C++ programming language

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a **compiled** language.

1. C++ is a high-level, general-purpose programming language designed for system and application programming. It was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C programming language. C++ is an object-oriented, multi-paradigm language that supports procedural, functional, and generic programming styles.
2. One of the key features of C++ is its ability to support low-level, system-level programming, making it suitable for developing operating systems, device drivers, and other system software. At the same time, C++ also provides a rich set of libraries and features for high-level application programming, making it a popular choice for developing desktop applications, video games, and other complex applications.
3. C++ has a large, active community of developers and users, and a wealth of resources and tools available for learning and using the language.
4. Machine Independent but Platform Dependent**:** A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent. Some of the key features of C++ include:
5. Object-Oriented Programming: C++ supports object-oriented programming, allowing developers to create classes and objects and to define methods and properties for these objects. C++ is an Object-Oriented Programming Language, unlike C which is a procedural programming language. This is the most important feature of **C++**. It can create/destroy objects while programming. Also, It can create blueprints with which objects can be created.
6. Concepts of Object-oriented programming Language:

- Class
- Objects
- Encapsulation
- Polymorphism
- Inheritance
- Abstraction

7. Strong Type System: C++ has a strong type system, which means that variables have a specific type and that type must be respected in all operations performed on that variable.
8. Cross-platform Compatibility: C++ can be compiled and run on multiple platforms, including Windows, MacOS, and Linux, making it a versatile language for developing cross-platform applications.
9. Performance: C++ is a compiled language, which means that code is transformed into machine code before it is executed. This can result in faster execution times and improved performance compared to interpreted languages like Python.
10. Memory Management: C++ requires manual memory management, which can lead to errors if not done correctly. However, this also provides more control over the program's memory usage and can result in more efficient memory usage.
11. Syntax: C++ has a complex syntax that can be difficult to learn, especially for beginners. However, with practice and experience, it becomes easier to understand and work with.
12. Templates: C++ templates allow developers to write generic code that can work with any data type, making it easier to write reusable and flexible code.
13. Standard Template Library (STL): The STL provides a wide range of containers and algorithms for working with data, making it easier to write efficient and effective code.
14. Exception Handling: C++ provides robust exception handling capabilities, making it easier to write code that can handle errors and unexpected situations.
15. Case-sensitive: It is clear that C++ is a case-sensitive programming language. For example, **cin** is used to take input from the input stream. But if the **"Cin"** won't work. Other languages like HTML and MySQL are not case-sensitive languages.
16. Compiler Based C++ is a compiler-based language, unlike Python. That is C++ programs used to be compiled and their executable file is used to run them. C++ is a relatively faster language than Java and Python.

Overall, C++ is a powerful and versatile programming language that is widely used for a range of applications and is well-suited for both low-level system programming and high-level application development.

Here are some simple C++ code examples to help you understand the language:

Program:

- #include <iostream>

```
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Output: Hello, World!

Applications of C++:

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming. e.g. *Linux-based OS (Ubuntu etc.)*
- Browsers *(Chrome & Firefox)*
- Graphics & Game engines *(Photoshop, Blender, Unreal-Engine)*
- Database Engines *(MySQL, MongoDB, Redis etc.)*
- Cloud/Distributed Systems

Disadvantages of C++:

1. Steep Learning Curve: C++ can be challenging to learn, especially for beginners, due to its complexity and the number of concepts that need to be understood.
2. Verbose Syntax: C++ has a verbose syntax, which can make code longer and more difficult to read and maintain.
3. Error-Prone: C++ provides low-level access to system resources, which can lead to subtle errors that are difficult to detect and fix.

Some interesting facts about C++:

Here are some awesome facts about C++ that may interest you:

- The name of C++ signifies the evolutionary nature of the changes from C. “++” is the C increment operator.
- C++ is one of the predominant languages for the development of all kind of technical and commercial software.
- C++ introduces Object-Oriented Programming, not present in C. Like other things, C++ supports the four primary features of OOP: encapsulation, polymorphism, abstraction, and inheritance.
- C++ got the OOP features from Simula67 Programming language.
- A function is a minimum requirement for a C++ program to run.(at least main() function)

Differences between C and C++ are:

C++ is often viewed as a superset of C. C++ is also known as a "C with class" This was very nearly true when C++ was originally created, but the two languages have evolved over time with C picking up a number of features that either weren't found in the contemporary version of C++ or still haven't made it into any version of C++. That said, C++ is still mostly a superset of C adding Object-Oriented Programming, Exception Handling, Templating, and a more extensive standard library.

Below is a table of some of the more obvious and general differences between C and C++. There are many more subtle differences between the languages and between versions of the languages.

C	**C++**
C was developed by Dennis Ritchie between the year 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979.
C does no support polymorphism, encapsulation, and inheritance which means that C does not support object oriented programming. C is (mostly) a subset of C++.	C++ supports polymorphism, encapsulation, and inheritance because it is an object oriented programming language. C++ is (mostly) a superset of C.
Number of keywords in C: * C90: 32 * C99: 37 * C11: 44 * C23: 59 For the development of code, C supports procedural programming.	Number of keywords in C++: * C++98: 63 * C++11: 73 * C++17: 73 * C++20: 81 C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.

C	C++
Data and functions are separated in C because it is a procedural programming language.	Data and functions are encapsulated together in form of an object in C++.
C does not support information hiding.	Data is hidden by the Encapsulation to ensure that data structures and operators are used as intended.
Built-in data types is supported in C.	Built-in & user-defined data types is supported in C++.
C is a function driven language because C is a procedural programming language.	C++ is an object driven language because it is an object oriented programming.
Function and operator overloading is not supported in C.	Function and operator overloading is supported by C++.
C is a function-driven language.	C++ is an object-driven language
Functions in C are not defined inside structures.	Functions can be used inside a structure in C++.
Namespace features are not present inside the C.	Namespace is used by C++, which avoid name collisions.
Standard IO header is stdio.h.	Standard IO header is iostream.h.
Reference variables are not supported by C.	Reference variables are supported by C++.

C	C++
Virtual and friend functions are not supported by C.	Virtual and friend functions are supported by C++.
C does not support inheritance.	C++ supports inheritance.
Instead of focusing on data, C focuses on method or process.	C++ focuses on data instead of focusing on method or procedure.
C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.
Direct support for exception handling is not supported by C.	Exception handling is supported by C++.
scanf() and printf() functions are used for input/output in C.	cin and cout are used for input/output in C++.
C structures don't have access modifiers.	C ++ structures have access modifiers.
C follows the top-down approach	C++ follows the Bottom-up approach
There is no strict type checking in C programming language.	Strict type checking in done in C++. So many programs that run well in C compiler will result in many warnings and errors under C++ compiler.
C does not support overloading	C++ does support overloading

C	C++
File extension is “.c”	File extension is “.cpp” or “.c++” or “.cc” or “.cxx”
Meta-programming: macros + _Generic()	Meta-programming: templates (macros are still supported but discouraged)
There are 32 keywords in the C	There are 97 keywords in the C++

C++ Hello World Program

Below is the C++ program to print Hello World.

