MODULE-2 Classes and Objects

Definition of class, object, Difference between class and structure, class definitions, member functions, access specifiers, Creation of Objects, Passing and Returning objects, Object assignment and array of objects. Constructors and its Types, Destructors, Nesting member function, Private member function, Inline functions. Static class members, Call by value, Call by reference, Friend functions, this pointer.

Introduction

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

One of the unique features of the C language is structures. They provide a method for packing together data of different types. A structure is a convenient tool for handling group of logically related data items. It is a user-defined datatype 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 declaration:

```
struct student {
        char name[20];
        int ro11_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 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 , "John");
```

A.ro1l_number = 999;

A.total marks = 595.5;

Final_total = A. total_marks* 5;

Structures 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 types. For example, consider the following structure:

```
struct complex{
        float x;
        float y;
        };
struct complex c1, 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 = c1 + 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 the capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types 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 variables 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 variables. For example, we can declare the student variable A as

student A; // C++ declaration

Remember, this is an Error in 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 structure for holding only data, and classes to hold both the data and functions.

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.

The following table summarizes all of the fundamental differences.

Sl.No.	**Structure**	**Class**
1	Members of a structure are public by default.	Members of a class are private by default.
2	It is declared using the **struct** keyword.	It is declared using the **class** keyword.
3	It is normally used for the grouping of different datatypes.	It is normally used for data abstraction and inheritance.

4	**Syntax:**
	struct structure_name {
	structure_member1;
	structure_member2;
	};
		Syntax:
		class class_name {
		data_member;
		member_function;
		};

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.

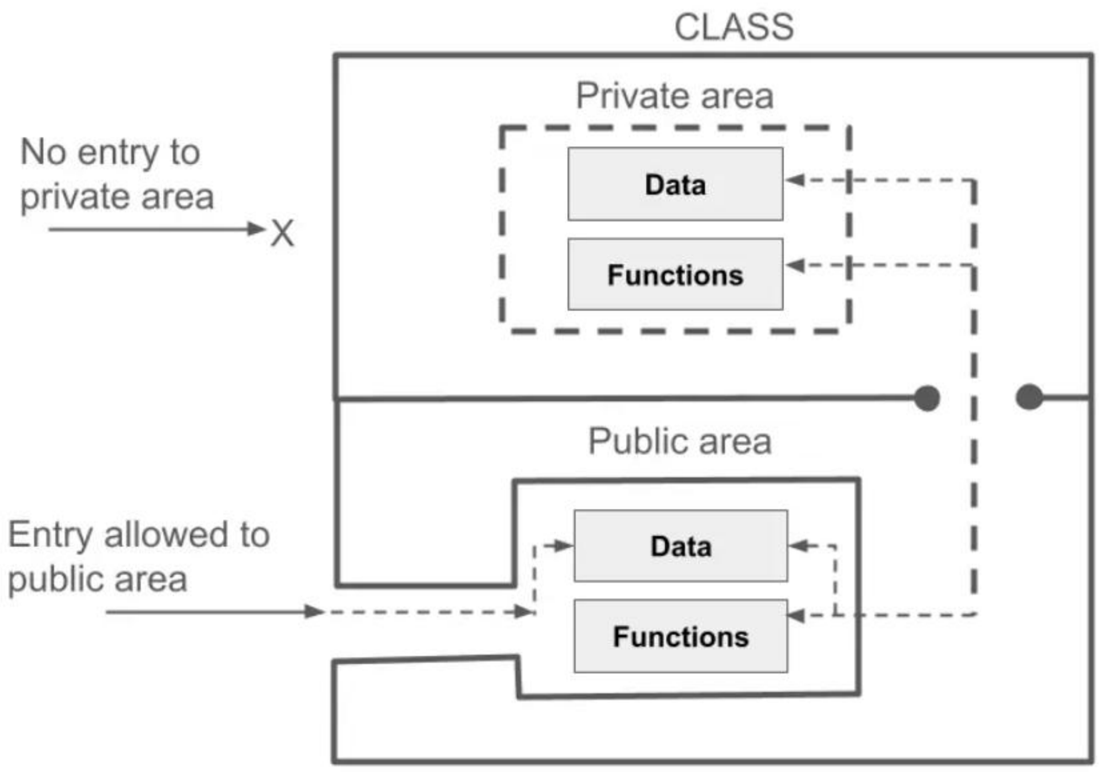
The general form of a class declaration is:

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

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 arc 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 the below 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; // variables declaration
        float cost; // private by default
 public:
        void getdata(int a, float b); // functions declaration
        void putdata(void); // using prototype
}; // ends with semicolon
```

class item 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. The data members are usually declared as private and the member functions as public.

