Object Oriented Programming
14CS55

UNIT - 1

The Origins of C++ :

C++ began as an expanded version of C. The C++ extensions were first invented by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey

In C, once a program exceeds from 25,000 to 100,000 lines of code, it becomes so complex that it is ifficult to grasp as a totality. The purpose of C++ is to allow this barrier to be broken

What Is Object-Oriented Programming?

Object-oriented programming took the best ideas of structured programming and combined them with several new concepts a program written in a structured language such as C is defined by its functions, any of which may operate on any type of data used by the program. Object-oriented programs work the other way around. They are organized around data, with the key principle being "data controlling access to code." In an object-oriented language, you define the data and the routines that are permitted to act on that data

To support the principles of object-oriented programming, all OOP languages have three traits in common: encapsulation, polymorphism, and inheritance

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

polymorphism, which is characterized by the phrase "one interface, multiple methods." Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions

Inheritance is the process by which one object can acquire the properties of another object

C++ Fundamentals

A Sample C++ Program :

#include <iostream> **/* This header supports C++-style I/O operations. (<iostream> is to C++ what stdio.h is to C.*/**
using namespace std; **/* Namespaces are a recent addition to C++. A namespace creates a declarative region in which various program elements can be placed*/**

```
int main()
{
        int i;
        cout << "This is output.\n"; // this is a single line comment
                /* you can still use C style comments */
                // input a number using >>
        cout << "Enter a number: ";
        cin >> i;
                // now, output a number using <<
        cout << i << " squared is " << i*i << "\n";
        return 0;
}
```

The word **cout** is an identifier that is linked to the screen. The identifier **cin** refers to the standard input device, which is usually the keyboard

Introducing C++ Classes

A class is similar syntactically to a structure

```
#define SIZE 100
// This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
void init();
void push(int i);
int pop();
};
```

- A class may contain private as well as public parts. By default, all items defined in a class are private
- To make parts of a class public (that is, accessible to other parts of your program), you must declare them after the public keyword
- Once you have defined a class, you can create an object of that type by using the class name. In essence, the class name becomes a new data type specifier : **stack mystack;**

When you declare an object of a class, you are creating an instance of that class

The general form of a simple class declaration is

```
class class-name {
private data and functions
public:
public data and functions
```

} object name list;

When it comes time to actually code a function that is the member of a class, you must tell the compiler which class the function belongs to by qualifying its name with the name of the class of which it is a member

```
void stack::push(int i)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;
}
stck[tos] = i;
tos++;
}
```

The :: is called the scope resolution operator. Essentially, it tells the compiler that this version of push( ) belongs to the stack class

complete example

```
#define SIZE 100
#include <iostream>
using namespace std;
// This creates the class stack.
class stack {
int stck[SIZE];
int tos;
public:
void init();
void push(int i);
int pop();
};
void stack::init()
{
tos = 0;
}
void stack::push(int i)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;
}
stck[tos] = i;
tos++;
}
```

```
int stack::pop()
{
if(tos==0) {
cout << "Stack underflow.\n";
return 0;
}
tos--;
return stck[tos];
}

int main()
{
stack stack1, stack2;
// create two stack objects
stack1.init();
stack2.init();
stack1.push(1);
stack2.push(2);
stack1.push(3);
stack2.push(4);
cout<<stack1.pop()<<” ”;
cout<<stack1.pop()<<” ”;
cout<<stack2.pop()<<” ”;
cout<<stack2.pop()<<” \n”;
return 0;
}
}
```

Function Overloading

One way that C++ achieves polymorphism is through the use of function overloading. In C++, two or more functions can share the same name as long as their parameter declarations are different In this situation, the functions that share the same name are said to be overloaded, and the process is referred to as function overloading.

Exampe

```
#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;
}
```

One important restriction is 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. They must differ in the types or number of their parameters.

Operator Overloading

Polymorphism is also achieved in C++ through operator overloading. As you know, in C++, it is possible to use the << and >> operators to perform console I/O operations. They can perform these extra operations because in the <iostream> header, these operators are overloaded. When an operator is overloaded, it takes on an additional meaning relative to a certain class.

