UNIT-III
C++ Inheritance

Syllabus:

Inheritance: Introduction to inheritance, Defining Derived Classes, Single Inheritance, Multiple Inheritance, Multi level Inheritance, Hierarchical Inheritance, Hybrid Inheritance.

Pointers: Introduction to Memory management, new operator and delete operator, Pointers to objects, Pointers to Derived Classes.

Introduction:

Reusability is one of the important characteristics of Object Oriented Programming (OOP). Instead of trying to write programs repeatedly, using existing code is a good practice for the programmer to reduce development time and avoid mistakes. In C++, reusability is possible by using Inheritance.

Definition: The technique of deriving a new class from an old one is called inheritance.

- The old class is referred to as base class and the new class is referred to as derived class or subclass.
- Base class – It is also known as a superclass or a parent class. It is responsible for sharing its properties with its derived class(es).
- Derived class – It is also known as a *subclass or a child class*. It is responsible for inheriting some of all of the properties of the base class(es).
- Inheritance concept allows programmers to define a class in terms of another class, which makes creating and maintaining application easier.
- When writing a new class, instead of writing new data member and member functions all over again, programmers can make a bonding of the new class

with the old one that the new class should inherit the members of the existing class.

- A class can get derived from one or more classes, which means it can inherit data and functions from multiple base classes.

<u>the syntax how inheritance is performed in C++:</u>

```
class derived_class_name :: visibility-mode base_class_name
{
            AccessSpecifier:
                            //List of Variables
          AccessSpecifier:
                      //List of Member_Functions
};
```

Where,

derived_class_name: It is the name of the derived class.

base_class_name: It is the name of the base class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the private members of the base class are not accessible by the objects of the derived class,but that can be accessed only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived

class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

Importance of Inheritance in C++

There are numerous reasons as to why the concept of inheritance was introduced in C++. Here are some of the reasons why:

- It drastically reduces code redundancy as it allows the derived class to inherit all the properties of the base class, leaving no room for duplicate data to achieve the same task.
- Inheritance in C++ offers the feature of code reusability. We can use the same fragment of code multiple times. To sum it up, inheritance helps us save our development time, maintain data in a simplified manner and gives us the provision to make our code extensible.
- It increases the code reliability by providing a definite body to the program.
- The transitive nature of inheritance makes things much easier for us.

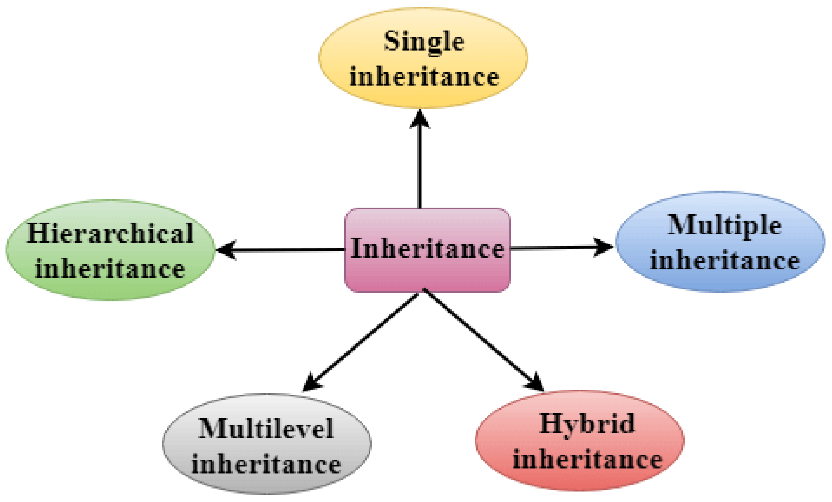
Note:

- The constructor and destructor of the parent class cannot be inherited.
- The assignment operator cannot be inherited
- Friend function and friend classes of the parent class cannot be inherited.

Types Of Inheritance

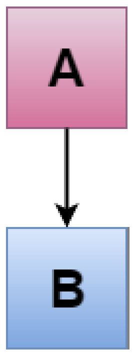
C++ supports five types of inheritance:

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance



1. Single Inheritance : The Process of deriving a single derived class from a single base class is class “Single (or) Simple Inheritance”

   (i.e)



Syntax-

```
class Base
{
// BODY OF THE BASE CLASS
};
class Derived : Acess_specifier Base
{
// BODY OF THE DERIVED CLASS
};
```

Example:

C++ program to demonstrate Simple Inheritance.

