Object Oriented Programming in C++

Segment-2

Course Code: 2301

Prepared by Sumaiya Deen Muhammad

Lecturer, CSE, IIUC

Contents

Introducing Classes, Arrays, Pointers and References:

-Access Specifies
-Constructor and Destructor
-Constructors with parameters
-Object Pointers
-Relation between Classes
-Structures and unions
-In-line functions
-Automatic in-line functions
-Assigning objects
-Passing objects to functions
-Returning objects from function
-Friend functions
-Static member functions
-Array of objects
-Pointer to objects,
-this pointer,
-using *new* and *delete*,
-passing references,
-returning references

Introducing Class

- A class is declared using the ***class*** keyword. The general form is:

```
class class_name{
private:
//private variables, and functions
public:
// public variables and functions
};
```

A Simple Class

```
#include <iostream>
using namespace std;
class Student           //define a class
{
        private:
        int num;        //class data

        public:
        void setdata(int d)       //member function to set data
        {
                num = d;
         }
        void showdata()           //member function to display data
        {
                cout << "Data is " << num << endl;
        }
};
```

```
int main()
{
        Student obj1, obj2;      //define two objects of class Student

        obj1.setdata(10);        //call member function to set data
        obj2.setdata(99);

        obj1.showdata();         //call member function to display data
        obj2.showdata();
        return 0;
}
```

Access Specifier

private and public

The body of the class contains two unfamiliar keywords: **private** and **public**.

-Private data or functions can only be accessed **within the class**.

-Public data or functions, on the other hand, are accessible both by other members of the class and by any other part of the program that contains the class.

Access Specifier (cont.)

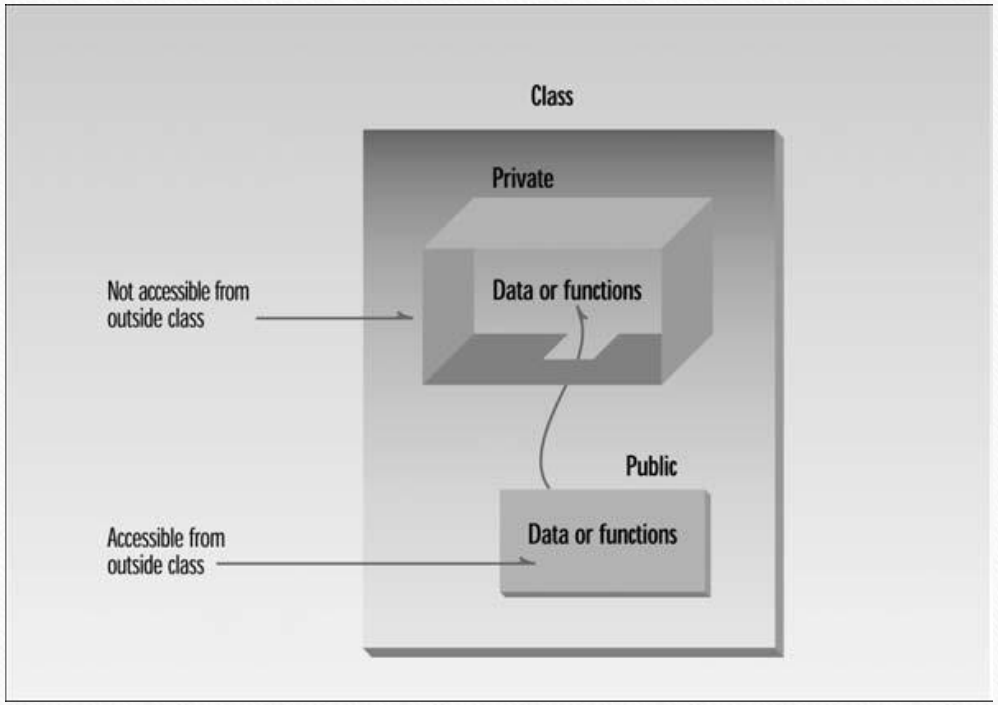

FIGURE : *private and public*.**

Constructor

A constructor is a member function that is executed automatically whenever an object is created.
A class's constructor is called each time an object of that class is created.
A constructor function has the **same name** of that class.
It has **no return type.**

```
class DemoCons
{
        private:
        int a;
        public:
        DemoCons() : a(10)              //constructor
        { }
        void show()
        {
           cout<<a;
        }
};
```

```
int main()
{
        DemoCons obj;
        obj.show();
        return 0;
}
```

Constructor (cont.)

```
#include <iostream>
using namespace std;
class Counter
{
        private:
        int count;                      //count
        public:

        Counter() : count(0)            //constructor
        {       }

        void inc_count()                //increment count
        {
                count++;
        }
        int get_count()                 //return count
        {
                return count;
        }
};
```

```
int main()
{
Counter c1, c2;                         //define and initialize
cout << c1.get_count();                 //display
cout << c2.get_count();
c1.inc_count();                         //increment c1
c2.inc_count();                         //increment c2
c2.inc_count();                         //increment c2
cout << c1.get_count();                 //display again
cout << c2.get_count();
return 0;
}
```

Output:

```
c1=0
c2=0
c1=1
c2=2
```

Constructor (cont.)

