Technical Round CE/IT

Prof. Sweety S. Patel

sweety.patel@aitindia.in

C++ Interview Questions

1) What is C++?

C++ is an object-oriented programming language created by Bjarne Stroustrup. It was released in 1985.
C++ is a superset of C with the major addition of classes in C language.
Initially, Stroustrup called the new language "C with classes". However, after sometime the name was changed to C++. The idea of C++ comes from the C increment operator ++.

2) What are the advantages of C++?

C++ doesn't only maintains all aspects from C language, it also simplifies memory management and adds several features like:

- C++ is a highly portable language means that the software developed using C++ language can run on any platform.
- C++ is an object-oriented programming language which includes the concepts such as classes, objects, inheritance, polymorphism, abstraction.
- C++ has the concept of inheritance. Through inheritance, one can eliminate the redundant code and can reuse the existing classes.
- Data hiding helps the programmer to build secure programs so that the program cannot be attacked by the invaders.
- Message passing is a technique used for communication between the objects.
- C++ contains a rich function library.

3) What is the difference between C and C++?

Following are the differences between C and C++:

C	**C++**
C language was developed by Dennis Ritchie.	C++ language was developed by Bjarne Stroustrup.
C is a structured programming language.	C++ supports both structural and object-oriented programming language.
C is a subset of C++.	C++ is a superset of C.

In C language, data and functions are the free entities.	In the C++ language, both data and functions are encapsulated together in the form of a object.
C does not support the data hiding. Therefore, the data can be used by the outside world.	C++ supports data hiding. Therefore, the data cannot be accessed by the outside world.
C supports neither function nor operator overloading.	C++ supports both function and operator overloading.
In C, the function cannot be implemented inside the structures.	In the C++, the function can be implemented inside the structures.
Reference variables are not supported in C language.	C++ supports the reference variables.
C language does not support the virtual and friend functions.	C++ supports both virtual and friend functions.
In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.

4) What is the difference between reference and pointer?

Following are the differences between reference and pointer:

Reference	Pointer
Reference behaves like an alias for an existing variable, i.e., it is a temporary variable.	The pointer is a variable which stores the address of a variable.
Reference variable does not require any indirection operator to access the value. A reference variable can be used directly to access the value.	Pointer variable requires an indirection operator to access the value of a variable.
Once the reference variable is assigned, then it cannot be reassigned with different address values.	The pointer variable is an independent variable means that it can be reassigned to point to different objects.
A null value cannot be assigned to the reference variable.	A null value can be assigned to the reference variable.
It is necessary to initialize the variable at the time of declaration.	It is not necessary to initialize the variable at the time of declaration.

5) What is a class?

The class is a user-defined data type. The class is declared with the keyword class. The class contains the data members, and member functions whose access is defined by the three modifiers are private, public and protected. The class defines the type definition of the category of things. It defines a data type, but it does not define the data it just specifies the structure of data.
You can create N number of objects from a class.

---

6) What are the various OOPs concepts in C++?

The various OOPS concepts in C++ are:

- **Class:**

The class is a user-defined data type which defines its properties and its functions. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space. Therefore, we can say that the class is the only logical representation of the data.
The syntax of declaring the class:

```
class student
{
//data members;
//Member functions
}
```

- **Object:**

An object is a run-time entity. An object is the instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions. The class does not occupy any memory space. When an object is created using a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created without a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.

```
class Student
{
//data members;
//Member functions
}
```

The syntax for declaring the object:

```
Student s = new Student();
```

- **Inheritance:**

Inheritance provides reusability. Reusability means that one can use the functionalities of the existing class. It eliminates the redundancy of code. Inheritance is a technique of deriving a new class from the old class. The old class is known as the base class, and the new class is known as derived class.
Syntax

```
class derived_class :: visibility-mode base_class;
```

Note: The visibility-mode can be public, private, protected.

- **Encapsulation:**

Encapsulation is a technique of wrapping the data members and member functions in a single unit. It binds the data within a class, and no outside method can access the data. If the data member is private, then the member function can only access the data.
int age;
setAge(int a){}
getAge(){}
data members are privates and to access variable setter and getter methods are used

- **Abstraction:**

Abstraction is a technique of showing only essential details without representing the implementation details. If the members are defined with a public keyword, then the members are accessible outside also. If the members are defined with a private keyword, then the members are not accessible by the outside methods.