Inheritance

Inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a base class, which defines those qualities common to all objects to be derived from the base. The classes derived from the base are usually referred to as derived classes A derived class includes all features of the generic base class and then adds qualities specific to the derived class

The general form for inheritance is

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

Example 1:

```
#inlcude<iostream>
using namespace std;
 //Base class
class Parent
{
      public:
         int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
      public:
         int id_c;
};

//main function
int main()
     {
            Child obj1;
            // An object of class child has all data members
            // and member functions of class parent
            obj1.id_c = 7;
            obj1.id_p = 91;
            cout << "Child id is " <<  obj1.id_c << endl;
            cout << "Parent id is " <<  obj1.id_p << endl;
            return 0;
     }
```

Exxample 2:

```
class building
        {
                int rooms;
                int floors;
                int area;
        public:
                void set_rooms(int num);
                int get_rooms();
                void set_floors(int num);
                int get_floors();
                void set_area(int num);
                int get_area();
        };
// house is derived from building
```

```
class house : public building
        {
                int bedrooms;
                int baths;
        public:
                void set_bedrooms(int num);
                void set_baths(int num);
                int get_bedrooms();
                int get_baths();
        };
```

Constructors and Destructors

A constructor is a special function that is a member of a class and has the same name as that class, constructors cannot return values, and constructors are calledd when an object is created,most constructors will not output or input anything,they will simply perform various initializations.

```
/#include <iostream>
using namespace std;
 class test
{
public:
      int a, b;

      // Default Constructor
      test()
      {
            a = 10;
            b = 20;
      }
};

int main()
{
            // Default constructor called automatically
            // when the object is created
      test c;
      cout << "a: "<< c.a << endl << "b: "<< c.b;
      return 1;
}
```

The complement of the constructor is the destructor, When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor may be needed. For example,

an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened
The destructor has the same name as the constructor, but it is preceded by a ~

example : ~test();

The General Form of a C++ Program

#includes
base-class declarations
derived class declarations
nonmember function prototypes
int main( )
{
//...
}
nonmember function definitions

Classes
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

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;

Access-specifier is one of these three C++ keywords: public, private, protected

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

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The public access specifier allows functions or data to be accessible to other parts of your program. The protected access specifier is needed only when inheritance is involved

Functions that are declared within a class are called member functions. Member functions may access any element of the class of which they are a part. This includes all private elements. Variables that are elements of a class are called member variables or data members. (The term instance variable is also used.) Collectively, any element of a class can be referred to as a member of that class.

There are a few restrictions that apply to class members.

- A non-static member variable cannot have an initializer.
- No member can be an object of the class that is being declared. (Although a member can be a pointer to the class that is being declared.)
- No member can be declared as auto, extern, or register

Example

```
#include <iostream>
#include <cstring>
using namespace std;
void employee::putname(char *n)
{
strcpy(name, n);
}
void employee::getname(char *n)
{
strcpy(n, name);
}
void employee::putwage(double w)
{
wage = w;
}
double employee::getwage()
{
return wage;
}
int main()
{
employee ted;
char name[80];
ted.putname("Ted Jones");
ted.putwage(75000);
class employee {
char name[80]; // private by default
public:
void putname(char *n); // these are public
```

```
void getname(char *n);
private:
double wage; // now, private again
public:
void putwage(double w); // back to public
double getwage();
};
ted.getname(name);
cout << name << " makes $";
cout << ted.getwage() << " per year.";
return 0;
}
class employee {
char name[80];
double wage;
public:
void putname(char *n);
void getname(char *n);
void putwage(double w);
double getwage();
};
```

Structures and Classes Are Related :

Structures are part of the C subset and were inherited from the C language. As you have seen, a class is syntactically similar to a struct.

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.

```
// 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 program displays 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();
} ;
```

what is the need for Struct in C++ when already there is class?

- *increase the capabilities of a structure in c++ in comparision with C*
- *In C, structures already provide a means of grouping data. Therefore, it is a small step to allow them to include member functions*

- *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*

Unions and Classes Are Related :

Like a structure, a union may also be used to define a class. In C++, unions may contain both member functions and variables

A union in C++ retains all of its C-like features, the most important being that all data elements share the same location in memory

Like the structure, union members are public by default and are fully compatible with C.