Accessing Class Members

The private data of a class can be accessed only through the member functions of that class. The main ( ) cannot contain statements that access number and cost directly. The following is the format for calling a member function:

```
object-name.function-name (actual-arguments);
```

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 getdata() 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.

Defining Member Functions

Member functions can be defined in two places:

- Outside the class definition.
- 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.

Outside the Class Definition

Member functions that are declared inside a class have to be defined separately outside the class. 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 putdata() as discussed above. They may be coded as follows:

```
void item :: getdata(int a, float b)
{
        number = a;
        cost = b;
}
void item :: putdata(void)
{
        cout << "Number :" << number « "\n";
        cout << "Cost :" << cost << "\n";
}
```

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

- Several different classes can use the same function name. The membership label will resolve their scope.
- Member functions can access the private data of the class. A non-member function cannot do so.
- 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 the function declaration by the actual function definition inside the class. For example, we could define the item class as follows:

```
class item
{
        int number;
        float cost;
    public:
        void getdata(int a, float b); // declaration
         // inline function
        void putdata(void)      // definition inside the class
        {
                cout << numher << "\n";
                cout << cost << "\n";
```

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

Access Specifiers

One of the main features of object-oriented programming languages such as C++ is **data hiding**. Data hiding refers to restricting access to data members of a class. This is to prevent other functions and classes from tampering with the class data.

However, it is also important to make some member functions and member data accessible so that the hidden data can be manipulated indirectly.

The access modifiers of C++ allows us to determine which class members are accessible to other classes and functions, and which are not.

In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

public Access Modifier

- The public keyword is used to create public members (data and functions).
- The public members are accessible from any part of the program.

Example 1: C++ public Access Modifier

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

// define a class
class Sample {

    // public elements
   public:
    int age;

    void displayAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // declare a class object
    Sample obj1;

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

    // call class function
    obj1.displayAge();

    return 0;
}
```

Run Code

Output

```
Enter your age: 20
Age = 20
```

In this program, we have created a class named Sample, which contains a public variable age and a public function displayAge().

In main(), we have created an object of the Sample class named obj1. We then access the public elements directly by using the codes obj1.age and obj1.displayAge().

Notice that the public elements are accessible from main(). This is because public elements are accessible from all parts of the program.

private Access Modifier

- The private keyword is used to create private members (data and functions).
- The private members can only be accessed from within the class.
- However, friend classes and friend functions can access private members.

Example 2: C++ private Access Specifier

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

// define a class
class Sample {

    // private elements
   private:
    int age;

    // public elements
   public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }
};

int main() {

    int ageInput;

    // declare an object
    Sample obj1;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call function and pass ageInput as argument
    obj1.displayAge(ageInput);
    return 0;
}
```

Run Code

Output

```
Enter your age: 20
Age = 20
```

In `main()`, the object `obj1` cannot directly access the class variable `age`.

```
// error
cin >> obj1.age;
```

We can only indirectly manipulate `age` through the public function `displayAge()`, since this function initializes `age` with the value of the argument passed to it i.e. the function parameter `int a`.

protected Access Modifier

Before we learn about the `protected` access specifier, make sure you know about inheritance in C++.

- The `protected` keyword is used to create protected members (data and function).
- The protected members can be accessed within the class and from the derived class.

Example 3: C++ protected Access Specifier

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

// declare parent class
class Sample {
    // protected elements
   protected:
    int age;
};

// declare child class
class SampleChild : public Sample {

  public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }

};

int main() {
    int ageInput;

    // declare object of child class
    SampleChild child;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call child class function
    // pass ageInput as argument
    child.displayAge(ageInput);

    return 0;
}
```
Run Code

Output

```
Enter your age: 20
```

```
Age = 20
```

Here, SampleChild is an inherited class that is derived from Sample. The variable age is declared in Sample with the protected keyword.

This means that SampleChild can access age since Sample is its parent class.

We see this as we have assigned the value of age in SampleChild even though age is declared in the Sample class.

Summary: public, private, and protected

- public elements can be accessed by all other classes and functions.
- private elements cannot be accessed outside the class in which they are declared, except by friend classes and functions.
- protected elements are just like the private, except they can be accessed by derived classes.

Specifiers	Same Class	Derived Class	Outside Class
public	Yes	Yes	Yes
private	Yes	No	No
protected	Yes	Yes	No

Note: By default, class members in C++ are private, unless specified otherwise.

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 type variable). For example.

