Unit -2

C++ Classes and Objects:

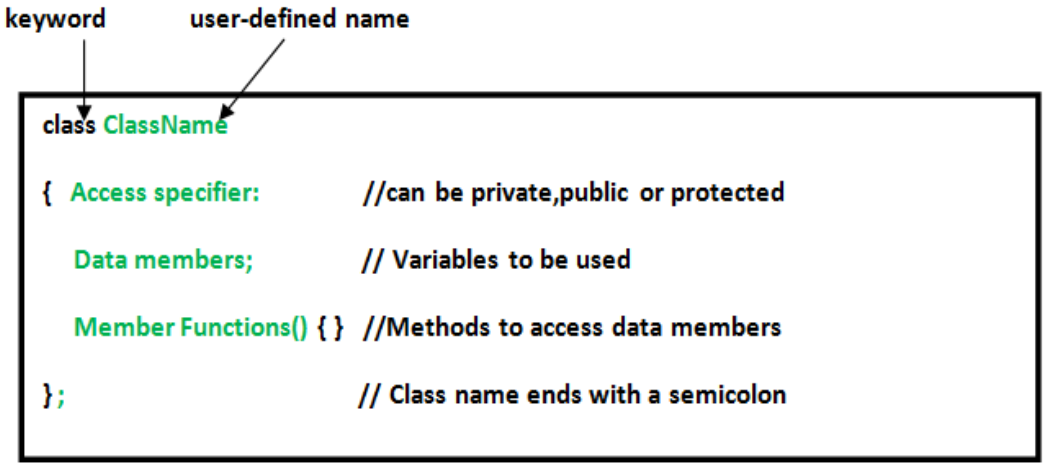
Class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, and mileage are their properties.

- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behavior of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage, etc, and member functions can be applying brakes, increasing speed, etc.

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects:

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



Declaring Objects:

When a 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

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of the class can be accessed using the dot('.') operator with the object. For example, if the name of the object is obj and you want to access the member function with the name printName() then you will have to write obj.printName().

Accessing Data Members:

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member. This access control is given by Access modifiers in C++. There are three access modifiers: public, private, and protected.

```
// C++ program to demonstrate accessing of data members
#include <bits/stdc++.h>
using namespace std;
class Geeks {
        // Access specifier
public:
        // Data Members
        string geekname;
        // Member Functions()
        void printname() { cout << "Geekname is:" << geekname; }
};
int main()
{
        // Declare an object of class geeks
        Geeks obj1;
        // accessing data member
```

```
        obj1.geekname = "Abhi";
        // accessing member function
        obj1.printname();
        return 0;
}
```

Output

Geekname is:Abhi

Access Modifiers in C++:

Access modifiers are used to implement an important aspect of Object-Oriented Programming known as Data Hiding. Consider a real-life example:
The Research and Analysis Wing (R&AW), having 10 core members, has come into possession of sensitive confidential information regarding national security. Now we can correlate these core members to data members or member functions of a class, which in turn can be correlated to the R&A Wing. These 10 members can directly access the confidential information from their wing (the class), but anyone apart from these 10 members can't access this information directly, i.e., outside functions other than those prevalent in the class itself can't access the information (that is not entitled to them) without having either assigned privileges (such as those possessed by a friend class or an inherited class, as will be seen in this article ahead) or access to one of these 10 members who is allowed direct access to the confidential information (similar to how private members of a class can be accessed in the outside world through public member functions of the class that have direct access to private members). This is what data hiding is in practice.
Access Modifiers or Access Specifiers in a class are used to assign the accessibility to the class members, i.e., they set some restrictions on the class members so that they can't be directly accessed by the outside functions.
There are 3 types of access modifiers available in C++:

1. **Public**
2. **Private**
3. **Protected**

Note: If we do not specify any access modifiers for the members inside the class, then by default the access modifier for the members will be **Private.**

