Classes And Objects
Key Concepts

Using structures	Static data members
Creating a class	Static member functions
Defining member functions	Using arrays of objects
Creating objects	Passing objects as parameter
Using objects	Making functions friendly to classes
Inline member functions	Functions returning object
Nested member functions	const member functions
Private member functions	Pointers to members
Arrays as class members	Using dereferencing operator
Storage of objects	Local classes

The most important feature of C++ is the "class". Its significance is highlighted that Stroustrup initially gave the name "C with classes" to his new language. A class is an extension of structure used in C. It is a new way of creating and implementing a user-defined data type. The concept of class by first reviewing the traditional structures found in C and the ways in which classes can be designed, implemented and applied.

C Structures Revisited

We know that one of the unique features of the C language in structure. They provide a method for packing together data of different types. A structure is a convenient tool for handling a group of logically related data items of different data types. It is a user defined data type with a template that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declarations:

```
struct student
{
        char    name[20];
        int     roll_number;
        float   total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as ***structure members*** or ***elements*.** The identifier student which is referred to as ***structure name*** **or** ***structure tag,*** can be used to create variables of type student. Example :

struct student A; // C declaration

A is a variable of type student and has three member variables as defined by the template. Member variables can be accessed using the ***dot*** or ***period operator*** as follows:

```
strcpy(A,name, “”Sachin”);
A.roll_number = 1234;
A.total_marks = 451;
Final_total – A.total_marks + 5;
```

Structure can have arrays, pointers or structures as members.

Limitations of C Structure

The standard C does not allow the struct data type to be treated like built-in data types. For example, consider the following structure :

```
struct  complex
{
        float x;
        float y;
 };

        struct complex cl, c2, c3;
```

The complex numbers cl, c2, and c3 can easily be assigned values using the dot operator but we cannot add two complex numbers or subtract one from the other. For example

c3 = cl + c2; is illegal in C.

Another important limitation of C structures is that they do not permit ***data hiding.*** Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In other words, the structure members are public members.

Extensions to Structures

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its **OOP philosophy.** It attempts to bring the user-defined type as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of **OOP.** ***Inheritance,*** a mechanism by which one type can inherit characteristics from other types, is also supported by **C++.**

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure varables. For example, we can declare the student variable A as

student A; // C++ *declaration* Remember, this is an error C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

The only difference between a structure and a class in C++ is that, by default, the members of a **class are *private,*** while, by default, the members of a **structure *are public.***

Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a **new *abstract data type*** that can be treated like any other built-in data type.

Generally, a class specification has two parts

1. **Class declaration**
2. **Class function definitions**

The class declaration describes the type and scope of its members. The class function definitions describe how the class functions are implemented.
The general form of a class declaration is:

```
class  class_name
{
        private:
             variable declarations;
             function  declarations;
        public:
             variable declarations;
             function declaration;
};
```

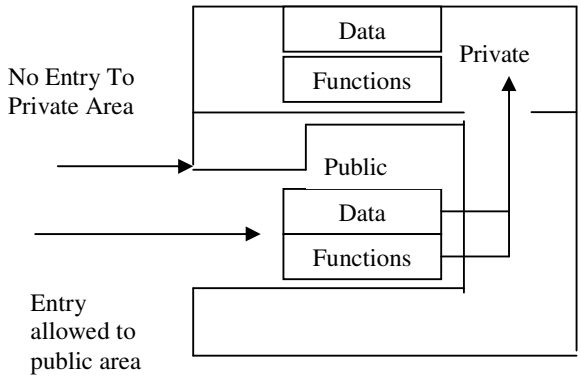
The **class** declaration is similar to a **struct** declaration. The keyword **class** specifies that what follows is an abstract data of type ***class_name.*** The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called ***class members.***

They are usually grouped under two sections, namely, ***private*** and ***public*** to denote which of the members ***are private*** and which of them are ***public***. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. **The data hiding (using private declaration) is the key feature of object-oriented programming.**

The use of the keyword private is optional. By default, the members of a class are **private.** If both the labels are missing, then, by default, all the members are **private.** Such a class is completely hidden from the outside world and does not serve any purpose,

The variables declared inside the class are known as ***data members*** and the functions are known as ***member functions***. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data can be accessed from outside the class. This is illustrated in following figure. The binding of data and functions together into a single class-type variable is referred to as ***encapsulation.***



Data Hiding in Classes

A Simple Class Example

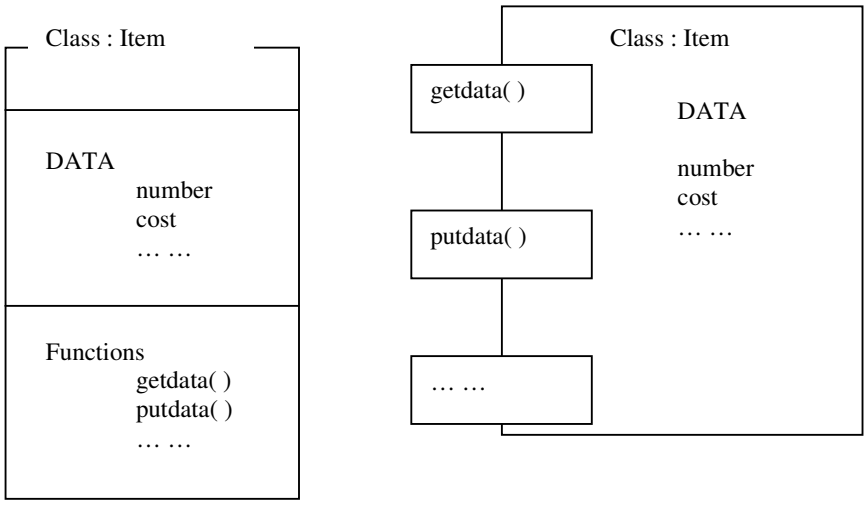
A typical class declaration would look like:

```
class  item
{
        int number;
        float cost;
        public:

        void getdata(int a,   float  b);
        void putdata(void);

}; / / ends with semi c o l on
```

We usually give a class some meaningful name, such as **item.** This name now becomes a new type identifier that can be used to declare ***instances*** of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration.

The function **getdata( )** can be used to assign values to the member variables number and cost, and **putdata( )** for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item.**

Note that the functions are declared, not defined. Actual function definition will appear later in the program. The data members are usually declared as private and the member functions as **public.** Following Figure shows two different notations used by the OOP analysts to represent a class.



Representation of Class

Creating Objects

Remember that the declaration of **item** as shown above does not define any objects of item but only specifies ***what*** they will contain. Once a class has been declared, we can create variables of that type by using the class name like any other built in data type variable. For example,

item x; // memory for x is created

This creates a variable x of type item. In C++, the class variables are known as objects. Therefore, x is called an object of type item. We may also declare more than one object in one statement.

Example:

Item x, y, z;

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification like a structure provides only a template and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class    item
{
        … …
        … …
        … …
} x, y, z;
```

Would create the objects x, y and z of type item. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

Accessing Class Members

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main ( )** function cannot contain statement that access number and cost directly. The following format for calling a member function :

object-name.function-name (actual-argument);

For example, the function call statement

x.getdata(100,75.5);

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the **object x** by implementing the **getdataO** function. The assignments occur in the actual function.

Similarly, the statement

x. putdata( ) ;

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

getdata (100,75.5);
has no meaning.
Similarly, the statement
x.number = 100;

is also illegal. Although x is an object of the type **item** to which **number** belongs, the number declared private can be accessed only through a member function and not by the object directly.|

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

x.putdata();
sends a message to the object x requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class  xyz
{
        int  x;
        int y;
        public :
        int z;
};

  xyz p;
  p.x = 0;          // error, x is private
  p.z = 10;         // OK, z is public
```

Defining Member Functions

Member functions can be defined in two places:

1. Outside the class definition.
2. Inside the class definition.

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, The ANSI prototype form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership **"identity label"** in the header. This **'label'** tells the compiler which **class** the function belongs to. The general form of a member function definition is:

```
return-type class-name :: function-name   (argument declaration)
{
     Function  body
}
```

The membership label class-name :: tells the compiler that the function ***function—name*** belongs to the class ***class-name***. That is, the scope of the function is restricted to the class-name specified in the header line. The symbol **::** is called the ***scope resolution*** operator.

For instance, consider the member functions **getdata () and putdataO** as discussed above. They may be coded as follows:

```
void item  ::  getdata(int a,  float b)

{
     number = a;
     cost  = b;
}


void item  ::  putdata(int a,  float b)

{
     cout << "Number : " << number  << "\n";
     cout  << "Cost : " << cost << "\n";
}
```

Since these functions do not return any value, their return type is void. Function arguments are declared using the ANSI prototype.

The member functions have some special characteristics that are often used in program development. These characteristics are :

1. Several different classes can use the same function name. The membership function will resolve their scope.
2. Member functions can access the private data of the class. A non member function cannot do so. However exception to this rule is a ***friend function*** .
3. A member function can call another member function directly without using the dot operator.

Inside the Class **Definition**

Another method of defining a member function is to replace function definition inside the class. for example, we can define item class as follows:

```
class  Item
{
      int  number;
      float cost:
      public;
      void getdata (int a,   float b);          //  declaration
            // inline  function
      void putdata'(void)
       {
            cout << "number  : " << number <<  "\n";
            cout << "cost : " << cost <<  "\n";
       }
  };
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an inline function are also applicable here. Normally, only small functions are defined inside the class definition.

A C++ program with a class.

```
#include<iostream.h>

class item
{
      int number;
      float cost;

      public :

      void getdata (int a, float b);
```

```
        void putdata (void)
        {
                cout << "Number : " << number << endl;
                cout << "Cost       : " << cost << endl;
        }
};

void item :: getdata (int a, float b)
{
        number = a;
        cost  = b;
}

int main( )
{
        item x;

        cout << "Object x : \n";

        x.getdata ( 100, 299.95);
        x.putdata( );
        item y;
        cout << "Object y : \n";
        y.getdata ( 200, 175.50);
        y.putdata( );
        return 0;
}
```

This program features the class **item.** This class contains two private variables and two public functions. The member function **getdata ( )** which has been defined outside the class supplies values to both the variables. Note the use of statements such as number = a;
In the function definition of **getdata( ).** This shows that the member functions can have direct access to private data items.

The member function **putdata( )** has been defined inside the class and therefore behaves like an **inline** function. This function displays the values of the private variables **number** and **cost**. The program creates two objects, x and y in two different statements. This can be combined in one statement.

```
item x,  y;       // creates a list of objects
```

Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition, It is therefore good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by using the qualifier **inline** in the header line of function definition. Example:

```
class item
{
        … …
        ... …
        public :
                void getdata(int a, int b);
};
inline void item :: getdata(int a, float b)        // definition
{
        number = a;
        cost = b;
}
```

Nesting Of Member functions

A member function can be called b using its name inside another member function of same class, This is known as nesting of member function.

Private Member Function

Although it is normal practice to place all the data items in a private section and all the function in public, some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator.

Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size=10;        // provides value for array size

class array
{
        int a [size];      //  'a'  is int type array
        public:
                setval (void);
                void display (void);
}
```

The array variable a[ ] declared as a private member of the class array can be used in functions, like any other array variable. We can perform any operations on it. In the above class definition, the member function setval ( ) sets the values of elements of the array a[ ], and display( ) function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

Memory Allocation for Objects

We have stated that the memory space for objects is allocated when they are declared not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different values for different objects.

Static Data Members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are :

1. It is initialized to zero when the first object of its class is created. No other initialization is permitted.
2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
3. It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all objects.

The type and scope of each static member variable must be defined outside class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as ***class variables***. The static member variable is initialized to zero when the objects are created.

Static Member Functions

Like static member variable, we can also have **static** member functions. A member function that is declared **static** has the following properties:

1. **A static** function can have access to only other static members (functions or variables) declared in the same class.

2. **A static** member function can be called using the class name (instead of its objects) as follows: classname :: static-function-name;

Arrays of Objects

We know that an array can be of any data type including **struct.** Similarly, we can also have arrays of variables that are of the type **class.** Such variables are called arrays of object. Consider the following class definition:

```
class employee
{
        char name[30];
        float age;
        public:
        void getdata(void);
        void putdata(void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];        //array of manager
employee foreman[15];       //array of foreman
employee worker[75];        //array of worker
```

An array of objects is stored inside the memory in the same way as a multi-dimensional. Note that only the space for data of the objects is created. Member functions are stored separately and will be used by all the objects.

Objects as Function Arguments

Like any other data type, an object may be used as a function argument. This can be done in two ways.

1. A copy of the entire object is passed to the function.
2 Only the address of the object is transferred to the function

The first method is called ***pass-by-value.*** Since a copy of the object is passed to function, any changes; made to the object inside the function do not affect the object used to call the function. The second method is called ***pass-by -reference***. When an address of the object is passed, the called function works directly on the actual object used in the function call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes manager and **scientist,** have been defined. We would like to use a function **income_tax** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes.

To make an outside function “friendly” to a class, we have to simplify declare this function as a friend of the class as shown below.

```
class ABC
{       … …
        … …
        public:
                … …
                … …
                friend void xyz(void); // declaration
}
```

The function declaration should be preceded by the keyword **friend.** The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope operator ::. The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any number of classes. A friend function, although not a member function. has rights to the private members of the class.

A friend function possesses certain special characteristics:

1. It is not in the scope of the class to which it has been declared as **friend.**
2. Since it is not in the scope of the class, it cannot be called using the object of that class.
3. It can be invoked like a normal function without the help of any object.
4. Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name. (i.e. A.x)
5. It can be declared either in the public or the private part of a class without affecting its meaning.
6. Usually, it has the objects as arguments.
7. Friend functions are often used in operator overloading.

Returning Objects

A function cannot only receive objects as arguments but also can return them. The following sample code illustrates how an object can be created (within a function) and returned to another function.

```
complex  complex :: sum (complex c2)

{
        complex c3;
        c3.x = x + c2.x;
        c3.y = y + c2.y;
        return c3;
}
```

Above member function can be invoked using following statement;

```
complex c1, c2, c3;
c3 = c1.sum (c2);
```

const Member Functions

If a member function does not alter any data in the class, then we may declare it as a const member function as follows:

```
void mul(int,  int)  const;
double get_balance()  const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition. The compiler will generate an error message if such functions try to alter the data values).

Pointers to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" member name. A class member pointer can be declared using the operator ::* with the class name. For example, given the class

```
class A
{
    private:
        int m;
    public:
        void show( );
};
```

We can define a pointer to the member m as follows:

```
int A::* ip = &A :: m;
```

The ip pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase A::* means "pointer-to-member of a class" The phrase &A::m means the "address of the m member of A class".

Remember, the following statement is not valid:

```
int *ip = &m;       // won't work
```

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer ip can now be used to access the member m inside member functions (or friend functions). Let us assume that a is an object of A declared in a member function. We can access m using the pointer ip as follows:

```
cout << a.*ip;     // display m
cout <<a.m;        // same as above
```

Now look at the following code:

```
ap = &a;                  // ap is pointer to object a
cout << ap  -> *ip;   // display m
cout << ap  -> m;     // same as above
```

The dereferencing operator ->* is used to access a member when we use pointers to both object and the member. The dereferencing operator .* is used when the object itself is used with the member pointer.

Note that *ip is used like a member name.
We can also design pointers to member functions which, then, can be invoked using the dereferencing operators in the main as shown below :

```
(object-name  .* pointer-to-member function)   (10);
(pointer-to-object  ->* pointer-to-member function)   (10);
```

The precedence of ( ) is higher than that of .* and ->*, so the parentheses are necessary.