Lecture-1 Date: 16.08.2016

C++ has 3 features

1. Encapsulation:
2. Polymorphism
3. Inheritance

Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created. When code and data are linked together in this fashion, an object is created.

In other words, an object is the device that supports encapsulation. Within an object, code, data, or both may be private to that object or public. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program may access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. For all intents and purposes, an object is a variable of a user-defined type. It may seem strange that an object that links both code and data can be thought of as a variable. However, in object-oriented programming, this is precisely the case.

Each time you define a new type of object, you are creating a new data type. Each specific instance of this data type is a compound variable.

Typdef int new type
Create a new type
Structure can contain function
Class like a structure
Object like a user defined data type.
Abs() – absolute value find int
Fabs()-floating absolute value find floating
Labs()-absolute value finds long integer.

In object oriented such as C++. One function abs() can do all task of those function, without user having to do anything.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example:

```
class Box
{
```

```
  public:
    double getVolume(void)
    {
      return length * breadth * height;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

Data encapsulation led to the important OOP concept of data hiding.

Data Encapsulation Example:

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

class Adder{
  public:
    // constructor
    Adder(int i = 0)
    {
      total = i;
    }
    // interface to outside world
    void addNum(int number)
    {
       total += number;
    }
    // interface to outside world
    int getTotal()
    {
       return total;
    };
  private:
    // hidden data from outside world
    int total;
};
int main( )
{
  Adder a;

  a.addNum(10);
  a.addNum(20);
  a.addNum(30);

  cout << "Total " << a.getTotal() <<endl;
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

Polymorphism:

Object-oriented programming languages support polymorphism, which is characterized by the phrase "one interface, multiple methods." **In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation.**

A real-word example of polymorphism is a thermostat. No matter what type of furnace your hou has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (meth you have. For example, if you want a 70-degree temperature, you set the thermostat 70 degrees. It doesn't matter what type of furnace actually provides the heat.
This same principle can also apply to programming.

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
Consider the following example where a base class has been derived by other two classes:

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

class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape{
  public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
```

```
      {
        cout << "Rectangle class area :" <<endl;
        return (width * height);
      }
};
class Triangle: public Shape{
  public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
// Main function for the program
int main( )
{
Shape *shape;
Rectangle rec(10,7);
Triangle  tri(10,5);

// store the address of Rectangle
shape = &rec;
// call rectangle area.
shape->area();

// store the address of Triangle
shape = &tri;
// call triangle area.
shape->area();

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Parent class area
Parent class area
```

For example, you might have a program that defines three different types of stacks. **One stack is used for integer values, one for character values, and one for floating-point values.**

Because of polymorphism, you can define one set of names, push() and pop(), that can be u for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being store Thus, the interface to a stack—the functions push() and pop()—are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data. Polymorphism helps reduce complexity by allowing the same interface to be use to

access a general class of actions. It is the compiler's job to select the specific action (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the general interface.

Stack are like's an array
In c
Push-int(i)
Pop-int-(i)
In c++ simply push() and pop() function work for all type.

Inheritance:

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. If you think about it, most knowledge is made manageable by hierarchical classifications.
For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. As you will see, inheritance is an
important aspect of object-oriented programming.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on

Base & Derived Classes:
A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es).

A class derivation list names one or more base classes and has the form:
class derived-class: access-specifier base-class Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.
Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
```

```
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape
{
  public:
    int getArea()
    {
      return (width * height);
    }
};

int main(void)
{
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Total area: 35

Header:

To begin, the header **<iostream>** is included. This header supports C++-style I/O operations. (**<iostream>**is to C++ what **stdio.h** is to C.) Notice one other thing: there is no **.h** extension to the name **iostream**. The reason is that **<iostream>** is one of the new-style headers defined by Standard C++. New-style headers do not use the **.h** extension. The next line in the program is using namespace std;

This tells the compiler to use the **std** namespace. Namespaces are a recent addition to C++. **A namespace** creates a declarative region in which various program elements can be placed. Namespaces help in the organization of large programs.

The **using** statement informs the compiler that you want to use the **std** namespace. This is the namespace in which the entire Standard C++ library is declared. By using the **std** namespace you simplify access to the standard library.

Output:
In C++, the << has an expanded role. It is still the left shift operator, but when it is used as shown in this example, it is also an output operator. The word **cout** is an identifier that is linked to the screen. You can use **cout** and the **<<** to output any of the built-in data types, as well as strings of characters.

Input:
cin >> i;
In C++, the **>>** operator still retains its right shift meaning. However, when used as shown, it also is C++'s input operator.

Formal way for flashing buffer
Std:: cout<< ::std

Another terms:
In c++ variable can be declared any where unlike c program
C++ has a new data type that is bool which represent either true or false.

Object oriented Program (Oop):,

Classes
Class is a collection of data member and member function. Class is a user define data type

Classes are created using the keyword **class**. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.
A class declaration is similar syntactically to a structure. In Chapter 11, a simplified general form of a class declaration was shown. Here is the entire general form of a **class** declaration that does not inherit any other class.

```
class class-name {
private data and functions
access-specifier:
data and functions
access-specifier:
data and functions
// ...
access-specifier:
data and functions
} object-list;
```

The object-list is optional. If present, it declares objects of the class. Here, access-specifier is one of these three C++ keywords:

```
public
private
protected
```

C++ CLASS DEFINITIONS:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

```
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

Object is a class type variable. **Objects** are also called instance of the class. Each **object** contains all members(variables and functions) declared in the class.

Constructor: A class constructor is a special function in a class that is called when a new object of the class is created.
A constructor function is a special function that is a member of a class and has the same name as that class. For example, here is how the **stack** class looks when converted to use a constructor function for initialization:
// This creates the class stack.

```
class stack {
int stck[SIZE];
int tos;
public:

stack(); // constructor
void push(int i);
int pop();

};
```

Notice that the constructor **stack()** has no return type specified. In C++, constructor functions cannot return values and, thus, have no return type.

The **stack()** function is coded like this:

```
// stack's constructor function
stack::stack()
{
tos = 0;
cout << "Stack Initialized\n";
```

```
}
```

Destructo: A destructor is also a special function which is called when created object is deleted.

The complement of the constructor is the destructor. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called

Lecture-2 Date: 18.08.2016

Operator overloading:

```
#include <iostream>
using namespace std;
// abs is overloaded three ways

int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
cout << abs(-10) << "\n";
cout << abs(-11.0) << "\n";
cout << abs(-9L) << "\n";
return 0;
}

int abs(int i)
{
cout << "Using integer abs()\n";
return i<0 ? -i : i;
}

double abs(double d)
{
cout << "Using double abs()\n";
return d<0.0 ? -d : d;
}

long abs(long l)
{
cout << "Using long abs()\n";
```

```
return l<0 ? -l : l;
}

The output from this program is shown here.
Using integer abs()
10
Using double abs()
11
Using long abs()
```

This program creates three similar but different functions called **abs(),** each of which returns the absolute value of its argument. The compiler knows which function to call in each situation because of the type of the argument. The value of overloaded functions is that they allow related sets of functions to be accessed with a common name. Thus, the name **abs()** represents the general action that is being performed. It is left to the compiler to choose the right specific method for a particular circumstance. You need only remember the general action being performed. Due to polymorphism, three things to remember have been reduced to one. This example is fairly trivial, but if you expand the concept, you can see how polymorphism can help you manage very complex programs. In general, to overload a function, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction when overloading a function: the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types

Lecture-3 Date: 21.08.2016

General From of Class declaration:

Classes are created using the keyword class. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.

A class declaration is similar syntactically to a structure

```
class class-name {
private data and functions
access-specifier:

data and functions
access-specifier:

data and functions
// ...
access-specifier:
data and functions
} object-list;
```

The object-list is optional. If present, it declares objects of the class. Here,

Class specifier:

Access-specifier is one of these three C++ keywords:

public
private
protected

Private: Functions and data declared within a class are private to that class and may be accessed only by other members of the class.

A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

By default all the members of a class would be private, for example in the following class **width** is a private member, which means until you label a member, it will be assumed a private member:

```
class Box
{
  double width;
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );
};
```

Public: The public access specifier allows functions or data to be accessible to other parts of your program.
A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function

Protected: The protected access specifier is needed only when inheritance is involved . Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

Structures and Classes Are Related

Structures are part of the C subset and were inherited from the C language. the only difference between a **class** and a **struct** is that by default all members are public in a **struct** and private in a **class**. In all other respects, structures and classes are equivalent in C++, a structure defines a class typ

Structure Example:
which uses a structure to declare a class that controls access to a string:

```
// Using a structure to define a class.
#include <iostream>
#include <cstring>
using namespace std;

struct mystr {
     void buildstr(char *s); // public
    void showstr();
```

```
private: // now go private
        char str[255];
} ;
void mystr::buildstr(char *s)
{
if(!*s) *str = '\0'; // initialize string
else strcat(str, s);
}
void mystr::showstr()
{
cout << str << "\n";
}
int main()
{
mystr s;
s.buildstr(""); // init
s.buildstr("Hello ");
s.buildstr("there!");
s.showstr();
return 0;
}
```

This programdisplays the string **Hello there!**.
The class **mystr** could be rewritten by using **class** as shown here:

```
class mystr {
char str[255];
public:
void buildstr(char *s); // public
void showstr();
} ;
```

You might wonder why C++ contains the two virtually equivalent keywords **struct** and **class**. This seeming redundancy is justified for several reasons. First, there is no fundamental reason not to increase the capabilities of a structure. In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions. Second, because structures and classes are related, it may be easier to port existing C programs to C++. Finally, although **struct** and **class** are virtually equivalent today, providing two different keywords allows the definition of a **class** to be free to evolve. In order for C++ to remain compatible with C, the definition of **struct** must always be tied to its C definition.

Friend Functions :

A friend function that is a "friend" of a given class is allowed access to private and protected data in that class that it would not normally be able to as if the data was public .

Normally, A **function** that is defined outside of a class cannot access such information. It is possible to grant a nonmember function access to the private members of a class by using a **friend**.

A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**.

Friend Function Example:

Consider this program:

```
#include <iostream>
using namespace std;
class myclass {
        int a, b;
public:
        friend int sum(myclass x);
        void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}
        // Note: sum() is not a member function of any class.
int sum(myclass x)
{
        /* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}
int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

Lecture-4 Date: 23.08.2016

Friend class Example:

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

```
#include <iostream>
using namespace std;
class TwoValues {
        int a;
        Int b;
public:
TwoValues(int i, int j) { a = i; b = j; }
 friend class Min;
  };

class Min {
```

```
        public:
                int min(TwoValues x);
            };
int min(TwoValues x)
    {
     return x.a < x.b ? x.a : x.b;
     }
int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
 return 0;
}
```

class Min has access to the private variables a and b declared within the TwoValues class.
It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class.

Inline function:

There is an important feature in C++, called an inline function, that is commonly used with classes. you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called,precede its definition with the inline keyword.

Inline Function Example:

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
return a>b ? a : b;
}
int main()
{
cout << max(10, 20);
cout << " " << max(99, 88);
return 0;
}
```

As far as the compiler is concerned, the preceding program is equivalent to this one:

```
#include <iostream>
using namespace std;
  int main()
   {
    cout << (10>20 ? 10 : 20);
     cout << " " << (99>88 ? 99 : 88);
    }
```

Inline Class Member Example:

inline functions may be class member functions. For example, this is a perfectly

valid C++ program:

```
#include <iostream>
using namespace std;
class myclass {
                int a, b;
        public:
                void init(int i, int j);
                void show();
         };



// Create an inline function.
inline void myclass::init(int i, int j)
{
a = i;
b = j;
}
// Create another inline function.

inline void myclass::show()
{
cout << a << " " << b << "\n";
}

int main()
{
myclass x;
        x.init(10, 20);
        x.show();
return 0;
}
```

Lecture-5 Date: 28.08.2016

Parameterized Constructor:

It is possible to pass arguments to constructor functions. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

Example:

```
#include <iostream>
    using namespace std;
    class myclass {
                int a, b;
        public:
                myclass(int i, int j) {a=i; b=j;}
                void show() {cout << a << " " << b;}
            };
```

```
int main()
{
        myclass ob(3, 5);
        ob.show();
return 0;
}
```

Static Class Member:

Both function and data members of a class can be made static. This section explains the consequences of each.

Static data member:

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable All static variables are initialized to zero before the first object is created.

When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs.

To understand the usage and effect of a static data member, consider this program:

```
#include <iostream>
using namespace std;
class shared {
        static int a;
         int b;
    public:
        void set(int i, int j) {a=i; b=j;}
         void show();
} ;
int shared::a; // define a
void shared::show()
{
        cout << "This is static a: " << a;
        cout << "\nThis is non-static b: " << b;
        cout << "\n";
}
int main()
{
        shared x, y;
        x.set(1, 1); // set a to 1
        x.show();
        y.set(2, 2); // change a to 2
        y.show();
        x.show(); /* Here, a has been changed for both x and y
        because a is shared by both objects. */
return 0;
}
```

This program displays the following output when run.

```
static a: 1
non-static b: 1
static a: 2
non-static b: 2
static a: 2
non-static b: 1
```

Static member function:

Member functions may also be declared as static. There are several restrictions placed on static member functions. They may only directly refer to other static members of the class. (Of course, global functions and data may be accessed by static member functions.) A static member function does not have a this pointer.

There cannot be a static and a non-static version of the same function. A static member function may not be virtual. Finally, they cannot be declared as const or volatile.

For example that get_resource( ) is now declared as static.

illustrates, get_resource( ) may be called either by itself, independent of any object, by using the class name and the scope resolution operator, or in connection with an object.

```
#include <iostream>
using namespace std;
class cl {
        static int resource;
 public:
        static int get_resourc ();
         void free_resource() { resource = 0; }
};
int cl::resource; // define resource
int cl::get_resource()
{
          if(resource) return 0;  // resource already in use
else {
           resource = 1;
          return 1; // resource allocated to this object
}
}
int main()
{
cl ob1, ob2;
/* get_resource() is static so may be called independent
of any object. */
     if(cl::get_resource()) cout << "ob1 has resource\n";
     if(!cl::get_resource()) cout << "ob2 denied resource\n";
        ob1.free_resource();
       if(ob2.get_resource()) // can still call using object syntax
cout << "ob2 can now use resource\n";
```

```
return 0;
}
```

When Constructor and destructor are Execution:

As a general rule, an object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed. Precisely when these events occur is discussed here.

A local object's constructor function is executed when the object's declaration statement is encountered. The destructor functions for local objects are executed in the reverse order of the constructor functions.
Global objects have their constructor functions execute before main( ) begins execution. Global constructors are executed in order of their declaration, within the same file. You cannot know the order of execution of global constructors spread among several files. Global destructors execute in reverse order after main( ) has terminated.

Example:

```
#include <iostream>
using namespace std;
        class myclass {
                public:
                        int who;
                        myclass(int id);
                        ~myclass();
} glob_ob1(1), glob_ob2(2);
myclass::myclass(int id)
        {
                cout << "Initializing " << id << "\n";
               who = id;
}
 myclass::~myclass()
       {
            cout << "Destructing " << who << "\n";
}
int main()
{
myclass local_ob1(3);
                cout << "This will not be first line displayed.\n";
                myclass local_ob2(4);
return 0;
}
```

It displays this output:

```
                Initializing 1
                Initializing 2
                Initializing 3
This will not be first line displayed.
                Initializing 4
                Destructing 4
```

The scope resolution operator:

The :: operator links a class name with a member name in order to tell the compiler what class the member belongs to. However, the scope resolution operator has another related use: it can allow access to a name in an enclosing scope that is "hidden" by a local declaration of the same name.

Nested class

It is possible to define one class within another. Doing so creates a nested class. Since a class declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class. Frankly, nested classes are seldom used.

Local class

A class may be defined within a function. For example, this is a valid C++ program:

```
#include <iostream>
using namespace std;
        void f();
int main()
        {
            f();
              // myclass not known here
return 0;
}
void f()
{
  class myclass {
          int i;
        public:
           void put_i(int n) { i=n; }
int get_i() { return i; }
} ob;
ob.put_i(10);
cout << ob.get_i();
}
```

When a class is declared within a function, it is known only to that function and unknown outside of it.
Several restrictions apply to local classes. First, all member functions must be defined within the class declaration. The local class may not use or access local variables of the function in which it is declared (except that a local class has access

Passing object to function

Objects may be passed to functions in just the same way that any other type of variable can. Objects are passed to functions through the use of the standard call-by value mechanism. This means that a copy of an object is made when it is passed to a function. However, the fact that a copy is created means, in essence, that another object is created.