```
item x; // memory for x is created
```

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.

Passing and Returning objects

In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables. No special keyword or header file is required to do so.
In C++ programming, we can pass objects to a function in a similar manner as passing regular arguments.

Passing an Object as argument

To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

Syntax:

```
function_name (object_name)
```

Example 1: C++ Pass object to a Function

An object can be passed to a function just like we pass structure to a function. Here in class A we have a function disp() in which we are passing the object of class A. Similarly we can pass the object of another class to a function of different class.

```
#include <iostream>
using namespace std;
class A {
   public:
    int age = 20;
    char ch = 'A';

    // function that has objects as parameters
    void display(A &a) {

       cout << "Age = " << a.age << endl;
       cout << "Character = " << a.ch << endl;

    }
};
int main() {
    A obj;
    obj.display(obj);
    return 0;
}
```

Output

```
Age = 20
Character = A
```

Example 2: C++ Program to calculate the average age of two students

```
// C++ Program to understand Pass Objects to Function

#include <iostream>
using namespace std;
class Student {
   public:
    int age;

    // constructor to initialize age
    Student(int a) {
        age = a;
    }
};
// function that has objects as parameters
void calculateAverage(Student s1, Student s2) {
    // calculate the average of age of s1 and s2
    int average = (s1.age + s2.age) / 2;

   cout << "Average Age = " << average << endl;

}
int main() {
    Student student1(10), student2(20);
    // pass the objects as arguments
        calculateAverage(student1, student2);

    return 0;
}
```

Output

```
Average age = 15
```

Here, we have passed two `Student` objects `student1` and `student2` as arguments to the `calculateAverage()` function.

Returning an Object from a Function

In this example we have two functions, the function input() returns the Student object and display() takes Student object as an argument.

Example 3: C++ Return object to a Function

To print the content of a void pointer, we use the `static_cast` operator. It converts the pointer from `void*` type to the respective data type of the address the pointer is storing:

```
#include <iostream>
using namespace std;
class Student {
   public:
    int stId;
    int stAge;
    string stName;

    // In this function we returning the student object
    Student input(int n, int a, string s) {
        Student obj:
        obj.stId = n;
        obj.stAge = a;
        obj.stName = s;
        return obj;
    }
    // In this function we pass object as an argument
    void display(Student obj) {

       cout << "Name = " << onj.stName << endl;
       cout << "Id = " << onj.stId << endl;
       cout << "Age = " << onj.stAge << endl;
    }
};
int main() {
    Student s;
    s = s.input (1005, 20, James)
    s.display(s);
    return 0;
}
```

Output

```
Name = James
Id = 1005
Age = 20
```

Example 4: C++ Return Object from a Function

```
#include <iostream>
using namespace std;
class Student {
   public:
    int age1, age2;
};
// function that returns object of Student
Student newStudent() {
    Student student;

    // Initialize member variables of Student
    student.age1 = 10;
    student.age2 = 20;

    // print member variables of Student
    cout << "Age of Student 1 = " << student.age1 << endl;
    cout << "Age of Student 2 = " << student.age2 << endl;

    return student;
}
int main() {
    Student student1;

    // Call function
    student1 = newStudent();

    return 0;
}
```

Output

```
Age of Student 1 = 15
Age of Student 2 = 20
```

In this program, we have created a function newStudent() that returns an object of Student class.

We have called newStudent() from the main() method.

```
// Call function
student1 = newStudent();
```

Here, we are storing the object returned by the newStudent() method in the student1.

Object assignment and array of objects

In C++, an array of objects allows you to store and work with multiple objects of the same class in a structured manner. It provides a convenient way to group related objects together and perform operations on them collectively. With the help of an array of objects, you can efficiently manage and manipulate a collection of objects with similar properties and behaviors.

An array of objects in C++ is a collection of objects of the same class type stored in contiguous memory locations. It allows you to create & manage multiple objects with similar properties & behaviors in a single data structure. Each element in the array represents an individual object, and you can access these objects using their respective indices. By using an array of objects, you can efficiently perform operations on multiple objects simultaneously, like initialization, data manipulation, and iteration. This helps in code organization, reduces redundancy, and simplifies common tasks when working with a group of related objects.

Declaration of Array Of Objects In C++

To declare an array of objects in C++, you need to specify the class name followed by the array name and the desired size of the array. The syntax for declaring an array of objects is:

```
ClassName arrayName[size];
```

Here, `ClassName` is the name of the class for which you want to create an array of objects, `arrayName` is the name you choose for the array, and `size` is the number of objects you want to store in the array.

Initializing Array Of Objects In C++

After declaring an array of objects, you need to initialize the objects with appropriate values. There are several ways to initialize an array of objects in C++, like:

1. Using the default constructor: If the class has a default constructor (a constructor that takes no arguments), you can simply declare the array, and each object will be initialized with the default constructor.

