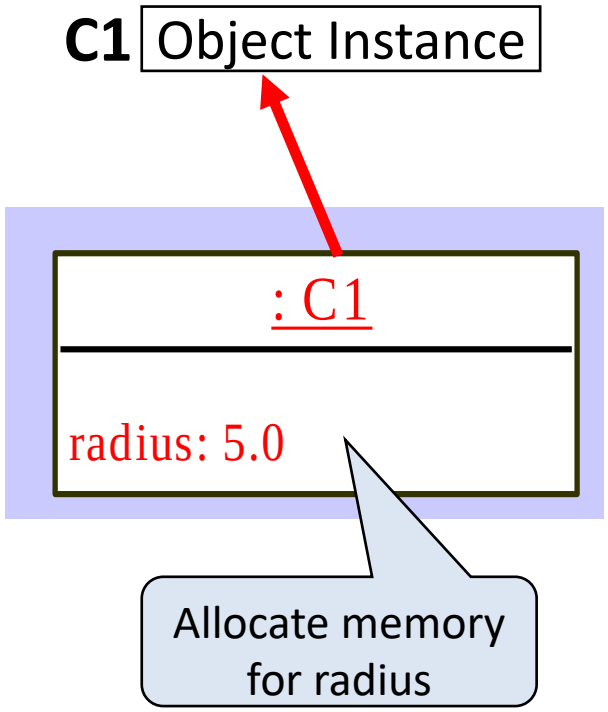
Hidden from Whom?

- ***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***

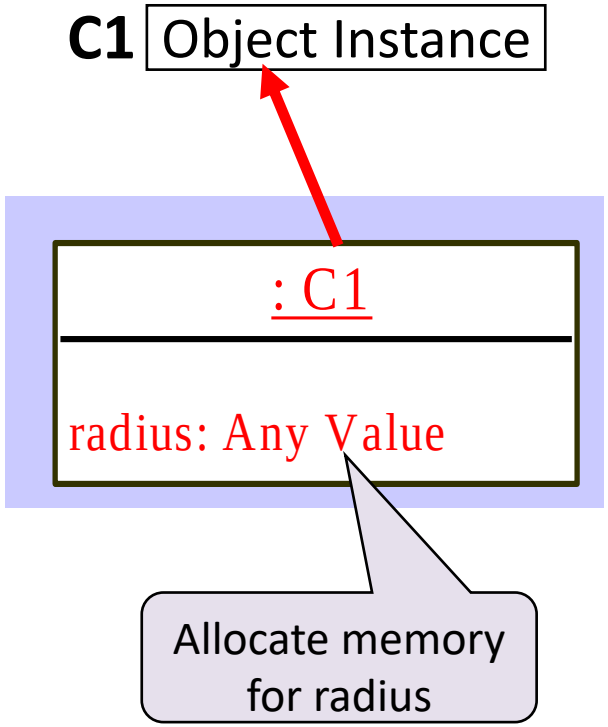
```
class  Circle
{
    private:
        double radius;

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




```
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:

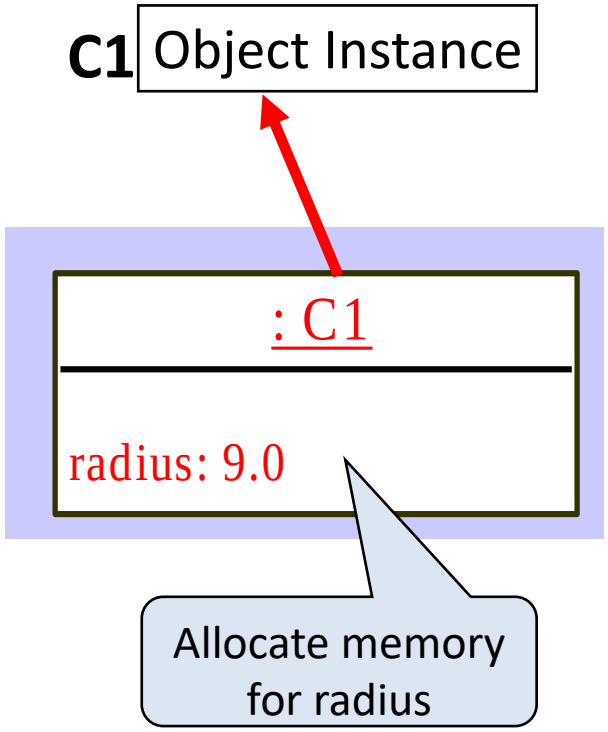
ClassName objectName(arguments);

- 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; }
};
```




```
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**

```
1   // Define a Rectangle pointer.
2   Rectangle *rectPtr = nullptr;
3
4   // Dynamically allocate a Rectangle object.
5   rectPtr = new Rectangle;
6
7   // Store values in the object's width and length.
8   rectPtr->setWidth(10.0);
9   rectPtr->setLength(15.0);
10
11  // Delete the object from memory.
12  delete rectPtr;
13  rectPtr = nullptr;
```

