C++ Interview Questions With Code Examples

Also read => Top C Programming Interview Questions

Basic C++

Structure of C++ Program

Q #1) What is the basic structure of a C++ program?

Answer: The basic structure of a C++ program is shown below:

```
#include<iostream.h>
int main()
{
                cout<<"Hello,World!";
                return 0;
}
```

The first line that begins with "#" is a **preprocessor directive**. In this case, we are using ***include*** as a directive which tells the compiler to include a header while "***iostream.h***" which will be used for basic input/output later in the program.
Next line is the "main" function that returns an Integer. The main function is the starting point of execution for any C++ program. Irrespective of its position in the source code file, the contents of the main function are always executed first by the C++ compiler.

In the next line, we can see open curly braces that indicate the start of a block of a code. After this, we see the programming instruction or the line of code that uses the count which is the standard output stream (its definition is present in iostream.h).

This output stream takes a string of characters and prints it to a standard output device. In this case it is, "Hello, World!". Please note that each C++ instruction ends with a semicolon (;), which is very much necessary and omitting it will result in compilation errors.

Before closing the braces}, we see another line "return 0;". This is the returning point to the main function.

Every C++ program will have a basic structure as shown above with a preprocessor directive, main function declaration followed by a block of code and then a returning point to the main function which indicates successful execution of the program.

Q #2) What are the Comments in C++?

Answer: Comments in C++ are simply a piece of source code ignored by the compiler. They are only helpful for a programmer to add a description or additional information about their source code.
In C++ there are two ways to add comments:
//single-line comment

/* block comment */

The first type will discard everything after the compiler encounters "//". In the second type, the compiler discards everything between "/*" and "*/".

Variables, Data Types, and Constants

Q #3) Difference between Declaration and Definition of a variable.

Answer: Declaration of a variable is merely specifying the data type of a variable and the variable name. As a result of the declaration, we merely tell the compiler to reserve the space for a variable in the memory according to the data type specified.

Example:

```
int Result;
char c;
int a,b,c;
```

All the above are valid declarations. Also, note that as a result of the declaration, the value of the variable is undetermined.

Whereas, a definition is an implementation/instantiation of the declared variable where we tie up appropriate value to the declared variable, so that linker will be able to link references to the appropriate entities.

From above Example,
Result = 10;

C = 'A';

These are valid definitions.

Q #4) Comment on Local and Global scope of a variable.

Answer: The scope of a variable is defined as the extent of the program code within which the variable remains active i.e. it can be declared, defined or worked with.

There are two types of scope in C++:

1. **Local Scope:** A variable is said to have a local scope or is local when it is declared inside a code block. The variable remains active only inside the block and is not accessible outside the code block.
2. **Global Scope:** A variable has a global scope when it is accessible throughout the program. A global variable is declared on top of the program before all the function definitions.

Example:

```
#include <iostream.h>
Int globalResult=0; //global variable
int main()
{
Int localVar = 10; //local variable.
…..

}
```

Q #5) What is the precedence when there is a Global variable and a Local variable in the program with the same name?

Answer: Whenever there is a local variable with the same name as that of a global variable, the compiler gives precedence to the local variable.

Example:

```
#include <iostream.h>
 int globalVar = 2;
int main()
{
 int globalVar = 5;
 cout<<globalVar<<endl;
```

```
}
```
The output of the above code is 5. This is because, although both the variables have the same name, the compiler has given preference to the local scope.

Q #6) When there is a Global variable and Local variable with the same name, how will you access the global variable?

Answer: When there are two variables with the same name but different scope, i.e. one is a local variable and the other is a global variable, the compiler will give preference to a local variable.
In order to access the global variable, we make use of "***scope resolution operator (::)***". Using this operator, we can access the value of the global variable.

Example:

```
#include<iostream.h>
int x= 10;
int main()
{
 int x= 2;
 cout<<"Global Variable x = "<<::x;
 cout<<"\nlocal Variable x= "<<x;
}
```

Output:
Global Variable x = 10
local Variable x= 2

Q #7) How many ways are there to initialize an int with a Constant?

Answer: There are two ways:
The first format uses traditional C notation.
int result = 10;

The second format uses the constructor notation.
int result (10);

Constants

Q #8) What is a Constant? Explain with an example.

Answer: A constant is an expression that has a fixed value. They can be divided into integer, decimal, floating point, character or string constants depending on their data type.
Apart from decimal, C++ also supports two more constants i.e. octal (to the base 8) and hexadecimal (to the base 16) constants.

