## General

### What is C++?

C++ is a **mid-level** programming language that was developed as an **extension of the C language**. It’s known for its efficiency, versatility, and direct access to hardware, making it suitable for a wide range of applications. C++ supports both procedural and object-oriented programming (**OOP**) paradigms, allowing developers to write code that is both efficient and organized.

Real-world applications of C++:

* Operating systems
* GUI based applications
* Distributed systems
* Database software
* Banking applications
* Advanced computations and graphics
* Embedded systems

### Advantages of C++

* C++ is an OOP language. That means the data is considered as **objects**. They can work with the concept of *polymorphism*, *inheritance*, *abstraction*, *encapsulation*.
* C++ program uses [multi-paradigm programming](https://www.geeksforgeeks.org/introduction-of-programming-paradigms/) (and [here](https://www.youtube.com/watch?v=B1p5OlO5tWg&ab_channel=TheCodePrism)). By **paradigm**, we mean you can program the logic, structure and procedure of program. Also, **multi-paradigm** means it follows three paradigms Generic, Imperative, Object-Oriented.
* It is a **mid-level** programming language. It can develop OS kernels, embedded systems, drivers, DB engines, graphic engines, desktop applications, games, etc.
* C++ has a **large and mature ecosystem of libraries and frameworks** that can be used to accelerate development and simplify complex tasks like network programming, graphics rendering, etc.
* C++ runs **super fast**. A program written in C++ gets **compiled directly into machine code** – without an intermediary interpretation required at runtime. By contrast, some interpreted languages – such as Java – have to rely on a Java Virtual Machine (JVM). Its code is compiled into an intermediary form, called *bytecode*, before it is converted into machine code during runtime. This intermediary interpretation consumes resources and cause speed issues.
* C++ runs on **multiple platforms** like Windows, Linux, Mac, etc. Of course, with the helps of suitable compilers.

### Disadvantages of C++

* C++ is **not a high-level language**. Its syntax is **complex**. Hence, it can be more **difficult** to learn and to become master.
* C++ program **doesn’t support garbage collector**. It requires careful memory and resource managements. So, it’s not secure in terms of memory management. If the coder is not careful enough, the program can face issues like **memory leak**, segment fault, etc.
* C++ is a compiled language. It has to be compiled into **machine code specific to the target hardware** and OS on which it’s intended to run. By contrast, Java bytecode can run on any computer with a JVM installed, which is the basis of Java's slogan "write-once, run-anywhere".

### C vs C++

|  |  |
| --- | --- |
| **C** | **C++** |
| It is a **procedural** programming language. So, it doesn’t support classes and objects. It also doesn’t support **OOPs properties** like inheritance, polymorphism, abstraction, encapsulation. | It is a mixture of both **procedural** and **object-oriented** programming languages. So, it supports classes and objects. |
| DOES NOT supports: function overload, operator overload, namespace, STL, exception handlings, … | Supports |
| **Functions CANNOT be defined inside structures**.  However, you can include *function pointers* within a structure as members of a structure. So you can achieve a form of encapsulation and associate functions with the structure.  Example:  // Structure definition  struct MyStruct {  int data;  void (\*functionPtr)(int); // function pointer member  };  void myFunction(int value) {  printf("Value: %d\n", value);  }  int main() {  struct MyStruct myStruct;  myStruct.data = 42;  myStruct.functionPtr = myFunction;  // calling the function through the function pointer  myStruct.functionPtr(myStruct.data);  return 0;  } | Functions can be defined inside structures. |

## Variables

### global vs static

…

### extern keyword

…

### Can variables be DEFINED in header file?

## Structs

### Struct vs Class in C++

|  |  |
| --- | --- |
| **Struct** | **Class** |
| Members of the struct are always by default **public** mode. | Members of the class can be in **private, protected, and public** modes. |
| Structure does **not support inheritance**. | Classes support the concept of **inheritance**. |
|  |  |

### In which cases, structs should prefered to class

### Volatile

…

## References

### References in C++

**What and How:**

When a variable is described as a reference it becomes an **alias of the already existing variable**. Because they both share the **same memory address**, changes made in the reference variable will be reflected in the already existing variable. A reference variable is preceded with a **‘&’** symbol.

**Example:**

int GFG = 10;

int& ref = GFG;

**When and Why:**

* Function parameters:
  + More convenient: **simpler syntax**
  + Safer: **cannot be null** and must be initialized, so don’t need to check for null pointers or handle potential null pointer exceptions.
  + Efficiency: unlike passing by value, passing by reference **eliminates the need for creating a new copy of the object**, which can save both time and memory.
* Function return value:
  + Efficiency: unlike returning by value, returning by reference **eliminates the need for creating a new copy of the object**, which can save both time and memory.
  + Enabling *function chaining*: multiple function calls can be chained together in a single expression. This can lead to more readable code.

