Module 2

Functions, Classes and Objects

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main (). You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

- The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
- A function is known with various names like a method or a sub-routine or a procedure etc.

• *Defining a Function*

The general form of a C++ function definition is as follows −

```
return_type function_name (parameter list)
{
  //Body of the function
}
```

A function definition consists of a function header and a function body. Here are all the parts of a function −

1. Return Type : A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
2. Function Name : This is the actual name of the function. The function name and the parameter list together constitute the function signature.
3. Parameters : A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
4. Function Body : The function body contains a collection of statements that define what the function does.

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two −

```
int max(int num1, int num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;
  return result; }
```

- ***Function Declarations***

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts .
return_type function_name(parameter list);
For the above defined function max(), following is the function declaration.
int max(int num1, int num2);
Parameter names are not important in function declaration only their type is required, so following is also valid declaration.
int max(int, int);
Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

- ***Calling a Function***

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

- ***Function Arguments***

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
While calling a function, there are two ways that arguments can be passed to a function −

S.No	**Call Type & Description**
1	Call by Value: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	Call by Pointer: This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
3	Call by Reference: This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in

	the call. This means that changes made to the parameter affect the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Function call by value

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```
void swap(int x, int y)
{
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
```

Now, let us call the function swap() by passing actual values as in the following example.

```
#include <iostream>
using namespace std;
// function declaration
void swap(int x, int y);
int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
   // calling a function to swap the values.
   swap(a, b);
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

Function call by Pointer

The call by pointer method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument. To pass the value by pointer, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
void swap(int *x, int *y)
{
   int temp;
   temp = *x; /* save the value at address x */
   *x = *y; /* put y into x */
   *y = temp; /* put x into y */
   return;
}
```

For now, let us call the function swap() by passing values by pointer as in the following example.

```
#include <iostream>
using namespace std;
// function declaration
void swap(int *x, int *y);
int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
   /* calling a function to swap the values.
      * &a indicates pointer to a ie. address of variable a and
      * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

Function call by reference

The call by reference method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument. To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```
void swap(int &x, int &y)
{
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */
   return;
}
#include <iostream>
using namespace std;
void swap(int &x, int &y);
int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;
   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
   /* calling a function to swap the values using variable 
reference.*/
   swap(a, b);
   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;
   return 0;
}
```

- ***Inline Functions***

Inline function is one of the important features of C++. When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.

The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
   // function code   }
```

Remember, in-lining is only a request to the compiler, not a command. Compiler can ignore the request for in lining. Compiler may not perform in lining in such circumstances like:

1) If a function contains a loop. (For, while, do-while).
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn't exist in function body.
5) If a function contains switch or goto statement.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube(int s)
{
   return s*s*s;
}
int main()
{
   cout << "The cube of 3 is: " << cube(3)
<< "\n";
   return 0;
}
```

Overloading Functions

If any class have multiple functions with same names but different parameters then they are said to be overloaded. Function overloading allows you to use the same name for different functions, to perform, either same or different functions in the same class.

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

When you call an overloaded function or operator, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

- ***Different ways to Overload a Function***

  1. ***By changing number of Arguments.***
  2. ***By having different types of argument.***

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

- *Example 1: Function Overloading*

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

void display(int);
void display(float);
void display(int, float);

int main() {

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);

   return 0;
}

void display(int var) {
   cout << "Integer number: " << var << endl;
}
```