Reference to Objects

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

```
6   // class Count definition
7   class Count
8   {
9   public: // public data is dangerous
10     // sets the value of private data member x
11     void setX( int value )
12     {
13        x = value;
14     } // end function setX
15
16     // prints the value of private data member x
17     void print()
18     {
19        cout << x << endl;
20     } // end function print
21
22  private:
23     int x;
24  }; // end class Count
```

Reference to Objects

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

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*—*not even void***.
- ***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.

```
10 // Time abstract data type definition
11 class Time {
12 public:
13    Time( int = 0, int = 0, int = 0 );  // default constructor
14    void setTime( int, int, int ); // set hour, minute, second
15    void printMilitary();          // print military time format
16    void printStandard();          // print standard time format
17 private:
18    int hour;      // 0 - 23
19    int minute;    // 0 - 59
20    int second;    // 0 - 59
21 };
22
23 #endif
```

Default Parameters in Constructor

Same constructor, used in overloaded style

```
70
71 int main()
72 {
73    Time t1,          // all arguments defaulted
74         t2(2),       // minute and second defaulted
75         t3(21, 34),  // second defaulted
76         t4(12, 25, 42), // all values specified
77         t5(27, 74, 99); // all bad values specified
78
79    cout << "Constructed with:\n"
80         << "all arguments defaulted:\n   ";
81    t1.printMilitary();
82    cout << "\n   ";
83    t1.printStandard();
84
85    cout << "\nhour specified; minute and second defaulted:"
86         << "\n   ";
87    t2.printMilitary();
88    cout << "\n   ";
89    t2.printStandard();
90
91    cout << "\nhour and minute specified; second defaulted:"
92         << "\n   ";
93    t3.printMilitary();
```

```
94     cout << "\n   ";
95     t3.printStandard();
96
97     cout << "\nhour, minute, and second specified:"
98          << "\n   ";
99     t4.printMilitary();
100    cout << "\n   ";
101    t4.printStandard();
102
103    cout << "\nall invalid values specified:"
104         << "\n   ";
105    t5.printMilitary();
106    cout << "\n   ";
107    t5.printStandard();
108    cout << endl;
109
110    return 0;
111 }
```

```
 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`

```
7  class CreateAndDestroy {
8  public:
9     CreateAndDestroy( int );  // constructor
10    ~CreateAndDestroy();      // destructor
11 private:
12    int data;
13 };
14
15 #endif
```

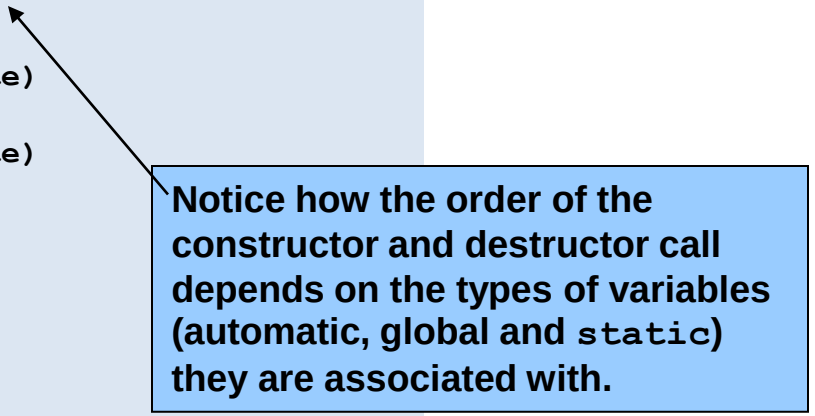
```
24
25 CreateAndDestroy::CreateAndDestroy( int value )
26 {
27    data = value;
28    cout << "Object " << data << "   constructor";
29 }
30
31 CreateAndDestroy::~CreateAndDestroy()
32    { cout << "Object " << data << "   destructor " << endl; }
```

Constructor and Destructor changed to print when they are called.

```
42
43 void create( void );   // prototype
44
45 CreateAndDestroy first( 1 );  // global object
46
47 int main()
48 {
49    cout << "   (global created before main)" << endl;
50
51    CreateAndDestroy second( 2 );        // local object
52    cout << "   (local automatic in main)" << endl;
53
54    static CreateAndDestroy third( 3 );  // local object
55    cout << "   (local static in main)" << endl;
56
57    create();  // call function to create objects
58
59    CreateAndDestroy fourth( 4 );        // local object
60    cout << "   (local automatic in main)" << endl;
61    return 0;
62 }
```

```
63
64 // Function to create objects
65 void create( void )
66 {
67    CreateAndDestroy fifth( 5 );
68    cout << "   (local automatic in create)" << endl;
69
70    static CreateAndDestroy sixth( 6 );
71    cout << "   (local static in create)" << endl;
72
73    CreateAndDestroy seventh( 7 );
74    cout << "   (local automatic in create)" << endl;
75 }
```

```
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
```



Destructor Example

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

}
```

When do Constructors Get Called?