Examples of Constants:
75 //integer (decimal)
0113 //octal
0x4b //hexadecimal
3.142 //floating point
'c' //character constant
"Hello, World" //string constant

Note: When we have to represent a single character, we use single quotes and when we want to define a constant with more than one character, we use double quotes.

Q #9) How do you define/declare constants in C++?

Answer: In C++, we can define our own constants using the **#define** preprocessor directive.
#define Identifier value
Example:

```
#include<iostream.h>
#define PI 3.142
int main ()
{
                float radius =5, area;
                area = PI * r * r;
                cout<<"Area of a Circle = "<<area;
}
```

Output: Area of a Circle = 78.55
As shown in the above example, once we define a constant using #define directive, we can use it throughout the program and substitute its value.

We can declare constants in C++ using "***const***" keyword. This way is similar to that of declaring a variable, but with a const prefix.
Examples of declaring a constant
const int pi = 3.142;
const char c = "sth";
const zipcode = 411014;

In the above examples, whenever the type of a constant is not specified, C++ compiler defaults it to an integer type.

Operators

Q #10) Comment on Assignment Operator in C++.
Answer: Assignment operator in C++ is used to assign a value to another variable.
a = 5;

This line of code assigns the integer value **5** to variable **a**.
The part at the left of the =operator is known as ***lvalue*** (left value) and the right as ***rvalue*** (right value). **L*value*** must always be a variable whereas the right side can be a constant, a variable, the result of an operation or any combination of them.
The assignment operation always takes place from the right to left and never at the inverse.

One property which C++ has over the other programming languages is that the assignment operator can be used as the *rvalue* (or part of an *rvalue*) for another assignment.
Example:
a = 2 + (b = 5);

is equivalent to:

b = 5;
a = 2 + b;

Which means, first assign **5** to variable **b** and then assign to **a,** the value **2** plus the result of the previous expression of **b**(that is 5), leaves **a** with a final value of **7**.

Thus, the following expression is also valid in C++:
a = b = c = 5;

assign 5 to variables **a**, **b** and **c**.

Q #11) What is the difference between equal to (==) and Assignment Operator (=)?

Answer: In C++, equal to (==) and assignment operator (=) are two completely different operators.
Equal to (==) is equality relational operator that evaluates two expressions to see if they are equal and returns true if they are equal and false if they are not.

The assignment operator (=) is used to assign a value to a variable. Hence, we can have a complex assignment operation inside the equality relational operator for evaluation.

Q #12) What are the various Arithmetic Operators in C++?

Answer: C++ supports the following arithmetic operators:

- + addition
- – subtraction
- * multiplication
- / division
- % module

Let's demonstrate the various arithmetic operators with the following piece of code.

Example:

```
#include <iostream.h>
int main ()
{
                int a=5, b=3;
cout<<"a + b = "<<a+b;
cout<"\na - b ="<<a-b;
cout<<"\na * b ="<<a*b;
cout<<"\na / b ="<<a/b;
cout<<"\na % b ="<<a%b;

return 0;
 }
```

Output:
a + b = 8
a – b =2
a * b =15
a / b =2
a % b=1

As shown above, all the other operations are straightforward and the same as actual arithmetic operations, except the modulo operator which is quite different. Modulo operator divides a and b and the result of the operation is the remainder of the division.

Q #13) What are the various Compound Assignment Operators in C++?

Answer: Following are the Compound assignation operators in C++:
+=, -=, *=, /=, %=, >>=, <<=, &=, ^=,|=
Compound assignation operator is one of the most important features of C++ language which allow us to change the value of a variable with one of the basic operators:

Example:

```
value += increase; is equivalent to value = value + increase;
if base_salary is a variable of type int.
                int base_salary = 1000;
                base_salary += 1000; #base_salary = base_salary + 1000
                base_salary *= 5; #base_salary = base_salary * 5;
```

Q #14) State the difference between Pre and Post Increment/Decrement Operations.

Answer: C++ allows two operators i.e ++ (increment) and –(decrement), that allow to add 1 to the existing value of a variable and subtract 1 from the variable respectively. These operators are in turn, called increment (++) and decrement (–).