```
  ClassName arrayName[size];
```

For example:

```
  Rectangle rectangles[5];
```

2. Using an initialization list: You can use an initialization list to provide initial values for each object in the array at the time of declaration.

```
  ClassName arrayName[size] = { { arguments_for_object1 }, { arguments_for_object2 }, ... };
```

For example:

```
  Rectangle rectangles[3] = { {3, 4}, {5, 6}, {7, 8} };
```

In this case, each object in the `rectangles` array is initialized with the provided length and width values.

3. Using a loop to initialize objects: You can use a loop to iterate over the array and initialize each object individually.

```
  for (int i = 0; i < size; i++) {
      // Initialize each object using member functions or assignment
      arrayName[i].memberFunction(arguments);
      arrayName[i].dataMember = value;
  }

  For example:



  for (int i = 0; i < 5; i++) {
      rectangles[i].setDimensions(i+1, i+2);
  }
```

In this case, each `Rectangle` object in the `rectangles` array is initialized with different length and width values using the `setDimensions()` member function.

For Example,

```
class Student {
public:
    string name;
    int age;
    void displayInfo() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
int main() {
    const int SIZE = 3;
    Student students[SIZE];
    for (int i = 0; i < SIZE; i++) {
        cout << "Enter name & age for student " << i + 1 << ": ";
        cin >> students[i].name >> students[i].age;
    }
    cout << "\nStudent Information:" << endl;
    for (int i = 0; i < SIZE; i++) {
        students[i].displayInfo();
    }
    return 0;
}
```

In this example, we define a `Student` class with `name` and `age` attributes and a `displayInfo()` member function. We create an array of `Student` objects called `students` with a size of 3. We then use loops to input data for each student object and display their information.
The array of objects helps us store and manage multiple `Student` objects efficiently, providing a structured way to work with related data.

Constructors and its Types, Destructors

In C++, **constructor** is a special method that is invoked automatically at the time an object of a class is created. It is used to initialize the data members of new objects. The constructor in C++ has the same name as the class or structure.

Example:

```
#include <iostream>
using namespace std;
class A {
public:
      // Constructor of the class without
    // any parameters
    A() {
        cout << "Constructor called" << endl;
    }
};
int main() {
    A obj1;
    return 0;
}
```

Output

```
Constructor called
```

Types of Constructors in C++

Constructors can be classified based on in which situations they are being used. There are 4 types of constructors in C++:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Move Constructor**

1. Default Constructor

A default constructor is a constructor that doesn't take any argument. It has no parameters. It is also called a zero-argument constructor. The compiler automatically creates an implicit default constructor if the programmer does not define one.

Example:

```
#include <iostream>
using namespace std;
// Class with no explicity defined constructors
class A {
public:
};
int main() {
      // Creating object without any parameter
    A a;
      cout << "In Main";
    return 0;
}
```

Output

```
In Main
```

The object of the **class A** is created without any parameters, hence the default constructor exists.

2. Parameterized Constructor

Parameterized constructors make it 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.

Example:

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

class A {
public:
    int val;

      // Parameterized Constructor
    A(int x) {
        val = x;
    }
};

int main() {

      // Creating object with a parameter
    A a(10);
    cout << a.val;
    return 0;
}
```

Output

```
10
```

In this code, the parameterized constructor is called when we create object **a** with integer argument **10** . As defined, it initializes the member variable **val** with the value **10**.

***Note**: If a parameterized constructor is defined, the non-parameterized constructor should also be defined as compiler does not create the default constructor.*

3. Copy Constructor

A copy constructor is a member function that initializes an object using another object of the same class. Copy constructor takes a reference to an object of the same class as an argument.

Example:

```
#include <iostream>
using namespace std;
class A {
public:
    int val;
        // Parameterized constructor
    A(int x) {
        val = x;
    }
        // Copy constructor
    A(A& a) {
        val = a.val;
    }
};

int main() {
    A a1(10);
        // Creating another object from a1
    A a2(a1);
          cout << a2.val;
    return 0;
}
```

Output

```
10
```

In this example, the copy constructor is used to create a new object **a2** as a copy of the object **a1**. It is called automatically when the object of **class A** is passed as constructor argument.