- Class object creation time
- Dynamically allocated object
- Class argument passed by value
- Class object returned by value
- Object Array element (created)

What Constructors Do

- **Help in initializing:** class data members

```
Employee( ) { id = 0; }
```

- **Allocate memory** for **dynamic members**

```
Employee() { char* nameptr = new char[20];}
```

- **Allocate** any **needed resources**
  - **Such as to open files, etc.**

Constructing Arrays of Objects

```
Complex c_arr[10];
Date date_arr[20];
```

Issue: There is no way to call argument-based constructors (non-default) for array members

Arrays of Objects and Non-Default Constructors

- **Trick:** declare an **array** of pointer to objects
- **Allocate** and **initialize each object in a loop**

```
Date *dates[31];

for (int day = 0; day < 31; ++day)
{
  dates[day] = new Date(3, day, 2020);
}
```

Default Member-wise Assignment

- **Assignment operator** (**=**) can be used **to assign** an **object** to **another object** of the **same type**.

- **Member-wise assignment**: each **data member** of the **object** on the **right of the assignment** operator is **assigned individually** to the ***same*** **data member** in the **object on the left**

```
class Date
{
public:
   Date( int = 1, int = 1, int = 2000 ); // default constructor
   void print();
private:
   int month;
   int day;
   int year;
}; // end class Date

#endif
```

```
// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
} // end constructor Date

// print Date in the format mm/dd/yyyy
void Date::print()
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

```
int main()
{
   Date date1( 7, 4, 2004 );
   Date date2; // date2 defaults to 1/1/2000

   cout << "date1 = ";
   date1.print();
   cout << "\ndate2 = ";
   date2.print();

   date2 = date1; // default memberwise assignment

   cout << "\n\nAfter default memberwise assignment, date2 = ";
   date2.print();
   cout << endl;
} // end main
```

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

Default copy constructor

- **A type of constructor that is used to initialize an object with another object of the same type is known as default copy constructor.**

- It is **by default available in all classes**

- syntax is **ClassName(ClassName &Variable)**

Copy Constructor for Class Date

```
Date::Date(Date &date)
{
// no need to check passed date arg
  month = date.month;
  day   = date.day;
  year  = date.year;
}
```

Uses of the Copy Constructor

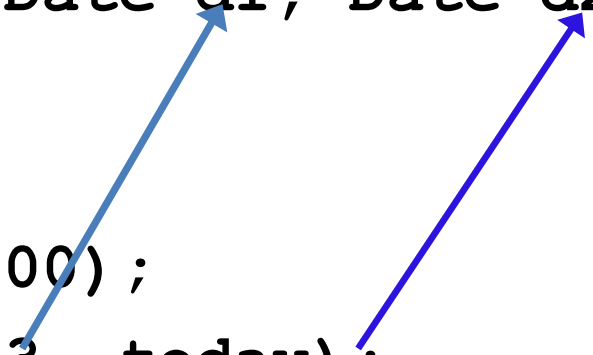
- **Implicitly called in 3 situations:**
  1. **defining** a **new object** from an **existing object**
  2. **passing an object by value**
  3. **returning an object by value**

Copy Constructor: Defining a New Object

```
Date d1(02,28,2020);

// init 2 local objects from d1
Date d2(d1);// pass by value
Date d3 = d1;// return value