- **Data binding:**

Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.

- **Polymorphism:**

Polymorphism means multiple forms. Polymorphism means having more than one function with the same name but with different functionalities. Polymorphism is of two types:

1. Static polymorphism is also known as early binding.
2. Dynamic polymorphism is also known as late binding.

---

7) What are the different types of polymorphism in C++?

Polymorphism: Polymorphism means multiple forms. It means having more than one function with the same function name but with different functionalities.
Polymorphism is of two types:

- **Runtime polymorphism**

Runtime polymorphism is also known as dynamic polymorphism. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class. Hence, the

child class overrides the method of the parent class. In case of function overriding, parent and child class both contains the same function with the different definition. The call to the function is determined at runtime is known as runtime polymorphism.
Let's understand this through an example:

```
#include <iostream>
using namespace std;
class Base
{
    public:
    virtual void show()
    {
        cout<<"java";
     }
};
class Derived:public Base
{
    public:
    void show()
    {
        cout<<"java tutorial";
    }
};

int main()
{
    Base* b;
    Derived d;
    b=&d;
    b->show();
                return 0;
}
```

Output:

```
java tutorial
```

- **Compile time polymorphism**

Compile-time polymorphism is also known as static polymorphism. The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Method overloading is an example of compile-time polymorphism.
Method overloading: Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

- The return type of the overloaded function.
- The type of the parameters passed to the function.

- The number of parameters passed to the function.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class Multiply
{
   public:
   int mul(int a,int b)
   {
       return(a*b);
   }
   int mul(int a,int b,int c)
   {
       return(a*b*c);
  }
 };
int main()
{
    Multiply multi;
    int res1,res2;
    res1=multi.mul(2,3);
    res2=multi.mul(2,3,4);
    cout<<"\n";
    cout<<res1;
    cout<<"\n";
    cout<<res2;
    return 0;
}
```

Output:

```
6
24
```

- In the above example, mul() is an overloaded function with the different number of parameters.

---

8) Define namespace in C++.

- The namespace is a logical division of the code which is designed to stop the naming conflict.

- The namespace defines the scope where the identifiers such as variables, class, functions are declared.
- The main purpose of using namespace in C++ is to remove the ambiguity. Ambiquity occurs when the different task occurs with the same name.
- For example: if there are two functions exist with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.
- C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.
- **Syntax of namespace:**

```
namespace namespace_name
{
 //body of namespace;
}
```

Syntax of accessing the namespace variable:

```
namespace_name::member_name;
```

Let's understand this through an example:

```
#include <iostream>
using namespace std;
namespace addition
{
    int a=5;
    int b=5;
    int add()
    {
        return(a+b);
    }
}

int main() {
int result;
result=addition::add();
cout<<result;
return 0;
}
```

Output:

```
10
```

9) Define token in C++.

A token in C++ can be a keyword, identifier, literal, constant and symbol.

---

10) Who was the creator of C++?

Bjarne Stroustrup.

---

11) Which operations are permitted on pointers?

Following are the operations that can be performed on pointers:

- **Incrementing or decrementing a pointer**: Incrementing a pointer means that we can increment the pointer by the size of a data type to which it points.

There are two types of increment pointers:
1. Pre-increment pointer: The pre-increment operator increments the operand by 1, and the value of the expression becomes the resulting value of the incremented. Suppose ptr is a pointer then pre-increment pointer is represented as ++ptr.
Let's understand this through an example:

```
#include <iostream>
using namespace std;
int main()
{
int a[5]={1,2,3,4,5};
int *ptr;
ptr=&a[0];
cout<<"Value of *ptr is : "<<*ptr<<"\n";
cout<<"Value of *++ptr : "<<*++ptr;
return 0;
}
```

Output:

```
Value of *ptr is : 1

Value of *++ptr : 2
```

2. Post-increment pointer: The post-increment operator increments the operand by 1, but the value of the expression will be the value of the operand prior to the incremented value of the operand. Suppose ptr is a pointer then post-increment pointer is represented as ptr++.
Let's understand this through an example:

```
#include <iostream>
```

```cpp
using namespace std;
int main()
{
int a[5]={1,2,3,4,5};
int *ptr;
ptr=&a[0];
cout<<"Value of *ptr is : "<<*ptr<<"\n";
cout<<"Value of *ptr++ : "<<*ptr++;
return 0;
}
```

Output:

```
Value of *ptr is : 1
Value of *ptr++ : 1
```

- **Subtracting a pointer from another pointer:** When two pointers pointing to the members of an array are subtracted, then the number of elements present between the two members are returned.

12) Define 'std'.

Std is the default namespace standard used in C++.

13) Which programming language's unsatisfactory performance led to the discovery of C++?

C++was discovered in order to cope with the disadvantages of C.

14) How delete [] is different from delete?

Delete is used to release a unit of memory, delete[] is used to release an array.

15) What is the full form of STL in C++?

STL stands for Standard Template Library.

16) What is an object?

The Object is the instance of a class. A class provides a blueprint for objects. So you can create an object from a class. The objects of a class are declared with the same sort of declaration that we declare variables of basic types.

17) What are the C++ access specifiers?

The access specifiers are used to define how to functions and variables can be accessed outside the class.
There are three types of access specifiers:

- **Private**: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.
- **Public**: Functions and variables declared under public can be accessed from anywhere.
- **Protected**: Functions and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

---

18) What is Object Oriented Programming (OOP)?

OOP is a methodology or paradigm that provides many concepts. The basic concepts of Object Oriented Programming are given below:
Classes and Objects: Classes are used to specify the structure of the data. They define the data type. You can create any number of objects from a class. Objects are the instances of classes.
Encapsulation: Encapsulation is a mechanism which binds the data and associated operations together and thus hides the data from the outside world. Encapsulation is also known as data hiding. In C++, It is achieved using the access specifiers, i.e., public, private and protected.
Abstraction: Abstraction is used to hide the internal implementations and show only the necessary details to the outer world. Data abstraction is implemented using interfaces and abstract classes in C++.
Some people confused about Encapsulation and abstraction, but they both are different.
Inheritance: Inheritance is used to inherit the property of one class into another class. It facilitates you to define one class in term of another class.

---

19) What is the difference between an array and a list?

- An Array is a collection of homogeneous elements while a list is a collection of heterogeneous elements.
- Array memory allocation is static and continuous while List memory allocation is dynamic and random.
- In Array, users don't need to keep in track of next memory allocation while In the list, the user has to keep in track of next location where memory is allocated.

---

20) What is the difference between new() and malloc()?

- new is a preprocessor while malloc() is a function.
- There is no need to allocate the memory while using "new" but in malloc() you have to use sizeof().

- "new" initializes the new memory to 0 while malloc() gives random value in the newly allotted memory location.
- The new operator allocates the memory and calls the constructor for the object initialization and malloc() function allocates the memory but does not call the constructor for the object initialization.
- The new operator is faster than the malloc() function as operator is faster than the function.

---

22) Define friend function.

Friend function acts as a friend of the class. It can access the private and protected members of the class. The friend function is not a member of the class, but it must be listed in the class definition. The non-member function cannot access the private data of the class. Sometimes, it is necessary for the non-member function to access the data. The friend function is a non-member function and has the ability to access the private data of the class.
To make an outside function friendly to the class, we need to declare the function as a friend of the class as shown below:

```
class sample
{
  // data members;
 public:
friend void abc(void);
};
```

Following are the characteristics of a friend function:

- The friend function is not in the scope of the class in which it has been declared.
- Since it is not in the scope of the class, so it cannot be called by using the object of the class. Therefore, friend function can be invoked like a normal function.
- A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
- Friend function uses objects as arguments.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class Addition
{
 int a=5;
 int b=6;
 public:
```

```
 friend int add(Addition a1)
 {
    return(a1.a+a1.b);
 }
};
int main()
{
int result;
Addition a1;
 result=add(a1);
 cout<<result;
return 0;
}
```

Output:

```
11
```

23) What is a virtual function?

- A virtual function is used to replace the implementation provided by the base class. The replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer.
- A virtual function is a member function which is present in the base class and redefined by the derived class.
- When we use the same function name in both base and derived class, the function in base class is declared with a keyword virtual.
- When the function is made virtual, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class pointer. Thus, by making the base class pointer to point different objects, we can execute different versions of the virtual functions.

Rules of a virtual function:

- The virtual functions should be a member of some class.
- The virtual function cannot be a static member.
- Virtual functions are called by using the object pointer.
- It can be a friend of another class.
- C++ does not contain virtual constructors but can have a virtual destructor.