Just like the default constructor, the C++ compiler also provides an implicit copy constructor if the explicit copy constructor definition is not present.

Here, it is to be noted that, unlike the default constructor where the presence of any type of explicit constructor results in the deletion of the implicit default constructor, the implicit copy constructor will always be created by the compiler if there is no explicit copy constructor or explicit move constructor is present.

4. Move Constructor

The move constructor is a recent addition to the family of constructors in C++. It is like a copy constructor that constructs the object from the already existing objects, but instead of copying the object in the new memory, it makes use of move semantics to transfer the ownership of the already created object to the new object without creating extra copies.

It can be seen as stealing the resources from other objects.

Syntax:

className (className&& obj) {

// body of the constructor

}

The move constructor takes the rvalue reference of the object of the same class and transfers the ownership of this object to the newly created object.

Like a copy constructor, the compiler will create a move constructor for each class that does not have any explicit move constructor.

Characteristics of Constructors

- The name of the constructor is the same as its class name.
- Constructors are mostly declared as public member of the class though they can be declared as private.
- Constructors do not return values, hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Multiple constructors can be declared in a single class
- In case of multiple constructors, the one with matching function signature will be called.

Destructors in C++

Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

Syntax to Define Destructor

The syntax for defining the destructor within the class:

```
~ <class-name>() {
   // some instructions
}
```

Just like any other member function of the class, we can define the destructor outside the class too:

```
<class-name> {
public:
    ~<class-name>();
}

<class-name> :: ~<class-name>() {
   // some instructions
}
```

But we still need to at least declare the destructor inside the class.

Characteristics of a Destructor

- A destructor is also a special member function like a constructor. Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence, destructor cannot be overloaded.
- It cannot be declared static or const.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

Examples of Destructor

The below programs demonstrate the behaviour of the destructor in different cases:

Example 1

The below code demonstrates the automatic execution of constructors and destructors when objects are created and destroyed, respectively.

```
// C++ program to demonstrate the execution of constructor
// and destructor
#include <iostream>
using namespace std;
class Test {
public:
    // User-Defined Constructor
    Test() { cout << "\n Constructor executed"; }
    // User-Defined Destructor
    ~Test() { cout << "\nDestructor executed"; }
};
main()
{
    Test t;
    return 0;
}
```

Output

```
 Constructor executed
Destructor executed
```

Example 2

The below C++ program demonstrates the number of times constructors and destructors are called.

```
// C++ program to demonstrate the number of times
// constructor and destructors are called
#include <iostream>
using namespace std;
// It is static so that every class object has the same
// value
static int Count = 0;
class Test {
public:
    // User-Defined Constructor
    Test()
    {
        // Number of times constructor is called
        Count++;
        cout << "No. of Object created: " << Count << endl;
    }
    // User-Defined Destructor
    ~Test()
    {
        // It will print count in decending order
        cout << "No. of Object destroyed: " << Count
             << endl;
        Count--;
        // Number of times destructor is called
    }
};
// driver code
int main()
{
    Test t, t1, t2, t3;
    return 0;
}
```

Output

```
No. of Object created: 1
No. of Object created: 2
No. of Object created: 3
No. of Object created: 4
No. of Object destroyed: 4
No. of Object destroyed: 3
No. of Object destroyed: 2
No. of Object destroyed: 1
```

Note: *Objects are destroyed in the reverse order of their creation. In this case, t3 is the first to be destroyed, while t is the last.*

When is the destructor called?

A destructor function is called automatically when the object goes out of scope or is deleted. Following are the cases where destructor is called:

1. Destructor is called when the function ends.
2. Destructor is called when the program ends.
3. Destructor is called when a block containing local variables ends.
4. Destructor is called when a delete operator is called.

How to call destructors explicitly?

Destructor can also be called explicitly for an object. We can call the destructors explicitly using the following statement:

```
object_name.~class_name()
```

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, the compiler creates a default destructor for us. **The default destructor works fine unless we have dynamically allocated memory or pointer in class.** When a class contains a pointer to memory allocated in the class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leaks.