```
// C++ program to display "Hello World"
// Header file for input output functions
#include <iostream>
using namespace std;
int main()                 // Main() function: where the execution of program begins
{
        cout << "Hello World";   // Prints hello world
    return 0;
}
```

Working of Hello World Program in C++

Let us now understand every line and the terminologies of the above program.

1. // C++ program to display “Hello World”

This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic.

When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with ‘//’ without quotes OR in between /*…*/ in C++ is a comment.

2. #include

This is a preprocessor directive. The **#include** directive tells the compiler to include the content of a file in the source code.
For example, **#include<iostream>** tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions.

3. using namespace std

This is used to import the entity of the std namespace into the current namespace of the program. The statement using namespace std is generally considered a bad practice. When we import a namespace we are essentially pulling all type definitions into the current scope.

The std namespace is huge. The alternative to this statement is to specify the namespace to which the identifier belongs using the scope operator(::) each time we declare a type. For example, std::cout.

4. int main() { }

A function is a group of statements that are designed to perform a specific task. The main() function is the entry point of every C++ program, no matter where the function is located in the program.

The opening braces '{' indicates the beginning of the main function and the closing braces '}' indicates the ending of the main function.

5. cout<<"Hello World";

std::cout is an instance of the std::ostream class, that is used to display output on the screen. Everything followed by the character << in double quotes " " is displayed on the output device. The semi-colon character at the end of the statement is used to indicate that the statement is ending there.

6. return 0

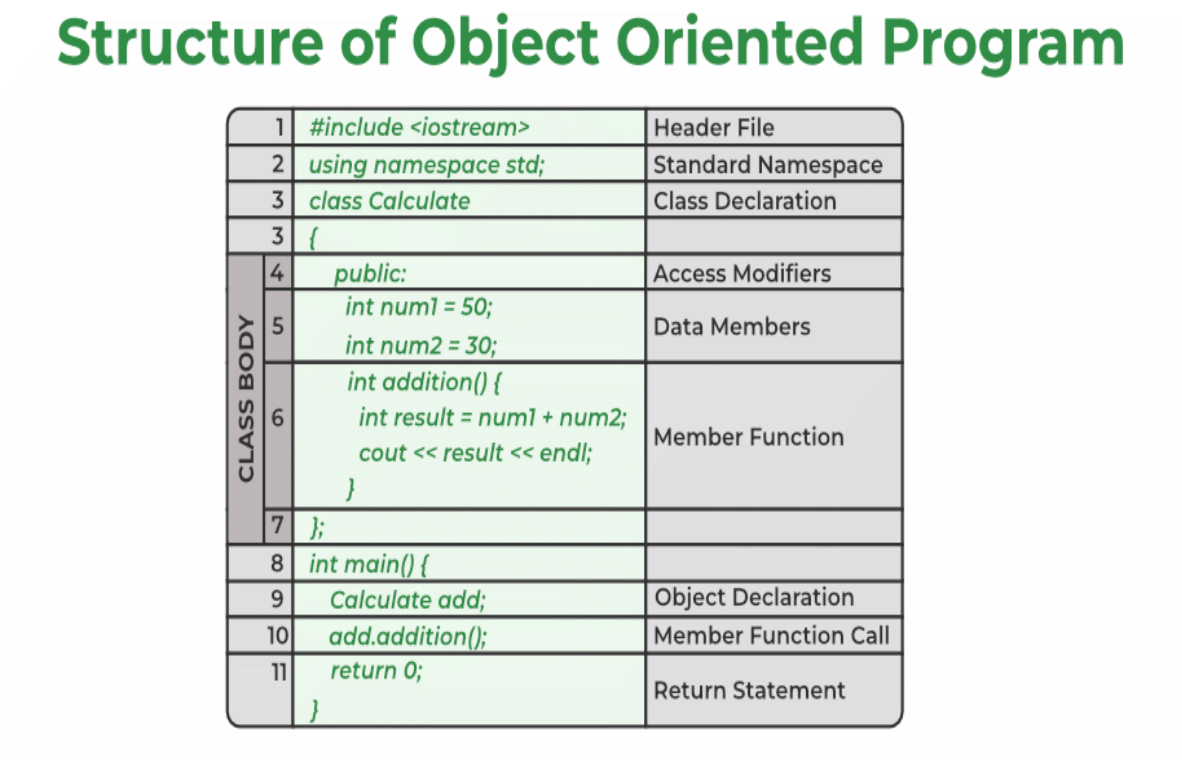
This statement is used to return a value from a function and indicates the finishing of a function. This statement is basically used in functions to return the results of the operations performed by a function.

Important Points

- Always include the necessary header files for the smooth execution of functions. For example, **<iostream>** must be included to use **std::cin** and **std::cout**.
- The execution of code begins from the **main()** function.
- It is a good practice to use **Indentation** and **comments** in programs for easy understanding.
- **cout** is used to print statements and **cin** is used to take inputs.

Object-Oriented Programming in C++

C++ programming language supports both procedural-oriented and object-oriented programming. The above example is based on the procedural-oriented programming paradigm.



1. Class

A class is a template of an object. For example, the animal is a class and dog is the object of the animal class. It is a user-defined data type. A class has its own attributes (data members) and behavior (member functions). The first letter of the class name is always capitalized & use the ***class*** keyword for creating the class.

In **line #3**, we have declared a class named **Calculate** and its body expands from **line #3** to **line #7**.

Syntax:

```
class class_name
{
   // class body
};
```

2. Data Members & Member Functions

The attributes or data in the class are defined by the data members & the functions that work on these data members are called the member functions.

Example:

```
class Calculate{

   public:

      int num1 = 50;    // data member
      int num2 = 30;    // data member


      // member function
      int addition() {
         int result = num1 + num2;
         cout << result << endl;
      }
};
```

In the above example, num1 and num2 are the data member & addition() is a member function that is working on these two data members. There is a keyword here **public** that is **access modifiers**. The access modifier decides who has access to these data members &

member functions. **public** access modifier means these data members & member functions can get access by anyone.

3. Object

The object is an instance of a class. The class itself is just a template that is not allocated any memory. To use the data and methods defined in the class, we have to create an object of that class.

They are declared in the similar way we declare variables in C++.

Syntax:

class_name object_name;

We use dot operator (**.**) for accessing data and methods of an object.