24) When should we use multiple inheritance?

You can answer this question in three manners:

1. Never
2. Rarely

3. If you find that the problem domain cannot be accurately modeled any other way.

---

25) What is a destructor?

A Destructor is used to delete any extra resources allocated by the object. A destructor function is called automatically once the object goes out of the scope.
Rules of destructor:

- Destructors have the same name as class name and it is preceded by tilde. ~
- It does not contain any argument and no return type.

---

26) What is an overflow error?

It is a type of arithmetical error. It happens when the result of an arithmetical operation been greater than the actual space provided by the system.

---

27) What is overloading?

- When a single object behaves in many ways is known as overloading. A single object has the same name, but it provides different versions of the same function.
- C++ facilitates you to specify more than one definition for a function name or an operator in the same scope. It is called function overloading and operator overloading respectively.
- **Overloading is of two types:**

1. Operator overloading: Operator overloading is a compile-time polymorphism in which a standard operator is overloaded to provide a user-defined definition to it. For example, '+' operator is overloaded to perform the addition operation on data types such as int, float, etc.
Operator overloading can be implemented in the following functions:

- Member function
- Non-member function
- Friend function

Syntax of Operator overloading:

```
Return_type classname :: Operator Operator_symbol(argument_list)
{
    // body_statements;
}
```

2. Function overloading: Function overloading is also a type of compile-time polymorphism which can define a family of functions with the same name. The function would perform different operations based on the argument list in the function call. The function to be invoked depends on the number of arguments and the type of the arguments in the argument list.

---

28) What is function overriding?

If you inherit a class into a derived class and provide a definition for one of the base class's function again inside the derived class, then this function is called overridden function, and this mechanism is known as function overriding.

---

29) What is virtual inheritance?

Virtual inheritance facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.

---

30) What is a constructor?

A Constructor is a special method that initializes an object. Its name must be same as class name.

---

31) What is the purpose of the "delete" operator?

The "delete" operator is used to release the dynamic memory created by "new" operator.

---

32) Explain this pointer?

This pointer holds the address of the current object.

---

33) What does the Scope Resolution operator do?

A scope resolution operator(::) is used to define the member function outside the class.

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

int my_var = 0;
int main(void) {
   int my_var = 0;
   ::my_var = 1;   // set global my_var to 1
   my_var = 2;     // set local my_var to 2
   cout << ::my_var << ", " << my_var;
   return 0;

}
```

1 2

```cpp
#include <iostream>
using namespace std;
class X {
   public:
   static int count;
};
int X::count = 10;  // define static data member

int main () {
   int X = 0;    // hides class type X
   cout << X::count << endl;    // use static member of class X

}
```

```
10
```

---

34) What is the difference between delete and delete[]?

Delete [] is used to release the array of allocated memory which was allocated using new[] whereas delete is used to release one chunk of memory which was allocated using new.

---

35) What is a pure virtual function?

The pure virtual function is a virtual function which does not contain any definition. The normal function is preceded with a keyword virtual. The pure virtual function ends with 0.
Syntax of a pure virtual function:

```cpp
virtual void abc()=0;   //pure virtual function.
```

Let's understand this through an example:

```cpp
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual void show()=0;
};

class Derived:public Base
{
    public:
```

```
    void show()
    {
        cout<<"java";
    }
};
int main()
{
    Base* b;
    Derived d;
    b=&d;
    b->show();
    return 0;
}
```

Output:

```
java
```

---

35) What is Data Encapsulation

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

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

```
Total 60
```

36) What is the Data Abstraction?

Data abstraction provides two important advantages −

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Any C++ program where you implement a class with public and private members is an example of data abstraction

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

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

```
Total 60
```

37) What is the difference between struct and class?

Structures	class
A structure is a user-defined data type which contains variables of dissimilar data types.	The class is a user-defined data type which contains member variables and member functions.
The variables of a structure are stored in the stack memory.	The variables of a class are stored in the heap memory.
We cannot initialize the variables directly.	We can initialize the member variables directly.
If access specifier is not specified, then by default the access specifier of the variable is "public".	If access specifier is not specified, then by default the access specifier of a variable is "private".
The instance of a structure is a "structure variable".	

Declaration of a structure:
struct structure_name
{
  // body of structure;
} ;
	Declaration of class:
	class class_name
	{
	  // body of class;
	}

A structure is declared by using a struct keyword.	The class is declared by using a class keyword.

The structure does not support the inheritance.	The class supports the concept of inheritance.
The type of a structure is a value type.	The type of a class is a reference type.

37) What is a class template?

A class template is used to create a family of classes and functions. For example, we can create a template of an array class which will enable us to create an array of various types such as int, float, char, etc. Similarly, we can create a template for a function, suppose we have a function add(), then we can create multiple versions of add().
The syntax of a class template:

```
template<class T>
class classname
{
  // body of class;
};
```

Syntax of a object of a template class:

```
classname<type> objectname(arglist);
```

Function Template

```
#include <iostream>
#include <string>

using namespace std;

template <typename T>
inline T const& Max (T const& a, T const& b) {
   return a < b ? b:a;
}

int main () {
   int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;

   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;

   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;

   return 0;
}
```

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Class Template

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
   private:
      vector<T> elems;     // elements

   public:
      void push(T const&);  // push element
      void pop();               // pop element
      T top() const;            // return top element

      bool empty() const {      // return true if empty.
         return elems.empty();
      }
};

template <class T>
void Stack<T>::push (T const& elem) {
    // append copy of passed element
    elems.push_back(elem);
}

template <class T>
void Stack<T>::pop () {
    if (elems.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    }

    // remove last element
    elems.pop_back();
}

template <class T>
T Stack<T>::top () const {
    if (elems.empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }

    // return copy of last element
    return elems.back();
}

int main() {
    try {
        Stack<int>         intStack;  // stack of ints
        Stack<string> stringStack;    // stack of strings

        // manipulate int stack
        intStack.push(7);
        cout << intStack.top() <<endl;

        // manipulate string stack
        stringStack.push("hello");
        cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    } catch (exception const& ex) {
        cerr << "Exception: " << ex.what() <<endl;
        return -1;
    }
```

```
}
7
hello
Exception: Stack<>::pop(): empty stack
```

---

38) What is the difference between function overloading and operator overloading?

Function overloading: Function overloading is defined as we can have more than one version of the same function. The versions of a function will have different signature means that they have a different set of parameters.
Operator overloading: Operator overloading is defined as the standard operator can be redefined so that it has a different meaning when applied to the instances of a class.

---

39) What is a virtual destructor?

A virtual destructor in C++ is used in the base class so that the derived class object can also be destroyed. A virtual destructor is declared by using the ~ tilde operator and then virtual keyword before the constructor.
Note: Constructor cannot be virtual, but destructor can be virtual.
Let's understand this through an example

- Example without using virtual destructor

In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.

But virtual destructor is possible

```
#include<iostream>

using namespace std;

class b {

   public:

      b() {

         cout<<"Constructing base \n";
      }
      virtual ~b() {
         cout<<"Destructing base \n";
      }
};
class d: public b {
   public:
      d() {
         cout<<"Constructing derived \n";
      }
      ~d() {
         cout<<"Destructing derived \n";
      }
};
int main(void) {
   d *derived = new d();
   b *bptr = derived;
   delete bptr;
   return 0;
}
```

o/p

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

```cpp
#include <iostream>
using namespace std;
class Base
{
    public:
    Base()
    {
        cout<<"Base constructor is called"<<"\n";
    }
    ~Base()
    {
        cout<<"Base class object is destroyed"<<"\n";
    }
};
class Derived:public Base
{
    public:
    Derived()
    {
        cout<<"Derived class constructor is called"<<"\n";
    }
    ~Derived()
    {
        cout<<"Derived class object is destroyed"<<"\n";
    }
};
int main()
{
  Base* b= new Derived;
  delete b;
  return 0;

}
```

Output:

```
Base constructor is called
Derived class constructor is called
Base class object is destroyed
```

In the above example, delete b will only call the base class destructor due to which derived class destructor remains undestroyed. This leads to the memory leak.

- Example with a virtual destructor

```
#include <iostream>
using namespace std;
class Base
{
    public:
    Base()
    {
        cout<<"Base constructor is called"<<"\n";
    }
    virtual ~Base()
    {
        cout<<"Base class object is destroyed"<<"\n";
    }
};
class Derived:public Base
{
    public:
    Derived()
    {
        cout<<"Derived class constructor is called"<<"\n";
    }
    ~Derived()
    {
        cout<<"Derived class object is destroyed"<<"\n";
    }
};
int main()
{
  Base* b= new Derived;
  delete b;
  return 0;

}
```

Output:

```
Base constructor is called
Derived class constructor is called
Derived class object is destroyed
Base class object is destroyed
```

Inline virtual function

Virtual functions in C++ use to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. Virtual functions are resolved late, at the runtime.