```
// Passing an object to a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
myclass(int n);
~myclass();
void set_i(int n) { i=n; }
```

```
    int get_i() { return i; }
    };
    myclass::myclass(int n)
    {
    i = n;
    cout << "Constructing " << i << "\n";
    }
    myclass::~myclass()
    {
    cout << "Destroying " << i << "\n";
    }
    void f(myclass ob);
int main()
{
myclass o(1);
f(o);
cout << "This is i in main: ";
cout << o.get_i() << "\n";
return 0;
}
void f(myclass ob)
{
ob.set_i(2);
cout << "This is local i: " << ob.get_i();
cout << "\n";
}
This program produces this output:
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying
```

Object return

A function may return an object to the caller. For example, this is a valid C++ program:

```
// Returning objects from a function.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
myclass f(); // return object of type myclass
int main()
{
myclass o;
o = f();
cout << o.get_i() << "\n";
```

```
return 0;
}
myclass f()
{
myclass x;
x.set_i(1);
return x;
}
```

When an object is returned by a function, a temporary object is automatically created that holds the return value. It is this object that is actually returned by the function. After the value has been returned, this object is destroyed. The destruction of this temporary object may cause unexpected side effects in some situations. For example, if the object returned by the function has a destructor that frees dynamically allocated memory, that memory will be freed even though the object that is receiving the return value is still using it.

Object assignment bit by bit

Assuming that both objects are of the same type, you can assign one object to another. This causes the data of the object on the right side to be copied into the data of the object on the left. For example, this program displays **99**:

```
// Assigning objects.
#include <iostream>
using namespace std;
class myclass {
int i;
public:
void set_i(int n) { i=n; }
int get_i() { return i; }
};
int main()
{
myclass ob1, ob2;
ob1.set_i(99);
ob2 = ob1; // assign data from ob1 to ob2
cout << "This is ob2's i: " << ob2.get_i();
return 0;
}
```

By default, all data from one object is assigned to the other by use of a bit-by-bit copy.

Chapter 13

Arrays of Objects:

The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
      return 0;
}
```

```
  #include <iostream>
  using namespace std;
  class cl {
  int i;
  public:
  cl(int j) { i=j; } // constructor
  int get_i() { return i; }
  };
  int main()
  {
  cl ob[3] = {1, 2, 3}; // initializers
  int i;
  for(i=0; i<3; i++)
  cout << ob[i].get_i() << "\n";
  return 0;
  }
```

Creating Initialized vs. Uninitialized Arrays:

A special case situation occurs if you intend to create both initialized and uninitialized arrays of objects. Consider the following **class.**

```
  class cl {
  int i;
  public:
  cl(int j) { i=j; }
  int get_i() { return i; }
  };
```

Here, the constructor function defined by **cl** requires one parameter. This implies that any array declared of this type must be initialized. That is, it precludes this array declaration:

```
cl a[9];
```

error constructor requires initializers

To solve this problem, you need to overload the constructor function, adding one that takes no parameters. In this way, arrays that are initialized and those that are not are both allowed.

```
class cl {
int i;
public:
cl() { i=0; } // called for non-initialized arrays
cl(int j) { i=j; } // called for initialized arrays
int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:

```
cl a1[3] = {3, 5, 6}; // initialized
cl a2[34]; // uninitialized
```

Pointers to Objects:

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (–>) operator instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

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

class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer.

For example, this program uses a pointer to access all three elements of array **ob** after being assigned **ob**'s starting address:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
```

```
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob[3] = {1, 2, 3};
cl *p;
int i;
p = ob; // get start of array
for(i=0; i<3; i++) {
cout << p->get_i() << "\n";
p++; // point to next object
}
return 0;
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

```
#include <iostream>
using namespace std;
class cl {
public:
int i;
cl(int j) { i=j; }
};
int main()
{
cl ob(1);
int *p;
p = &ob.i; // get address of ob.i
cout << *p; // access ob.i via p
return 0;
}
}
```

The this Pointer:

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**. To understand **this**, first consider a program that creates a class called **pwr** that computes the result of a number raised to some power:

```
#include <iostream>
using namespace std;
class pwr {
double b;
int e;
double val;
public:
pwr(double base, int exp);
double get_pwr() { return val; }
};
```

```
pwr::pwr(double base, int exp)
{
b = base;
e = exp;
val = 1;
if(exp==0) return;
for( ; exp>0; exp--) val = val * b;
}
int main()
{
pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
cout << x.get_pwr() << " ";
cout << y.get_pwr() << " ";
cout << z.get_pwr() << "\n";
return 0;
}
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **pwr(),** the statement b = base;
means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

this->b = base

The **this** pointer points to the object that invoked **pwr().** Thus, **this –>b** refers to that object's copy of **b**.
Here is the entire **pwr()** function written using the **this** pointer:

```
pwr::pwr(double base, int exp)
{
this->b = base;
this->e = exp;
this->val = 1;
if(exp==0) return;
for( ; exp>0; exp--)
this->val = this->val * this->b;
}
```

Remember that the **this** pointer is automatically passed to all member functions. Therefore, **get_pwr()** could also be rewritten as shown here:

double get_pwr() { return this->val; }

In this case, if **get_pwr()** is invoked like this:

y.get_pwr();

then **this** will point to object **y.**
Two final points about **this.** First, **friend** functions are not members of a class and, therefore, are not passed a **this** pointer. Second, **static** member functions do not have a **this** pointer.

Pointers to Derived Types:

In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes. To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B *** may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type **D *** may not point to an object of type **B**.

Here is a short program that illustrates the use of a base pointer to access derived objects.

```
#include <iostream>
using namespace std;
class base {
int i;
public:
void set_i(int num) { i=num; }
int get_i() { return i; }
};
class derived: public base {
int j;
public:
void set_j(int num) { j=num; }
int get_j() { return j; }
};
int main()
{
base *bp;
derived d;
bp = &d; // base pointer points to derived object
// access derived object using base pointer
bp->set_i(10);
cout << bp->get_i() << " ";
/* The following won't work. You can't access element of
a derived class using a base class pointer.
bp->set_j(88); // error
cout << bp->get_j(); // error
*/
return 0;
}
```

As you can see, a base pointer is used to access an object of a derived class. Although you must be careful, it is possible to cast a base pointer into a pointer of the derived type to access a member of the derived class through the base pointer. For example, this is valid C++ code:

```
// access now allowed because of cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

It is important to remember that pointer arithmetic is relative to the base type of the pointer.

References:

C++ CONTAINS A FEATURE THAT IS RELATED TO THE POINTER CALLED A REFERENCE. A REFERENCE IS ESSENTIALLY AN IMPLICIT POINTER. THERE ARE THREE WAYS THAT A REFERENCE CAN BE USED: AS A FUNCTION PARAMETER, AS A FUNCTION RETURN VALUE, OR AS A STAND-ALONE REFERENCE

Reference Parameters:

The most important use for a reference is to allow you to create functions that Automatically use call-by-reference parameter passing.
Arguments can be passed to functions in one of two ways:

- Using call-by-value or
- Call-by-reference.

Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing.

- First, you can explicitly pass a pointer to the argument.
- Second, you Call-by-reference passes the address of the argument to the function.

To fully understand what a reference parameter is and why it is valuable, we will begin by reviewing how a call-by-reference can be generated using a pointer parameter. The following program manually creates a call-by-reference parameter using a pointer in the function called neg(), which reverses the sign of the integer variable pointed to by its argument

Manually create a call-by-reference using a pointer.

```
#include <iostream>
using namespace std;
void neg(int *i);
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(&x);
cout << x << "\n";
return 0;
}
void neg(int *i)
{
*i = -*i;
}
```

In this program, neg() takes as a parameter a pointer to the integer whose sign it will reverse. Therefore, neg() must be explicitly called with the address of x. Further, inside neg() the * operator must be used to access the variable pointed to by i. This is how you generate a "manual" call-by-reference in C++, and it is the only way to obtain a call-by-reference using the C subset. Fortunately, in C++ you can automate this feature by using a reference parameter.

To create a reference parameter, precede the parameter's name with an &. For example, here is how to declare neg() with i declared as a reference parameter:

```
void neg(int &i);
```

```
// Use a reference parameter.
#include <iostream>
using namespace std;
void neg(int &i); // i now a reference
int main()
{
int x;
x = 10;
cout << x << " negated is ";
neg(x); // no longer need the & operator
cout << x << "\n";
return 0;
}
void neg(int &i)
{
i = -i; // i is now a reference, don't need *
}
```

Here is another example. This program uses reference parameters to swap the values of the variables it is called with. The swap() function is the classic example of call-by-reference parameter passing.

```
#include <iostream>
using namespace std;
void swap1(int  i, int  j);
void swap2(int &i, int &j);

int main()
{
int a, b, c, d;
a = 1;
b = 2;
c = 3;
d = 4;
cout << "a and b: " << a << " " << b << "\n";
swap1(a, b); // no & operator needed
cout << "a and b: " << a << " " << b << "\n";

cout << "c and d: " << c << " " << d << "\n";
swap2(c, d);
cout << "c and d: " << c << " " << d << "\n";
return 0;
}

void swap1(int  i, int  j)
{
int t;
t = i; // no * operator needed
i = j;
```

```
j = t;
}

void swap2(int &i, int &j)
{
int t;
t = i; // no * operator needed
i = j;
j = t;
}
```

This program displays the following:

```
a and b: 1 2
a and b: 1 2
c and d: 3 4
c and d: 4 3
```

Passing References to Objects:

when an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called. If for some reason you do not want the destructor function to be called, simply pass the object by reference.

When you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called. For example, try this program.