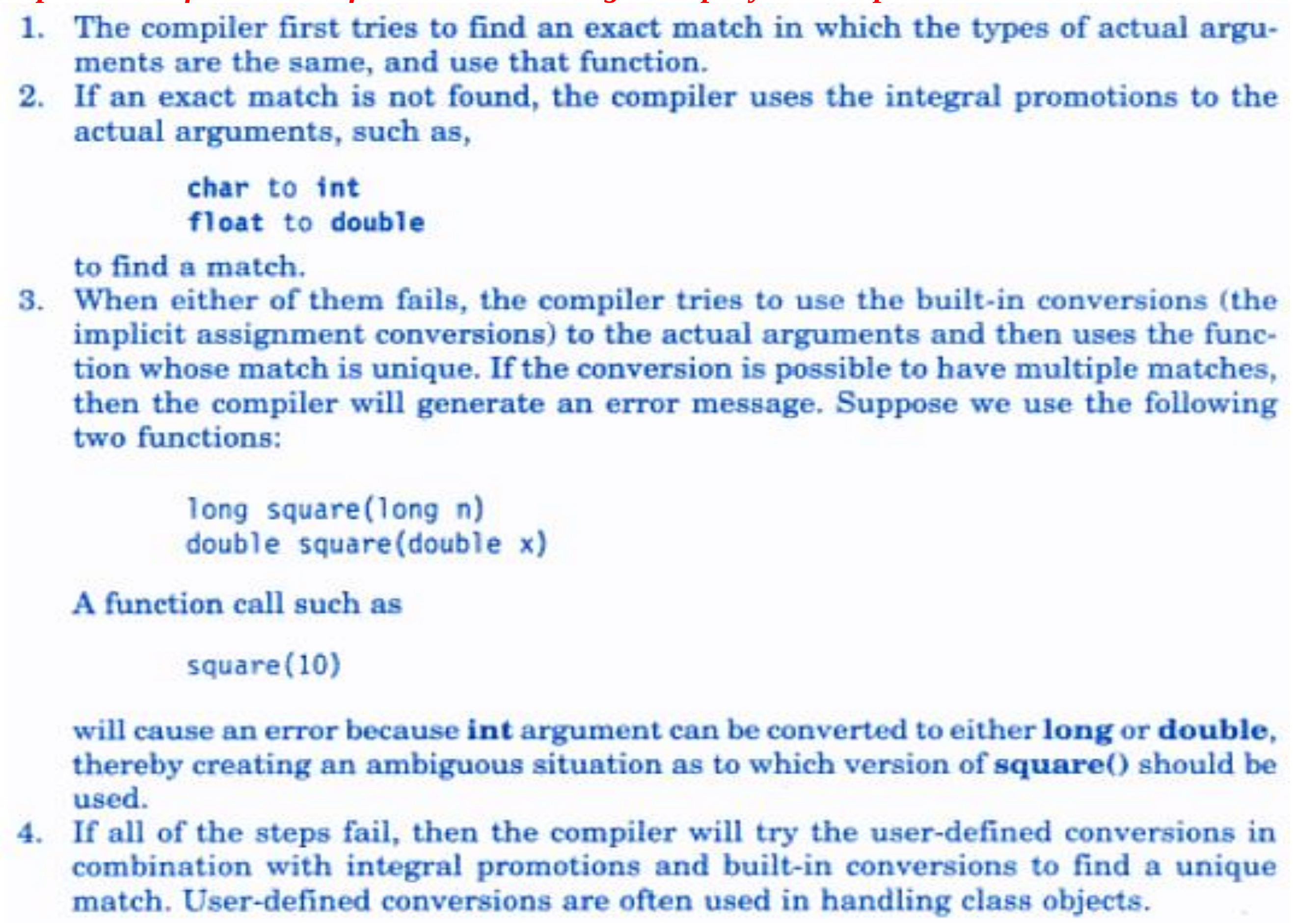
```
void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the display() function is called three times with different type or number of arguments. The return type of all these functions are same but it's not necessary.

Steps to select functions in function overloading concept by the compiler.



1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   ```
   char to int
   float to double
   ```

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square(long n)
   double square(double x)
   ```

   A function call such as

   ```
   square(10)
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

- ***Functions with Default Arguments***

When we mention a default value for a parameter while declaring the function, it is said to be as default argument. In this case, even if we make a call to the function without passing any value for that parameter, the function will take the default value specified.

```
sum(int x, int y=0){
   cout << x+y;
}
```

Here we have provided a default value for y, during function definition.

```
int main(){
   sum(10);
   sum(10,0);
   sum(10,10);
}
```

First two function calls will produce the exact same value, for the third function call, y will take 10 as value and output will become 20. By setting default argument, we are also overloading the function. Default arguments also allow you to use the same function in different situations just like function overloading.