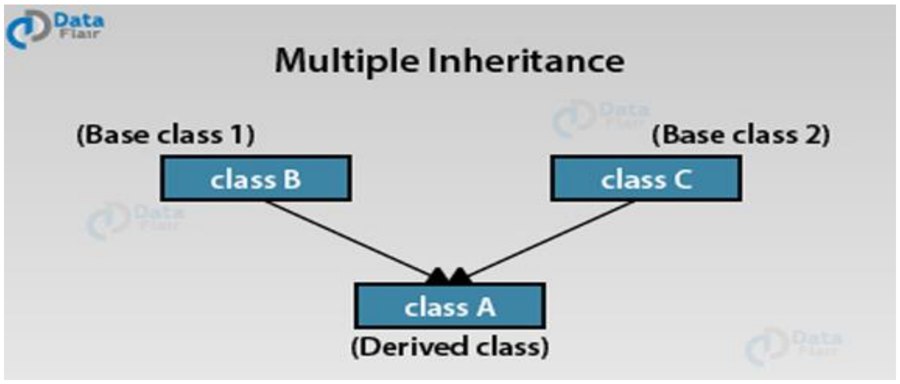
Code:

```
#include<iostream>
 Class A
 {
         Public:
             int a;
  };
  Class B:public A
  {
          Public:
                 int b;
          public:
                 void get_ab();
                  void display_ab();
 };
  Void B::get_ab()
  {
            Cout<<"Enter the values of a,b:";
             Cin>>a>>b;
   }
   Void B::display_ab()
     {
            Cout<<"a="<<a<<"\n"<<"b="<<b;
```

```
                }

                  Void main()
`                  {

                          B    b1;

                          b1.get_ab();

                          b1.display_ab();

                   }
```

2.Multiple Inheritance

- **Def:** This type of inheritance happens *when the child class inherits its properties from more than one base class*.
- In other others, the derived class inherits properties from multiple base classes.



Syntax

```
class A                // Base class of B
{
          // BODY OF THE CLASS A
};
class B             // Derived class of A and Base class
{
        // BODY OF THE CLASS B
};
class C : acess_specifier A, access_specifier A           // Derived class of A and B
{
```

```
                    // BODY OF CLASS C
};
```

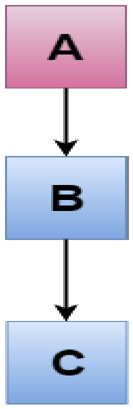
Example: Write a C++ program to describe Multiple Inheritance.

3.<u>Multilevel Inheritance</u>

<u>Def:</u> The Process of deriving a subclass from another subclass is called"Multi-level Inheritance".

This type of inheritance is the *best way to represent the transitive nature of inheritance*.

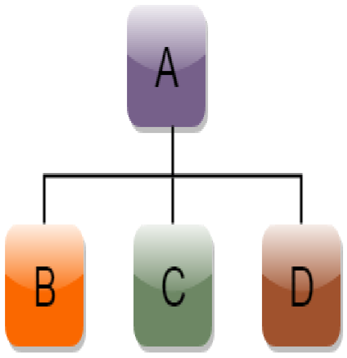
In multilevel inheritance, a derived class inherits all its properties from a class that itself inherits from another class.



Example: Write a C++ program to demonstrate Multi-level Inheritance

4. Hierarchical Inheritance

Def: Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax :

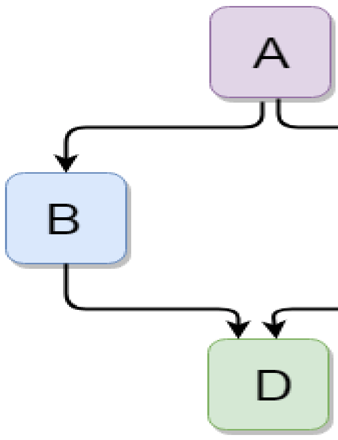
```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Example: Write a C++ program to demonstrate Hierarchical Inheritance

5.Hybrid Inheritance

Def: Hybrid inheritance is a combination of more than one type of inheritance,(i.e)

Combination of Multiple and Hierarchcal(or)Multilevel and Multiple Inheritance.



Example: Write a C++ program to demonstrate Hybrid Inheritance.

Pointers

Introduction to Memory Management:

Arrays can be used to store multiple homogenous data but there are serious drawbacks of using arrays.(i.e) there are 2 drawbacks

1.Wastage of Memory: If we allocate the memory for arrays as

int A[30];

Let us assume that, while storing the data into an array if we got the situation to store only 25 elements then remaining 5 elements space will be wasted.(i.e) Wastage of Memory.

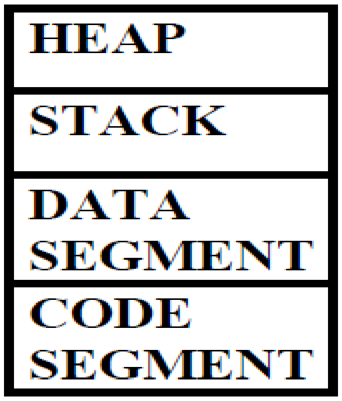
2.Insufficient Memory: If we allocate the memory for arrays as

int A[30];

Let us assume that, while storing the data into an array if we got the situation to store 60 elements then there is no such memory(i.e) Insufficient Memory.

To avoid Wastage of memory and Insufficient memory, we can dynamically allocate required memory during runtime using new and delete operators in C++.

Below is a basic memory architecture used by any C++ program:



- **Code Segment**: Compiled program with executive instructions are kept in code segment. It is read only. In order to avoid over writing of stack and heap, code segment is kept below stack and heap.
- **Data Segment**: Global variables and static variables are kept in data segment. It is not read only.
- **Stack**: A stack is usually pre-allocated memory. The stack is a LIFO data structure. Each new variable is pushed onto the stack. Once variable goes out of scope, memory is freed. Once a stack variable is freed, that region of memory becomes available for other variables. The stack grows and shrinks as functions push and pop local variables. It stores local data, return addresses, arguments passed to functions and current status of memory.
- **Heap**: Memory is allocated during program execution. Memory is allocated using new operator and deallocating memory using delete operator.

new operator

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, 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: To allocate memory of any data type, the syntax is:

```
pointer-variable = new data-type;
```

Here, 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 types 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 using new operator. (i.e)

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

Example:

```
int *p = new int(25);
float *q = new float(75.25);
```

Allocate block of memory(or)Memory allocation for Array:

new operator is also used to allocate a block(an array) of memory for array elements.

Syntax:

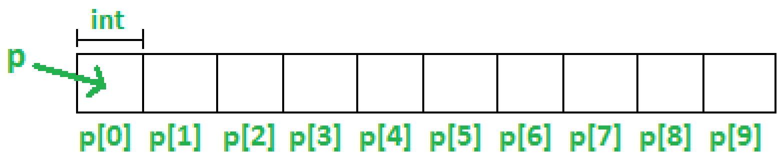
```
pointer-variable = new data-type[size];
```

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 pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new:

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by programmer or program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless "nothrow" is used with the new operator, in which case it returns a NULL pointer (scroll to section "Exception handling of new operator" in this article). Therefore, it may be good idea to check for the pointer variable produced by new before using it program.

```
int *p = new(nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

delete operator

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

Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, 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 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.h>
```

```
int main ()

{

            // Pointer initialization to null

            Int * p = NULL;

              // Request memory for the variable  using new operator

            p = new 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 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;

}
```

Pointer to Objects

C++ allows you to have pointers to objects. The pointers pointing to objects are referred to as Object Pointers. Just like other pointers, the object pointers are declared by placing in front of a object pointer's name. It takes the following general form :

Syntax:

```
class-name    * object-pointer ;
```

where class-name is the name of an already defined class and object-pointer is the pointer to an object of this class type.

For example, to declare optr as an object pointer of Sample class type, we shall write as

```
Sample   *optr ;
```

where Sample is already defined class.

Note: When accessing members of a class using an object pointer, the arrow operator (->) is used instead of dot operator.

//Write a C++ Program to describe pointers to objects

```
#include <iostream>

using namespace std;

class Box
```

```
{
   public:
      // Constructor definition
      Box(double l = 2.0, double b = 2.0, double h = 2.0)
     {
                   cout <<"Constructor called." << endl;
                   length = l;
                   breadth = b;
                   height = h;
      }
      double Volume()
     {
                   return length * breadth * height;
      }

             private:
                double length;     // Length of a box
                double breadth;    // Breadth of a box
                double height;     // Height of a box
};

int main(void)
{
```

```
                Box Box1(3.3, 1.2, 1.5);    // Declare box1
                Box Box2(8.5, 6.0, 2.0);    // Declare box2
                Box *ptrBox;                // Declare pointer to a class.

                // Save the address of first object
                ptrBox = &Box1;
                // Now try to access a member using member access operator
                cout << "Volume of Box1: " << ptrBox->Volume() << endl;
                // Save the address of second object
                  ptrBox = &Box2;
                // Now try to access a member using member access operator
                cout << "Volume of Box2: " << ptrBox->Volume() << endl;
                  return 0;
        }
```

Pointers to Derived Classes

In C++, we can declare a pointer points to the base class as well as derive class.

/*Write C++ program to demonstrate Pointers to Derived Classes*/

```
#include <iostream.h>
class Base
{
        public:
        int x;
        void display ()
        {
              cout<<"X="<<x<<endl;
        }
};
class Derive: public Base
{
        public:
        int y;
        void display ();
        {
              cout<<"X="<<x<<endl;
              cout<<"Y="<<y<<endl;
        }
};
int main ()
{
        Base B1;
        Base *ptr;

        //Pointer pointing Base class Object

        ptr = &B1;

        ptr->x = 10;

        ptr->display();

        //Pointer pointing Derived  class Object

        Derive D1;

        Derive *ptr1;

        ptr1 = &D1;

        ptr1->x = 10;

        ptr1->y = 20;

        ptr1->display ();

}
```