// init a dynamic object from d1
Date* pdate = new Date(d1);
```

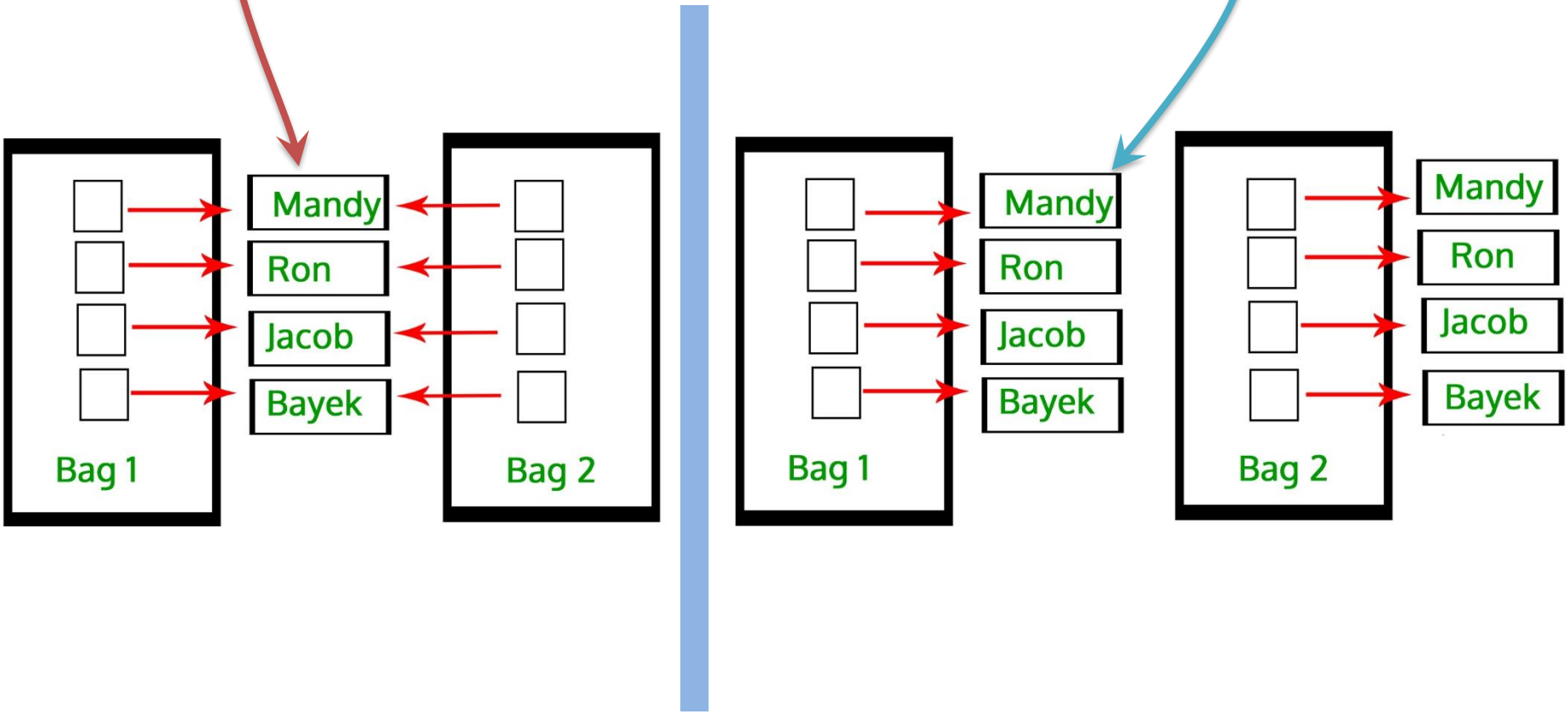
Copy Constructor:Passing Objects by Value

```
//copy constructor called for each value arg
unsigned dateDiff(Date d1, Date d2);
...
Date today;
Date d3(02, 21, 2000);
cout << dateDiff(d3, today);
```



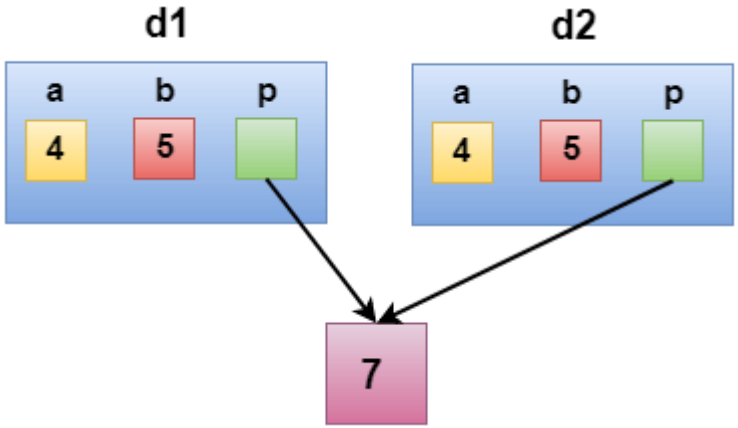
User-defined Copy Constructor, when required?

- Default-copy Constructor do only “Shallow Copy”
- We need user-defined copy-constructor,
  - When we need “Deep Copy” (for Dynamic Memory)



Shallow Copy

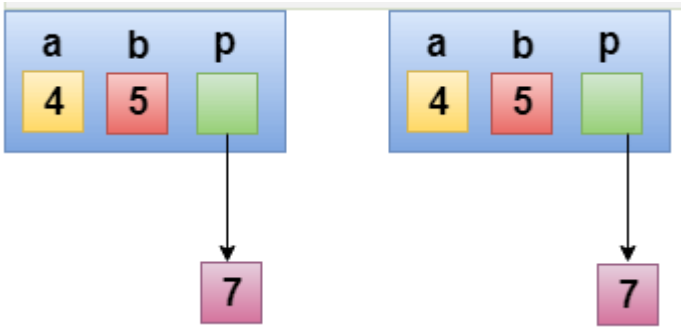
```
1.  class Demo
2.  {
3.      int a;
4.      int b;
5.      int *p;
6.      public:
7.      Demo()
8.      {
9.          p=new int;
10.     }
11.     void setdata(int x,int y,int z)
12.     {
13.         a=x;
14.         b=y;
15.         *p=z;
16.     }
17.     void showdata()
18.     {
19.         std::cout << "value of a is : " <<a<< std::endl;
20.         std::cout << "value of b is : " <<b<< std::endl;
21.         std::cout << "value of *p is : " <<*p<< std::endl;
22.     }
```



Deep Copy