```
#include <iostream>
using namespace std;
class cl {
int id;
public:
int i;
cl(int i);
~cl();
void neg(cl &o) { o.i = -o.i; } // no temporary created
};
cl::cl(int num)
{
cout << "Constructing " << num << "\n";
id = num;
}
cl::~cl()
{
cout << "Destructing " << id << "\n";
```

```
}
int main()
{
cl o(1);
o.i = 10;
o.neg(o);
cout << o.i << "\n";
return 0;
}
```

Here is the output of this program:

```
Constructing 1
-10
Destructing 1
```

As you can see, only one call is made to cl's destructor function. Had o been passed by value, a second object would have been created inside neg(), and the destructor would have been called a second time when that object was destroyed at the time neg() terminated. As the code inside neg() illustrates, when you access a member of a class through a reference, you use the dot operator. The arrow operator is reserved for use with pointers only.

Returning References:

A function may return a reference. This has the rather startling effect of allowing a function to be used on the left side of an assignment statement! For example, consider this simple program.

```
#include <iostream>
using namespace std;
char &replace(int i); // return a reference
char s[80] = "Hello There";
int main()
{
replace(5) = 'X'; // assign X to space after Hello
cout << s;
return 0;
}
char &replace(int i)
{
return s[i];
}
```

This program replaces the space between Hello and There with an X. That is, the program displays HelloXthere. Take a look at how this is accomplished.

First, replace() is declared as returning a reference to a character. As replace() is coded, it returns a reference to the element of s that is specified by its argument i. The reference returned by replace() is then used in main() to assign to that element the character X.

Independent References:

By far the most common uses for references are to pass an argument using call-by-reference and to act as a return value from a function. However, you can declare a reference that is simply a variable. This type of reference is called an independent reference.

When you create an independent reference, all you are creating is another name for an object variable. All independent references must be initialized when they are created. The reason for this is easy to understand. Aside from initialization, you cannot change what object a reference variable points to. Therefore, it must be initialized when it is declared. (In C++, initialization is a wholly separate operation from assignment.)

The following program illustrates an independent reference:
Chapter 13: Arrays, Pointers, References, and the Dynamic Allocation Operators 347

```
#include <iostream>
using namespace std;
int main()
{
int a;
int &ref = a; // independent reference
a = 10;
cout << a << " " << ref << "\n";
ref = 100;
cout << a << " " << ref << "\n";
int b = 19;
ref = b; // this puts b's value into a
cout << a << " " << ref << "\n";
ref--; // this decrements a
// it does not affect what ref refers to
cout << a << " " << ref << "\n";
return 0;
}
```

The program displays this output:

```
10 10
100 100
19 19
18 18
```

Actually, independent references are of little real value because each one is, literally, just another name for another variable. Having two names to describe the same object is likely to confuse, not organize, your program.

References to Derived Types:

Similar to the situation as described for pointers earlier, a base class reference can be used to refer to an object of a derived class. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

Restrictions to References:

There are a number of restrictions that apply to references. You cannot reference another reference. Put differently, you cannot obtain the address of a reference. You cannot create arrays of

references. You cannot create a pointer to a reference. You cannot reference a bit-field. A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value. Null references are prohibited.

C++'s Dynamic Allocation Operators:

C++ provides two dynamic allocation operators: new and delete. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part of almost all real-world programs. As explained in Part One, C++ also supports dynamic memory allocation functions, called malloc() and free(). These are included for the sake of compatibility with C. However, for C++ code, you should use the new and delete operators because they have several advantages.

The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new. The general forms of new and delete are shown here:

p_var = new type; delete p_var;

Here, p_var is a pointer variable that receives a pointer to memory that is large enough to hold an item of type type. Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then new will fail and a bad_alloc exception will be generated. This exception is defined in the header <new>. Your program should handle this exception and take appropriate action if a failure occurs. (Exception handling is described in Chapter 19.) If this exception is not handled by your program, then your program will be terminated.

The actions of new on failure as just described are specified by Standard C++. The trouble is that not all compilers, especially older ones, will have implemented new in compliance with Standard C++. When C++ was first invented, new returned null on failure. Later, this was changed such that new caused an exception on failure. Finally, it was decided that a new failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, new has been implemented differently, at different times, by compiler manufacturers. Although all compilers will eventually implement new in compliance with Standard C++, currently the only way to know the precise action of new on failure is to check your compiler's documentation.

Since Standard C++ specifies that new generates an exception on failure, this is the way the code in this book is written. If your compiler handles an allocation failure differently, you will need to make the appropriate changes. Here is a program that allocates memory to hold an integer:

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try { p = new int; // allocate space for an int
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
*p = 100;
cout << "At " << p << " ";
```

```
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```

This program assigns to p an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements new such that it returns null on failure, you must change the preceding program appropriately.

The delete operator must be used only with a valid pointer previously allocated by using new. Using any other type of pointer with delete is undefined and will almost certainly cause serious problems, such as a system crash. Although new and delete perform functions similar to malloc() and free(), they have several advantages. First, new automatically allocates enough memory to hold an object of the specified type. You do not need to use the sizeof operator. Because the size is computed automatically, it eliminates any possibility for error in this regard. Second, new automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using malloc(). Finally, both new and delete can be overloaded, allowing you to create customized allocation systems. Although there is no formal rule that states this, it is best not to mix new and delete with malloc() and free() in the same program. There is no guarantee that they are mutually compatible.

The nothrow Alternative:

it is possible to have **new** return **null** instead of throwing an exception when an allocation failure occurs. This form of **new** is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to **malloc()** with **new**. (This is common when updating C code to C++.) This form of **new** is shown here:

p_var = new(nothrow) *type*;

Here, *p_var* is a pointer variable of *type*. The **nothrow** form of **new** works like the original version of **new** from years ago. Since it returns null on failure, it can be "dropped into" older code without having to add exception handling. However, for new code, exceptions provide a better alternative. To use the **nothrow** option, you must include the header <**new**>.

The following program shows how to use the **new(nothrow)** alternative.

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p, i;
p = new(nothrow) int[32]; // use nothrow option
if(!p) {
cout << "Allocation failure.\n";
return 1;
}
for(i=0; i<32; i++) p[i] = i;
for(i=0; i<32; i++) cout << p[i] << " ";
delete [] p; // free the memory
return 0;
}
```

As this program demonstrates, when using the **nothrow** approach, you must check the pointer returned by **new** after each allocation request.

Chapter 14

Function Overloading :

Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation.

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in types of parameters
double myfunc(double i);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(5.4); // calls myfunc(double i)
return 0;
}
double myfunc(double i)
{
return i;
}
int myfunc(int i)
{
return i;
}
```

The next program overloads **myfunc()** using a different number of parameters:

```
#include <iostream>
using namespace std;
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
int main()
{
cout << myfunc(10) << " "; // calls myfunc(int i)
cout << myfunc(4, 5); // calls myfunc(int i, int j)
return 0;
}
int myfunc(int i)
{
return i;
}
int myfunc(int i, int j)
{
return i*j;
}
```

As mentioned, the key point about function overloading is that the functions must differ in regard to the types and/or number of parameters. Two functions differing only in their return types cannot be overloaded. For example, this is an invalid attempt
to overload **myfunc():**

```
int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

Sometimes, two function declarations will appear to differ, when in fact they do not. For example, consider the following declarations.

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]
```

Overloading Constructor Functions :

Constructor functions can be overloaded; in fact, overloaded constructors are very common. **There are three main reasons why you will want to overload a constructor function: to gain flexibility, to allow both initialized and uninitialized objects to be created, and to define copy constructors Overloading a Constructor to Gain Flexibility Many times you will create a class for which there are two or more possible ways to construct an object**. In these cases, you will want to provide an overloaded constructor function for each way. This is a self-enforcing rule because if you attempt to create an object for which there is no matching constructor, a compile-time error results.

The user is free to choose the best way to construct an object given the specific circumstance. Consider this program that creates a class called **date**, which holds a calendar date.

```
#include <iostream>
#include <cstdio>
using namespace std;
class date {
int day, month, year;
public:
date(char *d);
date(int m, int d, int y);
void show_date();
};
// Initialize using string.
date::date(char *d)
{
scanf(d, "○%d%*c%d%*c%d", &month, &day, &year);
}
// Initialize using integers.
date::date(int m, int d, int y)
{
day = d;
month = m;
year = y;
}
void date::show_date()
{
cout << month << "/" << day;
cout << "/" << year << "\n";
```

```
}
int main()
{
date ob1(12, 4, 2001), ob2("10/22/2001");
ob1.show_date();
ob2.show_date();
return 0;
}
```

Allowing Both Initialized and Uninitialized Objects Another common reason constructor functions are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created. This is especially important if you want to be able to create dynamic arrays of objects of some class, since it is not possible to initialize a dynamically allocated array. To allow uninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not. initialized and the other is not. It also dynamically allocates an array.

