C++ interview questions

29 July 2023 12:46

What is C++?

C++ is a general-purpose programming language that extends the capabilities of the C programming language. It is an object oriented programming language where we use object, class, inheritance, polymorphism, abstraction and encapsulation.

What is reference in C++?

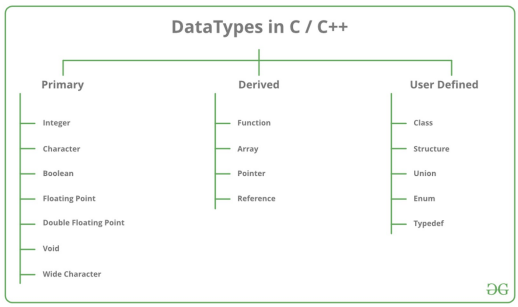
Reference or reference variable is variable which access address or memory of other variable.
Ex- int dc = 5;
Int &d= dc;

Difference between C and C++.

C is a structural or procedural programming language that was used for system applications and low-level programming applications. Whereas C++ is an object-oriented programming language having some additional features like Encapsulation, Data Hiding, Data Abstraction, Inheritance, Polymorphism, etc. This helps to make a complex project more secure and flexible.

Approach C follows a top-down approach || C++ follows the bottom-up approach.
Reference Variable C does not support reference variable||C++ support reference variable.

Data type in c++.



1. Primitive Data Types: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char, float, bool, etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

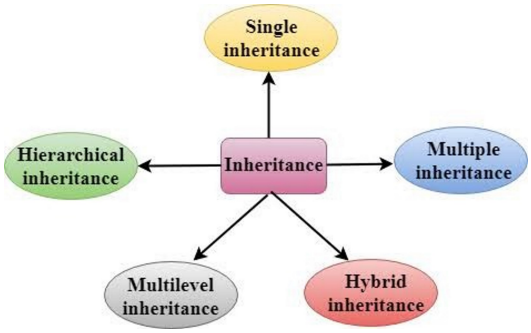
2. Derived Data Types: Derived data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:

- Function
- Array
- Pointer
- Reference

3. Abstract or User-Defined Data Types: Abstract or User-Defined data types are defined by the user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

- Class
- Structure
- Union
- Enumeration
- Typedef defined Datatype

What is inheritance?



The capability of a class to derive properties and characteristics from another class is called Inheritance.

Types of Inheritance in C++

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.Single inheritance in C++.

2.Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one subclass is inherited from more than one base class.Multiple inheritance in C++

3.Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

5.Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance.

A special case of hybrid inheritance: Multipath inheritance: A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

```
// C++ program demonstrating ambiguity in Multipath
// Inheritance

#include <iostream>
using namespace std;

class ClassA {
public:
    int a;
};
```

```
class ClassB : public ClassA {
public:
    int b;
};

class ClassC : public ClassA {
public:
    int c;
};

class ClassD : public ClassB, public ClassC {
public:
    int d;
};

int main()
{
    ClassD obj;

    // obj.a = 10;                  // Statement 1, Error
    // obj.a = 100;                 // Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a from ClassB  : " << obj.ClassB::a;
    cout << "\n a from ClassC  : " << obj.ClassC::a;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}
```

Output
a from ClassB : 10
a from ClassC : 100
b : 20
c : 30
d : 40

There are 2 Ways to Avoid this Ambiguity:

1) Avoiding ambiguity using the scope resolution operator:

Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

CPP

```
obj.ClassB::a = 10;       // Statement 3
obj.ClassC::a = 100;      // Statement 4
```

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

CPP

```
#include<iostream>
```

```
class ClassA
{
  public:
    int a;
};

class ClassB : virtual public ClassA
{
  public:
    int b;
};

class ClassC : virtual public ClassA
{
  public:
    int c;
};

class ClassD : public ClassB, public ClassC
{
  public:
    int d;
};

int main()
{
    ClassD obj;

    obj.a = 10;       // Statement 3
    obj.a = 100;      // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}

Output:
 a : 100
b : 20
c : 30
d : 40
```

What is polymorphism?

The word “polymorphism” means having many forms. In simple words, we can define polymorphism as ability of function which can display massage more than one form . A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.