Example:

a=5;
a++;

The second statement, a++, will cause 1 to be added to the value of a. Thus a++ is equivalent to

a = a+1; or
a += 1;

A unique feature of these operators is that we can prefix or suffix these operators with the variable. Hence, if a is a variable and we prefix the increment operator it will be

++a;

This is called Pre-increment. Similarly, we have pre-decrement as well.

If we prefix the variable a with an increment operator, we will have,

a++;

This is the post-increment. Likewise, we have post-decrement too.

The difference between the meaning of pre and post depends upon how the expression is evaluated and the result is stored.

In the case of the pre-increment/decrement operator, the increment/decrement operation is carried out first and then the result passed to a lvalue. Whereas for post-increment/decrement operations, the lvalue is evaluated first and then increment/decrement is performed accordingly.

Example:

a = 5; b=6;
++a; #a=6
b–; #b=6
–a; #a=5
b++; #6

I/O through Console

Q #15) What are the Extraction and Insertion operators in C++? Explain with examples.

Answer: In the iostream.h library of C++, **cin**, and **cout** are the two data streams that are used for input and output respectively. Cout is normally directed to the screen and cin is assigned to the keyboard.

"cin" (extraction operator): By using overloaded operator >> with cin stream, C++ handles the standard input.

```
int age;
cin>>age;
```

As shown in the above example, an integer variable 'age' is declared and then it waits for cin (keyboard) to enter the data. "cin" processes the input only when the RETURN key is pressed.

"cout" (insertion operator): This is used in conjunction with the overloaded << operator. It directs the data that followed it into the cout stream.

Example:

```
cout<<"Hello, World!";
 cout<<123;
```

Control Structures and Functions

Control Structures and Loops

Q #16) What is the difference between while and do while loop? Explain with examples.

Answer: The format of while loop in C++ is:

While (expression)
{statements;}

Statement block under while is executed as long as the condition in the given expression is true.

Example:

```
#include <iostream.h>
 int main()
{
                int n;
                cout<<"Enter the number : ";
                cin>>n;
               while(n>0)
               {
                                cout<<" "<<n;
                                --n;
               }
               cout<<"While loop complete";
 }
```

In the above code, the loop will directly exit if n is 0. Thus in while loop, the terminating condition is at the beginning of the loop and if it's fulfilled, no iterations of the loop are executed.

Next, we consider the do-while loop.

The general format of do-while is:
do {statement;} while(condition);

Example:

```
#include<iostream.h>
```

```
int main()
{
                int n;
                cout<<"Enter the number : ";
                cin>>n;
                do {
                            cout<<n<<",";
                            --n;
                     }while(n>0);
                    cout<<"do-while complete";
}
```

In the above code, we can see that the statement inside the loop is executed at least once as the loop condition is at the end. These are the main differences between the while and do-while.

In case of while, we can directly exit the loop at the beginning if the condition is not met whereas in the do-while loop we execute the loop statements at least once.

Functions

Q #17) What do you mean by 'void' return type?

Answer: All functions should return a value as per the general syntax.
However, in case, if we don't want a function to return any value, we use "**void**" to indicate that. This means that we use "**void**" to indicate that the function has no return value or it returns "**void**".

Example:

```
void myfunc()
{
                          Cout<<"Hello,This is my function!!";
}
int main()
{
myfunc();
return 0;
}
```

Q #18) Explain Pass by value and Pass by reference.

Answer: While passing parameters to the function using "Pass by Value", we pass a copy of the parameters to the function.
Hence, whatever modifications are made to the parameters in the called function are not passed back to the calling function. Thus the variables in the calling function remain unchanged.

Example:

```
void printFunc(int a,int b,int c)
{
                    a *=2;
                    b *=2;
                    c *=2;
}

int main()

{

                  int x = 1,y=3,z=4;
                  printFunc(x,y,z);
```

```
                cout<<"x = "<<x<<"\ny = "<<y<<"\nz = "<<z;
}
```

Output:

x=1
y=3
z=4

As seen above, although the parameters were changed in the called function, their values were not reflected in the calling function as they were passed by value.

However, if we want to get the changed values from the function back to the calling function, then we use "Pass by Reference" technique.

To demonstrate this we modify the above program as follows:

```
void printFunc(int& a,int& b,int& c)
{
                                a *=2;
                                b *=2;
                                c *=2;
}
  int main()
{
                 int x = 1,y=3,z=4;
                 printFunc(x,y,z);
                  cout<<"x = "<<x<<"\ny = "<<y<<"\nz = "<<z;
}
```

Output:

x=2
y = 6
z = 8

As shown above, the modifications done to the parameters in the called functions are passed to the calling function when we use "Pass by reference" technique. This is because, using this technique we do not pass a copy of the parameters but we actually pass the variable's reference itself.