1. **Public**: All the class members declared under the public specifier will be available to everyone. The data members and member functions declared as public can be accessed

by other classes and functions too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
// C++ program to demonstrate public
// access modifier
#include<iostream>
using namespace std;
// class definition
class Circle
{
    public:
        double radius;
        double compute_area()
        {
            return 3.14*radius*radius;
        }
};
// main function
int main()
{
    Circle obj;
    // accessing public datamember outside class
    obj.radius = 5.5;
    cout << "Radius is: " << obj.radius << "\n";
    cout << "Area is: " << obj.compute_area();
```

```
    return 0;
}
```

Output:

Radius is: 5.5

Area is: 94.985

In the above program, the data member radius is declared as public so it could be accessed outside the class and thus was allowed access from inside main().

2. **Private:** The class members declared as private can be accessed only by the member functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of the class.

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
    // private data member
    private:
        double radius;

    // public member function
    public:
        double compute_area()
        { // member function can access private
            // data member radius
```

```
                return 3.14*radius*radius;
        }
    };
// main function
int main()
{
    // creating object of the class
    Circle obj;

    // trying to access private data member
    // directly outside the class
    obj.radius = 1.5;

    cout << "Area is:" << obj.compute_area();
    return 0;
}
```

Output:

```
 In function 'int main()':
11:16: error: 'double Circle::radius' is private
      double radius;
             ^
31:9: error: within this context
   obj.radius = 1.5;
```

The output of the above program is a compile time error because we are not allowed to access the private data members of a class directly from outside the class. Yet an access to obj.radius

is attempted, but radius being a private data member, we obtained the above compilation error.

However, we can access the private data members of a class indirectly using the public member functions of the class.

```
// C++ program to demonstrate private

// access modifier


#include<iostream>

using namespace std;


class Circle

{

        // private data member

        private:

                double radius;

// public member function

        public:

                void compute_area(double r)

                { // member function can access private

                        // data member radius

                        radius = r;

double area = 3.14*radius*radius;

cout << "Radius is: " << radius << endl;

                        cout << "Area is: " << area;

                }
```

```
};

// main function

int main()

{

// creating object of the class

        Circle obj;

        // trying to access private data member

        // directly outside the class

        obj.compute_area(1.5);

        return 0;

}
```

Output:

Radius is: 1.5

Area is: 7.065

3. **Protected:** The protected access modifier is similar to the private access modifier in the sense that it can't be accessed outside of its class unless with the help of a friend class. The difference is that the class members declared as Protected can be accessed by any subclass (derived class) of that class as well.

Note: This access through inheritance can alter the access modifier of the elements of base class in derived class depending on the mode of Inheritance.

```
// C++ program to demonstrate

// protected access modifier

#include <bits/stdc++.h>

using namespace std;

// base class

class Parent
```

```
{
        // protected data members
        protected:
        int id_protected;
};
// sub class or derived class from public base class
class Child : public Parent
{
        public:
        void setId(int id)
        {
// Child class is able to access the inherited
// protected data members of base class
id_protected = id;


        }
        void displayId()
        {
                cout << "id_protected is: " << id_protected << endl;
        }
};
// main function
int main() {
        Child obj1;
```

```
    // member function of the derived class can
    // access the protected data members of the base class


    obj1.setId(81);
    obj1.displayId();
    return 0;
}
```

Output:

id_protected is: 81

Nested Classes in C++ :

A nested class is a class which is declared in another enclosing class. A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed. For example, program 1 compiles without any error and program 2 fails in compilation.

Program 1

```
#include<iostream>
using namespace std;
/* start of Enclosing class declaration */
class Enclosing {
private:
    int x;
    /* start of Nested class declaration */
class Nested {
    int y;
```

```
        void NestedFun(Enclosing *e) {
                cout<<e->x; // works fine: nested class can access
                                // private members of Enclosing class
        }
}; // declaration Nested class ends here
}; // declaration Enclosing class ends here
int main()
{


}
```

Program 2

```
#include<iostream>
using namespace std;
/* start of Enclosing class declaration */
class Enclosing {
int x;
/* start of Nested class declaration */
class Nested {
        int y;
}; // declaration Nested class ends here

void EnclosingFun(Nested *n) {
                cout<<n->y; // Compiler Error: y is private in Nested
}
```

```
}; // declaration Enclosing class ends here


int main()

{

}
```

Local Classes in C++ :

A class declared inside a function becomes local to that function and is called Local Class in C++.