```
#include <iostream>
#include <new>
using namespace std;
class powers {
int x;
public:
// overload constructor two ways
powers() { x = 0; } // no initializer
powers(int n) { x = n; } // initializer
int getx() { return x; }
void setx(int i) { x = i; }
};
int main()
{
powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
powers ofThree[5]; // uninitialized
powers *p;
int i;
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {
cout << ofTwo[i].getx() << " ";
}
cout << "\n\n";
// set powers of three
ofThree[0].setx(1);
ofThree[1].setx(3);
ofThree[2].setx(9);
ofThree[3].setx(27);
ofThree[4].setx(81);
// show powers of three
```

```
cout << "Powers of three: ";
for(i=0; i<5; i++) {
cout << ofThree[i].getx() << " ";
}
cout << "\n\n";
// dynamically allocate an array
try {
p = new powers[5]; // no initialization
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
// initialize dynamic array with powers of two
for(i=0; i<5; i++) {
p[i].setx(ofTwo[i].getx());
}
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) {
cout << p[i].getx() << " ";
}
cout << "\n\n";
delete [] p;
return 0;
}
```

Copy Constructors :

One of the more important forms of an overloaded constructor is the copy constructor. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another.

when one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created.

For example, assume a class called MyClass that allocates memory for each object when it is created, and an object A of that class. This means that A has already allocated its memory. Further, assume that A is used to initialize B, as shown here:

MyClass B= A;

If a bitwise copy is performed, then B will be an exact copy of A. This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if MyClass includes a destructor that frees the memory, then the same piece of memory will be freed twice when A and B are destroyed! The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances. To solve the type

of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. When a copy constructor exists, the default, bitwise copy is bypassed.

The most common generalform of a copy constructor is

```
classname (const classname &o) {
// body of constructor
}
```

Here, o is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them.
The first is assignment. The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing x
func(y); // y passed as a parameter
y = func(); // y receiving a temporary, return object
```

Finding the Address of an Overloaded Function :

If **myfunc()** is not overloaded, there is one and only one function called **myfunc(),** and the compiler has no difficulty assigning its address to **p**. However, if **myfunc()** is overloaded, how does the compiler know which version's address to assign to **p**? The answer is that it depends upon how **p** is declared. For example, consider this program:

```
#include <iostream>
using namespace std;
int myfunc(int a);
int myfunc(int a, int b);
int main()
{
int (*fp)(int a); // pointer to int f(int)
fp = myfunc; // points to myfunc(int)
cout << fp(5);
return 0;
}
int myfunc(int a)
{
return a;
}
int myfunc(int a, int b)
{
```

```
return a*b;
}
```

Here, there are two versions of **myfunc()** . Both return **int**, but one takes a single integer argument; the other requires two integer arguments. In the program, **fp** is declared as a pointer to a function that returns an integer and that takes one integer argument. When **fp** is assigned the address of **myfunc()** , C++ uses this information to select the **myfunc(int a)** version of **myfunc().** Had **fp** been declared like this:

```
int (*fp)(int a, int b);
```

Default Function Arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. For example, this declares **myfunc()** as taking one **double** argument with a default value of 0.0:

```
void myfunc(double d = 0.0)
{
// ...
}
```

Now, **myfunc()** can be called one of two ways, as the following examples show:

```
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

The first call passes the value 198.234 to **d**. The second call automatically gives **d** the default value zero.

One reason that default arguments are included in C++ is because they provide another method for the programmer to manage greater complexity. To handle the widest variety of situations, quite frequently a function contains more parameters than are required for its most common usage. Thus, when the default arguments apply, you need specify only the arguments that are meaningful to the exact situation, not all those needed by the most general case. For example, many of the C++ I/O functions make use of default arguments for just this reason. A simple illustration of how useful a default function argument can be is shown by the **clrscr()** function in the following program. The **clrscr()** function clears the screen by outputting a series of linefeeds (not the most efficient way, but sufficient for this example). Because a very common video mode displays 25 lines of text, the default argument of 25 is provided. However, because some terminals can display more or less than 25 lines (often depending upon what type of video mode is used), you can override the default argument by specifying one explicitly.

```
#include <iostream>
using namespace std;
void clrscr(int size=25);
int main()
{
register int i;
for(i=0; i<30; i++ ) cout << i << endl;
cin.get();
clrscr(); // clears 25 lines
for(i=0; i<30; i++ ) cout << i << endl;
cin.get();
clrscr(10); // clears 10 lines
return 0;
```

```
}
void clrscr(int size)
{
for(; size; size--) cout << endl;
}
```

Default Arguments vs. Overloading :

In some situations, default arguments can be used as a shorthand form of function overloading. The **cube** class's constructor just shown is one example. Let's look at another. Imagine that you want to create two customized versions of the standard **strcat()** function. The first version will operate like **strcat()** and concatenate the entire contents of one string to the end of another.

```
void mystrcat(char *s1, char *s2, int len);
void mystrcat(char *s1, char *s2);
```

The first version would copy **len** characters from **s2** to the end of **s1**. The second version would copy the entire string pointed to by **s2** onto the end of the string pointed to by **s1** and would operate like **strcat()**

Using Default Arguments Correctly Although default arguments can be a very powerful tool when used correctly, they can also be misused. The point of default arguments is to allow a function to perform its job in an efficient, easy-to-use manner while still allowing considerable flexibility. Toward this end, all default arguments should reflect the way a function is generally used, or a reasonable alternate usage. When there is no single value that is normally associated with a parameter, there is no reason to declare a default argument. In fact, declaring default arguments when there is insufficient basis for doing so destructures your code, because they are liable to mislead and confuse anyone reading your program.

```
// A customized version of strcat().
#include <iostream>
#include <cstring>
using namespace std;
void mystrcat(char *s1, char *s2, int len = -1);
int main()
{
char str1[80] = "This is a test";
char str2[80] = "0123456789";
mystrcat(str1, str2, 5); // concatenate 5 chars
cout << str1 << '\n';
strcpy(str1, "This is a test"); // reset str1
mystrcat(str1, str2); // concatenate entire string
cout << str1 << '\n';
return 0;
}
// A custom version of strcat().
void mystrcat(char *s1, char *s2, int len)
{
// find end of s1
while(*s1) s1++;
if(len == -1) len = strlen(s2);
```

```
while(*s2 && len) {
*s1 = *s2; // copy chars
s1++;
s2++;
len--;
}
*s1 = '\0'; // null terminate s1
}
```

Here, **mystrcat()** concatenates up to **len** characters from the string pointed to by **s2** onto the end of the string pointed to by **s1**. However, if **len** is –1, as it will be when it is allowed to default, **mystrcat()** concatenates the entire string pointed to by **s2** onto **s1**.
(Thus, when **len** is –1, the function operates like the standard **strcat()** function.) By using a default argument for **len**, it is possible to combine both operations into one function. In this way, default arguments sometimes provide an alternative to function
overloading.

Function Overloading and Ambiguity :

You can create a situation in which the compiler is unable to choose between two (or more) overloaded functions. When this happens, the situation is said to be ambiguous.Ambiguous statements are errors, and programs containing ambiguity will not compile.
By far the main cause of ambiguity involves C++'s automatic type conversions. Asyou know, C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function. For example, consider thisfragment:

```
int myfunc(double d);
// ...
cout << myfunc('c'); // not an error, conversion applied
```

As the comment indicates, this is not an error because C++ automatically converts the character **c** into its **double** equivalent. In C++, very few type conversions of this sort are actually disallowed. Although automatic type conversions are convenient, they are also a prime cause of ambiguity. For example, consider the following program:

```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);
int main()
{
cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
cout << myfunc(10); // ambiguous
return 0;
}
float myfunc(float i)
{
return i;
}
double myfunc(double i)
{
return -i;
```

```
}
```

Here, **myfunc()** is overloaded so that it can take arguments of either type **float** or type **double**. In the unambiguous line, **myfunc(double)** is called because, unless explicitly specified as **float**, all floating-point constants in C++ are automatically of type **double**.

Chapter 15

Operator Overloading

An operator function is created using the keyword operator. Operator functions can be either members or non-members of a class. Nonmember operator functions are almost always friend functions of the class

Creating a Member Operator Function:

A member operator function takes this general form:

ret-type class-name::operator#(arg-list)
{
// operations
}

Operator Overloading Often, operator functions return an object of the class they operate on, but ret-type can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #.

For example, if you are overloading the / operator, use operator/. When you are overloading a unary operator, arg-list will be empty. When you are overloading binary operators, arg-list will contain one parameter.

First example of operator overloading.

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
};
// Overload + for loc.
loc loc::operator+(loc op2)
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
```

```
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}
```

- **operator+()** has only one parameter even though it overloads the binary **+** operator.
- The reason that **operator+()** takes only one parameter is that the operand on the left side of the **+** is passed implicitly to the function through
- the **this** pointer.
- The operand on the right is passed in the parameter **op2**.
- The fact that the left operand is passed using **this** also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function.
- it is common for an overloaded operator function to return an object of the class it operates upon.