Initializer List

One of the most common tasks a constructor carries out is initializing data members. In the Counter class the constructor must initialize the count member to 0. You might think that this would be done in the constructor's function body, like this:

count()
{ count = 0; }

However, this is not the preferred approach (although it does work). Here's how you should initialize a data member:

count() : count(0)
{ }

The initialization takes place following the member function declarator but before the function body. It's preceded by a colon. The value is placed in parentheses following the member data. If multiple members must be initialized, they're separated by commas. The result is the *initializer list*.

someClass() : m1(7), m2(33), m2(4) ← initializer list
{ }

Why not initialize members in the body of the constructor? The reasons are complex, but have to do with the fact that members initialized in the initializer list are given a value before the constructor even starts to execute. This is important in some situations. For example, the initializer list is the only way to initialize const member data and references.

Constructor (cont.)

The constructor functions have some special characteristics. These are :

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be **virtual**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators **new** and **delete** when memory allocation is required.

Destructor

We've seen that a special member function—the constructor—is called automatically when an object is first created. Another function is called automatically when an object is destroyed. Such a function is called a *destructor*. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a **tilde (~)**

```
class Hello{
public:

   Hello(){
      cout<<"Constructor is called here"<<endl;
   }
   ~ Hello(){
      cout<<"Destructor is called here"<<endl;
   }
};
```

Like constructors, destructors do not have a return value. They also take no arguments (the assumption being that there's only one way to destroy an object). The most common use of destructors is to deallocate memory that was allocated for the object by the constructor.

```
int main(){

  Hello h1;

  return 0;
}
```

Constructors with Parameters

```
class Sum
{
        private:
        int num1,num2,total;
        public:
        Sum() : num1(0), num2(0)        //constructor (no args)
        { }
        Sum(int p, int q) : num1(p), num2(q)  //constructor (two args)
        {
          total = p + q;
        }
        void display()
        {
                cout<<"Sum value is= "<<total<<endl;
        }
};
```

```
int main()
{
   Sum ob;          //calls first constructor
   Sum ob2(10,20);      //class second constructor
   ob.display();
   ob2.display();
  //find the difference between these two function calls
    return 0;
}
```

Member Functions Defined Outside the Class

```
class Distance
{
   private:
   int feet;
   float inches;
   public:
   Distance() : feet(0), inches(0.0)
   { }
   Distance(int ft, float in) : feet(ft), inches(in)
   { }
   void display( Distance, Distance );
};
```

```
void Distance :: display(Distance d2, Distance d3)
{
   cout<<"INCH(d2)="<<d2.inches<<", FEET(d2)="<<d2.feet<<endl;
   cout<<"INCH(d3)="<<d3.inches<<", FEET(d2)="<<d3.feet<<endl;
}

int main()
{
   Distance dist1, dist3;
   Distance dist2(10, 20.9);
   dist3.display(dist1,dist2);        //object as argument
}
```

Objects as Arguments

In previous example objects have been passed as arguments:

```
dist3.display(dist1,dist2)
```

We can add two values d2.inches and d3.inches like this:

```
inches = d2.inches + d3.inches;
feet = d2.feet + d3.feet;
```

Object Pointers

Pointers can point to objects as well as to simple data types and arrays. Sometimes, however, we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use new to create objects while the program is running. new returns a pointer to an unnamed object.

```
class Distance
{
        private:
        int d;
        public:
        void getdist()
        {   cin>>d; }
        void showdist()
        {   cout<<d; }
};
```

```
int main()
{
        Distance dist;
        dist.getdist();             //access object members with dot operator
        dist.showdist();
        Distance* distptr;                  //pointer to Distance
        distptr = new Distance;       //points to new Distance object
        distptr->getdist();         //access object members with -> operator
        distptr->showdist();
        cout << endl;
        return 0;
}
```

Structures and Unions

A structure is a collection of simple variables. The variables in a structure can be of **different data types**. Some can be ***int***, some can be ***float***, and so on. The data items in a structure are called the *members* of the structure. The structure definition tells how the structure is organized: It specifies what members the structure will have.

```
struct part                    //declare a structure definition
{
        int modelnumber;
        int partnumber
        float cost;
};
```

```
int main()
{
part part1 = { 6244, 373, 217.55F };     //initialize structure variable
cout << "Model " << part1.modelnumber;
cout << ", part " << part1.partnumber;
cout << ", costs $" << part1.cost << endl;
return 0;
}
```

Structure (cont.)