- A local class name can only be used locally i.e., inside the function and not outside it.
- The methods of a local class must be defined inside it only.
- A local class can have static functions but, not static data members.

  For example, in the following program, Test is a local class in fun().

```
// C++ program without any compilation error
// to demonstrate a Local Class
#include <iostream>
using namespace std;

// Creating the class
void fun()
{
        // local to fun
        class Test {
                // members of Test class
        };
}

// Driver Code
int main() { return 0; }
```

Empty Class in C++ :

Empty class: It is a class that does not contain any data members (e.g. int a, float b, char c, and string d, etc.) However, an empty class may contain member functions.

Why actually an empty class in C++ takes one byte?
Simply a class without an object requires no space allocated to it. The space is allocated when the class is instantiated, so 1 byte is allocated by the compiler to an object of an empty class for its unique address identification.

If a class has multiple objects they can have different unique memory locations. Suppose, if a class does not have any size, what would be stored on the memory location? That's the reason when we create an object of an empty class in a C++ program, it needs some memory to get stored, and the minimum amount of memory that can be reserved is 1 byte. Hence, if we create multiple objects of an empty class, every object will have a unique address.

```
// C++ program without any compilation
// error to demonstrate the size of
// an Empty Class
#include <iostream>
using namespace std;

// Creating an Empty Class
class Empty_class {
};

// Driver Code
int main()
{
        cout << "Size of Empty Class is = "
                << sizeof(Empty_class);
        return 0;
}
```

Output:

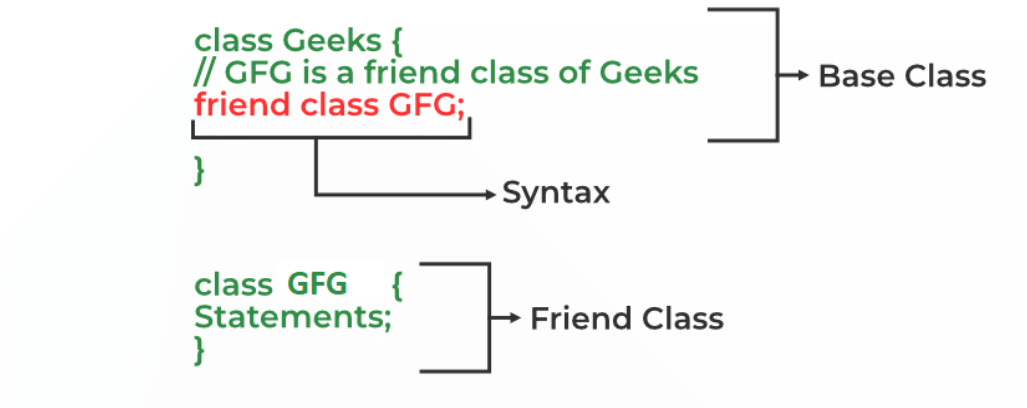
Size of Empty Class is = 1

Friend Class and Function in C++:

A friend class can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.

We can declare a friend class in C++ by using the friend keyword.
Syntax:

friend class class_name; // declared in the base class



Example:

```
// C++ Program to demonstrate the
// functioning of a friend class
#include <iostream>
using namespace std;

class GFG {
private:
```

```
    int private_variable;


protected:
    int protected_variable;


public:
    GFG()
    {
        private_variable = 10;
        protected_variable = 99;
    }
// friend class declaration
    friend class F;
};
// Here, class F is declared as a
// friend inside class GFG. Therefore,
// F is a friend of class GFG. Class F
// can access the private members of
// class GFG.
class F {
public:
    void display(GFG& t)
    {
        cout << "The value of Private Variable = "
```

```
                << t.private_variable << endl;
        cout << "The value of Protected Variable = "
                << t.protected_variable;
    }
};
// Driver code
int main()
{
    GFG g;
    F fri;
    fri.display(g);
    return 0;
}
```