1. Compile-Time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

A. Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.

```
#include <iostream>
using namespace std;
class Cal {
   public:
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;                                  //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Output:

30
55

A. Operator Overloading

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Example1: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix#include<iostream>usingnamespacestd;
classCount{private:
   intvalue;
public:
// Constructor to initialize count to 5Count() : value(5) {}
// Overload ++ when used as prefixvoidoperator++ () {
      ++value;
   }
```

```
voiddisplay(){
      cout<< "Count: "<< value << endl;
   }
};
intmain(){
   Count count1;
// Call the "void operator ++ ()" function++count1;
count1.display();
   return0;
}
```

Run Code

Output

```
Count: 6
```

Operator Overloading in Binary Operators

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, `+` is a binary operator that works on the operands `num` and `9`.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the `obj1` object and `obj2` is passed as an argument to the function.

Example 4: C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +// This program adds two complex
numbers#include<iostream>usingnamespacestd;
classComplex{private:
   floatreal;
   floatimag;
public:
   // Constructor to initialize real and imag to 0Complex() : real(0), imag(0) {}
voidinput(){
      cout<< "Enter real and imaginary parts respectively: ";
      cin>> real;
      cin>> imag;
   }
// Overload the + operatorComplex operator+ (constComplex& obj) {
      Complex temp;
      temp.real = real + obj.real;
      temp.imag = imag + obj.imag;
      returntemp;
   }
```

```
voidoutput(){
      if(imag < 0)
         cout<< "Output Complex number: "<< real << imag << "i";
      elsecout<< "Output Complex number: "<< real << "+"<< imag << "i";
   }
};
intmain(){
   Complex complex1, complex2, result;
cout<< "Enter first complex number:\n";
   complex1.input();
cout<< "Enter second complex number:\n";
   complex2.input();
// complex1 calls the operator function// complex2 is passed as an argument to the functionresult =
complex1 + complex2;
   result.output();
return0;
}
```

Run Code

Output

```
Enter first complex number:
Enter real and imaginary parts respectively: 9 5
Enter second complex number:
Enter real and imaginary parts respectively: 7 6
Output Complex number: 16+11i
```

Runtime Polymorphism

This type of polymorphism is achieved by Function Overriding. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

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

Function Overriding in C++

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {
```

```
    cout<<"Eating bread...";
  }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

Eating bread...

B.Virtual Function **A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.**

Some Key Points About Virtual Functions:

Virtual functions are Dynamic in nature. They are defined by inserting the keyword "virtual" inside a base class and are always declared with a base class and overridden in a child classA virtual function is called during RuntimeBelow is the C++ program to demonstrate virtual function:

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
```

```
 a->display();   //Late Binding occurs
}
```
Output:
Derived Class is invoked

What is Class?

A class can be understood as a template or a blueprint, which contains some values, known as data members, and some set of rules, known as behaviors or methods. The data and methods that are defined in the class are automatically taken when an object is created. The class is a template or blueprint for objects. One can make as many objects as they want to be based on a class.
For example, the template for the car is created first after that multiple units or objects of cars can be created based on the template. And each of the objects must have the data and methods mentioned in the blueprint.
A class is a set of traits and features that define an object. It is a blueprint which must be implemented by each of its objects.
For example: If we define a blueprint of human beings then it can have attributes like **Name, Age, Gender, Occupation** and functionalities like **Walk(), Eat(),** and **Sleep()**. Any human being must have these properties(data) and functionalities(methods) associated with them.

What is an Object?

object is an instance of a class which has those properties and behaviors associated with them

A class contains the properties and behaviour for a set of objects, many objects can be formed using a class.

- Take the example of a dog as an object.
- There are different varieties of dogs that means different varieties of objects according to the dog class. For example, one dog has black color, another dog may have grey color, or white, etc.
- However, we can notice that most of the behaviours are the same like barking, wagging their tail, eating, sleeping, etc.
- Properties and behaviour of these objects can be the same but their values would be different.

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;
	};

Types of Access Specifiers & How do they work in C++?

1. Public Access Specifier

This keyword is used to declare the functions and variables public, and any part of the entire program can access it. The members and member methods declared public can be accessed by other classes and functions.

The public members of a class can be accessed from anywhere in the program using the **(.)** with the object of that class.

Example - In the above example,employeeId and employeeName are public access specifiers.