```
1.  class Demo
2.  {
3.      public:
4.      int a;
5.      int b;
6.      int *p;
7.
8.      Demo()
9.      {
10.         p=new int;
11.     }
12.     Demo(Demo &d)
13.     {
14.         a = d.a;
15.         b = d.b;
16.         p = new int;
17.         *p = *(d.p);
18.     }
19.     void setdata(int x,int y,int z)
20.     {
21.         a=x;
22.         b=y;
23.         *p=z;
24.     }
```

```
1.
2.     void showdata()
3.     {
4.         std::cout << "value of a is : " <<a<< std::endl;
5.         std::cout << "value of b is : " <<b<< std::endl;
6.         std::cout << "value of *p is : " <<*p<< std::endl;
7.     }
8. };
9. int main()
10.{
11.  Demo d1;
12.  d1.setdata(4,5,7);
13.  Demo d2 = d1;
14.  d2.showdata();
15.  return 0;
16.}
```



static, const, and *this* Pointer

`static` Class Members

- **`static` class members**
  - **Shared** by **all objects of a class**
  - **Efficient, when a single copy of data is enough**
    - **Only the `static` variable has to be updated**
  - **May seems like global variables**, but have **class scope**
    - **only accessible to objects of same class**
  - **Initialized** at **file scope**
  - **Exist** even **if no instances (objects) of the class exist**
  - Both **variables** and **functions** can be **`static`**
  - Can be **`public`**, **`private`** **or** **`protected`**

`static` Class Variables

- **Two-Step Procedure:**
  1. **Declare (Inside Class):** `static int radius;`
  2. **Define (Outside Class):** `int Circle::radius=2;`

- **static Variables**
  - **Default Initialization: 0 or Null (for pointers)**
  - **Initialization: user defined value**
  - **Initialization** is **made just once**, at **compile time.**
  - **Accessibility: Private or Public**

Public static Class Variables

- Can be accessed using Class name:

```
cout<<Employee::count;
```

- Can be accessed via any class' object:

```
cout<<e1.count;
```

- Can be accessed via Non-Static member functions:

```
cout<<e1.getCount();
```

- Can be accessed via Static member functions:

```
cout<<Employee::Stat_getCount();
cout<<e1.Stat_getCount(); //public static
```

Private static Class Variables

- Cannot be accessed using Class name:

```
// ERROR → cout<<Employee::count;
```

- Cannot be accessed via class' object:

```
// ERROR → cout<<e1.count;
```

- Can be accessed via Non-Static member functions:

```
cout<<e1.getCount();
```

- Can be accessed via Static member functions:

```
cout<<Employee::Stat_getCount();
cout<<e1.Stat_getCount(); //public static
```

`static` Class Functions

- **Non-static function:**
  - **Can access: static/non-static data members** and **static/non-static methods**

- **Static functions:**
  - **Can access: static data and static functions**
  - **Cannot access: non-static data, non-static functions, and this pointer**

Public static Class Functions

–Can be invoked using class's any object:

```
cout<<e1.getCount();
```

–Can be invoked using Class name:

```
cout<<Employee::getCount();
```

`Private static` Class Functions

- Cannot be invoked using class's object

```
//ERROR → cout<<e1.getCount();
```

- Cannot be invoked using Class name

```
//ERROR → cout<<Employee::getCount();
```

- Can be invoked within Class:
  - Static member functions
  - Non-Static member functions

```
1  // Fig. 7.9: employ1.h
2  // An employee class
3  #ifndef EMPLOY1_H
4  #define EMPLOY1_H
5
6  class Employee {
7  public:
8     Employee( const char*, const char* );  // constructor
9     ~Employee();                           // destructor
10    const char *getFirstName() const;  // return first name
11    const char *getLastName() const;   // return last name
12
13    // static member function
14    static int getCount(); // return # objects instantiated
15
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data member
21    static int count;  // number of objects instantiated
22 };
23
24 #endif
```

static member function and variable declared.

