6. What is multiple inheritance? Explain with example.

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.

The constructors of inherited classes are called in the same order in which they are inherited. For example, in the following program, B's constructor is called before A's constructor.

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

class A
{
public:
A() { cout << "A's constructor called" << endl; }
};

class B
{
public:
B() { cout << "B's constructor called" << endl; }
};

class C: public B, public A // Note the order
{
public:
C() { cout << "C's constructor called" << endl; }
};

int main()
{
        C c;
        return 0;
}
```

Output:

```
B's constructor called
A's constructor called
C's constructor called
```

7. How is polymorphism achieved at
a) Compile time
b) Run time

Compile time polymorphism: This type of polymorphism is achieved by **function overloading** or **operator overloading**.

- **Overloading**: Overloading is where more than one methods share the same name with different parameters or signature and different return type.
- The call is resolved by the **compiler**.
- It is also known as **Static binding, Early binding** and **overloading** as well.
- It provides **fast execution** because known early at compile time.
- Compile time polymorphism is **less flexible** as all things execute at compile time.

→ **Runtime polymorphism:** This type of polymorphism is achieved by **Function Overriding.**

- **Function overriding:** Overriding is having same method with same parameters or signature, but associated within a class & its subclass.
- It is also known as **Dynamic binding, Late binding** and **overriding** as well.
- It is achieved by **virtual functions** and **pointers**.
- It provides **slow execution** as compare to early binding because it is known at runtime.
- Run time polymorphism is **more flexible** as all things execute at run time.

Compile time polymorphism -- method overloading

Run time time polymorphism -- method overriding

WHAT IS POLYMORPHISM?

- Polymorphism means having many forms or we can say we can define the polymorphism as the ability to display a message in many form.
- It happens when there is a hierarchy of classes and they are related by inheritance. It also means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- Let us have a look at the real life example to understand polymorphism, like a person can have different characteristic at the same time.
- Like a woman can behave as a mother, a wife and a employee at the same time so, a same person posses different behavior in different situations. This is known as polymorphism.

TYPES OF POLYMORPHISM

In C++, polymorphism can be divided into two types:

- Compile Time Polymorphism
- Runtime Polymorphism

COMPILE TIME POLYMORPHISM:

The compile time polymorphism can be achieved by function overloading or by operator overloading. The overloaded functions are invoked by matching the type and number of arguments and this is done at the compile time so, compiler selects the appropriate function at the compile time. The operator overloading is also known as static binding.

FUNCTION OVERLOADING

When there are multiple functions with same name but have different parameters then these functions are said to be overloaded. They can be overloaded by change in number of arguments or change in type of arguments.

LET US HAVE A LOOK AT THE EXAMPLE:

C++

```
#include <iostream>

using namespace std;

class Multiplication {

public:

    int mul(int a,int b) {

        return a*b;

    }

    int mul(int a,int b, int c) {

        return a*b*c;

    }

     double mul(double a,double b) {

        return a*b;

    }

};

```

```
int main(void) {

    Multiplication obj;

    cout<<obj.mul(2, 5)<<endl;

    cout<<obj.mul(7, 3, 1)<<endl;

    cout<<obj.mul(4.4, 10.8);

    return 0;

}
```

OPERATOR OVERLOADING

- In C++, operator overloading is a compile time polymorphism in which the operator is overloaded to provide the special meaning to the user defined data type.
- It is used to overload the operator in C++ and perform the operation on the user defined data type.

LET US HAVE A LOOK AT THE EXAMPLE:

C++

```
#include<iostream>

using namespace std;

class Complex {

private:

    int r, i;

public:

    Complex(int a = 0, int b =0)  {r = a;   i = b;}

    // This is automatically called when '+' is used with

    // between two Complex objects

    Complex operator + (Complex const &obj) {

          Complex res;

          res.r = r + obj.r;

          res.i = i + obj.i;

          return res;

    }

```

```
    void print() { cout << r << " + i" << i << endl; }

};

int main()

{

    Complex c1(20, 10), c2(5, 7);

    Complex c3 = c1 + c2; // An example call to "operator+"

    c3.print();

}
```

RUNTIME POLYMORPHISM:

The runtime polymorphism is achieved when the object method is invoked at the runtime instead of compile time. It is achieved by method overriding which is also known as dynamic binding.

FUNCTION OVERRIDING

The function overriding occurs when a derived class has a definition for one of the member functions of the base class. The base function is said to be overridden.

LET US HAVE A LOOK AT THE EXAMPLE:

C++

```
#include <iostream>

using namespace std;

class Fruits {

    public:

void taste(){

cout<<"Juicy...";

    }

};

class Apple: public Fruits

{

 public:

 void taste()

    {

       cout<<"Juicy Apple...";

    }

```

```
};

int main(void) {

   Apple a = Apple();

   a.taste();

   return 0;