- ***Rules for using Default Arguments***

1. Only the last argument must be given default value. You cannot have a default argument followed by non-default argument.
   ```
   sum (int x,int y);
   sum (int x,int y=0);
   sum (int x=0,int y);  // This is Incorrect
   ```
2. If you default an argument, then you will have to default all the subsequent arguments after that.
   ```
   sum (int x,int y=0);
   sum (int x,int y=0,int z);  // This is incorrect
   sum (int x,int y=10,int z=10);  // Correct
   ```
3. You can give any value a default value to argument, compatible with its datatype.

- ***Function with Placeholder Arguments***

When arguments in a function are declared without any identifier they are called placeholder arguments.

```
void sum (int, int);
```

Such arguments can also be used with default arguments.

```
void sum (int, int=0);
```

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 build-in data type.
Generally, a class specification has two parts:
1. Class declaration
2. Class function definitions
The class declaration describes the type 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 declaration;
          function declaration;
     public:
         variable declaration;
         function declaration;
};
```

A class is a blueprint for the object. We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.

- ***How to define a class in C++?***

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className
  {
  // some data
  // some functions
  };
```

- ***Example: Class in C++***

```
class Test {
    private:
        int data1;
        float data2;
    public:
        void function1(){
                data1 = 2;
         }
        float function2() {
            data2 = 3.5;
            return data2;
        }
  };
```

Here, we defined a class named Test.

This class has two data members: data1 and data2 and two member functions: function1() and function2().

- ***Keywords: private and public***

You may have noticed two keywords: private and public in the above example.
The private keyword makes data and functions private. Private data and functions can be accessed only from inside the same class.
The public keyword makes data and functions public. Public data and functions can be accessed out of the class.
Here, data1 and data2 are private members where as function1() and function2() are public members.
If you try to access private data from outside of the class, compiler throws error. This feature in OOP is known as data hiding.

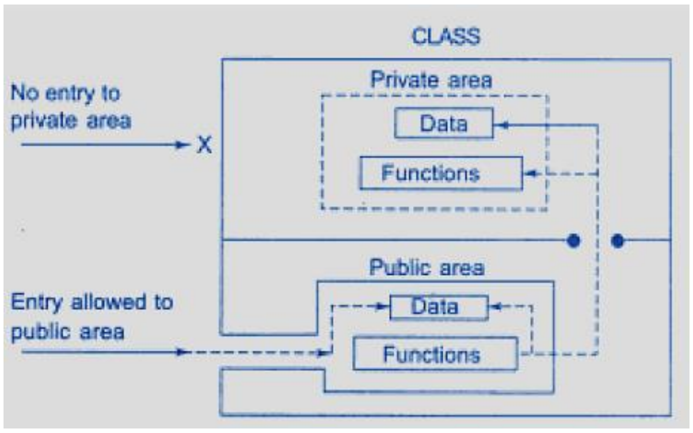


Figure: Data hiding in classes

Types of Member functions:

- *Access functions,*
- *Utility Functions,*
- *Constructors,*
- *Destructors,*
- *Overloaded Operators*

C++ Objects

When class is defined, only the specification for the object is defined; no memory or storage is allocated.
To use the data and access functions defined in the class, you need to create objects.
Syntax to Define Object in C++

className objectVariableName;

You can create objects of Test class (defined in above example) as follows:

```
class Test{
   private:
       int data1;
       float data2;
   public:
       void function1() {
        data1 = 2;  }

       float function2() {
           data2 = 3.5;
           return data2; }
  };

int main(){
   Test o1, o2;
}
```

Here, two objects o1 and o2 of Test class are created.
In the above class Test, data1 and data2 are data members and function1() and function2() are member functions.

- ***How to access data member and member function in C++?***

You can access the data members and member functions by using a . (dot) operator. For example,

o2.function1();

This will call the function1() function inside the Test class for objects o2.
Similarly, the data member can be accessed as:

o1.data2 = 5.5;

It is important to note that, the private members can be accessed only from inside the class.
So, you can use o2.function1(); from any function or class in the above example. However, the code o1.data2 = 5.5; should always be inside the class Test.