```
25 // Fig. 7.9: employ1.cpp
26 // Member function definitions for class Employee
27 #include <iostream>
28
29 using std::cout;
30 using std::endl;
31
32 #include <cstring>
33 #include <cassert>
34 #include "employ1.h"
35
36 // Initialize the static data member
37 int Employee::count = 0;
38
39 // Define the static member function that
40 // returns the number of employee objects instantiated.
41 int Employee::getCount() { return count; }
42
43 // Constructor dynamically allocates space for the
44 // first and last name and uses strcpy to copy
45 // the first and last names into the object
46 Employee::Employee( const char *first, const char *last )
47 {
48    firstName = new char[ strlen( first ) + 1 ];
49    assert( firstName != 0 );    // ensure memory allocated
50    strcpy( firstName, first );
51
52    lastName = new char[ strlen( last ) + 1 ];
53    assert( lastName != 0 );     // ensure memory allocated
54    strcpy( lastName, last );
55
56    ++count;  // increment static count of employees
```

`static` data member `count` and function `getCount( )` initialized at file scope (required).

Note the use of **`assert`** to **test for memory allocation.**

`static` data member **`count`** changed when a constructor/destructor called.

```
57    cout << "Employee constructor for " << firstName
58         << ' ' << lastName << " called." << endl;
59 }
60
61 // Destructor deallocates dynamically allocated memory
62 Employee::~Employee()
63 {
64    cout << "~Employee() called for " << firstName
65         << ' ' << lastName << endl;
66    delete [] firstName;  // recapture memory
67    delete [] lastName;   // recapture memory
68    --count;  // decrement static count of employees
69 }
70
71 // Return first name of employee
72 const char *Employee::getFirstName() const
73 {
74    // Const before return type prevents client from modifying
75    // private data. Client should copy returned string before
76    // destructor deletes storage to prevent undefined pointer.
77    return firstName;
78 }
79
80 // Return last name of employee
81 const char *Employee::getLastName() const
82 {
83    // Const before return type prevents client from modifying
84    // private data. Client should copy returned string before
85    // destructor deletes storage to prevent undefined pointer.
86    return lastName;
87 }
```

`static` data member **`count`** changed when a **constructor/destructor called**.

`Count` decremented because of destructor calls from **`delete`**.

```
88 // Fig. 7.9: fig07_09.cpp
89 // Driver to test the employee class
90 #include <iostream>
91
92 using std::cout;
93 using std::endl;
94
95 #include "employ1.h"
96
97 int main()
98 {
99    cout << "Number of employees before instantiation is "
100        << Employee::getCount() << endl;    //
101
102   Employee *e1Ptr = new Employee( "Susan", "Baker" );
103   Employee *e2Ptr = new Employee( "Robert", "Jones" );
104
105   cout << "Number of employees after instantiation is "
106        << e1Ptr->getCount();
107
108   cout << "\n\nEmployee 1: "
109        << e1Ptr->getFirstName()
110        << " " << e1Ptr->getLastName()
111        << "\nEmployee 2: "
112        << e2Ptr->getFirstName()
113        << " " << e2Ptr->getLastName() << "\n\n";
114
115   delete e1Ptr;   // recapture memory
116   e1Ptr = 0;
117   delete e2Ptr;   // recapture memory
118   e2Ptr = 0;
```

count incremented because of constructor calls from new.

If no Employee objects exist getCount must be accessed using the class name and (::).

Number of employees before instantiation is 0

e2Ptr->getCount() or Employee::getCount() would also work.

Number of employees after instantiation is 2

Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones

```
119
120    cout << "Number of employees after deletion is "
121         << Employee::getCount() << endl;
122
123    return 0;
124 }
```

count back to zero.

```
Number of employees before instantiation is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after instantiation is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after deletion is 0
```

const Class Members

- As with **member functions**, **data members** can also be **const**

- **Member Initializer List:**
  - **Can be used to initialize both const and non-const data members**
  - `consts` and **references must be initialized using member initializer**

Member Initializer List (Non-const Members)

```
class Point {
private:
    int x;
    int y;
public:
    Point(int i = 2, int j = 3):y(i) {x=j;}
    /* The above use of Initializer list is optional as the  constructor can also
    be written as:
    Point(int i = 0, int j = 0) {
            x = i;
            y = j;
    } */

    int getX() const {return x;}
    int getY() const {return y;}
};

int main() {
  Point t1(10, 15);
  cout<<"x = "<<t1.getX()<<", ";
  cout<<"y = "<<t1.getY();
  return 0;
}
```

Member Initializer List (non-static const)

```
#include<iostream>
using namespace std;

class Test {
    const int t;
public:
    Test(int x):t(x) {} //Initializer list must be used
    int getT() { return t; }
};

int main() {
    Test t1(10);
    cout<<t1.getT();
    return 0;
}
```

Member Initializer List (References)

```
#include<iostream>
using namespace std;

class Test {
    int &cRef;
public:
    Test(int &ref):cRef(ref) {}  //Initializer list must be used
    int getRef() { return cRef; }
};

int main() {
    int x = 20;
    Test t1(x);
    cout<<t1.getRef()<<endl;
    x = 30;
    cout<<t1.getRef()<<endl;
    return 0;
}
```

Member Initializer List (member object, no default constructor)

```
class A {
    int i;
public:
    A(int);
};

A::A(int arg) {
    i = arg;
    cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B contains object of A
class B {
    A a;
public:
    B(int );
};