Nesting of Member Functions

A member function of a class can be called only by an abject of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
#include<iostream>
using namespace std;
class nest
{
    int a;
    int square_num( )
    {
        return a* a;
    }
    public:
    void input_num( )
    {
    cout << ”\nEnter a number ”;
    cin>>a;
    }
    int cube_num( )
    {
        return a* a*a;
    }
void disp_num()
{
    int sq=square_num();        //nesting of member function
    int cu=cube_num();  //nesting of member function
    cout<<”\nThe square of “<<a<<” is  ” <<sq;
    cout<<”\nThe cube of “<<a<<” is  ” <<cu;
}
};
```

```
int main()
{
    nest n1;
    n1.input_num();
    n1.disp_num();
    return 0;
}
```

Output:

```
Enter a number 5
The square of 5 is 25
The cube of 5 is 125
```

In the above program *disp_num()* function calls two other member function *square_ num()* and *cube_num()*. Both are defined in different visibility mode.

Private member function

A function declared inside the class's private section is known as **"private member function"**. A private member function is accessible through the only public member function. Even an object cannot invoke a private function using (he dot operator. Consider a class as defined below:

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

class Student {
private:
   int rNo;
   float perc;
   //private member functions
   void inputOn(void)
   {
      cout << "Input start..." << endl;
   }
   void inputOff(void)
```

```
    {
        cout << "Input end..." << endl;
    }

public:
    //public member functions
    void read(void)
    {
        //calling first member function
        inputOn();
        //read rNo and perc
        cout << "Enter roll number: ";
        cin >> rNo;
        cout << "Enter percentage: ";
        cin >> perc;
        //calling second member function
        inputOff();
    }
    void print(void)
    {
        cout << endl;
        cout << "Roll Number: " << rNo << endl;
        cout << "Percentage: " << perc << "%" << endl;
    }
};

//Main code
int main()
{
    //declaring object of class student
    Student std;
    //reading and printing details of a student
    std.read();
    std.print();

    return 0;
}
```

Output

```
Input start...
Enter roll number: 101
Enter percentage: 84.02
Input end...

Roll Number: 101
Percentage: 84.02%
```

Error: When you try to call the private member function inside the main with the object name.

Example: Changing only main() part

```
//Main code
int main()
{
        //declaring object of class student
        Student std;
        //trying to call private data member
        std.inputOn(); //error - because it's a private member
        //reading and printing details of a student
        std.read();
        std.print();
        return 0;
}
```

Output

```
main.cpp: In function 'int main()':
main.cpp:10:8: error: 'void Student::inputOn()' is private
  void inputOn(void)
       ^
main.cpp:47:14: error: within this context
 std.inputOn();
```

Inline functions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function in called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads. One solution to this problem is to use macro definitions, popularly known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to a smalI functions, C++ proposes a new feature called inline function. An inline function is a function that is expanded in line when it is invoked. That in, the compiler replaces the function call with the corresponding function code. The inline functions are defined as follows:

```
inline function-header
{
        function body
}
```

Example:

inline double cube (double a)

```
{
        return{a*a*a);
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+l.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword inline to the function definition. All inline functions must be defined before they are called.

The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines, Example:

```
        inline double cube (double a) { return (a*a*a) ;}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request it the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

C++ Static Data Members

Static data members are class members that are declared using **static** keywords. A static member has certain special characteristics which are as follows:

- 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.

- It is initialized before any object of this class is created, even before the main starts outside the class itself.
- It is visible can be controlled with the class access specifiers.
- Its lifetime is the entire program.

Syntax

```
className {
    static data_type data_member_name;
    .....
}
```

Static data members are useful for maintaining data shared among all instances of a class.

Example

Below is the C++ program to demonstrate the working of static data members:

```cpp
// C++ Program to demonstrate the use of
// static data members
#include <iostream>
using namespace std;
// class definition
class A {
public:
    // static data member here
    static int x;
    A() { cout << "A's constructor called " << endl; }
};
// we cannot initialize the static data member inside the
// class due to class rules and the fact that we cannot
// assign it a value using constructor
int A::x = 2;

// Driver code
int main()
{
    // accessing the static data member using scope
    // resultion operator
    cout << "Accessing static data member: " << A::x
         << endl;
    return 0;
}
```

Output

```
Accessing static data member: 2
```

Defining Static Data Member

As told earlier, the static members are only declared in the class declaration. If we try to access the static data member without an explicit definition, the compiler will give an error.

To access the static data member of any class we have to define it first and static data members are defined outside the class definition. The only exception to this are static const data members of integral type which can be initialized in the class declaration.