2. Private Access Specifiers

The **private keyword** is used to create private variables or private functions. The private members can only be accessed from within the class. Only the member functions or the friend functions are allowed to access the private data of a class or the methods of a class.

Note -

- Protected and Private data members or class methods can be accessed using a function only if that function is declared as the friend function.
- We can use the keyword friend to ensure the compiler understands and make the data accessible to that function.

Example - In the above example,employeeSalary is a private access specifier.

3. Protected Access Specifiers

The protected keyword is used to create protected variables or protected functions. The protected members can be accessed within and from the derived/child class.

> **Note** - A class created or derived from another existing class(base class) is known as a derived class. The base class is also known as a superclass. It is created and derived through the process of inheritance.

For more details about derived class or inheritance, you can follow this blog.

Inheritance in cpp

Protected access specifier is similar to the private modifier. It cannot be accessed outside of its class except the derived class or subclass of that class. However, it can be accessed by the friend function of that class, similar to a private specifier.

Example - In the above example,setEmployeeSalary method/function is a protected specifier.

Encaptulation

In C++, encapsulation involves combining similar data and functions into a single unit called a class. By encapsulating these functions and data, we protect that data from change. This concept is also known as data or information hiding.

```
class MyClass {
private:
   int privateData;

public:
   void setData(int value) {
      privateData = value;
   }

   int getData() {
      return privateData;
   }
};
```

Constructor in C++?

A constructor is a particular member function having the same name as the class name. It calls automatically whenever the object of the class is created.

```
#include <iostream.h>
#include <conio.h>
using namespace std;
class hello {     // The class
  public:         // Access specifier
   hello () {    // Constructor
      cout << "Hello World! Program in C++ by using Constructor";
    }
   void display() {
     cout <<"Hello World!" <<endl;
   }
};
int main() {
  hello myObj;   /
  return 0;
}
```

1. Default Constructor in C++

- The default constructor is the constructor which doesn't take any argument. It has no parameters.
- In this case, as soon as the object is created the constructor is called which initializes its data members.
- A default constructor is so important for the initialization of object members, that

even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly

```
#include <iostream>
using namespace std;
class construct {
   public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};
int main()
{
   construct c;
   int sum = c.a + c.b;

   cout << "a : " << c.a << endl;
   cout << "b : " << c.b << endl;
   cout << "sum : " << sum << endl;

   return 0;
}
```

Output

```
a : 10
b : 20
sum : 30
```

2. Parameterized Constructor in C++

- Arguments can be passed to the parameterised constructors.
- These arguments help initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

We can also have more than one constructor in a class and that concept is called constructor overloading.

Uses of Parameterized constructor:

- It is used to initialize the various data elements of different objects with different values when they are created.

- It is used to overload constructors.

Sample Code

Run

```
#include <iostream>
using namespace std;
class PrepInsta {
   private:
   int a, b;

   public:

   PrepInsta(int a1, int b1)
   {
      a = a1;
      b = b1;
   }

   int getA()
   {
      return a;
   }

   int getB()
   {
      return b;
   }
};
int main()
{
  PrepInsta obj1(10, 15);

  cout << "a = " << obj1.getA() << ", b = " << obj1.getB();

  return 0;
}
```

Output

```
a = 10, b = 15
```

3. Copy Constructor in C++

- A copy constructor is a member function which initializes an object using another object of the same class.
- Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case.
- An object can be initialized with another object of same type. This is same as copying the contents of a class to another class.

- For a better understanding of Copy Constructor in C++ (click here).

What is a Copy Constructor?

Definition These are the special type of Constructors that takes an object as an argument and is used to copy values of data members of one object into another object.

```
class_name(class-name &){
....
}
```

- Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type.
- It is usually of the form X (X&), where X is the class name.
- The compiler provides a default Copy Constructor to all the classes.