(ob1+ob2).show(); // displays outcome of ob1+ob2 In this situation, **ob1+ob2** generates a temporary object that ceases to exist after the call to **show()** terminates.

It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates. One last point about the **operator+()** function: It does not modify either operand. Because the traditional use of the **+** operator does not modify either operand, it makes sense for the overloaded version not to do so either.

The next program adds three additional overloaded operators to the **loc** class: the **–,** the **=,** and the unary **++.**

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;

public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Overload + for loc.
loc loc::operator+(loc op2)
```

```
{
loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload prefix ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
ob1.show();
ob2.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob1.show(); // displays 12 22
ob2.show(); // displays 12 22
ob1 = ob2 = ob3; // multiple assignment
ob1.show(); // displays 90 90
ob2.show(); // displays 90 90
return 0;
}
```

First, examine the **operator–()** function. Notice the order of the operands in the subtraction. In keeping with the meaning of subtraction, the operand on the right side of the minus sign is subtracted from the operand on the left. Because it is the object on the left that generates the call to the **operator–()** function, **op2**'s data must be

Creating Prefix and Postfix Forms of the Increment and Decrement Operators

Standard C++ allows you to explicitly create separate prefix and postfix versions of increment or decrement operators. To accomplish this, you must define two versions of the **operator++()** function. One is defined as shown in the foregoing program. The other is declared like this:
loc operator++(int x); If the **++** precedes its operand, the **operator++()** function is called. If the **++** follows its operand, the **operator++(int x)** is called and **x** has the value zero. The preceding example can be generalized. Here are the general forms for the
prefix and postfix **++** and **– –** operator functions

Here are the general forms for the prefix and postfix **++** and **– –** operator functions.

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}
// Postfix increment
type operator++(int x) {
// body of postfix operator
}
// Prefix decrement
type operator– –( ) {
// body of prefix operator
}
// Postfix decrement
type operator– –(int x) {
// body of postfix operator
}
```

Overloading the Shorthand Operators

You can overload any of C++'s "shorthand" operators, such as **+=, –=,** and the like. For example, this function overloads **+=** relative to **loc**:

```
loc loc::operator+=(loc op2)
{
longitude = op2.longitude + longitude;
latitude = op2.latitude + latitude;
return *this;
}
```

When overloading one of these operators, keep in mind that you are simply combining an assignment with another type of operation.

Operator Overloading Restrictions

There are some restrictions that apply to operator overloading.
You cannot alter the precedence of an operator.
You cannot change the number of operands that an operator takes.

(You can choose to ignore an operand, however.) Except for the function call

Note
operator (described later), operator functions cannot have default arguments. Finally,these operators cannot be overloaded:
. : : .* ?

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a **friend** function is not a member of the class, it does not have a **this** pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter.

In this program, the **operator+()** function is made into a friend:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
friend loc operator+(loc op1, loc op2); // now a friend
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
loc temp;
temp.longitude = op1.longitude + op2.longitude;
temp.latitude = op1.latitude + op2.latitude;
return temp;
}
// Overload - for loc.
loc loc::operator-(loc op2)
{
```

```
loc temp;
// notice order of operands
temp.longitude = longitude - op2.longitude;
temp.latitude = latitude - op2.latitude;
return temp;
}
// Overload assignment for

loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Overload ++ for loc.
loc loc::operator++()
{
longitude++;
latitude++;
return *this;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1 = ob1 + ob2;
ob1.show();
return 0;
}
```

There are some restrictions that apply to friend operator functions. First, you may not overload the **=, ( ), [ ],** or **–>** operators by using a friend function. Second, as explained in the next section, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Using a Friend to Overload ++ or – –

If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter. his is because friend functions do not have **this** pointers. Assuming that you stay true to the original meaning of the **++** and – – operators, these operations imply the modification of the operand they operate upon. However, if you overload these operators by using a friend, then the operand is passed by value as a parameter. This means that a friend operator function has no way to modify the operand. Since the friend operator function is not passed a **this** pointer to the operand, but rather a copy of the operand, no changes made to that
parameter affect the operand that generated the call. However, you can remedy this situation by specifying the parameter to the friend operator function as a reference parameter. This causes any changes made to the parameter inside the function to affect the operand that generated the call. For example, this program uses friend functions to overload the prefix versions of **++** and **– –** operators relative to the **loc** class:

```
#include <iostream>
using namespace std;
class loc {
int longitude, latitude;
```

```
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator=(loc op2);
friend loc operator++(loc &op);
friend loc operator--(loc &op);
};
// Overload assignment for loc.
loc loc::operator=(loc op2)
{
longitude = op2.longitude;
latitude = op2.latitude;
return *this; // i.e., return object that generated call
}
// Now a friend; use a reference parameter.
loc operator++(loc &op)
{
op.longitude++;
op.latitude++;
return op;
}
// Make op-- a friend; use reference.
loc operator--(loc &op)
{
op.longitude--;
op.latitude--;
return op;
}
int main()
{
loc ob1(10, 20), ob2;
ob1.show();
++ob1;
ob1.show(); // displays 11 21
ob2 = ++ob1;
ob2.show(); // displays 12 22
--ob2;
ob2.show(); // displays 11 21
return 0;
}
```

Overloading new and delete

It is possible to overload **new** and **delete**. You might choose to do this if you want to use some special allocation method. For example, you may want allocation routines that automatically begin using a disk file as virtual memory when the heap has been exhausted. Whatever the reason, it is a very simple matter to overload these operators.

The skeletons for the functions that overload **new** and **delete** are shown here:

```
// Allocate an object.
void *operator new(size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor called automatically. */
return pointer_to_memory;
}
// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */
}
```

The type **size_t** is a defined type capable of containing the largest single piece of memory that can be allocated. (**size_t** is essentially an unsigned integer. The parameter **size** will contain the number of bytes needed to hold the object being allocated. This is the amount of memory that your version of **new** must allocate. The overloaded **new** function must return a pointer to the memory that it allocates, or throw a **bad_alloc** exception if an allocation error occurs. Beyond these constraints, the overloaded **new** function can do anything else you require. When you allocate an object using **new** (whether your own version or not), the object's constructor isautomatically called. The **delete** function receives a pointer to the region of memory to be freed. It then releases the previously allocated memory back to the system. When an object is deleted, its destructor function is automatically called. vThe **new** and **delete** operators may be overloaded globally so that all uses of these operators call your custom versions. They may also be overloaded relative to one or more classes. Lets begin with an example of overloading **new** and **delete** relative to a class. For the sake of simplicity, no new allocation scheme will be used. Instead, the overloaded operators will simply invoke the standard library functions **malloc()** and **free().** (In your own application, you may, of course, implement any alternative allocation scheme you like.) To overload the **new** and **delete** operators for a class, simply make the overloaded operator functions class members. For example, here the **new** and **delete** operators are

overloaded for the **loc** class**:**

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
int longitude, latitude;
public:
loc() {}
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
```

```
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
void *operator new(size_t size);
void operator delete(void *p);
};
// new
{
void *p;
cout << "In overloaded new.\n";
p = malloc(size);
if(!p) {
bad_alloc ba;
throw ba;
}
return p;
}
// delete overloaded relative to loc.
void loc::operator delete(void *p)
{
cout << "In overloaded delete.\n";
free(p);
}
int main()
{
loc *p1, *p2;
try {
p1 = new loc (10, 20);
} catch (bad_alloc xa) {
cout << "Allocation error for p1.\n";
return 1;
}
try {
p2 = new loc (-10, -20);
} catch (bad_alloc xa) {
cout << "Allocation error for p2.\n";
return 1;;
}
p1->show();
p2->show();
delete p1;
c e
delete p2;
return 0;
}
```

Output from this program is shown here.

```
In overloaded new.
```

```
In overloaded new.
10 20
-10 -20
In overloaded delete.
In overloaded delete.
```

.

Overloading the nothrow Version of new and delete

You can also create overloaded **nothrow** versions of **new** and **delete**. To do so, use these skeletons.

```
// Nothrow version of new.
void *operator new(size_t size, const nothrow_t &n)
{
// Perform allocation.
if(success) return pointer_to_memory;
else return 0;
}
// Nothrow version of new for arrays.
void *operator new[](size_t size, const nothrow_t &n)
{
// Perform allocation.
if(success) return pointer_to_memory;
else return 0;
}
void operator delete(void *p, const nothrow_t &n)
{
// free memory
}
void operator delete[](void *p, const nothrow_t &n)
{
// free memory
}
```

The type **nothrow_t** is defined in **<new>**. This is the type of the **nothrow** object. The **nothrow_t** parameter is unused

Chapter 16

Inheritance is one of the corner stones of OOP because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items.

a class that is inherited is referred to as a base class. The class that does the inheriting is called the derived class. Further, a derived class can be used as a base class for another derived class.