Q #19) What are Default Parameters? How are they evaluated in C++ function?

Answer: Default parameter is a value that is assigned to each parameter while declaring a function.

This value is used if that parameter is left blank while calling to the function. To specify a default value for a particular parameter, we simply assign a value to the parameter in the function declaration.

If the value is not passed for this parameter during the function call, then the compiler uses the default value provided. If a value is specified, then this default value is stepped on and the passed value is used.

Example:

```
int multiply(int a, int b=2)
{
            int r;
            r = a * b;
            return r;
}
```

```
int main()
 {

              Cout<<multiply(6);
              Cout<<"\n";
              Cout<<multiply(2,3);
 }
```

Output:
12
6

As shown in the above code, there are two calls to multiply function. In the first call, only one parameter is passed with a value. In this case, the second parameter is the default value provided. But in the second call, as both the parameter values are passed, the default value is overridden and the passed value is used.

Q #20) What is an Inline function in C++?

Answer: Inline function is a function that is compiled by the compiler as the point of calling the function and the code is substituted at that point. This makes compiling faster. This function is defined by prefixing the function prototype with the keyword "inline". Such functions are advantageous only when the code of the inline function is small and simple. Although a function is defined as Inline, it is completely compiler dependent to evaluate it as inline or not.

Advanced Data Structure

Arrays

Q #21) Why are arrays usually processed with for loop?

Answer: Array uses the index to traverse each of its elements.
If A is an array then each of its element is accessed as A[i]. Programmatically, all that is required for this to work is an iterative block with a loop variable i that serves as an index (counter) incrementing from 0 to A.length-1.

This is exactly what a loop does and this is the reason why we process arrays using for loops.

Q #22) State the difference between delete and delete[].

Answer: "delete[]" is used to release the memory allocated to an array which was allocated using new[]. "delete" is used to release one chunk of memory which was allocated using new.

Q #23) What is wrong with this code?

T *p = new T[10];
delete p;
Answer: The above code is syntactically correct and will compile fine.
The only problem is that it will just delete the first element of the array. Though the entire array is deleted, only the destructor of the first element will be called and the memory for the first element is released.

Q #24) What's the order in which the objects in an array are destructed?

Answer: Objects in an array are destructed in the reverse order of construction: First constructed, last destructed.

In the following Example, the order for destructors will be a[9], a[8], …, a[1], a[0]:

```
voiduserCode()
{
                  Car a[10];
                   ...
}
```

Pointers

Q #25) What is wrong with this code?

T *p = 0;
delete p;

Answer: In the above code, the pointer is a null pointer. Hence naturally, the program will crash in an attempt to delete the null pointer.

Q #26) What is a Reference Variable in C++?

Answer: A reference variable is an alias name for the existing variable. This means that both the variable name and the reference variable point to the same memory location. Hence, whenever the variable is updated, the reference is updated too.

Example:

```
int a=10;
 int& b = a;
```

Here, b is the reference of a.

Storage Classes

Q #27) What is a Storage Class? Mention the Storage Classes in C++.

Answer: Storage class determines the life or scope of symbols such as variable or functions.

C++ supports the following storage classes:

- Auto
- Static
- Extern
- Register
- Mutable

Q #28) Explain Mutable Storage class specifier.

Answer: The variable of a constant class object's member cannot be changed. However, by declaring the variables as "mutable", we can change the values of these variables.

Q #29) What is the keyword auto for?

Answer: By default, every local variable of the function is automatic i.e. *auto*. In the below function both the variables 'i' and 'j' are automatic variables.

```
void f()
 {
 int i;
 auto int j;
 }
```

NOTE: A global variable is not an automatic variable.

Q #30) What is a Static Variable?

Answer: A static variable is a local variable that retains its value across the function calls. Static variables are declared using the keyword "static". Numeric variables which are static have the default value as zero.

The following function will print 1 2 3 if called thrice.

```
void f()
{
static int i;
```

```
++i;
printf("%d ",i);
}
```
If a global variable is static, then its visibility is limited to the same source code.

Q #31) What is the purpose of Extern Storage Specifier?
Answer: "Extern" specifier is used to resolve the scope of a global symbol.
```
#include <iostream >
 using nam espace std;
 main()
 {
extern int i;
 cout<<i<<endl;
 }
 int i=20;
```
In the above code, "i" can be visible outside the file where it is defined.