Example:

class MyVector {

public:

MyVector& add(int value) {

// Add 'value' to the vector

// ...

return \*this; // Return a reference to the current object

}

MyVector& multiply(int factor) {

// Multiply each element of the vector by 'factor'

// ...

return \*this; // Return a reference to the current object

}

};

int main() {

MyVector v;

v.add(5).multiply(2); // Function chaining

return 0;

}

### Call by Value vs Call by Reference

To call a function we have 2 methods: **Call by Value**and **Call by Reference**

|  |  |
| --- | --- |
| **Call by Value** | **Call by Reference** |
| A **copy** of a variable is passed. | A variable **itself** is passed fundamentally. |
| The changes made in the function are never reflected outside the function on the variable. In short, the **original value is never altered** in Call by Value. | The changes made in the functions can be seen outside the function on the passed function. In short, the **original value is altered** in Call by reference. |
| Passed actual and formal parameters are stored in different memory locations. Therefore, causes a **little memory insufficient** | Passed actual and formal parameters are stored in the same memory location. Therefore, cause a little **more memory efficient**. |

### Reference vs Pointer

|  |  |
| --- | --- |
| **Reference** | **Pointer** |
| The value of a reference **cannot be reassigned**. | The value of a pointer **can be reassigned**. |
| It can **never hold a *null*value** as it needs an existing value to become an alias of. | It can **hold or point at a *null* value** and be termed as a *nullptr* or *null pointer.* |
| Once a reference variable is initialized to an object, it **cannot be changed to refer to another object**. | A pointer can be pointed to another object at any time. |
| To access the members of class/struct it uses a ‘ **.**‘ | To access the members of class/struct it uses a ‘ **->**‘. |
| The memory location of reference can be accessed easily or it can be used directly. | The memory location of a pointer cannot be accessed easily as we have to use a dereference ‘ **\***‘ . |
|  |  |
|  |  |

## Overload

### Function Overloading vs Operator Overloading

|  |  |
| --- | --- |
| **Function Overloading** | **Operator Overloading** |
| It is basically defining a function in numerous ways such that there are many ways to call it. In simple terms you have multiple versions of the same function. | It is basically giving practice of giving a special meaning to the existing meaning of an operator (+, -, \*, /, <=, etc). In simple terms redefining the pre-redefined meaning. |
| Parameterized functions are a good example of Function overloading as just by changing the argument of a function you make it useful for different purposes. | Polymorphism is a good example of an operator overloading as an object can be used and called by different classes for different purposes. |
| Example:  int GFG(int X, int Y);  int GFG(char X, char Y); | Example:  int GFG() = X() + Y();  int GFG() = X() – Y(); |
|  |  |

## Data Structures

### Array vs List

|  |  |
| --- | --- |
| **Arrays** | **Lists** |
| Array are **contiguous** memory locations of homogenous data types stored in a **fixed** **size**. | Lists are classic individual elements that are **linked** or connected to each other with the help of pointers and have **flexible size**. |
| Uses less memory than linked lists. | Uses more memory as it has to store the value and the pointer memory location. |
| Require using an **index** to access elements. | Require using an **iterator** to access elements. |

### Array vs Pointer

|  |  |
| --- | --- |
| **Array** | **Pointer** |
| A data structure storing a **group of elements** of the same type. | A variable storing **address** of a **single variable**, function or object. |
| Memory is allocated in a **contiguous** block, either **statically** or **dynamically**. | Memory is allocated **dynamically** allocated. Memory allocation is **random**. |
| Support indexing using either **square** **brackets** or **pointer arithmetic**. | Support indexing using **pointer arithmetic**. |

### Array vs Vector

|  |  |
| --- | --- |
| **Arrays** | **Vectors** |
| Array are contiguous memory locations of homogenous data types stored in a **fixed** **size**. | Vectors have **dynamic size** and can be resized at runtime. Elements can be added or removed easily. |
| Offer **limited functionality** and rely on manual algorithms for operations like sorting or searching. | Provide **various buit-in functions** and algorithms in the <vector> header, such as sorting, searching, inserting, and erasing elements, as well as exception handling. |
| Require using an **index** to access elements. | Provide either **index** or **iterator** to access elements. |

### How to pass arrays as function arguments?

C++ does NOT allow to pass an ENTIRE array as an argument to a function (only the first element of the array is passed). However, you can pass a pointer to the first element of the array by specifying the array's name without an index.

**Way 1: Parameters as a pointer**

void myFunction(int\* param, int size) {

...

}

**Way 2: Parameters as a sized array**

void myFunction(int param[10], int size) {

...

}

**Way 3: Parameters as an un-sized array**

void myFunction(int param[], int size) {

...

}

**Way 4: Pass array to function by reference**

void myFunction(int (&param)[3]) { // suppose: 3 is array size

...

}

### How to return arrays from function?

**Return type as a pointer** (to the first element of the array)

int\* myFunction() {

...