Code

Run

```
#include <iostream>
using namespace std;
class PrepInsta
{
private:
   int x, y;
public:
   PrepInsta()
   { // empty default constuctor
   }

   PrepInsta(int x1, int y1)
   {
      x = x1;
      y = y1;
cout << "Parameterized constructor called here" << endl;
}
// User defined Copy constructor
   PrepInsta(const PrepInsta &p2)
   {
      x = p2.x;
      y = p2.y;
      cout << "Copy constructor called here" << endl;
   }
int getX()
   {
      return x;
}
   int getY()
   {
      return y;
}
};
int main()
{
   // Trying to call parameterized constructor here
   PrepInsta p1(10, 15);

   // Trying to call copy constructor here
```

```
    PrepInsta p2 = p1;

    // Trying to call Copy constructor here (Another way of doing so)
    PrepInsta p3(p1);
    PrepInsta p4;

    // Here there is no copy constructor called only assignment operator happens
    p4 = p1;

    cout << "\nFor p4 no copy constructor called only assignment operation happens\n"
<< endl;
// displaying values for both constructors
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
cout << "\np3.x = " << p3.getX() << ", p3.y = " << p3.getY();
    cout << "\np4.x = " << p4.getX() << ", p4.y = " << p4.getY();

    return 0;
}
```

Output

```
Parameterized constructor called here
Copy constructor called here
Copy constructor called here
For p4 no copy constructor called only assignment operation happens
p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15
p3.x = 10, p3.y = 15
p4.x = 10, p4.y = 15
```

Very ImportantIn the above example there are two ways to call copy constructor –

```
PrepInsta p2 = p1;
PrepInsta p3(p1);
```

However, PrepInsta p4; followed by p4 = p1; doesn't call copy constructor, its just simply and assignment operator

Inline function

If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

The compiler may not perform inlining in such circumstances as:

If a function contains a loop. (*for, while and do-while*)
If a function contains static variables.
If a function is recursive.
If a function return type is other than void, and the return statement doesn't exist in a function body.
If a function contains a switch or goto statement.

```
#include <iostream>
using namespace std;
inline int cube(int s) { return s * s * s; }
int main()
```

```
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
#include<stdio.h>
void change(int num) {
   printf("Before adding value inside function num=%d \n",num);
   num=num+100;
   printf("After adding value inside function num=%d \n", num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at

the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>
void change(int *num) {
   printf("Before adding value inside function num=%d \n",*num);
   (*num) += 100;
   printf("After adding value inside function num=%d \n", *num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(&x);//passing reference in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

C++ Static Data Members

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

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is initialized before any object of this class is created, even before the main starts.
- It is visible only within the class, but its lifetime is the entire program.

Syntax:

```
static data_type data_member_name;
```

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

- C++

```
// C++ Program to demonstrate
// the working of static data member
#include <iostream>
using namespace std;

class A {
public:
    A()
    {
      cout << "A's Constructor Called " <<
                endl;
    }
```

```
};

class B {
    static A a;

public:
    B()
    {
      cout << "B's Constructor Called " <<
                endl;
    }
};

// Driver code
int main()
{
    B b;
    return 0;
}
```

Output

```
B's Constructor Called
```

Static Member Functions

The static member functions are special functions used to access the static data members or other static member functions. A member function is defined using the static keyword. A static member function shares the single copy of the member function to any number of the class' objects. We can access the static member function using the class name or class' objects. If the static member function accesses any non-static data member or non-static member function, it throws an error.

Syntax

class_name::function_name (parameter);

Here, the **class_name** is the name of the class.

function_name: The function name is the name of the static member function.

parameter: It defines the name of the pass arguments to the static member function.

Example 2: Let's create another program to access the static member function using the class name in the C++ programming language.

```
#include <iostream>
using namespace std;
class Note
{
// declare a static data member
static int num;

public:
// create static member function
static int func ()
{
return num;
}
};
// initialize the static data member using the class name and the scope resolution operator
int Note :: num = 5;
```

```
int main ()
{
// access static member function using the class name and the scope resolution
cout << " The value of the num is: " << Note:: func () << endl;
return 0;
}
```

Output

The value of the num is: 5

C++ Pointers

In C++, pointers are variables that store the memory addresses of other variables.