Output

The value of Private Variable = 10

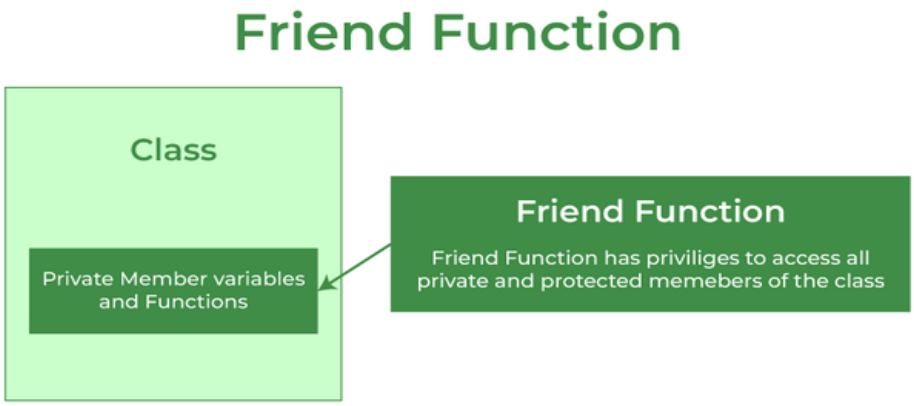
The value of Protected Variable = 99

Friend Function :

Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are the non-member functions that can access and manipulate the private and protected members of the class for they are declared as friends.

A friend function can be:

1. A global function
2. A member function of another class



1. Global Function as Friend Function

We can declare any global function as a friend function. The following example demonstrates how to declare a global function as a friend function in C++:

Example:

```
// C++ program to create a global function as a friend

// function of some class

#include <iostream>

using namespace std;

class base {

private:

    int private_variable;

protected:

    int protected_variable;

public:

    base()

    {

        private_variable = 10;
```

```
        protected_variable = 99;
    }

    // friend function declaration
    friend void friendFunction(base& obj);
};


// friend function definition
void friendFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
        << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}
// driver code
int main()
{
    base object1;
    friendFunction(object1);
return 0;
}
```

Output

```
Private Variable: 10
```

Protected Variable: 99

In the above example, we have used a global function as a friend function. In the next example, we will use a member function of another class as a friend function.

2. **Member Function of Another Class as Friend Function**

We can also declare a member function of another class as a friend function in C++. The following example demonstrates how to use a member function of another class as a friend function in C++:

Example:

```
// C++ program to create a member function of another class

// as a friend function

#include <iostream>

using namespace std;

class base; // forward definition needed

// another class in which function is declared

class anotherClass {

public:

        void memberFunction(base& obj);

};

// base class for which friend is declared

class base {

private:

        int private_variable;

protected:

        int protected_variable;

public:
```

```
        base()
        {
                private_variable = 10;
                protected_variable = 99;
        }
// friend function declaration
        friend void anotherClass::memberFunction(base&);
};
// friend function definition
void anotherClass::memberFunction(base& obj)
{
        cout << "Private Variable: " << obj.private_variable
                << endl;
        cout << "Protected Variable: " << obj.protected_variable;
}
// driver code
int main()
{
        base object1;
        anotherClass object2;
        object2.memberFunction(object1);
return 0;
}
```

Output

Private Variable: 10

Protected Variable: 99

Features of Friend Functions

- A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to access the private and protected data of a class.
- A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword “friend” inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword “friend” is placed only in the function declaration of the friend function and not in the function definition or call.
- A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.
- A friend function can be declared in any section of the class i.e. public or private or protected.