Structures are usually used to hold data only, and classes are used to hold both data and functions. However, in C++, structures can in fact hold both data and functions.

The program's **output** looks like this:
Model 6244, part 373, costs $217.55

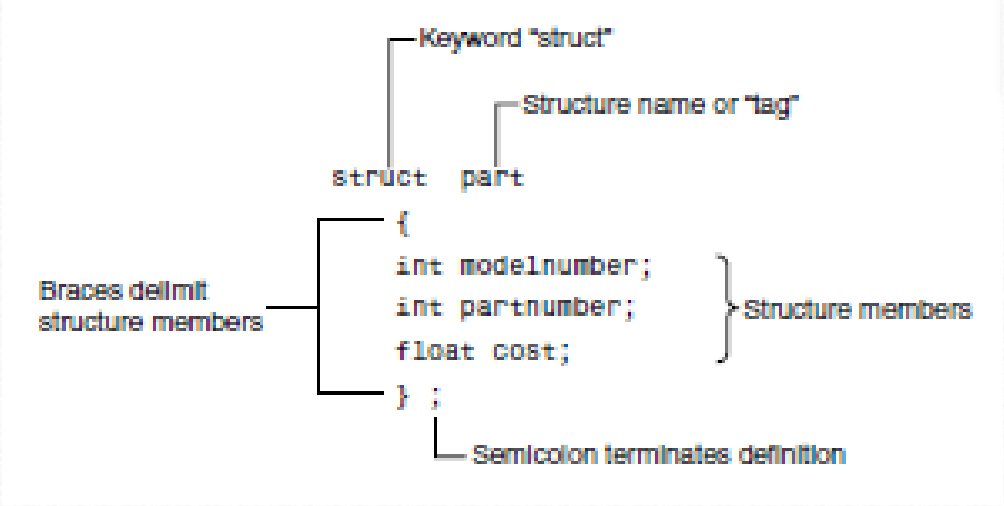

Fig: Syntax of the structure definition.

Union

```
union Employee
{
        int Id; char Name[25];
        int Age;
        long Salary;
};

int main()
{
        Employee E;
        cin >> E.Id;
        cin >> E.Name;
        cin >> E.Age;
        cin >> E.Salary;
    cout << "\n\nEmployee Id : " << E.Id;
    cout << "\nEmployee Name : " << E.Name;
    cout << "\nEmployee Age : " << E.Age;
    cout << "\nEmployee Salary : " << E.Salary;
    return 0;
}
```

Both structure and union are collection of different datatype. They are used to group number of variables of different type in a single unit.

Difference between Structure and Union

-Structure allocates different memory locations for all its members.

-Union allocates common memory location for all its members.

-The memory occupied by a union will be **large** enough to hold the largest member of the union.

In-line functions

One of the advantage of using function is to save memory space by making common block for the code we need to execute many times. When compiler invoke a function, it takes extra time to execute such as jumping to the function definition, saving registers, passing value to argument and returning value to calling function. This extra time can be avoidable for large functions but for small functions we use inline function to save extra time.

When we make an inline function, compiler will replace all the calling statements with the function definition at run-time.

```
#include<iostream>
inline int Add(int x, int y)
{
        return x+y;
}
```

```
int main()
{
        cout<<"The Sum is : " << Add(100,200);
        return 0;
}
```

Automatic in-line functions

If a member function's definition is short enough, the definition can be included inside the class declaration. This kind of function is called automatic inline function. When a function is defined within a class declaration, the inline keyword is no longer necessary.

Example: see Herbert Schildt book page-80

Assigning objects

```
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;
}
```

Passing objects to functions

```
class rational
{
private:
        int num;
        int dnum;
public:
        rational():num(1),dnum(1)
        {}
        void get ()
        {
                cout<<"enter numerator";
                cin>>num;
                cout<<"enter denomenator";
                cin>>dnum;
        }
        void print ()
       { cout<<num<<"/"<<dnum<<endl;  }
   void multi(rational r1,rational r2)
   {
            num=r1.num*r2.num;
            dnum=r1.dnum*r2.dnum;
    }
};
void main ()
{
        rational r1,r2,r3;
        r1.get();
        r2.get();
        r3.multi(r1,r2);      //passing object as argument
        r3.print();
}
```