- ***Example: Object and Class in C++ Programming***

// Program to illustrate the working of objects and class in C++ Programming

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

class Test {
   private:
           int data1;
           float data2;

   public:
             void insertIntegerData(int d) {
          data1 = d;
          cout << "Number: " << data1;
        }
```

```
      float insertFloatData()  {
          cout << "\nEnter data: ";
          cin >> data2;
          return data2;
       }
};

 int main() {
     Test o1, o2;
     float secondDataOfObject2;
     o1.insertIntegerData(12);
     secondDataOfObject2 = o2.insertFloatData();
     cout << "You entered " << secondDataOfObject2;
     return 0;
 }
```

Output

```
Number: 12
Enter data: 23.3
You entered 23.3
```

In this program, two data members data1 and data2 and two member functions insertIntegerData() and insertFloatData() are defined under Test class.Two objects o1 and o2 of the same class are declared. The insertIntegerData() function is called for the o1 object using:o1.insertIntegerData(12);This sets the value of data1 for object o1 to 12. Then, the insertFloatData() function for object o2 is called and the return value from the function is stored in variable secondDataOfObject2 using:secondDataOfObject2 = o2.insertFloatData(); In this program, data2 of o1 and data1 of o2 are not used and contains garbage value.

Arrays within Class:

Arrays can be declared as the members of a class. The arrays can be declared as private, public or protected members of the class. To understand the concept of arrays as members of a class, consider this example.

```
    #include<iostream>
    using namespace std;
    const int size=5;
    class student{
             int roll_no;
               int marks[size];
        public:
              void getdata ();
             void tot_marks ();
    } ;
void student :: getdata (){
     cout<<"\nEnter roll no: ";
    cin>>roll_no;
    for(int i=0; i<size; i++){
            cout<<"Enter marks in subject"<<(i+1)<<": ";
           cin>>marks[i] ;}
    }
void student :: tot_marks() //calculating total marks
{
int total=0;
for(int i=0; i<size; i++)
total+ = marks[i];
cout<<"\n\nTotal marks "<<total;
}
int main()
student stu;
stu.getdata() ;
stu.tot_marks() ;
return 0;
}
```

The output of the program is

```
Enter roll no: 101
Enter marks in subject 1: 67
Enter marks in subject 2 : 54
Enter marks in subject 3 : 68
Enter marks in subject 4 : 72
Enter marks in subject 5 : 82
Total marks = 343
```

In this example, an array marks is declared as a private member of the class student for storing a student's marks in five subjects. The member function tot_marks () calculates the total marks of all the subjects and displays the value.
Similar to other data members of a class, the memory space for an array is allocated when an object of the class is declared. In addition, different objects of the class have their own copy of the array. Note that the elements of the array occupy contiguous memory locations along with other data members of the object. For instance, when an object stu of the class student is declared, the memory space is allocated for both rollno and marks

Memory allocation for objects:

Once a class is defined, it can be used to create variables of its type known as objects. The relation between an object and a class is the same as that of a variable and its data type. The syntax for declaring an object is

class_name = object_list;

where,
class_name = the name of the class
object_list = a comma-separated list of objects.

To understand the concept of instantiation, consider this example.
Example : Declaring objects of a class

```
class book{
      //  body of the class
} ;
int main (){
       book bookl, book2, book3; //objects of class book
       return 0;
}
```

In this example, three objects namely, book1, book2 and book3 of the class book have been created in main ().
Note that the syntax for the declaration of objects of a particular class is same as that for the declaration of variables of a built-in data type. To understand this concept, consider these statements.