Advantages of Friend Functions

- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions

- Friend functions have access to private members of a class from outside the class which violates the law of data hiding.
- Friend functions cannot do any run-time polymorphism in their members.

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. To know more about static keywords refer to the article static Keyword in C++.

Static Member in C++

Static members of a class are not associated with the objects of the class. Just like a static variable once declared is allocated with memory that can't be changed every object points to the same memory. To know more about the topic refer to a static Member in C++.

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++ 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.

Example:

```
// 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

‘this’ pointer in C++ :

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

- Each object gets its own copy of the data member.
- 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

Difference Between Structure and Class in C++

In C++, a structure works the same way as a class, except for just two small differences. The most important of them is hiding implementation details. A structure will by default not hide its implementation details from whoever uses it in code, while a class by default hides all its implementation details and will therefore by default prevent the programmer from accessing them. The following table summarizes all of the fundamental differences.

Class	Structure
1. Members of a class are private by default.	1. Members of a structure are public by default.
2. An instance of a class is called an 'object'.	2. An instance of structure is called the 'structure variable'.
3. Member classes/structures of a class are private by default but not all programming languages have this default behavior eg Java etc.	3. Member classes/structures of a structure are public by default
4. It is declared using the class keyword.	4. It is declared using the struct keyword.
5. It is normally used for data abstraction and further inheritance.	5. It is normally used for the grouping of data

6. NULL values are possible in Class.	6. NULL values are not possible.

7. Syntax:
 	
class class_name{
 	
data_member;
 	
member_function;
 	
};
	7. Syntax:
	
	struct structure_name{
	
	type structure_member1;
	
	type structure_member2;
	
	};

new and delete Operators in C++ For Dynamic Memory :

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on Heap, and non-static and local variables get memory allocated on Stack (Refer to Memory Layout C Programs for details).

How is it different from memory allocated to normal variables?

For normal variables like “int a”, “char str[10]”, etc, memory is automatically allocated and deallocated. For dynamically allocated memory like “int *p = new int[10]”, it is the programmer’s responsibility to deallocate memory when no longer needed. If the programmer doesn’t deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).

How is memory allocated/deallocated in C++?

C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. C++ supports these functions and also has two operators new and delete, that perform the task of allocating and freeing the memory in a better and easier way.

new operator:

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator

pointer-variable = new data-type;

Here, the pointer variable is the pointer of type data-type. Data type could be any built-in data type including array or any user-defined data type including structure and class.

Example:

```
// Pointer initialized with NULL

// Then request memory for the variable

int *p = NULL;

p = new int;
```

OR

```
// Combine declaration of pointer

// and their assignment

int *p = new int;
```

Initialize memory: We can also initialize the memory for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

pointer-variable = new data-type(value);

Example:

```
int* p = new int(25);

float* q = new float(75.25);


// Custom data type

struct cust
```

```
{
        int p;
        cust(int q) : p(q) {}
        cust() = default;
        //cust& operator=(const cust& that) = default;
};
int main()
{
        // Works fine, doesn't require constructor
        cust* var1 = new cust;
//OR
// Works fine, doesn't require constructor
        var1 = new cust();
// Notice error if you comment this line
        cust* var = new cust(25);
        return 0;
}
```

Allocate a block of memory: a new operator is also used to allocate a block(an array) of memory of type data type.

pointer-variable = new data-type[size];

where size(a variable) specifies the number of elements in an array.

Example:

int *p = new int[10]

Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned top(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.

delete operator:

Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, the pointer variable is the pointer that points to the data object created by new.

Examples:

delete p;

delete q;

To free the dynamically allocated array pointed by pointer variable, use the following form of delete:

// Release block of memory

// pointed by pointer-variable

delete[] pointer-variable;

Example:

// It will free the entire array

  // pointed by p.

  delete[] p;

// C++ program to illustrate dynamic allocation

// and deallocation of memory using new and delete

#include <iostream>