Q #32) Explain Register Storage Specifier.
Answer: "Register" variable should be used whenever the variable is used. When a variable is declared with a "register" specifier, then the compiler gives CPU register for its storage to speed up the lookup of the variable.

Q #33) When to use "const" reference arguments in a function?
Answer: Using "const" reference arguments in a function is beneficial in several ways:
- "const" protects from programming errors that could alter data.
- As a result of using "const", the function is able to process both const and non-const actual arguments, which is not possible when "const" is not used.
- Using a const reference, allows the function to generate and use a temporary variable in an appropriate manner.

Structure & User Defined Data Types

Q #34) What is a Class?
Answer: Class is a user-defined data type in C++. It can be created to solve a particular kind of problem. After creation, the user need not know the details of the working of a class.
In general, class acts as a blueprint of a project and can include in various parameters and functions or actions operating on these parameters. These are called the members of the class.

Q #35) Difference between Class and Structure.
Answer:
Structure: In C language, the structure is used to bundle different type of data types together. The variables inside a structure are called the members of the structure. These members are by default public and can be accessed by using the structure name followed by a dot operator and then the member name.
Class: Class is a successor of the Structure. C++ extends the structure definition to include the functions that operate on its members. By default all the members inside the class are private.

Object-Oriented Programming with C++

Classes, Constructors, Destructors

Q #36) What is Namespace?
Answer: Namespaces allow us to group a set of global classes, objects and/or functions under a specific name.
The general form to use namespaces is:

namespace identifier { namespace-body }
Where identifier is any valid identifier and the namespace-body is the set of classes, objects, and functions that are included within the namespace. Namespaces are especially useful in the case where there is a possibility for more than one object to have the same name, resulting in name clashes.

Q #37) What is the use of 'using' declaration?
Answer: Using declaration is used to refer a name from the namespace without the scope resolution operator.

Q #38) What is Name Mangling?
Answer: C++ compiler encodes the parameter types with function/method into a unique name. This process is called name mangling. The inverse process is called demangling.
Example:
A::b(int, long) const is mangled as **'b__C3Ail'**.
For a constructor, the method name is left out.

That is **A:: A(int, long)** const is mangled as **'C3Ail'.**

Q #39) What is the difference between an Object and a Class?
Answer: Class is a blueprint of a project or problem to be solved and consists of variables and methods. These are called the members of the class. We cannot access methods or variables of the class on its own unless they are declared static.
In order to access the class members and put them to use, we should create an instance of a class which is called an Object. The class has an unlimited lifetime whereas an object has a limited lifespan only.

Q #40) What are the various Access Specifiers in C++?
Answer: C++ supports the following access specifiers:

- **Public:** Data members and functions are accessible outside the class.
- **Private:** Data members and functions are not accessible outside the class. The exception is the usage of a friend class.
- **Protected:** Data members and functions are accessible only to the derived classes.

Example:
Describe PRIVATE, PROTECTED and PUBLIC along with their differences and give examples.

```
class A{
              int x; int y;
              public int a;
              protected bool flag;
              public A() : x(0) , y(0) {} //default (no argument) constructor
 };

 main(){

A MyObj;

MyObj.x = 5; // Compiler will issue a ERROR as x is private

int x = MyObj.x; // Compiler will issue a compile ERROR MyObj.x is private

 MyObj.a = 10; // no problem; a is public member
 int col = MyObj.a; // no problem
```

```
MyObj.flag = true; // Compiler will issue a ERROR; protected values are read only
bool isFlag = MyObj.flag; // no problem
```

Q #41) What is a Constructor and how is it called?

Answer: Constructor is a member function of the class having the same name as the class. It is mainly used for initializing the members of the class. By default constructors are public.

There are two ways in which the constructors are called:

1. **Implicitly:** Constructors are implicitly called by the compiler when an object of the class is created. This creates an object on a Stack.
2. **Explicit Calling:** When the object of a class is created using new, constructors are called explicitly. This usually creates an object on a Heap.

Example:

```
class A{
 int x; int y;
 public A() : x(0) , y(0) {} //default (no argument) constructor
 };
 main()
 {
                    A Myobj; // Implicit Constructor call. In order to allocate memory
                                //the default constructor is implicitly called.
                  A * pPoint = new A(); // Explicit Constructor call. In order to alloca
                                                //memory on HEAP we call the default
 }
```

Q #42) What is a COPY CONSTRUCTOR and when is it called?