Example

```
#include <iostream>
using namespace std;
union swap_byte {
void swap();
void set_byte(unsigned short i);
void show_word();
unsigned short u;
unsigned char c[2];
};
void swap_byte::swap()
{
unsigned char t;
t = c[0];
c[0] = c[1];
c[1] = t;
```

```
}
void swap_byte::show_word()
{
cout << u;
}
void swap_byte::set_byte(unsigned short i)
{
u = i;
}
int main()
{
swap_byte b;
b.set_byte(49034);
b.swap();
b.show_word();
return 0;
}
```

There are several restrictions that must be observed when you use C++ unions

- First, a union cannot inherit any other classes of any type.
- Further, a union cannot be a base class.
- A union cannot have virtual member functions.
- No static variables can be members of a union.
- A reference member cannot be used.
- A union cannot have as a member any object that overloads the = operator.
- Finally, no object can be a member of a union if the object has an explicit constructor or destructor function

Anonymous Unions

There is a special type of union in C++ called an anonymous union. An anonymous union does not include a type name, and no objects of the union can be declared

Instead, an anonymous union tells the compiler that its member variables are to share the same location. However, the variables themselves are referred to directly, without the normal dot operator syntax.

For example, consider this program:

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
// define anonymous union
union {
long l;
double d;
char s[4];
} ;
// now, reference union elements directly
l = 100000;
cout << l << " ";
d = 123.2342;
cout << d << " ";
strcpy(s, "hi");
cout << s;
return 0;
}
```

Friend Functions

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

Example :

```
#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;
}
```

In this example, the sum( ) function is not a member of myclass. However, it still has full access to its private members. Also, notice that sum( ) is called without the use of the dot operator

friends can be useful :

- when you are overloading certain types of operators.
- friend functions make the creation of some types of I/O functions easier
- friend functions may be desirable is that in some cases, two or more classes may contain members that are interrelated relative to other parts of your program
-

Friend Classes

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.

For example:

```
// Using a friend 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::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;
}
```

In this example, class Min has access to the private variables a and b declared within the TwoValues class.

Inline Functions

In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation

To cause a function to be expanded in line rather than called, precede its definition with the inline keyword

```
#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;
}
```

Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code

Like the register specifier, inline is actually just a request, not a command, to the compiler. The compiler can choose to ignore it

Defining Inline Functions Within a Class

It is possible to define short functions completely within a class declaration. When a function is defined inside a class declaration, it is automatically made into an inline function (if possible)

```
#include <iostream>
using namespace std;
class myclass {
int a, b;
public:
// automatic inline
void init(int i, int j) { a=i; b=j; }
void show() { cout << a << " " << b << "\n"; }
};
int main()
{
myclass x;
x.init(10, 20);
x.show();
return 0;
}
```

Parameterized Constructors

It is possible to pass arguments to constructors. 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

```
#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;
}
```

there are diferent ways to define amnd call cunstructor

- myclass ob(3, 4);
- myclass ob = myclass(3, 4);
- myclass ob = 99; // special type of constructor called one parameter constructor
- myclass ob = myclass(99);//

Static Class Members

Both function and data members of a class can be made static.

Static Data Members

- 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**
- Unlike regular data members, individual copies of a static member variable are not made for each object
- 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

```
#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.

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

A static member variable exists before any object of its class is created. in the following short program, a is both public and static. Thus it may be **directly accessed** in main( ). Further, since a exists before an object of shared is created, a can be given a value at any time. As this program illustrates, the value of a is unchanged by the creation of object x. For this reason, both output statements display the same value: 99.

```
#include <iostream>
using namespace std;
class shared
	{
		public:
		static int a;
	} ;

int shared::a; // define a

int main()
	{
		// initialize a before creating any objects
		shared::a = 99;
		cout << "This is initial value of a: " << shared::a;
		cout << "\n";
		shared x;
		cout << "This is x.a: " << x.a;
		return 0;
	}
```

- a static member is referred to through the use of the class name and the scope resolution operator. In general, to refer to a static member independently of an object, you must qualify it by using the name of the class of which it is a member
- One use of a static member variable is to provide access control to some shared resource used by all objects of a class

Another interesting use of a static member variable is to keep track of the number of objects of a particular class type that are in existence.