}

**Way 1: Define the local array as static** (reason: memory will be keep outside of the function)

int\* getRandom()

{

static int r[5];

...

return r;

}

**Way 2: Use dynamic array** (reason: memory will be keep outside of the function)

int\* getRandom()

{

int\* r = new int[5];

...

return r;

}

### Dynamic array

…

### Why in vector, list, and some other STL, pre-increment (++i) is prefered to post-increment (i++)?

…

## Memory

### How C/C++ Program Structure The Memory Area

When a program is stared, its execuable data is loaded into the memory (RAM). Its memory area will be structured as following:



#### Text Segment (Code Segment)

It’s one of the sections of a program in an object file or in memory, which **contains executable instructions (compiled binary code)**.

Usually, the text segment is **sharable** so that only a single copy needs to be in memory for frequently executed programs, such as text editors, C compiler, shells, and so on. Also, the text segment is often **read-only**, to prevent a program from accidentally modifying its instructions.

#### Data Segment (Initialized Data Segment)

It’s a portion of virtual address space of a program, which **contains the global variables and static variables** that are initialized (!= 0) by the programmer.

This segment can be further classified into initialized **read-only** area and initialized **read-write** area.

For example:

* The global string defined by char s[] = "hello world" or static int debug = 1 is stored in initialized **read-write** area.
* The global statement like const char\* string = "hello world" makes the string literal to be stored in **read-only** area and the character pointer variable in **read-write** area.

#### BSS (Uninitialized Data Segment)

Uninitialized data segment, often called the "BSS" segment, named after an ancient assembler operator that stood for "block started by symbol."

This segment **contains the global variables and static variables** that are uninitialized or initialized to 0 by the programmer.

For example, a global variable declared int i or static int j would be contained in the BSS segment.

#### Stack (Automatic Memory Storage Segment)

Stack is a region of the computer's memory which **stores temporary variables created by each function**. It is a "LIFO" (last in, first out) data structure. Every time a function declares a new variable, it is "pushed" onto the stack. **Then every time a function exits, all of the variables pushed onto the stack by that function, are deleted**. Once a stack variable is deleted, that region of memory becomes available for other stack variables.

Because of being stored temporarily, stack variables are local in nature. This is related to a concept we often saw – variable scope (or local vs global variables).

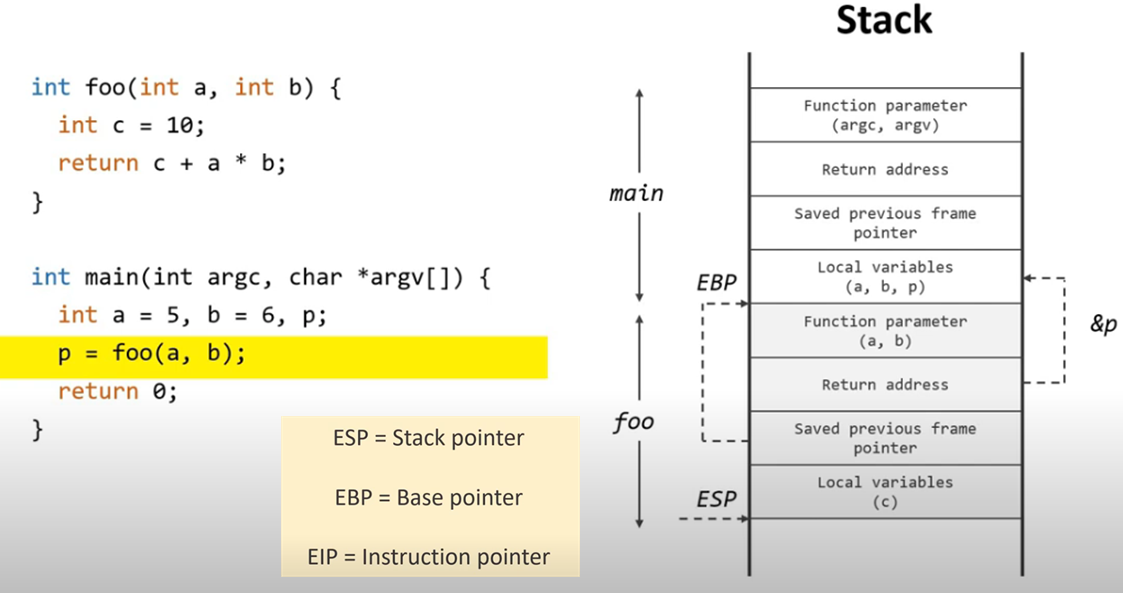
Advantages:

* Data in stack **is automatically managed by OS**. You don't have to allocate memory by hand, or free it once you don't need it any more.
* Because CPU organizes stack memory so efficiently, reading from and writing to stack variables is **very fast**.

Disadvantages:

* There is a **size** **limit** (varies with OS) of variables that can be stored on the stack.

How data is stored in stack:



#### Heap (Dynamic Memory Allocation Segment)

Like the stack, the heap is a region of the computer's memory. However, its data is accessible by any function, anywhere in your program. So, heap variables are essentially global in scope.