```
using namespace std;
int main ()
{
        // Pointer initialization to null
        int* p = NULL;
    // Request memory for the variable
        // using new operator
        p = new(nothrow) int;
        if (!p)
                cout << "allocation of memory failed\n";
        else
        {
// Store value at allocated address
                *p = 29;
                cout << "Value of p: " << *p << endl;
        }
// Request block of memory
        // using new operator
        float *r = new float(75.25);
cout << "Value of r: " << *r << endl;
// Request block of memory of size n
        int n = 5;
        int *q = new(nothrow) int[n];
if (!q)
```

```
            cout << "allocation of memory failed\n";
        else
        {
            for (int i = 0; i < n; i++)
                q[i] = i+1;
cout << "Value store in block of memory: ";
            for (int i = 0; i < n; i++)
                cout << q[i] << " ";
        }
// freed the allocated memory
        delete p;
        delete r;
// freed the block of allocated memory
        delete[] q;

        return 0;
}
```

Output

Value of p: 29

Value of r: 75.25

Value store in block of memory: 1 2 3 4 5

Pointer To Object In C++:

A pointer is a variable that stores the memory address of another variable (or object) as its value. A pointer aims to point to a data type which may be int, character, double, etc.

Pointers to objects aim to make a pointer that can access the object, not the variables. Pointer to object in C++ refers to accessing an object.

There are two approaches by which you can access an object. One is directly and the other is by using a pointer to an object in C++.

A pointer to an object in C++ is used to store the address of an object. For creating a pointer to an object in C++, we use the following syntax:

Syntax:

classname*pointertoobject;

For storing the address of an object into a pointer in c++, we use the following syntax:

Syntax:

pointertoobject=&objectname;

The above syntax can be used to store the address in the pointer to the object. After storing the address in the pointer to the object, the member function can be called using the pointer to the object with the help of an arrow operator.

Pointers to Class Members in C++:

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

```
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
}
```

Pointer to Data Members of Class:

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 ;

Wild Pointers:

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave unexpectedly.

Example of Wild Pointers:

```
// C program that demonstrated wild pointers
int main()
{
        /* wild pointer */
        int* p;
        /* Some unknown memory location is being corrupted.
        This should never be done. */
```

```
    *p = 12;
}
```

How can we avoid wild pointers?

If a pointer points to a known variable then it's not a wild pointer.

Example:

```
int main()
{
    int* p; /* wild pointer */
    int a = 10;
    /* p is not a wild pointer now*/
    p = &a;
    /* This is fine. Value of a is changed */
    *p = 12;
}
```

If we want a pointer to a value (or set of values) without having a variable for the value, we should explicitly allocate memory and put the value in the allocated memory.

Dangling Pointer in C:

A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

There are three different ways where a pointer acts as a dangling pointer:

1. De-allocation of Memory

When a memory pointed by a pointer is deallocated the pointer becomes a dangling pointer.

Example:

The below program demonstrates the deallocation of a memory pointed by ptr.

```
// C program to demonstrate Deallocating a memory pointed by
// ptr causes dangling pointer
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* ptr = (int*)malloc(sizeof(int));
       // After below free call, ptr becomes a dangling pointer
        free(ptr);
        printf("Memory freed\n");

        // removing Dangling Pointer
        ptr = NULL;

        return 0;
}
```

Output:

Memory freed

Smart Pointers in C++:

Pointers are used for accessing the resources which are external to the program – like heap memory. So, for accessing the heap memory (if anything is created inside heap memory), pointers are used. When accessing any external resource we just use a copy of the resource. If we make any changes to it, we just change it in the copied version. But, if we use a pointer to the resource, we'll be able to change the original resource.

Example:

```
// C++ program to demonstrate working of a Pointers
#include <iostream>
using namespace std;

class Rectangle {
private:
        int length;
        int breadth;
};
void fun()
{
       // By taking a pointer p and
        // dynamically creating object
        // of class rectangle
        Rectangle* p = new Rectangle();
}
int main()
{
        // Infinite Loop
        while (1) {
                fun();
        }
}
```

Output:

Memory limit exceeded