For example:

```
#include <iostream>
using namespace std;
class Counter
	{
		public:
		static int count;
		Counter() { count++; }
		~Counter() { count--; }
	};
int Counter::count;
void f();
int main(void)
	{
		Counter o1;
```

```
		cout << "Objects in existence: ";
		cout << Counter::count << "\n";
		Counter o2;
		cout << "Objects in existence: ";
		cout << Counter::count << "\n";
		f();
		cout << "Objects in existence: ";
		cout << Counter::count << "\n";
		return 0;
void f()
	{
		Counter temp;
		cout <<"Objects in existence: ";
		cout <<Counter::count << "\n";
		// temp is destroyed when f() returns
}
```

Static Member Functions

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
- 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.
- They cannot be declared as const or volatile

Example :

```
#include <iostream>
using namespace std;
class cl
	{
		static int resource;
		public:
			static int get_resource();
			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;
        }
```

Actually, static member functions have limited applications, but one good use for them is to "**preinitialize**" private static data before any object is actually created

Example

```
#include <iostream>
using namespace std;
class static_type
        {
                static int i;
                public:
                        static void init(int x)
                                {
                                        i = x;
                                }
                        void show()
                                {
                                        cout << i;
                                }
        };
int static_type::i; // define i
int main()
        {
                // init static data before object creation
```

```
        static_type::init(100);
        static_type x;
        x.show(); // displays 100
        return 0;
    }
```

When Constructors and Destructors Are Executed

- A local object's constructor is executed when the object's declaration statement is encountered. The destructors for local objects are executed in the reverse order of the constructor functions.
- Global objects have their constructors execute before main( ) begins execution. Global constructors are executed in order of their declaration, within the same file

```
#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
Destructing 3
Destructing 2
Destructing 1
```

The Scope Resolution Operator

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. For example, consider this fragment:

```
int i; // global i
void f()
        {
                int i; // local i
                i = 10; // uses local i
                .
                .
                .
        }
```

As the comment suggests, the assignment i = 10 refers to the local i. But what if function f( ) needs to access the global version of i? It may do so by preceding the i with the :: operator, as shown here.

```
int i; // global i
void f()
        {
                int i; // local i
                ::i = 10; // now refers to global i
                .
                .
                .
        }
```

Nested Classes

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

Local Classes

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

- 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 to static local variables declared within the function or those declared as extern)
- It may access type names and enumerators defined by the enclosing function
- however. No static variables may be declared inside a local class.

Because of these restrictions, local classes are not common in C++ programming.

Passing Objects to Functions

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

Although the passing of objects is straightforward, some rather unexpected events occur that relate to constructors and destructors.

To understand why, consider this short program.

// 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);  //constructor is not called, but destructor is being called once function call ends
		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 1
```

Few points to remember

- When a copy of an argument is made during a function call, the normal constructor is not called Instead, the object's copy constructor is called. A copy constructor defines how a copy of an object is made
- When passing an object to a function, you want to use the current state of the object, not its initial state.
- when the function terminates and the copy of the object used as an argument is destroyed, the destructor is called

Returning Objects

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

Object Assignment

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

Arrays of Objects

In C++, it is possible to have 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.

```
#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;
}
```

If a class defines a parameterized constructor, you may initialize each object inan array by specifying an initialization list, just like you do for other types of arrays

For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. As each element in the array is created, a value from the list is passed to the constructor's parameter. For example, here is a slightly different version of the preceding program that uses an initialization:

```
#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;
}
```

If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example,

```
#include <iostream>
using namespace std;
class cl {
int h;
int i;
public:
cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
int get_i() {return i;}
int get_h() {return h;}
};
int main()
{
cl ob[3] = {
cl(1, 2), // initialize
cl(3, 4),
```

```
cl(5, 6)
};
int i;
for(i=0; i<3; i++) {
cout << ob[i].get_h();
cout << ", ";
cout << ob[i].get_i() << "\n";
}
return 0;
}
```

Pointers to Objects

ust 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