Advantages:

* It’s a more free-floating region of memory.
* It’s **size is much larger** than stack.

Disadvantages:

* It’s NOT managed automatically for you and NOT as tightly managed by the CPU.

An extremely important note is that once you have allocated memory on the heap, you are responsible for **deallocating that memory once you don't need it any more.** If you forget to do this, your program will have what is known as a [memory leak](#_1ci93xb). That is, memory on the heap will still be set aside (and won't be available to other processes). There is a tool called [valgrind](http://www.valgrind.org/)that can help you detect memory leaks.

* Heap memory is **slightly slower** to be read from and written to, because one has to use *pointers* to access memory on the heap.

In C/C++, to store variables in heap, you must use built-in functions and operators like malloc(), calloc(), realloc(), or new. And to delete variables after being used, you must use free() or delete.

### new vs malloc()

|  |  |
| --- | --- |
| **new** | **malloc()** |
| an **operator** which performs an operation. | a **function** that returns and accepts values |
| calls the **constructors**. So, it doesn't just allocate memory but also constructs objects with initialized values. | cannot call a constructor. |
| **returns the exact data type**. | **returns void\***. So, type-casting is a must-to-do extra step. |
| **throws a *bad\_alloc* exception if memory allocation fails** | just returns a **null pointer**. |
| automatically calculates the size of the object based on its type. | requires manually specifying the size through a parameter. |

In general, it is **recommended to use operator *new* and *delete* in C++ code**, as they provide type safety and automatic object construction.

### malloc() vs calloc() vs realloc()

…

### delete[] vs delete

|  |  |
| --- | --- |
| **delete[]** | **delete** |
| It is used for deleting the objects of ***new[]****.*  By this, we can say that delete[] is used to **delete an array of objects**. | It is used for deleting the objects of ***new***.  By this, we can say that delete is used to **delete a single object**. |
|  |  |

### What is an Overflow Error?

Overflow Error occurs when the **number is too large for the data type to handle**. In simple terms, it is a type of error that is valid for the defined but exceeds used the defined range where it should coincide/lie.

For example, the range of an *int* is **–2,147,483,648**to **2,147,483,647**. If we declare a variable of size**2,247,483,648**, it will generate a overflow error.

### When Stack? When Heap?

When stack:

* If you are dealing with relatively **small variables** and only need to **live a short life** as long as the function using them alive.

When heap:

* If you need to allocate a **large block of memory** (e.g., a large array, or a big struct), and you need to keep that variable around a long time (like a **global**).
* If you are not aware in advance how much memory you will need to store particular information in a defined variable, so you want it to **change size dynamically** (e.g., dynamic arrays that can grow or shrink as needed).

### "The memory in structures is stored as stacks. The memory in classes is stored as heaps."?

No.

In C++, the memory allocation for both structures and classes can happen on either the stack or the heap, depending on how the objects are defined and created.

When stack:

By default, when you declare an object of a structure or class type as a **local variable within a function or a block**, the memory for that object is typically allocated on the stack. The stack is a region of memory used for automatic storage of local variables and function parameters. When the block or function execution ends, the memory for the variables on the stack is automatically deallocated.

Here's an example:

struct MyStruct {

int value;

};

int main() {

MyStruct struct1; // Memory for struct1 is allocated on the stack

struct1.value = 10;

// ...

return 0; // Memory for struct1 is deallocated when main() exits

}

When heap:

However, you can also allocate memory for objects dynamically on the heap using the **new** operator. This applies to both structures and classes. The heap is a region of memory used for dynamic memory allocation.

struct MyStruct {

int value;

};

int main() {

MyStruct\* struct2 = new MyStruct; // Memory for struct2 is allocated on the heap

struct2->value = 20;

// ...

delete struct2; // Memory for struct2 is deallocated using delete when it is no longer needed

return 0;

}

In this case, the memory for struct2 is allocated on the heap using new, and you need to explicitly deallocate the memory using delete to prevent memory leaks.

### Shallow Copy vs Deep Copy

|  |  |
| --- | --- |
| **Shallow Copy** | **Deep Copy** |
| A copy of the original object is stored and only the reference address is finally copied. In simple terms, Shallow Copy duplicates as little as possible. | A copy of the original object and the repetitive copies both are stored. In simple terms, Deep Copy duplicates everything. |
| A shallow copy of a collection is a copy of the collection structure, not the elements. With a shallow copy, two collections now share individual elements. | A deep copy of a collection is two collections with all of the elements in the original collection duplicated. |
| faster | comparatively slower |
| Shallow copy reflects changes made to the new/copied object in the original object. | Deep copy doesn’t reflect changes made to the new/copied object in the original object. |

### Tool For Finding Memory Leaks and How to Use

### Data Structure Alignment

…

### What Cause Segmentation Faults? Preventation?

In summary, a segfault will be triggered by one of following cases:

* Access invalid memory location (NULL, random, out of range, out of stack space, etc.).
* Write a read-only location.
* Out of stack space.