Base-Class Access Control:

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name {
// body of class
};
```

The base-class access specifier must be either **public, private,** or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier. When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class. In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class. For example, as illustrated in this program, objects of type **derived** can directly access the public members of **base**:

```
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // access member of base
ob.show(); // access member of base
ob.showk(); // uses member of derived class
return 0;
}
```

When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class. For example, the following program will not even compile because both **set()** and **show()**
are now private elements of **derived**:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n";}
};
// Public elements of base are private in derived.
class derived : private base {
int k;

public:
derived(int x) { k=x; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob(3);
ob.set(1, 2); // error, can't access set()
ob.show(); // error, can't access show()
return 0;
}
```

When a base class' access specifier is **private,** public and protected members of thebase become private members of the derived class. This means that they are still accessible by members of the derived class but cannot be accessed by parts of your
program that are not members of either the base or derived class.

Inheritance and protected Members

The **protected** keyword is included in C++ to provide greater flexibility in the inheritance mechanism. When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program. With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to
this is when a protected member is inherited. In this case, a protected member differs substantially from a private one.

protected members behave differently. If the base class is inherited as **public**, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class.

```
#include <iostream>
using namespace std;
class base {
```

```
Remember
protected:
int i, j; // private to base, but accessible by derived
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
class derived : public base {
int k;
public:
// derived may access base's i and j
void setk() { k=i*j; }
void showk() { cout << k << "\n"; }
};
int main()
{
derived ob;
ob.set(2, 3); // OK, known to derived
ob.show(); // OK, known to derived
ob.setk();
ob.showk();
return 0;
}
```

In this example, because **base** is inherited by **derived** as **public** and because **i** and **j** are declared as **protected, derived**'s function **setk()** may access them. If **i** and **j** had been declared as **private** by **base**, then **derived** would not have access to them, and the program would not compile. When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class. For example, this program is correct, and **derived2** does indeed have access to **i** and **j**.

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
// i and j inherited as protected.
class derived1 : public base {
int k;
public:
void setk() { k = i*j; } // legal
void showk() { cout << k << "\n"; }
};
// i and j inherited indirectly through derived1.
class derived2 : public derived1 {
int m;
```

```
public:
void setm() { m = i-j; } // legal
void showm() { cout << m << "\n"; }
};
int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(2, 3);
ob1.show();
ob1.setk();
ob1.showk();
ob2.set(3, 4);
ob2.show();
ob2.setk();
ob2.setm();
ob2.showk();
ob2.showm();
return 0;
```

If, however, **base** were inherited as **private**, then all members of **base** would become private members of **derived1**, which means that they would not be accessible by **derived2**. (However, **i** and **j** would still be accessible by **derived1**.) This situation is illustrated by the following program, which is in error (and won't compile). The comments describe each error:

```
// This program won't compile.
#include <iostream>
using namespace std;
class base {
protected:
int i, j;
public:
void set(int a, int b) { i=a; j=b; }
void show() { cout << i << " " << j << "\n"; }
};
// Now, all elements of base are private in derived1.
class derived1 : private base {
int k;
public:
// this is legal because i and j are private to derived1
void setk() { k = i*j; } // OK
void showk() { cout << k << "\n"; }
};
// Access to i, j, set(), and show() not inherited.
class derived2 : public derived1 {
int m;
public:
// illegal because i and j are private to derived1
void setm() { m = i-j; } // Error
void showm() { cout << m << "\n"; }
};
```

```
int main()
{
derived1 ob1;
derived2 ob2;
ob1.set(1, 2); // error, can't use set()
ob1.show(); // error, can't use show()
ob2.set(3, 4); // error, can't use set()
ob2.show(); // error, can't use show()
return 0;
}
```

Even though **base** is inherited as **private** by **derived1, derived1** still has access to **base**'s **public** and **protected** elements. However, it cannot pass along this privilege.

Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructor functions that require arguments. In cases where only the derived class' constructor requires one or more parameters,

The general form
of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}
```

Here, *base1* through *baseN* are the names of the base classes inherited by the derived class.

Consider this program:

```
#include <iostream>
using namespace std;
class base {
protected:
int i;
public:
base(int x) { i=x; cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
int j;
public:
// derived uses x; y is passed along to base.
derived(int x, int y): base(y)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << "\n"; }
```

```
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 4 3
return 0;
}
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**. However, **derived()** uses only **x; y** is passed along to **base().**

Any parameters required by the base class are passed to it in the base class' argument list specified after the colon.
Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class requires it. In this situation, the arguments passed to the derived class are simply passed along to the base. For example, in this program, the derived class' constructor
takes no arguments, but **base1()** and **base2()** do:

```
#include <iostream>
```

```
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
/* Derived constructor uses no parameter,
but still must be declared as taking them to
pass them along to base classes.
*/
derived(int x, int y): base1(x), base2(y)
{ cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4);
ob.show(); // displays 3 4
return 0;
}
```

A derived class' constructor function is free to make use of any and all parameters that it is declared as taking, even if one or more are passed along to a base class. Put differently, passing an argument along to a base class does not preclude its use by the derived class as well. For example, this fragment is perfectly valid:

```
class derived: public base {
int j;
public:
// derived uses both x and y and then passes them to base.
derived(int x, int y): base(x, y)
{ j = x*y; cout << "Constructing derived\n"; }
```

Granting Access

When a base class is inherited as **private**, all public and protected members of that class ecome private members of the derived class. However, in certain circumstances, you may want to restore one or more inherited members to their original access specification.

For example, you might want to grant certain public members of the base class public status in the derived class even though the base class is inherited as **private**.

two ways to accomplish this.
First, you can use a **using** statement, which is the preferred way. The **using** statement is designed primarily to support namespaces .

The second way to restore an inherited member's access specification is to employ an *access declaration* within the derived class.

An access declaration takes this general form:

base-class::member

The access declaration is put under the appropriate access heading in the derived class' declaration.

To see how an access declaration works, let's begin with this short fragment:

```
class base {
public:
int j; // public in base
};
// Inherit base as private.
class derived: private base {
public:
// here is access declaration
base::j; // make j public again
.
.
.
};
```

Because **base** is inherited as **private** by **derived**, the public member **j** is made a private member of **derived**. However, by including

```
base::j;
```

Virtual Base Classes

An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:

```
// This program contains an error and will not compile.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
```

```
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{

derived3 ob;
ob.i = 10; // this is ambiguous, which i???
ob.j = 20;
ob.k = 30;
// i ambiguous here, too
ob.sum = ob.i + ob.j + ob.k;
// also ambiguous, which i?
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As the comments in the program indicate, both **derived1** and **derived2** inherit **base**. However, **derived3** inherits both **derived1** and **derived2**. This means that there are two copies of **base** present in an object of type **derived3**. Therefore, in an expression like ob.i = 10;
which **i** is being referred to, the one in **derived1** or the one in **derived2?** Because there are two copies of **base** present in object **ob**, there are two **ob.i**s! As you can see, the statement is inherently ambiguous. There are two ways to remedy the preceding program. The first is to apply the scope resolution operator to **i** and manually select one **i.** For example, this version of the program does compile and run as expected:

```
// This program uses explicit scope resolution to select i.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base.
class derived1 : public base {
public:
int j;
};
// derived2 inherits base.
class derived2 : public base {
public:
int k;
};
```

```
/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
int sum;
};
int main()
{
derived3 ob;
ob.derived1::i = 10; // scope resolved, use derived1's i
ob.j = 20;
ob.k = 30;
// scope resolved
ob.sum = ob.derived1::i + ob.j + ob.k;
// also resolved here
cout << ob.derived1::i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

As you can see, because the **::** was applied, the program has manually selected **derived1**'s version of **base**.

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;
class base {
public:
int i;
};
// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
int j;
};
// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
int k;
};
/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
```

```
int sum;
};
int main()
{
derived3 ob;
ob.i = 10; // now unambiguous
ob.j = 20;
ob.k = 30;
// unambiguous
ob.sum = ob.i + ob.j + ob.k;
// unambiguous
cout << ob.i << " ";
cout << ob.j << " " << ob.k << " ";
cout << ob.sum;
return 0;
}
```

Protected Base-Class Inheritance

It is possible to inherit a base class as **protected.** When this is done, all public and protected members of the base class become protected members of the derived class.

For example,

```
#include <iostream>
using namespace std;
class base {
protected:
int i, j; // private to base, but accessible by derived
public:
void setij(int a, int b) { i=a; j=b; }
void showij() { cout << i << " " << j << "\n"; }
};
// Inherit base as protected.
class derived : protected base{
int k;
public:
// derived may access base's i and j and setij().
void setk() { setij(10, 12); k = i*j; }
// may access showij() here
void showall() { cout << k << " "; showij(); }
};
int main()
ote
{
derived ob;
// ob.setij(2, 3); // illegal, setij() is
// protected member of derived
ob.setk(); // OK, public member of derived
ob.showall(); // OK, public member of derived
// ob.showij(); // illegal, showij() is protected
```

```
// member of derived
return 0;
}
```

As you can see by reading the comments, even though **setij()** and **showij()** are public members of **base,** they become protected members of **derived** when it is inherited using the **protected** access specifier. This means that they will not be
accessible inside **main()** .