```
#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

```
#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

```
#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;
}
```

Type Checking C++ Pointers

There is one important thing to understand about pointers in C++: You may assign one pointer to another only if the two pointer types are compatible. For example, given:

```
int *pi;
float *pf;
```

in C++, the following assignment is illegal:

```
pi = pf; // error -- type mismatch
```

The this Pointer

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object This pointer is called this

Here is the entire pwr( ) constructor 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;
}
```

Actually, no C++ programmer would write pwr( ) as just shown because nothing is gained, and the standard form is easier. However, the this pointer is very important when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it.

Pointers to Derived Types

In general, a pointer of one type cannot point to an object of a different type

- 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

- you can access only the members of the derived type that were inherited from the base. That is, you won't be able to access any members added by the derived class

```
#include <iostream>
using namespace std;
class base
	{
		int i;
	public:
		 void set_i_b(int num)
			{
				i=num;
			}
		 int get_i_b()
			{
				return i;
			 }
	};
class derived: public base
	{
		int j;
		public:
		void set_j_d(int num)
		{
			 j=num;
		}
		int get_j_d()
		 {
			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 elements of
a derived class using a base class pointer.
bp->set_j(88); // error
cout << bp->get_j(); // error
*/
return 0;
}
```

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();
```

Pointers to Class Members

C++ allows you to generate a special type of pointer that "points" generically to a member of a class, not to a specific instance of that member in an object This sort of pointer is called a pointer to a class member or a pointer-to-member, for short. A pointer to a member is not the same as a normal C++ pointer

To access a member of a class given a pointer to it, you must use the special

pointer-to-member operators **.* and –>***. Their job is to allow you to access a member of a class given a pointer to that member.

```
#include <iostream>
using namespace std;
class cl {
public:
cl(int i) { val=i; }
int val;
int double_val() { return val+val; }
};
int main()
{
int cl::*data; // data member pointer
int (cl::*func)(); // function member pointer
cl ob1(1), ob2(2); // create objects
data = &cl::val; // get offset of val
func = &cl::double_val; // get offset of double_val()
cout << "Here are values: ";
cout << ob1.*data << " " << ob2.*data << "\n";
cout << "Here they are doubled: ";
cout << (ob1.*func)() << " ";
cout << (ob2.*func)() << "\n";
return 0;
}
```

When you are accessing a member of an object by using an object or a reference (discussed later in this chapter), you must use the .* operator. However, if you are using a pointer to the object, you need to use the –>* operator, as illustrated in this version of the preceding program:

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

```
cl(int i) { val=i; }
int val;
int double_val() { return val+val; }
};
int main()
{
int cl::*data; // data member pointer
int (cl::*func)(); // function member pointer
cl ob1(1), ob2(2); // create objects
cl *p1, *p2;
p1 = &ob1; // access objects through a pointer
p2 = &ob2;
data = &cl::val; // get offset of val
func = &cl::double_val; // get offset of double_val()
cout << "Here are values: ";
cout << p1->*data << " " << p2->*data << "\n";
cout << "Here they are doubled: ";
cout << (p1->*func)() << " ";
cout << (p2->*func)() << "\n";
return 0;
}
```

Remember, pointers to members are different from pointers to specific instances of elements of an object. Consider this fragment (assume that cl is declared as shown in the preceding programs):

```
int cl::*d;
int *p;
cl o;
p = &o.val // this is address of a specific val
d = &cl::val // this is offset of generic val
```

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 :** 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 can use a reference parameter. For most circumstances the best way is to use a reference parameter.

// Manually create a call-by-reference using a pointer i. Without reference parameter(automatic)

```
#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;
}
```

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 swap(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";
```

```
swap(a, b); // no & operator needed
cout << "a and b: " << a << " " << b << "\n";
cout << "c and d: " << c << " " << d << "\n";
swap(c, d);
cout << "c and d: " << c << " " << d << "\n";
return 0;
}

void swap(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 2 1
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. However, ***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

```
#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 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.