Answer: A copy constructor is a constructor that accepts an object of the same class as its parameter and copies its data members to the object on the left part of the assignment. It is useful when we need to construct a new object of the same class.

Example:

```
class A{
              int x; int y;
              public int color;
              public A() : x(0) , y(0) {} //default (no argument) constructor
              public A( const A& ) ;
};
A::A( const A & p )
{
               this->x = p.x;
               this->y = p.y;
               this->color = p.color;
}
main()
{
            A Myobj;
            Myobj.color = 345;
            A Anotherobj = A( Myobj ); // now Anotherobj has color = 345
}
```

Q #43) What is a Default Constructor?

Answer: Default constructor is a constructor that either has no arguments or if there are any, then all of them are default arguments.

Example:

```
class B {
 public: B (int m = 0) : n (m) {} int n;
 };
 int main(int argc, char *argv[])
{
 B b; return 0;
 }
```

Q #44) What is a Conversion Constructor?
Answer: It is a constructor that accepts one argument of a different type. Conversion constructors are mainly used for converting from one type to another.

Q #45) What is an Explicit Constructor?
Answer: A conversion constructor is declared with the explicit keyword. The compiler does not use an explicit constructor to implement an implied conversion of types. Its purpose is reserved explicitly for construction.

Q #46) What is the role of Static keyword for a class member variable?
Answer: Static member variable shares a common memory across all the objects created for the respective class. We need not refer to the static member variable using an object. However, it can be accessed using the class name itself.

Q #47) Explain the Static Member Function.
Answer: A static member function can access only the static member variable of the class. Same as the static member variables, a static member function can also be accessed using the class name.

Q #48) What's the order in which the local objects are destructed?
Answer: Consider following a piece of code:

```
Class A{
 ….
 };
 int main()
 {
 A a;
 A b;
 ...
 }
```

In the main function, we have two objects created one after the other. They are created in an order, first a then b. But when these objects are deleted or if they go out of the scope, the destructor for each will be called in the reverse order in which they were constructed.

Hence, the destructor of b will be called first followed by a. Even if we have an array of objects, they will be destructed in the same way in the reverse order of their creation.

Overloading

Q #49) Explain Function Overloading and Operator Overloading.
Answer: C++ supports OOPs concept Polymorphism which means “many forms”.
In C++ we have two types of polymorphism, i.e. Compile-time polymorphism, and Run-time polymorphism. Compile time polymorphism is achieved by using an Overloading technique. Overloading simply means giving additional meaning to an entity by keeping its base meaning intact.

C++ supports two types of overloading:
Function Overloading:
Function overloading is a technique which allows the programmer to have more than one function with the same name but different parameter list. In other words, we overload the function with different arguments i.e. be it the type of arguments, number of arguments or the order of arguments.

Function overloading is never achieved on its return type.

Operator Overloading:

This is yet another type of compile-time polymorphism that is supported by C++. In operator overloading, an operator is overloaded, so that it can operate on the user-defined types as well with the operands of the standard data type. But while doing this, the standard definition of that operator is kept intact.

For Example, Addition operator (+) that operates on numerical data types can be overloaded to operate on two objects just like an object of complex number class.

Q #50) What is the difference between Method Overloading and Method Overriding in C++?

Answer: Method overloading is having functions with the same name but different argument list. This is a form of compile-time polymorphism.
Method overriding comes into picture when we rewrite the method that is derived from a base class. Method overriding is used while dealing with run-time polymorphism or virtual functions.

Q #51) What is the difference between a Copy Constructor and an Overloaded Assignment Operator?

Answer: A copy constructor and an overloaded assignment operator basically serve the same purpose i.e. assigning the content of one object to another. But still, there is a difference between the two.

Example:

```
complex c1,c2;
c1=c2; //this is assignment
complex c3=c2; //copy constructor
```

In the above example, the second statement c1 = c2 is an overloaded assignment statement.

Here, both c1 and c2 are already existing objects and the contents of c2 are assigned to the object c1. Hence, for overloaded assignment statement both the objects need to be created already.

Next statement, complex c3 = c2 is an example of the copy constructor. Here, the contents of c2 are assigned to a new object c3, which means the copy constructor creates a new object every time when it executes.

Q #52) Name the Operators that cannot be Overloaded.