B::B(int x):a(x) {  //Initializer list must be used
    cout << "B's Constructor called";
}

int main() {
    B obj(10);
    return 0;
}
```

Member Initializer List (parameter name same as data member)

```
#include <iostream>
using namespace std;

class A {
    int i;
public:
    A(int );
    int getI() const { return i; }
};

A::A(int i):i(i) { }  // Either Initializer list or this pointer must be used
/* The above constructor can also be written as
A::A(int i) {
    this->i = i;
}
*/

int main() {
    A a(10);
    cout<<a.getI();
    return 0;
}
```

Member Initializer List (base class members)

Will be discussed in chapters after midterm

The *this* Pointer

- ***this*** **keyword is a special built-in pointer (constant pointer) that references to the calling object.**

- **this** pointer is passed as a **hidden argument** to all **non-static member function** and is **available** as a **local variable** within the body of all **non-static functions**.
  - **Not part of the object itself (this pointer is not reflected with sizeof(object))**

- **Can be used** to **access instance variables** within **constructors** and **member functions**

Using the this Pointer

- **Examples using `this`**
  - For a **member function** **print data member `x`**, either

```
this->x;
```

or

```
(*this).x
```

- **Cascaded member function calls:**
  - **Function returns** a **reference pointer** to the **same object**

```
{ return *this; }
```

  - Other **functions** can **operate** on **that pointer**

Using the this Pointer

- **Example** of **cascaded member function calls:**
  - Member functions **`setHour`**, **`setMinute`**, and **`setSecond`** all return **`*this`** (***reference to an object***)
  - For **object** **`t`**, consider:

    ```
    t.setHour(1).setMinute(2).setSecond(3);
    ```

  - Executes **`t.setHour(1)`**, returns **`*this`** (reference to object) and the **expression becomes**

    ```
    t.setMinute(2).setSecond(3);
    ```

  - Executes **`t.setMinute(2)`**, **returns reference** and becomes

    ```
    t.setSecond(3);
    ```

  - Executes **`t.setSecond(3)`**, returns reference and becomes

    ```
    t;
    ```

    (Has no effect)

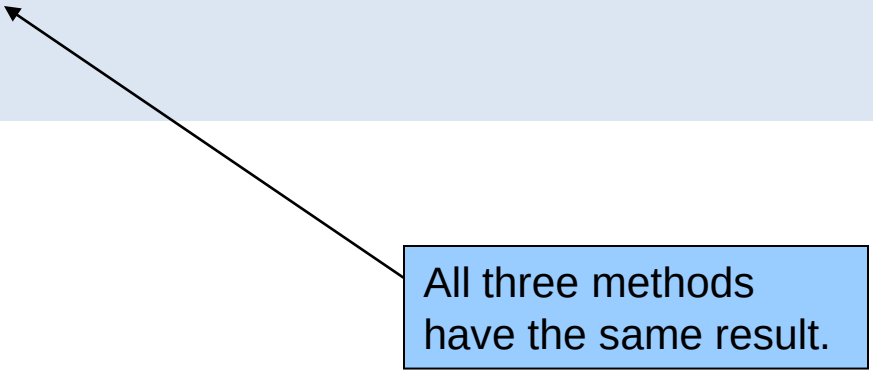
```
1  // Fig. 7.7: fig07_07.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  class Test {
9  public:
10    Test( int = 0 );              // default constructor
11    void print() const;
12 private:
13    int x;
14 };
15
16 Test::Test( int a ) { x = a; }  // constructor
17
18 void Test::print() const    // ( ) around *this required
19 {
20    cout << "         x = " << x
21         << "\n  this->x = " << this->x
22         << "\n(*this).x = " << ( *this ).x << endl;
23 }
24
25 int main()
26 {
27    Test testObject( 12 );
28
29    testObject.print();
30
31    return 0;
32 }
```

Printing **x** directly.

Print **x** using the arrow **->** operator off the **this** pointer.

Printing **x** using the dot (**.**) operator. Parenthesis required because dot operator has higher precedence than *****. Without, interpreted incorrectly as ***(this.x)**.