}
```

DIFFERENCE BETWEEN COMPILE TIME POLYMORPHISM AND RUNTIME POLYMORPHISM

Compile Time Polymorphism	Runtime Polymorphism
The function is invoked at the compile time.	The function is invoked at the runtime.
It is known as overloading, early binding and static binding.	It is also known as overriding, dynamic binding and late binding.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It is less flexible.	It is more flexible.
It provides fast execution.	It provides slow execution.

8. What is a file mode? Explain various file mode options available.

C++ File and File Modes

C++ language allows us to perform important Disk *input/output* operations such as -

- *Creating* a new file on the disk.
- *Reading* the file stored on the disk.
- *Writing* data to the file stored on the disk.
- *Appending* new data to the end of the file stored on the disk.
- *Modifying* the content of the file stored on the disk.

To perform any file operations, C++ provides us a few *file stream classes*, such as -

- **ifstream**, to perform the file input operations.
- **ofstream**, to perform the file output operations.
- **fstream**, to perform any file input and output operations.

How to open a file?

By using the object of any *file stream class*, we could call the ***open()*** function, using which we could provide location of the file stored on the disk and in return(*if the file is found*), the **open()** function opens the file in a specific *mode* to let you perform a specific file operation.

Let us take a look at a general syntax of ***open()*** function to open a file.

```
file-stream-object("filename", mode);
```

- ***file-stream-object***, is the an of a file stream class used to perform a specific file operation.
- ***filename***, is the name of a file on which we are going to perform file operations.
- ***mode***, is *single or multiple file modes* in which we are going to open a file.

File Modes

In C++, for every file operation, exists a specific *file mode*. These file modes allow us to *create, read, write, append or modify* a file. The file modes are defined in the class **ios**. Let's see all these different **modes** in which we could open a file on disk.

File Modes	Description
ios::in	Searches for the file and opens it in the **read** mode only(*if the file is found*).
ios::out	Searches for the file and opens it in the **write** mode. If the file is found, its content is overwritten. If the file is not found, a new file is created. *Allows you to **write** to the file.*
ios::app	Searches for the file and opens it in the **append** mode i.e. this mode allows you to **append** new data to the end of a file. If the file is not found, a new file is created.
"ios::binary"	Searches for the file and opens the file(if the file is found) in a binary mode to perform binary input/output file operations.
ios::ate	Searches for the file, opens it and positions the pointer at the end of the file. This mode when used with **ios::binary, ios::in** and **ios::out** modes, *allows you to **modify** the content of a file.*
"ios::trunc"	Searches for the file and opens it to truncate or deletes all of its content(*if the file is found*.

"ios::nocreate"	Searches for the file and if the file is not found, a new file will not be created.

9. What is generic programming? How is it implemented in c++?

Generic programming is a way to write functions and data types while making minimal assumptions about the type of data being used.

Getting Started

Let’s break it down in a more concrete way. Say you want to iterate over an array of numbers and perform a function with each element like print:

Now imagine if your program needs to do this over and over for many different arrays at various points throughout your application. Writing five lines each time would quickly become tedious. It could also degrade the readability of your code. To fix this we could create a helper function like this:

This a bit more convenient. We can now print all the values in an array and get the item count using one line. Even so, we’re still tying it to specific data type. We can still do better.

Keeping DRY

Imagine we had arrays all over our project and the array held different types of data. If we wanted to to iterate for an array that

held non-integer data types we would have to copy and paste this code and change the input type. This will quickly become the downfall of any effort to keep your code **DRY** (**D**on't **R**epeat **Y**ourself).

Templates

C++ implements generic programming concepts through templates. Templates give the compiler a framework to generate code for the types it gets implemented with. With a class, C++ will look at your template as well as the type specified when you created the object, and generate that typed class for you.

Think of it as a cookie cutter in the shape of a star. You can have various kinds of cookie dough, like shortbread, or chocolate-shortbread or chocolate chip cookie dough. The cookie cutter doesn't care about the ingredients, as long as it is dough.

Linked Lists

To prove the power of generics, let's build a doubly linked list. If you're new to linked lists, they are data types where each element hold a pointer to the element directly before and after it. A linked list will terminate when the pointer becomes NULL.

If you were using several linked lists with with different data types this could quickly degenerate into messy spaghetti code.

Let's Be Generic!

To simplify, let's change a few aspects of the code to make it compatible with generic data types:

We specify **template <class T>** at the top giving with generic type **T**. This **T** could eventually be implemented as **char** type like:

```
Linked_List<char> *generic_linked_list = new
Linked_List<char>();
```

Or an **int** type:

```
Linked_List<int> *generic_int_linked_list = new
Linked_List<int>();
```

10. Write a class template to represent a generic vector. Include member functions to perform the following tasks:

a) To create the vector.

b) To modify the value of a given element.

c) To multiply by a scalar value.