Answer:

- sizeof – sizeof operator
- . – Dot operator
- .* – dereferencing operator
- -> – member dereferencing operator
- :: – scope resolution operator
- ?: – conditional operator

Q #53) Function can be overloaded based on the parameter which is a value or a reference. Explain if the statement is true.

Answer: False. Both, Passing by value and Passing by reference look identical to the caller.

Q #54) What are the benefits of Operator Overloading?

Answer: By overloading standard operators on a class, we can extend the meaning of these operators, so that they can also operate on the other user-defined objects.

Function overloading allows us to reduce the complexity of the code and make it more clear and readable as we can have the same function names with different argument lists.

Inheritance

Q #55) What is Inheritance?

Answer: Inheritance is a process by which we can acquire the characteristics of an existing entity and form a new entity by adding more features to it.
In terms of C++, inheritance is creating a new class by deriving it from an existing class so that this new class has the properties of its parent class as well as its own.

Q #56) What are the advantages of Inheritance?

Answer: Inheritance allows code re-usability, thereby saving time on code development. By inheriting, we make use of a bug-free high-quality software that reduces future problems.

Q #57) Does C++ support Multilevel and Multiple Inheritances?

Answer: Yes.

Q #58) What are Multiple Inheritances (virtual inheritance)? What are its advantages and disadvantages?

Answer: In multiple inheritances, we have more than one base classes from which a derived class can inherit. Hence, a derived class takes the features and properties of more than one base class.
For Example, a class **driver** will have two base classes namely, **employee** and a person because a driver is an employee as well as a person. This is advantageous because the driver class can inherit the properties of the employee as well as the person class.
But in the case of an employee and a person, the class will have some properties in common. However, an ambiguous situation will arise as the driver class will not know the classes from which the common properties should be inherited. This is the major disadvantage of multiple inheritance.

Q #59) Explain the ISA and HASA class relationships. How would you implement each?

Answer: “ISA” relationship usually exhibits inheritance as it implies that a class “ISA” specialized version of another class. **Example,** An employee ISA person. That means an Employee class is inherited from the Person class.
Contrary to “ISA”, “HASA” relationship depicts that an entity may have another entity as its member or a class has another object embedded inside it.

So taking the same example of an Employee class, the way in which we associate the Salary class with the employee is not by inheriting it but by including or containing the Salary object inside the Employee class. “HASA” relationship is best exhibited by containment or aggregation.

Q #60) Does a derived class inherit or doesn't inherit?

Answer: When a derived class is constructed from a particular base class, it basically inherits all the features and ordinary members of the base class. But there are some exceptions to this rule. For instance, a derived class does not inherit the base class’s constructors and destructors.

Each class has its own constructors and destructors. The derived class also does not inherit the assignment operator of the base class and friends of the class. The reason is that these entities are specific to a particular class and if another class is derived or if it is the friend of that class, then they cannot be passed onto them.

Polymorphism

Q #61) What is Polymorphism?

Answer: The basic idea behind polymorphism is in many forms. In C++, we have two types of Polymorphism:

(i) Compile time Polymorphism

In compile time polymorphism, we achieve many forms by overloading. Hence, we have Operator overloading and function overloading. (We have already covered this above)

(ii) Run-time Polymorphism

This is the polymorphism for classes and objects. General idea is that a base class can be inherited by several classes. A base class pointer can point to its child class and a base class array can store different child class objects.

This means, that an object reacts differently to the same function call. This type of polymorphism can use virtual function mechanism.

Q #62) What are Virtual Functions?

Answer: A virtual function allows the derived classes to replace the implementation provided by the base class.

Whenever we have functions with the same name in the base as well as derived class, there arises an ambiguity when we try to access the child class object using a base class pointer. As we are using a base class pointer, the function that is called is the base class function with the same name.

To correct this ambiguity we use the keyword “virtual” before the function prototype in the base class. In other words, we make this polymorphic function Virtual. By using a Virtual function, we can remove the ambiguity and we can access all the child class functions correctly using a base class pointer.

Q #63) Give an example of Run-time Polymorphism/Virtual Functions.