…

### What Cause Buffer Overflow? Preventation?

…

### What Is Copy Constructor? When Is It Called? When Is It Useful?

…

## Functions

### Inline Function

**What:**

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**.

**Advantages:**

One of the important advantages of using an inline function is that it **eliminates the function calling overhead of a traditional function**.

**Disadvantages:**

**Usecases:**

### When the compiler reject inline functions?

…

### Inline functions vs. Macros?

…

### When to use static functions?

…

### Can static functions be DEFINED in header file?

…

## Pointers

### What Is Void Pointer? Null Pointer? The this Pointer? Pointer to Pointer?

…

### What is Dangling Pointer and Its Causes?

…

### What is Pointer Aliasing?

…

### Why Smart Pointers Are Recommended?

No Explicit Delete: …

Exception Safety: …

Multithread Safety: …

### Types of Smart Pointers and Their Usages and Notes?

…

### Why auto\_ptr is bad?

…

### Notes When Using Smart Pointers With Array?

…

### const char\* ptr vs char\* const ptr

…

## String

### What are the potential problems with using strcpy to copy strings? How can you mitigate them?

…

### How to use Unicode?

…

## OOP

### General

#### Classes vs Objects

A class is a **user-defined data type** where all the member functions and data members are tailor-made according to demands and requirements in addition to which these all can be accessed with the help of an **object**. To declare a user-defined data type we use a keyword ***class.***

An object is an **instance of a class** and an entity with value and state. In simple terms, it is used as a catalyst or to represent a class member. It may contain different parameters or none.

***Note:****A class is a blueprint that defines functions which are used by an object.*

#### What Are OOPs Properties?

* **Inheritance:** The ability of a class to derive properties and characteristics from another class
* **Polymorphism:**Polymorphism is known as many forms of the same thing
* **Abstraction:** It is a technique of showing only necessary details
* **Encapsulation:** Wrapping of data in a single unit

For more information, refer to [Various OOPs concepts in C++](https://www.geeksforgeeks.org/object-oriented-programming-in-cpp/)

**Explain:**

**Inheritance:** In simple terms, it is a technique of reusing and extending existing classes without modifying them.

**Polymorphism**: In simple terms, it is the ability to display a member function in multiple forms depending on the type of object that calls them.

#### Copy Constructor

**Copy Constructor:**A copy constructor is a member function that initializes an object using another object of the same class. Also, the Copy constructor takes a reference to an object of the same class as an argument.

Sample(Sample& t)

{

id = t.id;

}

#### Virtual Destructor

When destroying objects of a derived class using a base class object, a virtual destructor is invoked to free up memory space allocated by the derived class object.

Virtual destructor guarantees that first **the derived class’ destructor is called**. Then **the base class’s destructor is called** to release the space occupied by both destructors.

It is advised to **make your destructor virtual whenever your class is polymorphic**.

#### What Is Scope Resolution?

A scope resolution operator is denoted by a ‘**::**‘ symbol. It resolves the barrier of scope in a program. A scope resolution operator is used to reference a member out of their scope.

Scope Resolution is used for various tasks:

1. To access a global variable when there is a local variable with the same name
2. To define the function outside the class
3. In case of multiple inheritances
4. For namespace

For more information, refer to [Scope resolution operator](https://www.geeksforgeeks.org/scope-resolution-operator-in-c/)

#### Static Data Members and Static Member Functions

The static data member of a class is a normal data member, but preceded with a *static* keyword. It is only visible to a defined class but its **scope is of a lifetime**.

The static member function is the member function that is **used to access other static data members or other static member functions**. It is also defined with a *static* keyword. We can access the static member function using the class name or class objects. (*classname::function name(parameter);*)

#### Can we call a virtual function from a constructor?

Yes, we can call a virtual function from a constructor. But it can throw an exception of overriding.

**50. What are void pointers?**

Just like its name a void pointer is a pointer that is not associated with anything or with any data type. Nevertheless, a void pointer can hold the address value of any type and can be converted from one data type to another.

For more, information refers to [Void Pointer in C++](https://www.geeksforgeeks.org/void-pointer-c-cpp/)

**Bonus Question:**

**What is ‘*this*‘ pointer in C++?**

***this***pointer enables every object to have access to its own address through an essential pointer. All member functions take ***this*** pointer as an implicit argument. ***this***pointermay be used to refer to the calling object within a member function.

* *this*pointer is used to pass an object as a parameter to another method.
* Each object gets its own copy of the data member.
* *this*pointer is used to declare indexers.

For more information, refer to [*this*pointer in C++](https://www.geeksforgeeks.org/this-pointer-in-c/)

#### How to initialize static member variables of class?

…

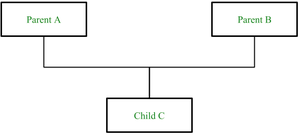
#### Using 'explicit' Keyword

…

### Inheritance

#### What Is Multiple Inheritance? Should We Use It?

Multiple inheritances mean that a derived class can inherit two or more base classes. It is useful when a derived class needs to inherit some or all of the implementation from base classes.