Returning objects from function

```
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist()
{
cout << "\nEnter feet: "; cin >> feet;
cout << "Enter inches: "; cin >> inches;
}
void showdist()
{ cout << feet << "\'-" << inches << '\"'; }
Distance add_dist(Distance); //add
};

//add this distance to d2, return the sum
Distance Distance::add_dist(Distance d2)
{
Distance temp;
temp.inches = inches + d2.inches;
temp.feet += feet + d2.feet;
return temp; //returning object
}

int main()
{
Distance dist1, dist3;
Distance dist2(11, 6.25);
dist1.getdist();
dist3 = dist1.add_dist(dist2);   //passing object
//dist3 = dist1 + dist2
cout << "\ndist1 = "; dist1.showdist();
cout << "\ndist2 = "; dist2.showdist();
cout << "\ndist3 = "; dist3.showdist();
cout << endl;
return 0;
}
```

Friend functions

One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data. But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions. This is done using a **friend function** or/and a **friend class.**

If a function is defined as a **friend function** then, the private and protected data of a class can be accessed using the function.

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

Friend functions (cont.)

```
class alpha
{
        private:
        int data;
        public:
        alpha() : data(3) { } //no-arg constructor
        friend void frifunc(alpha);   //friend function
};
```

```
void frifunc(alpha obf)       //function definition
{
        cout<<obf.data;       //accessing private variable
}
int main()
{
        alpha aa;
        cout << frifunc(aa) << endl; //call the friend function
        return 0;
}
```

Static member functions

- By declaring a function member as static, we make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **: :**
- A static member function can only access static data member, other static member functions and any other functions from outside the class.

  See the example from book.

Arrays of Object

```
#include
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<< ob[i].get_i() << "\n";
        return 0;
}
```

Pointer to objects

- Objects can be accessed via pointers. When a pointer to an object is used, the object's members are referenced using the arrow(->) operator instead of dot(.) operator.
- Example: see page 125, Herbert Schildt

this pointer

The member functions of every object have access to a pointer named **this**, which points to the object itself. Thus any member function can find out the address of the object of which it is a member.

```
class where
{
private:
char charray[10];          //occupies 10 bytes
public:
void reveal()
{ cout << "\nMy object's address is " << this; }
};
```

```
int main()
{
where w1, w2, w3;                  //make three objects
w1.reveal();                       //see where they are
w2.reveal();
w3.reveal();
cout << endl;
return 0;
}
```

new and *delete* operator

Sometimes programs need to dynamically allocate memory, for which C++ integrates the operators *new* and *delete*. Dynamic memory is allocated using *new* operator. The *new* operator obtains memory from operating system and returns a pointer to the starting point..

Syntax: `data-type *pointer = new data-type(initial-value)`

Example:

```
int *ptr = new int(25);
```
//ptr is a pointer

This expression is used to allocate memory to contain one single element of int type.

new and *delete* operator (cont.)

To allocate memory block for array the syntax is:

```
Data-type *pointer = new data-type[size]
```

Example:

```
int *foo = new int[10];   //foo is a pointer
```

In this case, the system dynamically allocates space for ten elements of type int and returns a pointer to the first element of the sequence, which is assigned to *foo* (a pointer).

new and *delete* operator (cont.)

In most cases , memory allocated dynamically is only needed during specific periods of time within a program. Once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of *delete* operator.

Syntax:

For single element, `delete pointer;`

`Example:` `delete ptr;`

For array: `delete[] pointer;`

`delete[] foo;`

Passing References

A *reference* provides an *alias*—a different name—for a variable. One of the most important uses for references is in passing arguments to functions. Passing arguments by reference uses a different mechanism. Instead of a value being passed to the function, a *reference to* the original variable, in the calling program, is passed. (It's actually the *memory address* of the variable that is passed).

```
#include<iostream>

using namespace std;

void f1 ( int *n);

int main()

{

        int i =0;

        f1(&i);

        cout<<"value of i ="<<i;

        return 0;

}

void f1 (int *n)

{

        *n = 100;

}
```

Returning References

A function can return a reference. This mechanism is very useful for operator overloading.

```
#include<iostream>
using namespace std;
int &f2();
int x;
int main()
{
        f2() = 100;// assign 100 to reference returned by f2
        cout<<x<<endl;
        return 0;
}

int &f2()
{
        return x; // returns a reference to x;
}
```

Reference Books

- 1. Object Oriented Programming in C++ by Robert Lafore
- 2. Teach Yourself C++ by Herbert Schildt
- 3. Object Oriented Programming with C++ by E Balagurusamy