Answer:

```
class SHAPE{
                    public virtual Draw() = 0; //abstract class with a pure virtual method
 };
 class CIRCLE: public SHAPE{
                    public int r;
                    public Draw() { this->drawCircle(0,0,r); }
 };
class SQUARE: public SHAPE{
                 public int a;
                 public Draw() { this->drawSquare(0,0,a,a); }
 };

 int main()
 {
                  SHAPE shape1*;
                  SHAPE shape2*;
```

```
                CIRCLE c1;
                SQUARE s1;

               shape1 = &c1;
              shape2 = &s1;
             cout<<shape1->Draw(0,0,2);
             cout<<shape2->Draw(0,0,10,10);
 }
```

In the above code, SHAPE class has a pure virtual function and is an abstract class (cannot be instantiated). Each class is derived from SHAPE implementing Draw () function in its own way.

Further, each Draw function is virtual so that when we use a base class (SHAPE) pointer each time with the object of the derived classes (Circle and SQUARE), then appropriate Draw functions are called.

Q #64) What do you mean by Pure Virtual Functions?

Answer: A Pure Virtual Member Function is a member function in which the base class forces the derived classes to override. Normally this member function has no implementation. Pure virtual functions are equated to zero.

Example:

```
class Shape { public: virtual void draw() = 0; };
```

Base class that has a pure virtual function as its member can be termed as an “Abstract class”. This class cannot be instantiated and it usually acts as a blueprint that has several sub-classes with further implementation.

Q #65) What are Virtual Constructors/Destructors?

Answer:

Virtual Destructors: When we use a base class pointer pointing to a derived class object and use it to destroy it, then instead of calling the derived class destructor, the base class destructor is called.

Example:

```
Class A{
               ….
               ~A();
};
Class B:publicA{
                …
                ~B();
};
B b;
A a = &b;
delete a;
```

As shown in the above example, when we say delete a, the destructor is called but it’s actually the base class destructor. This gives rise to the ambiguity that all the memory held by b will not be cleared properly.

This problem can be solved by using the “Virtual Destructor” concept.

What we do is, we make the base class constructor “Virtual” so that all the child class destructors also become virtual and when we delete the object of the base class pointing

to the object of the derived class, the appropriate destructor is called and all the objects are properly deleted.

This is shown as follows:

```
Class A{
                ….
                virtual ~A();
};
Class B:publicA{
                 …
                 ~B();
};
B b;
A a = &b;
delete a;
```

Virtual constructor: Constructors cannot be virtual. Declaring a constructor as a virtual function is a syntax error.

Friend

Q #66) What is a friend function?

Answer: C++ class does not allow its private and protected members to be accessed outside the class. But this rule can be violated by making use of the "**Friend**" function. As the name itself suggests, friend function is an external function which is a friend of the class. For friend function to access the private and protected methods of the class, we should have a prototype of the friend function with the keyword "friend" included inside the class.

Q #67) What is a friend class?

Answer: Friend classes are used when we need to override the rule for private and protected access specifiers so that two classes can work closely with each other. Hence, we can have a friend class to be the friend of another class. This way, friend classes can keep private, inaccessible things in the way they are.

When we have a requirement to access the internal implementation of a class (private member) without exposing the details by making the public, we go for friend functions.

Advanced C++

Templates

Q #68) What is a template?

Answer: Templates allow creating functions that are independent of data type (generic) and can take any data type as parameters and return value without having to overload the function with all the possible data types. Templates nearly fulfill the functionality of a macro.

Its prototype is any of the following ones:

template <class *identifier***>** *function_declaration;*

template <typename *identifier***>** *function_declaration;*

The only difference between both the prototypes is the use of keyword class or typename. Their basic functionality of being generic remains the same.

Exception Handling

Q #69) What is Exception Handling? Does C++ support Exception Handling?

Answer: Yes C++ supports exception handling.

We cannot ensure that code will execute normally at all times. There can be certain situations which might force the code written by us to malfunction, even though it's error-free. This malfunctioning of code is called **Exception**.
When an exception has occurred, the compiler has to throw it so that we know an exception has occurred. When an exception has been thrown, the compiler has to ensure that it is handled properly, so that the program flow continues or terminates properly. This is called **handling of an exception.**
Thus in C++, we have three keywords i.e. **try**, **throw** and **catch** which are in exception handling.
The general syntax for exception block is:

```
try{
 ….
 # Code that is potentially about to throw exception goes here
 ….
 throw exception;
 }
 catch(exception type) {
 …
 #code to handle exception goes here
 }
```

As shown above, the code that might potentially malfunction is put under the try block. When code malfunctions, an exception is thrown. This exception is then caught under the catch block and is handled i.e. appropriate action is taken.