…

#### What Is Virtual Inheritance?

Virtual inheritance is a technique that ensures only one copy of a base class’s member variables is inherited by grandchild-derived classes. Or in simple terms, virtual inheritance is used when we are dealing with a situation of multiple inheritances but want to prevent multiple instances of the same class from appearing in the inheritance hierarchy.

#### Friend Class and Friend Function

A *friend class* can access private, protected, and public members of other classes in which it is declared as friends.

Like friend class, *friend function* can also access private, protected, and public members. But, Friend functions are not member functions.

**Example:**

**class** GFG {

   statements;

**friend** dataype function\_Name(arguments);

   statements;

   }

   OR

**class** GFG{

   statements'

**friend** **int** divide(10,5);

   statements;

   }

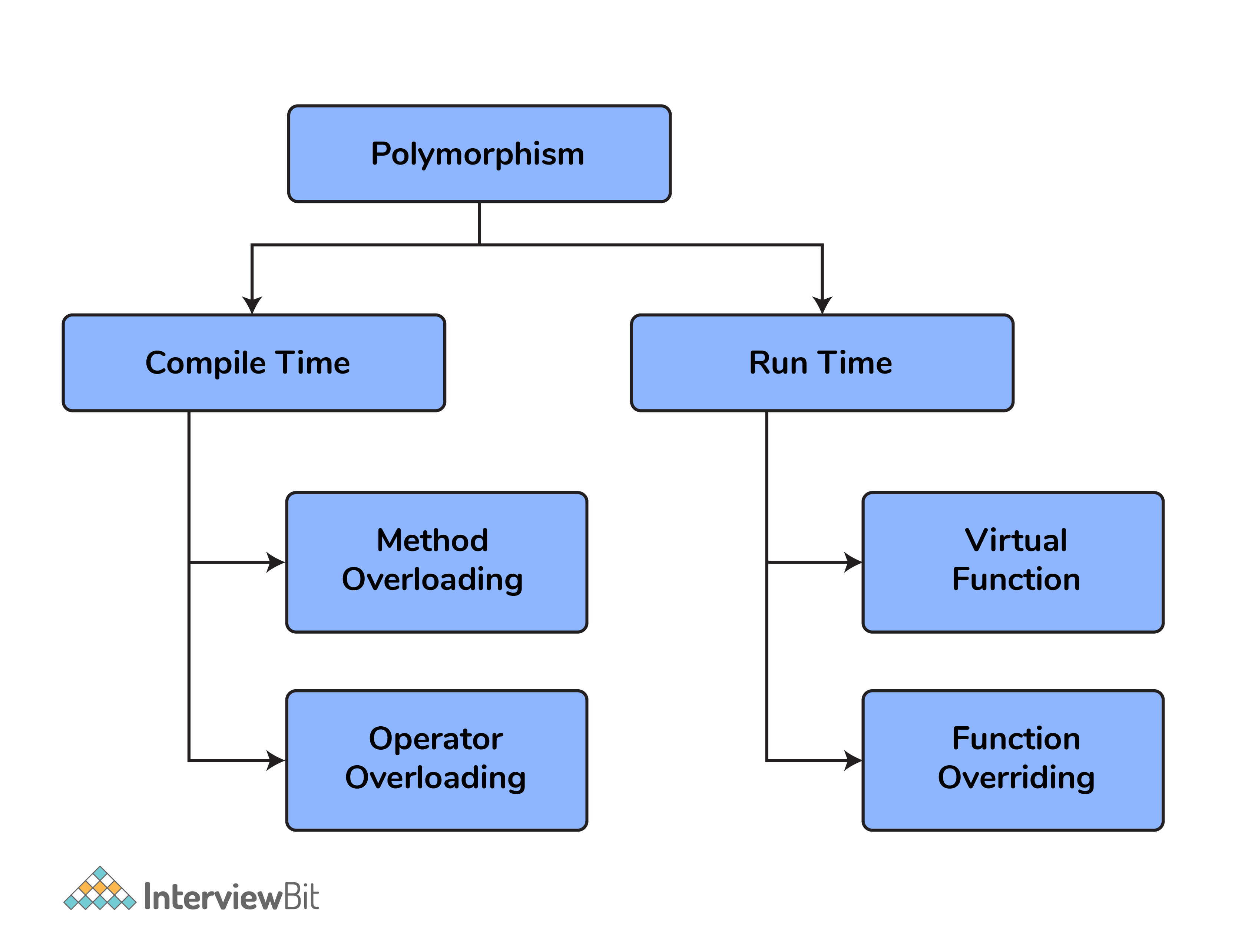
The advantage is that a *friend* function is not bound to the scope of the class. And once it is declared in a class, it cannot be called by an object of the class, therefore it can be called by other functions. Considering all the mentioned points we can say that a friend function is a global function**.**