```
int a,b,c; // declaration of variables
book bookl, book2, book3; // declaration of objects
```

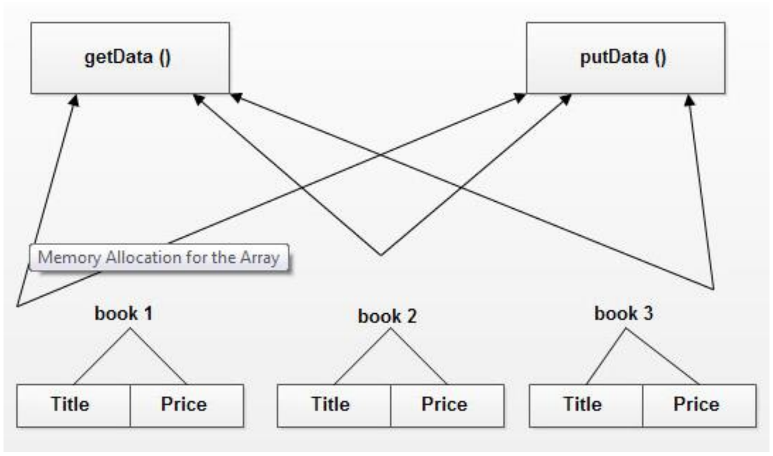
Like structures, objects of a class can also be declared at the time of defining the class as shown here.

```
class class_name{
    Object_list;
}
```

Hence, the objects of the class book can also be declared with the help of these statements.

```
class book{
    // body of class
}    bookl,book2,book3; // declaration of objects
```

1. Before using a member of a class, it is necessary to allocate the required memory space to that member. The way the memory space for data members and member functions is allocated is different regardless of the fact that both data members and member functions belong to the same class.
2. The memory space is allocated to the data members of a class only when an object of the class is declared, and not when the data members are declared inside the class. Since a single data member can have different values for different objects at the same time, every object declared for the class has an individual copy of all the data members.
3. On the other hand, the memory space for the member functions is allocated only once when the class is defined. In other words, there is only a single copy of each member function, which is shared among all the objects.
4. For instance, the three objects, namely, book1, book2 and book3 of the class book have individual copies of the data members title and price. However, there is only one copy of the member functions getdata () and putdata () that is shared by all the three objects.



Arrays of objects

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type. The syntax for declaring an array of objects is

class_name array_name [size] ;

To understand the concept of an array of objects, consider this example.

```
#include<iostream>
using namespace std;
class books{
        char tit1e [30];
        float price ;
    public:
        void getdata ();
        void putdata ();
} ;
void books :: getdata (){
    cout<<"Title:”;
    Cin>>title;
    cout<<"Price:”;
    cin>>price;
 }
void books :: putdata (){
    cout<<"Title:"<<title<< "\n";
    cout<<"Price:"<<price<< "\n”;
    const int size=3 ;
}
```

```
int main (){
    books book[size] ;
    for(int i=0;i<size;i++){
                                         cout<<"Enter details o£ book "<<(i+1)<<"\n";
               book[i].getdata();
    }
    for(int i=0;i<size;i++){
               cout<<"\nBook "<<(i+l)<<"\n";
               book[i].putdata() ;
    }
    return 0;
}
```

The output of the program is

```
Enter details of book 1
Title: c++
Price: 325
Enter details of book 2
Title: DBMS
Price:. 455
Enter details of book 3
Title: Java
Price: 255
Book 1
Title: c++
Price: 325
Book 2
Title: DBMS
Price: 455
Book 3
Title: Java
Price: 255
```

In this example, an array book of the type class books and size three is declared. This implies that book is an array of three objects of the class books. Note that every object in the array book can access public members of the class in the same way as any other object, that is, by using the dot operator. For example, the statement book [i] . getdata () invokes the getdata () function for the ith element of array book. When an array of objects is declared, the memory is allocated in the same way as to multidimensional arrays. For example, for the array book, a separate copy of title and price is created for each member book[0], book[l] and book[2]. However, member functions are stored at a different place in memory and shared among all the array members. For instance, the memory space is allocated to the the array of objects book of the class books,

Friend Function:

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function. The complier knows a given function is a friend function by the use of the keyword **friend**. For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.
Declaration of friend function

```
class class_name
{
   ... .. ...
   friend return_type function_name(argument/s);
   ... .. ...
}
```

Now, you can define the friend function as a normal function to access the data of the class.
No friend keyword is used in the definition.