Syntax

```
datatype class_name::var_name = value...;
```

For example, in the above program, we have initialized the static data member using the following statement:

```
int A::x = 10
```

Note: *The static data members are initialized at compile time so the definition of static members should be present before the compilation of the program*

Inline Definition of Static Data Member since C++ 17

The C++ 17 introduced the inline definition of the static data members of type integral or enumeration which was not allowed in the previous standards. This simplifies the definition of the static data members.

Syntax

```
// inside class definition
...
static inline data_type var_name = value/expression;
....
```

Accessing a Static Member

We can access the static data member without creating the instance of the class. Just remember that we need to initialize it beforehand. There are 2 ways of accessing static data members:

1. Accessing static data member using Class Name and Scope Resolution Operator

The class name and the scope resolution operator can be used to access the static data member even when there are no instances/objects of the class present in the scope.

Syntax

```
Class_Name :: var_name
```

Example

```
A::x
```

2. Accessing static data member through Objects

We can also access the static data member using the objects of the class using dot operator.

Syntax

```
object_name . var_name
```

Example

```
obj.x
```

Note: *The access to the static data member can be controlled by the class access modifiers.*

Example to Verify the Properties of the Static Data Members

The below example verifies the properties of the static data member that are told above:

```
// C++ Program to demonstrate
// the working of static data member
#include <iostream>
using namespace std;
// creating a dummy class to define the static data member
// it will inform when its type of the object will be
// created
class stMember {
public:
    int val;
    // constructor to inform when the instance is created
    stMember(int v = 10): val(v) {
        cout << "Static Object Created" << endl;
    }
};
// creating a demo class with static data member of type
// stMember
class A {
public:
    // static data member
    static stMember s;
    A() { cout << "A's Constructor Called " << endl; }
};
stMember A::s = stMember(11);
// Driver code
int main()
{
    // Statement 1: accessing static member without creating
    // the object
    cout << "accessing static member without creating the "
            "object: ";
    // this verifies the independency of the static data
    // member from the instances
    cout << A::s.val << endl;
    cout << endl;
    // Statement 2: Creating a single object to verify if
    // the seperate instance will be created for each object
    cout << "Creating object now: ";
    A obj1;
    cout << endl;
    // Statement 3: Creating multiple objects to verify that
    // each object will refer the same static member
    cout << "Creating object now: ";
    A obj2;
    cout << "Printing values from each object and classname"
         << endl;
    cout << "obj1.s.val: " << obj1.s.val << endl;
    cout << "obj2.s.val: " << obj2.s.val << endl;
    cout << "A::s.val: " << A::s.val << endl;
    return 0;
}
```

Output

```
Static Object Created
accessing static member without creating the object: 11
Creating object now: A's Constructor Called
Creating object now: A's Constructor Called
Printing values from each object and classname
obj1.s.val: 11
obj2.s.val: 11
A::s.val: 11
```

Note: *In C++, we cannot declare static data members in local classes.*

Static Member Function in C++

The static keyword is used with a variable to make the memory of the variable static once a static variable is declared its memory can’t be changed.

Example:

```
class Person{
     static int index_number;
};
```

Once a static member is declared it will be treated as same for all the objects associated with the class.

Example:

- C++

```
// C++ Program to demonstrate
// Static member in a class
#include <iostream>
using namespace std;
 class Student {
public:
    // static member
    static int total;

    // Constructor called
    Student() { total += 1; }
};
 int Student::total = 0;
 int main()
{
    // Student 1 declared
    Student s1;
    cout << "Number of students:" << s1.total << endl;

    // Student 2 declared
    Student s2;
    cout << "Number of students:" << s2.total << endl;
     // Student 3 declared
    Student s3;
    cout << "Number of students:" << s3.total << endl;
    return 0;
}
```

Output

```
Number of students:1
Number of students:2
```

Number of students:3

Static Member Function in C++

Static Member Function in a class is the function that is declared as static because of which function attains certain properties as defined below:

- A static member function is independent of any object of the class.
- A static member function can be called even if no objects of the class exist.
- A static member function can also be accessed using the class name through the scope resolution operator.
- A static member function can access static data members and static member functions inside or outside of the class.
- Static member functions have a scope inside the class and cannot access the current object pointer.
- You can also use a static member function to determine how many objects of the class have been created.

The reason we need Static member function:

- Static members are frequently used to store information that is shared by all objects in a class.
- For instance, you may keep track of the quantity of newly generated objects of a specific class type using a static data member as a counter. This static data member can be increased each time an object is generated to keep track of the overall number of objects.