One other point: Passing all but the smallest objects by reference is faster than passing them by value. Arguments are usually passed on the stack. Thus, large objects take a considerable number of CPU cycles to push onto and pop from the stack.

- **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!

```
#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.

One thing you must be careful about when returning references is that the object being referred to does not go out of scope after the function terminates.

- **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
- All independent references must be initialized when they are created

```
#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
```

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
- C++ also supports dynamic memory allocation functions, called malloc( ) and free( )
- 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.

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;
}
```

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.

Initializing Allocated Memory : You can initialize allocated memory to some known value by putting an initializer after the type name in the new statement

p_var = new var_type (initializer);

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p;
try {
p = new int (87); // initialize to 87
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
cout << "At " << p << " ";
cout << "is the value " << *p << "\n";
delete p;
return 0;
}
```

Allocating Arrays : You can allocate arrays using new by using this general form:

p_var = new array_type [size];

Here, size specifies the number of elements in the array.

To free an array, use this form of delete:
delete [ ] p_var;

Here, the [ ] informs delete that an array is being released.

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
int *p, i;
try {
p = new int [10]; // allocate 10 integer array
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
for(i=0; i<10; i++ )
p[i] = i;
for(i=0; i<10; i++)
cout << p[i] << " ";
delete [] p; // release the array
return 0;
}
```

Allocating Objects : You can allocate objects dynamically by using new. When you do this, an object is created and a pointer is returned to it

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance {
double cur_bal;
```

```
char name[80];
public:
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s);
}
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name);
}
};

int main()
{
balance *p;
char s[80];
double n;
try {
p = new balance;
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
p->set(12387.87, "Ralph Wilson");
p->get_bal(n, s);
cout << s << "'s balance is: " << n;
cout << "\n";
delete p;
return 0;
}
```

Because p contains a pointer to an object, the arrow operator is used to access members of the object.

As stated, dynamically allocated objects may have constructors and destructors. Also, the constructors can be parameterized

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance {
double cur_bal;
char name[80];
public:
balance(double n, char *s) {
cur_bal = n;
strcpy(name, s);
}
~balance() {
cout << "Destructing ";
cout << name << "\n";
}
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name);
}
};
int main()
{
balance *p;
char s[80];
double n;
/ this version uses an initializer
try {
p = new balance (12387.87, "Ralph Wilson");
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
p->get_bal(n, s);
cout << s << "'s balance is: " << n;
```

```
cout << "\n";
delete p;
return 0;
}
```

You can allocate arrays of objects, but there is one catch. Since no array allocated by new can have an initializer, you must make sure that if the class contains constructors, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance {
double cur_bal;
char name[80];
public:
balance(double n, char *s) {
cur_bal = n;
strcpy(name, s);
}
balance() {} // parameterless constructor
~balance() {
cout << "Destructing ";
cout << name << "\n";
}
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s);
}
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name);
```

```
}
};
int main()
{
balance *p;
char s[80];
double n;
int i;
try {
p = new balance [3]; // allocate entire array
} catch (bad_alloc xa) {
cout << "Allocation Failure\n";
return 1;
}
// note use of dot, not arrow operators
p[0].set(12387.87, "Ralph Wilson");
p[1].set(144.00, "A. C. Conners");
p[2].set(-11.23, "I. M. Overdrawn");
for(i=0; i<3; i++) {
p[i].get_bal(n, s);
cout << s << "'s balance is: " << n;
cout << "\n";
}
delete [] p;
return 0;
}
```

The output from this program is shown here.

```
Ralph Wilson's balance is: 12387.9
A. C. Conners's balance is: 144
I. M. Overdrawn's balance is: -11.23
Destructing I. M. Overdrawn
Destructing A. C. Conners
Destructing Ralph Wilson
```

The nothrow Alternative : In Standard C++ it is possible to have new return null instead of throwing an exception when an allocation failure occurs
p_var = new(nothrow) type;

The Placement Form of new : There is a special form of new, called the placement form, that can be used to specify an alternative method of allocating memory. It is primarily useful when overloading the new operator for special circumstances. Here is its general form:

p_var = new (arg-list) type;

Here, arg-list is a comma-separated list of values passed to an overloaded form of new.