```
        x = 12
  this->x = 12
(*this).x = 12
```



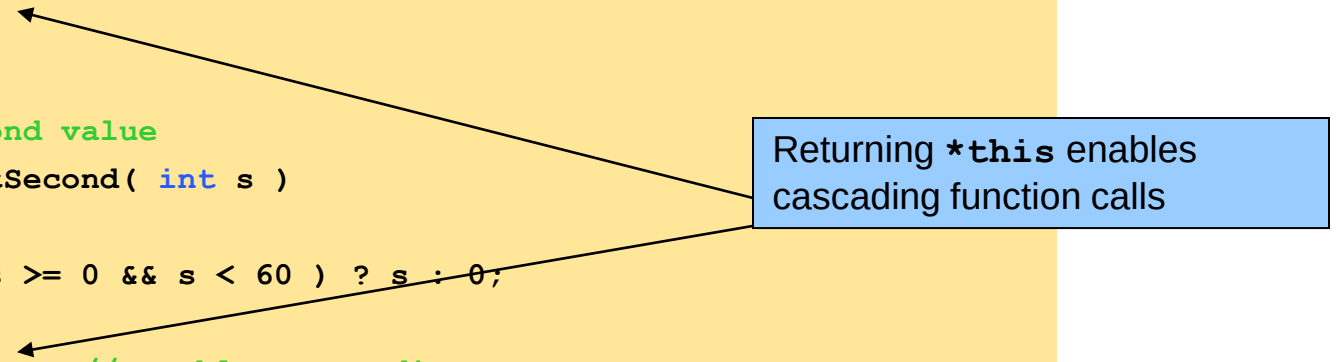
```
1  // Fig. 7.8: time6.h
2  // Cascading member function calls.
3
4  // Declaration of class Time.
5  // Member functions defined in time6.cpp
6  #ifndef TIME6_H
7  #define TIME6_H
8
9  class Time {
10 public:
11    Time( int = 0, int = 0, int = 0 );  // default constructor
12
13    // set functions
14    Time &setTime( int, int, int ); // set hour, minute, second
15    Time &setHour( int );     // set hour
16    Time &setMinute( int );   // set minute
17    Time &setSecond( int );   // set second
18
19    // get functions (normally declared const)
20    int getHour() const;      // return hour
21    int getMinute() const;    // return minute
22    int getSecond() const;    // return second
23
24    // print functions (normally declared const)
25    void printMilitary() const;  // print military time
26    void printStandard() const;  // print standard time
27 private:
28    int hour;              // 0 - 23
29    int minute;            // 0 - 59
30    int second;            // 0 - 59
31 };
32
33 #endif
```

Notice the **`Time &`** - function returns a **reference to a `Time` object**.
Specify object in function definition.

```
34 // Fig. 7.8: time.cpp
35 // Member function definitions for Time class.
36 #include <iostream>
37
38 using std::cout;
39
40 #include "time6.h"
41
42 // Constructor function to initialize private data.
43 // Calls member function setTime to set variables.
44 // Default values are 0 (see class definition).
45 Time::Time( int hr, int min, int sec )
46    { setTime( hr, min, sec ); }
47
48 // Set the values of hour, minute, and second.
49 Time &Time::setTime( int h, int m, int s )
50 {
51    setHour( h );
52    setMinute( m );
53    setSecond( s );
54    return *this;   // enables cascading
55 }
56
57 // Set the hour value
58 Time &Time::setHour( int h )
59 {
60    hour = ( h >= 0 && h < 24 ) ? h : 0;
61
62    return *this;   // enables cascading
63 }
64
```

Returning **`*this`** enables cascading function calls

```
65 // Set the minute value
66 Time &Time::setMinute( int m )
67 {
68    minute = ( m >= 0 && m < 60 ) ? m : 0;
69
70    return *this;   // enables cascading
71 }
72
73 // Set the second value
74 Time &Time::setSecond( int s )
75 {
76    second = ( s >= 0 && s < 60 ) ? s : 0;
77
78    return *this;   // enables cascading
79 }
80
81 // Get the hour value
82 int Time::getHour() const { return hour; }
83
84 // Get the minute value
85 int Time::getMinute() const { return minute; }
86
87 // Get the second value
88 int Time::getSecond() const { return second; }
89
90 // Display military format time: HH:MM
91 void Time::printMilitary() const
92 {
93    cout << ( hour < 10 ? "0" : "" ) << hour << ":"
94         << ( minute < 10 ? "0" : "" ) << minute;
```



```
95 }
96
97 // Display standard format time: HH:MM:SS AM (or PM)
98 void Time::printStandard() const
99 {
100    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
101         << ":" << ( minute < 10 ? "0" : "" ) << minute
102         << ":" << ( second < 10 ? "0" : "" ) << second
103         << ( hour < 12 ? " AM" : " PM" );
104 }
105 // Fig. 7.8: fig07_08.cpp
106 // Cascading member function calls together
107 // with the this pointer
108 #include <iostream>
109
110 using std::cout;
111 using std::endl;
112
113 #include "time6.h"
114
115 int main()
116 {
117    Time t;
118
119    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
120    cout << "Military time: ";
121    t.printMilitary();
122    cout << "\nStandard time: ";
123    t.printStandard();
124
125    cout << "\n\nNew standard time: ";
126    t.setTime( 20, 20, 20 ).printStandard();
```

printStandard does not return a reference to an object.

Notice cascading function calls.

Cascading function calls. **printStandard** must be called after **setTime** because **printStandard** does not return a reference to an object.
t.printStandard().setTime(); would cause an error.

```
127         cout << endl;
128
129         return 0;
130      }
```

```
Military time: 18:30
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```