Example:

- C++

```
// C++ Program to show the working of
// static member functions
#include <iostream>
using namespace std;
 class Box
{
    private:
    static int length;
    static int breadth;
    static int height;
     public:
         static void print()
    {
        cout << "The value of the length is: " << length << endl;
        cout << "The value of the breadth is: " << breadth << endl;
        cout << "The value of the height is: " << height << endl;
    }
};
 // initialize the static data members
 int Box :: length = 10;
int Box :: breadth = 20;
int Box :: height = 30;
 // Driver Code

int main()
{
         Box b;
         cout << "Static member function is called through Object name: \n" << endl;
    b.print();
         cout << "\nStatic member function is called through Class name: \n" << endl;
    Box::print();
         return 0;
}
```

Output

```
Static member function is called through Object name:


The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30

Static member function is called through Class name:

The value of the length is: 10
The value of the breadth is: 20
The value of the height is: 30
```

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Example

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
int data = 3;
change(data);
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
{
data = 5;
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Example

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Friend functions

The friend function in C++ is defined outside the scope of the class. It has the authority to access all protected members and private members of the class. Friends are not member functions, but the prototypes for friend functions appear in the class function only. It covers class template, class, function template, function, and member functions, in which the entire members of the class are friends.

Syntax of Friend Function in C++

You can declare the friend function using the keyword “friend” inside the body of the class.

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

Declaring a Friend Function in C++:

All the friend class members become friend functions when a class is declared as a friend class.

You can display the friend function in C++ in the following ways:

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

Another method of declaration can be:

```
class classB;
class classA {
     // classB is a friend class of classA
      friend class classB;
……
}
class classB {
……
}
```

The members of class A can be accessed from class B. The members of class B can't be accessed from inside class A.

Features of Friend Function in C++

- The friend function is invoked like a regular function without using the object and is declared in the public or private part.
- It is not in the scope of the class that declares it as a friend.
- It has to use the object name and dot membership operator with the member name to access the member names.

Advantages of Friend Function in C++

- The friend function allows the programmer to generate more efficient codes.
- It allows the sharing of private class information by a non-member function.
- It accesses the non-public members of a class easily.
- It is widely used in cases when two or more classes contain the interrelated members relative to other parts of the program.
- It allows additional functionality that is not used by the class commonly.

this pointer

In C++, ‘this’ pointers is a pointer to the current instance of a class. It is used to refer to the object within its own member functions.

Let’s take a look at an example:

```
#include <iostream>
using namespace std;
// Class that uses this pointer
class A {
  public:
    int a;
    A(int a) {

        // Assigning a of this object to
        // function argument a
        this->a = a;
    }
    void display() {

        // Accessing a of this object
        cout << "Value: " << this->a;
    }
};
int main() {
    // Checking if this works for the object
    A o(10);
    o.display();

    return 0;
}
```

Output

Value: 10

Understanding this Pointer

To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions. Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated? The compiler supplies an implicit pointer along with the names of the functions as ‘this’. The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name). For a class X, the type of this pointer is ‘X* ‘. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *’ (see this GFact) In the early version of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value. C++ lets object destroy themselves by calling the following code:

delete this;

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer. Following are the situations where ‘this’ pointer is used:

1) When local variable’s name is same as member’s name

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

/* local variable is same as a member's name */
class Test
{
```

```
private:
  int x;
public:
  void setX (int x)
  {
     // The 'this' pointer is used to retrieve the object's x
     // hidden by the local variable 'x'
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

Output:

```
x = 20
```

For constructors, initializer list can also be used when parameter name is same as member's name.

2) To return reference to the calling object

```
// Reference to the calling object can be returned
Test& Test::func () {
// Some processing
return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

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

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  Test &setX(int a) { x = a; return *this; }
  Test &setY(int b) { y = b; return *this; }
  void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
  Test obj1(5, 5);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Output:

```
x = 10 y = 20
```

Sample Questions:

1. What are classes and objects in C++? Give suitable examples to define them.
2. How is structure different from class?
3. How are member functions defined in a class?
4. Explain access specifiers in C++ with suitable examples.
5. How is an object created? How are they passed and returned as arguments?
6. Explain array of objects in C++ with an example.
7. What is a constructor? Explain the different types of constructor.
8. How are member functions nested?
9. What are private member functions?
10. Why do we need inline functions?
11. What do you understand by static data members?
12. Illustrate call by value and call by reference in C++
13. What are destructors in C++? Is it possible to overload a destructor? Give reasons for your answer.
14. What are friend functions?
15. Explain the use of this pointer.