```
class className
{
   ... .. ...
   friend return_type functionName(argument/s);
   ... .. ...
}

return_type functionName(argument/s)
{
   ... .. ...
   // Private and protected data of className can be accessed from
   // this function because it is a friend function of className.
   ... .. ...
}
```

/* C++ program to demonstrate the working of friend function.*/

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

class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0) { }
      //friend function
      friend int addFive(Distance);
};
```

```
// friend function definition
int addFive(Distance d)
{
   //accessing private data from non-member function
   d.meter += 5;
   return d.meter;
}

int main()
{
   Distance D;
   cout<<"Distance: "<< addFive(D);
   return 0;
}
```

Here, friend function addFive() is declared inside Distance class. So, the private data metercan be accessed from this function. A more meaningful use would to when you need to operate on objects of two different classes. That's when the friend function can be very helpful. You can definitely operate on two objects of different classes without using the friend function but the program will be long, complex and hard to understand.
Following are some important points about friend functions and classes:
1) Friends should be used only for limited purpose. too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.
2) Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.
3) Friendship is not inherited

- ***Using with Objects***

For accessing normal data members we use the dot . operator with object and -> qith pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

Object.*pointerToMember

and with pointer to object, it can be accessed by writing,

ObjectPointer->*pointerToMember

Lets take an example, to understand the complete concept.

```
class Data{
    public:
            int a;
             void print() { cout << "a is "<< a; }
};
```

```
int main(){
    Data d, *dp;
    dp = &d;     // pointer to object
    int Data::*ptr=&Data::a;   // pointer to data member 'a'
    d.*ptr=10;
    d.print();    // Using Object
    dp->*ptr=20;
    dp->print();   // Using Pointer to object
}
```

Output :

a is 10 a is 20

Pointers to class members

Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

- ***Defining a pointer of class type***

We can define pointer of class type, which can be used to point to class objects.

```
class Simple{
    public:
                int a;
};

int main(){
     Simple obj;
     Simple* ptr;   // Pointer of class type
     ptr = &obj;
     cout << obj.a;
     cout << ptr->a;  // Accessing member with pointer
}
```

Here you can see that we have declared a pointer of class type which points to class's object. We can access data members and member functions using pointer name with arrow -> symbol.

Pointer to Data Members of class

We can use pointer to point to class's data members (Member variables).

Syntax for Declaration :

datatype class_name :: *pointer_name ;

Syntax for Assignment :

pointer_name = &class_name :: datamember_name ;

Both declaration and assignment can be done in a single statement too.

datatype class_name::*pointer_name = &class_name::datamember_name ;

Pointer to Member Functions

Pointers can be used to point to class's Member functions.

Syntax :

return_type (class_name::*ptr_name) (argument_type) = &class_name::function_name ;

Below is an example to show how we use ppointer to member functions.

```
class Data{
    public:
              int f (float) { return 1; }
};
int (Data::*fp1) (float) = &Data::f;   // Declaration and assignment
int (Data::*fp2) (float);       // Only Declaration

int main(){
    fp2 = &Data::f;   // Assignment inside main()
}
```

Some Points to remember

1. *You can change the value and behaviour of these pointers on runtime. That means, you can point it to other member function or member variable.*
2. *To have pointer to data member and member functions you need to make them public.*

VTU Question Paper Questions :

1. Mention the restriction imposed by the compiler on inline functions?
2. Discuss the different types of function overloading in c++?
3. When do we use default arguments? State the rules that we need to follow while using default arguments in a function?
4. Draw a neat diagram and explain the process of memory allocation to objects in C++?
5. Develop a C++ program to define two classes namely husband and wife that hold a private member “salary” respectively. Claculate and display the total income of the family using friend function?
6. Design a class “trinagle” containing data items “base, “height” and four member functions set_data(), get_data(), display_data() and find_area(), to set values of “base” and “height” to get the user input , to display and to find the area of the triangle respectively. Write the main function which creats the object and uses the members of the class.
7. Explain dynamic memory allocation in C++. What is memory leak?
8. Explain different ways of argument passing in a function using swap() to swap two numbers?