#### Which Elements In a Base Class That a Derived Class CANNOT inherit?

…

### Polymophism

#### Different Types of Polymorphism



**Compile-Time Polymorphism or Static Binding**

This type of polymorphism is achieved **during the compile time** of the program which results in it making a bit faster than Run time. It is comprised of ***2 further techniques***:

* **Function Overloading:**

// C++

**int** GFG() { }

**int** GFG(**int** a) { }

**float** GFG(**double** a) { }

**int** GFG(**int** a, **double** b) { }

* **Operator Overloading:**

**Run-Time Polymorphism or Late Binding**

Run-time polymorphism takes place when functions are invoked **during run time**.

* **Function Overriding:**Function overriding occurs when a base class member function is redefined in a derived class with the same arguments and return type.

**Compare:**

|  |  |
| --- | --- |
| **Virtual Function** | **Pure Virtual Function** |
| A Virtual Function is a member function of a base class that can be redefined in another derived class. | A Pure Virtual Function is a member function of a base class that is only declared in a base class and defined in a derived class to prevent it from becoming an abstract class. |
| A virtual Function has its **definition** in its respective base class. | There is **no definition** in Pure Virtual Function and is initialized with a pure specifier (= 0). |
| The base class has a virtual function that can be represented or instanced. In simple words, its **object can be made**. | A base class having pure virtual function becomes abstract that cannot be represented or instanced. In simple words, it means its **object cannot be made**. |
|  |  |
|  |  |

#### Virtual Functions vs Pure Virtual Functions

|  |  |
| --- | --- |
| **Virtual Function** | **Pure Virtual Function** |
| A Virtual Function is a member function of a base class that **can be redefined in a derived class**.  When the function is made virtual, C++ determines which function is to be **invoked at the runtime based on the type of the object** pointed by the base class pointer. | A Pure Virtual Function is a member function of a base class that is **only declared in a base class and defined in a derived class** to prevent it from becoming an abstract class. |
| A virtual Function has its **definition** in its respective base class. | There is **no definition** in Pure Virtual Function and is initialized with a pure specifier (= 0). |
| The base class has a virtual function that can be represented or instanced. In simple words, its **object can be made**. | A base class having pure virtual function becomes abstract that cannot be represented or instanced. In simple words, it means its **object cannot be made**. |
|  |  |
|  |  |

#### What Is Function Overriding?

When a function of the same name, same arguments or parameters, and same return type already present/declared in the base class is used in a derived class is known as Function Overriding. It is an example of Runtime Polymorphism or Late Binding which means the overridden function will be executed at the run time of the execution.

#### Is Destructor Overloading Possible?

**NO**

We cannot overload a destructor. It is mandatory to only destructor per class in C++. Also, destructor neither take arguments nor they have a parameter that might help to overload.

**Which operations are permitted on pointers?**

1. Increment/Decrement of a Pointer
2. Addition and Subtraction of integer to a pointer
3. Comparison of pointers of the same type

#### Can We Call a Virtual Function From a Constructor?

Yes, we can call a virtual function from a constructor. But the behavior is a little different in this case. When a virtual function is called, the virtual call is resolved at runtime. It is always the member function of the current class that gets called. That is the virtual machine doesn’t work within the constructor.

For example:

class base{

private:

int value;

public:

base(int x){

value=x;

}

virtual void fun(){

}

}

class derived{

private:

int a;

public:

derived(int x, int y):base(x){

base \*b;

b=this;

b->fun(); //calls derived::fun()

}

void fun(){

cout<<”fun inside derived class”<<endl;

}

}

### Abstract

#### What Do You Mean By Abstraction

Abstraction is the process of **showing the essential details to the user and hiding the details** which we don’t want to show to the user or which are irrelevant to a particular user.

#### Abstract Class

An abstract class is a class that is **specifically designed to a base class**. An abstract class contains **at least one pure virtual function**. You **cannot declare an object of an abstract class**. However, it can be used to declare pointers and references to an abstract class.

An abstract class is used if you want to **provide a common, implemented functionality among all the implementations of the component**. Abstract classes will allow you to **partially implement your class**. They fit if you want to provide implementation details to your children, but don’t want to allow an instance of your class to be directly instantiated.

#### Interface

…

## Exception Handling

### Why using exception handling instead of conditional statement?

…

## Signal Handling

…

### What is the main use of the keyword “Volatile”?

Just like its name, things can change suddenly and unexpectantly. So it is **used to inform the compiler that the value may change anytime**.

Also, the volatile keyword prevents the compiler from performing optimization on the code. It was intended to be used when interfacing with memory-mapped hardware, signal handlers, and machine code instruction.