Inheriting Multiple Base Classes

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

```
// An example of multiple base classes.
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};
class base2 {
protected:
int y;
public:
void showy() {cout << y << "\n";}
};
// Inherit multiple base classes.
class derived: public base1, public base2 {
public:
void set(int i, int j) { x=i; y=j; }
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

As the example illustrates, to inherit more than one base class, use a comma-separated list. Further, be sure to use an access-specifier for each base inherited.

Constructors, Destructors, and Inheritance

There are two major questions that arise relative to constructors and destructors when inheritance is involved. First, when are base-class and derived-class constructor and destructor functions called? Second, how can parameters be passed to base-class constructor functions? This section examines these two important topics.

When Constructor and Destructor Functions Are Executed

It is possible for a base class, a derived class, or both to contain constructor and/or destructor functions.

It is important to understand the order in which these functions are executed when an object of a derived class comes into existence and when it goes out of existence. To begin, examine this short program:

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

As the comment in **main()** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**. When executed, this program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

As you can see, first **base**'s constructor is executed followed by **derived**'s. Next (because **ob** is immediately destroyed in this program), **derived**'s destructor is called, followed by **base**'s. The results of the foregoing experiment can be generalized.

When an object of a derived class is created, if the base class contains a constructor, it will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor, if it exists. Put differently, constructor functions are executed in their order of derivation. Destructor functions are executed in reverse order of derivation.

If you think about it, it makes sense that constructor functions are executed in order of derivation. Because a base class has no knowledge of any derived class, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the derived class. Therefore, it must be executed first. Likewise, it is quite sensible that destructors be executed in reverse order of derivation. Because the base class underlies the derived class, the destruction of the base object implies the destruction of the derived object. Therefore, the derived destructor must be called before the object is fully destroyed.

In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

```
#include <iostream>
```

```
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived1 : public base {
public:
derived1() { cout << "Constructing derived1\n"; }
~derived1() { cout << "Destructing derived1\n"; }
};
class derived2: public derived1 {
public:
derived2() { cout << "Constructing derived2\n"; }
~derived2() { cout << "Destructing derived2\n"; }
};
int main()
{
derived2 ob;
// construct and destruct ob
return 0;
}
```

displays this output:

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

The same general rule applies in situations involving multiple base classes. For example, this program

```
#include <iostream>
using namespace std;
class base1 {
public:
base1() { cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
public:
base2() { cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base2\n"; }
};
class derived: public base1, public base2 {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
```

```
{
derived ob;
// construct and destruct ob
return 0;
}
```

produces this output:

```
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1
```

As you can see, constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list. Destructors are called in reverse order, right to left. This means that had **base2** been specified before **base1** in **derived**'s list, as shown here: class derived: public base2, public base1 { then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

Pas

Inheriting Multiple Base Classes:

It is possible for a derived class to inherit two or more base classes. For example, in this short example, **derived** inherits both **base1** and **base2**.

// An example of multiple base classes.

```
#include <iostream>
using namespace std;
class base1 {
protected:
int x;
public:
void showx() { cout << x << "\n"; }
};

class base2 {
protected:
int y;
public:
void showy() {cout << y << "\n";}
};

// Inherit multiple base classes.
Class derived: public base1, public base2 {
public:
void set(int I, int j) { x=I; y=j; }
```

```
};
int main()
{
derived ob;
ob.set(10, 20); // provided by derived
ob.showx(); // from base1
ob.showy(); // from base2
return 0;
}
```

Constructors, Destructors, and Inheritance:

There are two major questions that arise relative to constructors and destructors when inheritance is involved.
First, when are base-class and derived-class constructor and destructor functions called?
Second, how can parameters be passed to base-class constructor functions?
This section examines these two important topics.

When Constructor and Destructor Functions Are Executed?

It is possible for a base class, a derived class, or both to contain constructor and/or destructor functions.

```
#include <iostream>
using namespace std;
class base {
public:
base() { cout << "Constructing base\n"; }
~base() { cout << "Destructing base\n"; }
};
class derived: public base {
public:
derived() { cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
};
int main()
{
derived ob;
// do nothing but construct and destruct ob
return 0;
}
```

As the comment in **main()** indicates, this program simply constructs and then destroys an object called **ob** that is of class **derived**. When executed, this program displays

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies: Constructors are called in order of derivation, destructors in reverse order. For example, this program

Passing Parameters to Base-Class Constructors:

how do you pass arguments to a constructor in a base class?
The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}
```

base1 through baseN are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes.

```
#include <iostream>
    using namespace std;
    class base {
    protected:
    int i;
    public:
    base(int x) { i=x; cout << "Constructing base\n"; }
    ~base() { cout << "Destructing base\n"; }
    };
    class derived: public base {
    int j;
    public:
    // derived uses x; y is passed along to base.
    derived(int x, int y): base(y)
    { j=x; cout << "Constructing derived\n"; }
    ~derived() { cout << "Destructing derived\n"; }
    void show() { cout << i << " " << j << "\n"; }
    };
    int main()
    {
    derived ob(3, 4);
    ob.show(); // displays 4 3
    return 0;
    }
```

Here, **derived**'s constructor is declared as taking two parameters, **x** and **y**.However, **derived()** uses only **x; y** is passed along to **base().** In general, the derived class' constructor must declare both the parameter(s) that it requires as well as any required by the base class. Here is an example that uses multiple base classes:

```
#include <iostream>
using namespace std;
class base1 {
protected:
int i;
public:
base1(int x) { i=x; cout << "Constructing base1\n"; }
~base1() { cout << "Destructing base1\n"; }
};
class base2 {
protected:
int k;
public:
base2(int x) { k=x; cout << "Constructing base2\n"; }
~base2() { cout << "Destructing base1\n"; }
};
class derived: public base1, public base2 {
int j;
public:
derived(int x, int y, int z): base1(y), base2(z)
{ j=x; cout << "Constructing derived\n"; }
~derived() { cout << "Destructing derived\n"; }
void show() { cout << i << " " << j << " " << k << "\n"; }
};
int main()
{
derived ob(3, 4, 5);
ob.show(); // displays 4 3 5
return 0;
}
```

It is important to understand that arguments to a base-class constructor are passed via arguments to the derived class' constructor. Therefore, even if a derived class' constructor does not use any arguments, it will still need to declare one if the base class

requires it. In this situation, the arguments passed to the derived class are simply passed along to the base.

Chapter 17

Virtual Functions and Polymorphism

A ***virtual function*** is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs.

```
include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().

inside **base**, the virtual function **vfunc()** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When
vfunc() is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed.

Calling a Virtual Function Through a Base Class Reference
virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference.

The most common situation in which a virtual function is invoked through a base class reference is when the reference is a function parameter. For example, consider the following variation on the preceding program.

```
/* Here, a base class reference is used to access
a virtual function. */
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
// Use a base class reference parameter.
void f(base &r) {
r.vfunc();
}
int main()
{
base b;
derived1 d1;
derived2 d2;
f(b); // pass a base object to f()
f(d1); // pass a derived1 object to f()
f(d2); // pass a derived2 object to f()
return 0;
}
```

This program produces the same output as its preceding version. In this example, the function **f()** defines a reference parameter of type **base**. Inside **main()** , the function is called using objects of type **base**, **derived1**, and **derived2**. Inside **f()** , the specific version of **vfunc()** that is called is determined by the type of object being referenced when the function is called.

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

Put differently, no matter how many times a virtual function is inherited, it remains virtual. For example, consider this program:

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
/* derived2 inherits virtual function vfunc()
from derived1. */
class derived2 : public derived1 {
public:
// vfunc() is still virtual
void vfunc() {
cout << "This is derived2's vfunc().\n";
}
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
```

```
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

As expected, the preceding program displays this output:
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
In this case, **derived2** inherits **derived1** rather than **base**, but **vfunc()** is still virtual.

Virtual Functions Are Hierarchical:
when a function is declared as **virtual** by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which **derived2** does not override **vfunc()** :

```
#include <iostream>
using namespace std;
class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};
class derived1 : public base {
public:
void vfunc() {
cout << "This is derived1's vfunc().\n";
}
};
class derived2 : public base {
public:
// vfunc() not overridden by derived2, base's is used
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;
// point to base
p = &b;
p->vfunc(); // access base's vfunc()
// point to derived1
p = &d1;
p->vfunc(); // access derived1's vfunc()
// point to derived2
p = &d2;
p->vfunc(); // use base's vfunc()
return 0;
}
```

The program produces this output:
This is base's vfunc().
This is derived1's vfunc().
This is base's vfunc().
Because **derived2** does not override **vfunc()** , the function defined by **base** is used when **vfunc()** is referenced relative to objects of type **derived2**.