```
type *var-name;
```

STL

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many po

Friend function.

A friend function in C++ is a function that is declared outside a class but is capable of accessing the private and protected members of the class. There could be situations in programming wherein we want two classes to share their members. These members may be data members, class functions or function templates.

https://www.programiz.com/cpp-programming/friend-function-class

friend Class in C++

We can also use a friend Class in C++ using the friend keyword. For example

C++ Exceptions

‹ PreviousNext ›

C++ Exceptions

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
When an error occurs, C++ will normally stop and generate an error message.
The technical term for this is: C++ will throw an **exception** (throw an error).

C++ try and catch

Exception handling in C++ consist of three keywords: `try`, `throw` and `catch`:
The `try` statement allows you to define a block of code to be tested for errors while it is being executed.
The `throw` keyword throws an exception when a problem is detected, which lets us create a custom error.
The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.
The `try` and `catch` keywords come in pairs:

Example

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

Consider the following example:

Example

```
try {
  int age = 15;
  if (age >= 18) {
    cout << "Access granted - you are old enough.";
  } else {
    throw (age);
  }
}
catch (int myNum) {
```

```
  cout << "Access denied - You must be at least 18 years old.\n";
  cout << "Age is: " << myNum;
}
```

C++ vs Java

There are many differences and similarities between the C++ programming language and Java. A list of top differences between C++ and Java are given below:

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Mainly used for	C++ is mainly used for system programming.	Java is mainly used for application programming. It is widely used in Windows-based, web-based, enterprise, and mobile applications.
Design Goal	C++ was designed for systems and applications programming. It was an extension of the C programming language.	Java was designed and created as an interpreter for printing systems but later extended as a support network computing. It was designed to be easy to use and accessible to a broader audience.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by using interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers. You can write a pointer program in C++.	Java supports pointer internally. However, you can't write the pointer program in java. It means java has restricted pointer support in java.
Compiler and Interpreter	C++ uses compiler only. C++ is compiled and run using the compiler which converts source code into machine code so, C++ is platform dependent.	Java uses both compiler and interpreter. Java source code is converted into bytecode at compilation time. The interpreter executes this bytecode at runtime and produces output. Java is interpreted that is why it is platform-independent.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports call by value only. There is no call by reference in java.

Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads. It relies on third-party libraries for thread support.	Java has built-in thread support.
Documentation comment	C++ doesn't support documentation comments.	Java supports documentation comment (/** ... */) to create documentation for java source code.
Virtual Keyword	C++ supports virtual keyword so that we can decide whether or not to override a function.	Java has no virtual keyword. We can override all non-static methods by default. In other words, non-static methods are virtual by default.
unsigned right shift >>>	C++ doesn't support >>> operator.	Java supports unsigned right shift >>> operator that fills zero at the top for the negative numbers. For positive numbers, it works same like >> operator.
Inheritance Tree	C++ always creates a new inheritance tree.	Java always uses a single inheritance tree because all classes are the child of the Object class in Java. The Object class is the root of the inheritance tree in java.
Hardware	C++ is nearer to hardware.	Java is not so interactive with hardware.
Object-oriented	C++ is an object-oriented language. However, in the C language, a single root hierarchy is not possible.	Java is also an object-oriented language. However, everything (except fundamental types) is an object in Java. It is a single root hierarchy as everything gets derived from java.lang.Object.

C++ Iterators

Iterators are just like pointers used to access the container elements.

Important Points:

- Iterators are used to traverse from one element to another element, a process is known as **iterating through the container**.
- The main advantage of an iterator is to provide a common interface for all the containers type.
- Iterators make the **algorithm independent** of the type of the container used.
- Iterators provide a generic approach to navigate through the elements of a container.

Syntax

1. <ContainerType> :: iterator;
2. <ContainerType> :: const_iterator;

Operations Performed on the Iterators:

- **Operator (*) :** The '*' operator returns the element of the current position pointed by the iterator.
- **Operator (++) :** The '++' operator increments the iterator by one. Therefore, an

iterator points to the next element of the container.

- **Operator (==) and Operator (!=) :** Both these operators determine whether the two iterators point to the same position or not.
- **Operator (=) :** The '=' operator assigns the iterator.

Difference b/w Iterators & Pointers

Iterators can be smart pointers which allow to iterate over the complex data structures. A Container provides its iterator type. Therefore, we can say that the iterators have the common interface with different container type.
The container classes provide two basic member functions that allow to iterate or move through the elements of a container:

- **begin()**: The begin() function returns an iterator pointing to the first element of the container.
- **end()**: The end() function returns an iterator pointing to the past-the-last element of the container.