The main use of the virtual function is to achieve Runtime Polymorphism. The inline functions are used to increase the efficiency of the code. The code of inline function gets substituted at the point of an inline function call at compile time, whenever the inline function is called.

Whenever a virtual function is called using base class reference or pointer it cannot be inlined, but whenever called using the object without reference or pointer of that class, can be inlined because the compiler knows the exact class of the object at compile time.

```
#include<iostream>
using namespace std;
class B {
   public:
      virtual void s() {
         cout<<" In Base \n";
      }
inline int add(int a,int b)
{
return a+b;
}
};

class D: public B {
   public:
      void s() {
         cout<<"In Derived \n";
      }
};

int main(void) {
   B b;
   D d; // An object of class D
   B *bptr = &d;// A pointer of type B* pointing to d
   b.s();//Can be inlined as s() is called through object of
class

int ans= b.add(12,32);
   bptr->s();// prints"D::s() called"
   //cannot be inlined, as virtualfunction is called through
pointer.
   return 0;
}
```

o/p

```
In Base
In Derived
```

C++ Break and Continue

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

int main() {
  for (int i = 0; i < 10; i++) {
    if (i == 4) {
      break;
    }
    cout << i << "\n";
  }
  return 0;
}
```

```
0
1
2
3
```

C++ Continue

The `continue` statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

This example skips the value of 4:

```
for (int i = 0; i < 10; i++) {
  if (i == 4) {
    continue;
  }
  cout << i << "\n";
}
```

```
0
1
2
3
5
6
7
8
9
```

Creating References

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

```
int main() {
  string food = "Pizza";
  string &meal = food;

  cout << food << "\n";
  cout << meal << "\n";
  return 0;
}
```

Function Overloading

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

int plusFuncInt(int x, int y) {
  return x + y;
}

double plusFuncInt(double x, double y) {
  return x + y;
}

int main() {
  int myNum1 = plusFuncInt(8, 5);
  double myNum2 = plusFuncInt(4.3, 6.26);
  cout << "Int: " << myNum1 << "\n";
  cout << "Double: " << myNum2;
  return 0;
}
```

Constructors

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

```
class Car {        // The class
  public:          // Access specifier
    string brand;  // Attribute
    string model;  // Attribute
    int year;      // Attribute
Car();

    Car(string x, string y, int z); // Constructor declaration
};
Car::Car()
{
  brand = “a”;
  model = “y”;
  year = 2020;

}
// Constructor definition outside the class
Car::Car(string x, string y, int z) {
  brand = x;
  model = y;
  year = z;
}

int main() {
  // Create Car objects and call the constructor with different values

  Car carObj();//default constructor

  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
```

```
 cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year <<
"\n";
 cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year <<
"\n";
 return 0;
}
```

Access Specifiers

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. You will learn more about Inheritance later.

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

class MyClass {
  public:    // Public access specifier
    int x;   // Public attribute
  private:   // Private access specifier
    int y;   // Private attribute
};

int main() {
  MyClass myObj;
  myObj.x = 25;  // Allowed (x is public)
  myObj.y = 50;  // Not allowed (y is private)
  return 0;
}
```

```
In function 'int main()':
Line 8: error: 'int MyClass::y' is private
Line 14: error: within this context
```

Encapsulation

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as `private` (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public get and set methods.

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

class Employee {
  private:
    int salary;

  public:
    void setSalary(int s) {
      salary = s;
    }
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

```
50000
```

Why Encapsulation?

- It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts
- Increased security of data

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a base class called `Animal` that has a method called `animalSound()`. Derived classes of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n" ;
  }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n" ;
  }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n" ;
  }
};
```

Now we can create `Pig` and `Dog` objects and override the `animalSound()` method:

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n" ;
  }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n" ;
   }
};
```

```
// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
     cout << "The dog says: bow wow \n" ;
  }
};

int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

```
The animal makes a sound
The pig says: wee wee
The dog says: bow wow
```