For more information, refer to this [Volatile](https://www.geeksforgeeks.org/understanding-volatile-qualifier-c-set-1-introduction/)

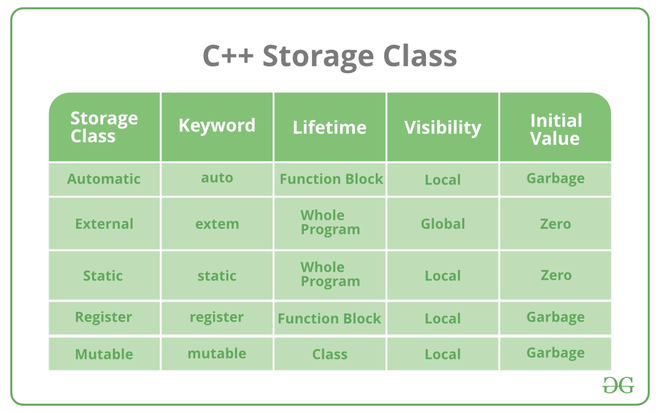
### Define storage class in C++ and name some

Storage class is used to define the features(lifetime and visibility) of a variable or function. These features usually help in tracing the existence of a variable during the runtime of a program.

**Syntax:**

storage\_class var\_data\_type var\_name;

**Some types of storage classes:**



For more information, refer to [Storage Class](https://www.geeksforgeeks.org/storage-classes-in-c-with-examples/)

**43. What is a mutable storage class specifier? How can they be used?**

Just like its name, the mutable storage class specifier is used only on a class data member to make it modifiable even though the member is part of an object declared as const. Static or const, or reference members cannot use the mutable specifier. When we declare a function as const, this pointer passed to the function becomes const.

**44. Define the Block scope variable.**

So the scope of a variable is a region where a variable is accessible. There are two scope regions, A global and block or local.

A block scope variable is also known as a local scope variable. A variable that is defined inside a function (like main) or inside a block (like loops and if blocks) is a local variable. It can be used ONLY inside that particular function/block in which it is declared. a block-scoped variable will not be available outside the block even if the block is inside a function.

For more information, refer to [Scope of a variable](https://www.geeksforgeeks.org/scope-of-variables-in-c/)

**45. What is the function of the keyword “Auto”?**

The auto keyword may be used to declare a variable with a complex type in a straightforward fashion.

You can use auto to declare a variable if the initialization phrase contains templates, pointers to functions, references to members, etc. With type inference capabilities, we can spend less time having to write out things the compiler already knows. As all the types are deduced in the compiler phase only, the time for compilation increases slightly but it does not affect the runtime of the program.

For more information, refer to[Auto in C++](https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/)

### Define Namespace in C++

Benefits:

* **Helps to organize different program parts (**variables, methods, and classes) **into distinct logical scopes with names**.
* Avoids name clashes, which might happen when you have many libraries in your code base.

## Multithread and Multiprocess

### Concurrency vs Parallel

…

### Mutext vs Semaphore vs Condition Variable

…

### Deadlock vs Livelock vs Starvation

…

### Thread Pool

…

## Move Semantics

…

Describe the concept of RAII (Resource Acquisition Is Initialization) in C++ and provide examples of how it can be used to manage resources safely.

Explain the concept of template metaprogramming (TMP) in C++ and provide examples of how it can be used for compile-time computations and optimizations.

Discuss the implications and potential issues related to multiple inheritance in C++, including the diamond problem and virtual inheritance.

Describe the concept of constexpr in C++ and explain how it enables compile-time evaluation of functions and data.

stack unwinding in exception

effective c++: rules of 3, rules of 5

Describe the concept of type erasure in C++ and discuss how it can be achieved using techniques such as virtual functions, templates, or the Any/Variant pattern.

Explain the concept of perfect forwarding in C++ and how it is related to universal references and template argument deduction.

Describe the concept of SFINAE (Substitution Failure Is Not An Error) and how it can be used for compile-time type checking and template specialization.

Explain the concept of type traits in C++ and how they can be used to extract and manipulate properties of types at compile time.

Explain the concept of move-only types in C++ and how they can be used to enforce move semantics and prevent copying of objects.

Discuss the concept of expression templates in C++ and how they can be used to optimize mathematical and numerical computations.

Explain the concept of function-level thread parallelism in C++ and how it can be achieved using features such as std::async, std::future, and std::promise.

Describe the concept of CRTP (Curiously Recurring Template Pattern) in C++ and provide examples of how it can be used to achieve static polymorphism.

Discuss the various techniques and best practices for optimizing C++ code, including loop unrolling, function inlining, cache optimization, and algorithmic improvements.

What are atomic operations in C++ and how do they relate to multithreading?

Remote Procedure Calls (RPC)

Discuss the concept of the Pimpl (Pointer to Implementation) idiom in C++.

Discuss the concept of the SFINAE (Substitution Failure Is Not An Error) principle in C++.

Explain the concept of the RAII (Resource Acquisition Is Initialization) idiom in C++.

Explain the concept of the copy-swap idiom and its usage in C++.