```
#include <iostream>
using namespace std;
class Calculate{
    // Access Modifiers
  public:
       // data member
      int num1 = 50;
      int num2 = 30;
      // memeber function
      int addition() {
          int result = num1 + num2;
          cout << result << endl;
      }
};
int main() {
  // object declaration
  Calculate add;
    // member function calling
  add.addition();
```

```
    return 0;
}
```

Types of Comments in C++

In C++ there are two types of comments in C++:

- **Single-line comment**
- **Multi-line comment**

1. Single Line Comment

In C++ Single line comments are represented as **//** double forward slash. It applies comments to a single line only. The compiler ignores any text after // and it will not be executed.

Syntax:

// Single line comment

```
// C++ Program to demonstrate single line comment
#include <iostream>
using namespace std;
int main()
{
    // Single line comment which will be ignored by the
    // compiler
    cout << "Hello!";
    return 0;
}
```

2. Multi-Line Comment

A multi-line comment can occupy many lines of code, it starts with **/*** and ends with ***/**, but it cannot be nested. Any text between /* and */ will be ignored by the compiler.

Syntax:

```
 /*
    Multiline Comment
 */
```

Example:

```
// C++ Program to demonstrate Multi line comment
#include <iostream>
using namespace std;
int main()
{
   /* Multi line comment which
      will be ignored by the compiler
   */
   cout << "Hello!";
   return 0;
}
```

How does the compiler process C++ Comments?

As a part of the compiler, the Lexical Analyzer scans the characters and transforms them into tokens with no passing of the commented text to the parser. Since Comments do not contribute to the functionality of the program they are simply omitted at the time of compilation. Accordingly, we can understand that comments are just text in programs that are ignored by the compiler.

Token

When the compiler is processing the source code of a C++ program, each group of characters separated by white space is called a token. Tokens are the smallest individual units in a program. A C++ program is written using tokens. It has the following tokens:

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords

Keywords (also known as **reserved words**) have special meanings to the C++ compiler and are always written or typed in short(lower) cases. Keywords are words that the language uses for a special purpose, such as **void**, **int**, **public**, etc. It can't be used for a variable name or

function name or any other identifiers. The total count of reserved keywords is 95. Below is the table for some commonly used C++ keywords.

C++ Keyword			
Asm	double	new	switch
Auto	else	operator	template
Break	enum	private	This
Case	extern	protected	Throw
Catch	float	public	Try
Char	for	register	typedef
Class	friend	return	Union
Const	goto	short	unsigned
continue	if	signed	Virtual
default	inline	sizeof	Void
delete	int	static	volatile
Do	long	struct	while

What is the identifier?

Identifiers refer to the name of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language.

Rules for naming identifiers:

- Identifier names can not start with a digit or any special character.
- A keyword cannot be used as s identifier name.
- Only alphabetic characters, digits, and underscores are permitted.
- The upper case and lower case letters are distinct. i.e., A and a are different in C++.
- The valid identifiers are GFG, gfg, and geeks_for_geeks.

Examples of good and bad identifiers

Invalid Identifier	Bad Identifier	Good Identifier
Cash prize	C_prize	Cashprize
Catch	catch_1	catch1
1list	list_1	list1

How keywords are different from identifiers?

So there are some main properties of keywords that distinguish keywords from identifiers:

Keywords	**Identifiers**
Keywords are predefined/reserved words	identifiers are the values used to define different programming items like a variable, integers, structures, and unions.
Keywords always start in lowercase	identifiers can start with an uppercase letter as well as a lowercase letter.
It defines the type of entity.	It classifies the name of the entity.
A keyword contains only alphabetical characters,	an identifier can consist of alphabetical characters, digits, and underscores.

It should be lowercase.	It can be both upper and lowercase.
No special symbols or punctuations are used in keywords and identifiers.	No special symbols or punctuations are used in keywords and identifiers. The only underscore can be used in an identifier.
Keywords: int, char, while, do.	Identifiers: Geeks_for_Geeks, GFG, Gfg1.

Variables in C++ is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In C++, all the variables must be declared before use.

How to Declare Variables?

A typical variable declaration is of the form:

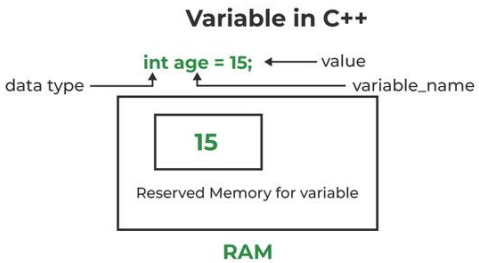
// Declaring a single variable

type variable_name;

// Declaring multiple variables:

type variable1_name, variable2_name, variable3_name;

A variable name can consist of alphabets (both upper and lower case), numbers, and the underscore '_' character. However, the name must not start with a number.



Examples:

```
// Declaring float variable

float simpleInterest;

// Declaring integer variable

int time, speed;

// Declaring character variable

char var;
```

We can also provide values while declaring the variables as given below:

```
int a=50,b=100;  //declaring 2 variable of integer type

float f=50.8;  //declaring 1 variable of float type

char c='Z';    //declaring 1 variable of char type
```

Rules For Declaring Variable

- The name of the variable contains letters, digits, and underscores.
- The name of the variable is case sensitive (ex Arr and arr both are different variables).
- The name of the variable does not contain any whitespace and special characters (ex #,$,%,*, etc).
- All the variable names must begin with a letter of the alphabet or an underscore(_).
- We cannot used C++ keyword (ex float,double,class)as a variable name.

Valid variable names:

```
int x;   //can be letters

int _yz; //can be underscores

int z40;//can be letters
```

Invalid variable names:

```
int 89; Should not be a number

int a b; //Should not contain any whitespace

int double;// C++ keyword CAN NOT BE USED
```

Difference Between Variable Declaration and Definition

The **variable declaration** refers to the part where a variable is first declared or introduced before its first use. A **variable definition** is a part where the variable is assigned a memory

location and a value. Most of the time, variable declaration and definition are done together. See the following C++ program for better clarification:

```
// C++ program to show difference between
// definition and declaration of a
// variable
#include <iostream>
using namespace std;
int main()
{
    // this is declaration of variable a
    int a;
    // this is initialisation of a
    a = 10;
    // this is definition = declaration + initialisation
    int b = 20;
    // declaration and definition
    // of variable 'a123'
    char a123 = 'a';
    // This is also both declaration and definition
    // as 'c' is allocated memory and
    // assigned some garbage value.
    float c;
    // multiple declarations and definitions
    int _c, _d45, e;
    // Let us print a variable
    cout << a123 << endl;
    return 0;
}
```

Types of Variables in C++

1. **Local Variables**: A variable defined within a block or method or constructor is called a local variable.

- These variables are created when entered into the block or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access this variable only within that block.
- Initialization of Local Variable is Mandatory.

2. **Instance Variables**: Instance variables are non-static variables and are declared in a class outside any method, constructor, or block.
   - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
   - Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
   - Initialization of Instance Variable is not mandatory.
   - Instance Variable can be accessed only by creating objects.
3. **Static Variables**: Static variables are also known as Class variables.
   - These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
   - Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
   - Static variables are created at the start of program execution and destroyed automatically when execution ends.
   - Initialization of Static Variable is not Mandatory. Its default value is 0
   - If we access the static variable like the Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name with the class name automatically.
   - If we access the static variable without the class name, the Compiler will automatically append the class name.

Instance Variable Vs Static Variable

- Each object will have its **own copy** of the instance variable whereas We can only have **one copy** of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of the instance variable. In the case of static, changes **will be reflected** in other objects as static variables are common to all objects of a class.

- We can access instance variables **through object references** and Static Variables can be accessed **directly using the class name.**
- The syntax for static and instance variables:

```
class Example
{
   static int a; // static variable
   int b;        // instance variable
}
```

Scope of Variables in C++

In general, the scope is defined as the extent up to which something can be worked with. In programming also the scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes:

1. Local Variables
2. Global Variables

```
#include<iostream>
using namespace std;
// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;

    cout << global << endl;
}
```

Global Variable

Local variable

Local Variables

Variables defined within a function or block are said to be local to those functions.

- Anything between '{' and '}' is said to inside a block.
- Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block.

- **Declaring local variables**: Local variables are declared inside a block.

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;
void func()
{
   // this variable is local to the
   // function func() and cannot be
   // accessed outside this function
   int age=18;
}
int main()
{
   cout<<"Age is: "<<age;
   return 0;
}
```

Output: Error: age was not declared in this scope

The above program displays an error saying “age was not declared in this scope”. The variable age was declared within the function func() so it is local to that function and not visible to portion of program outside this function.

Rectified Program : To correct the above error we have to display the value of variable age from the function func() only. This is shown in the below program:

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;
void func()
{
   // this variable is local to the
```

```
    // function func() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}
int main()
{
    cout<<"Age is: ";
    func();
    return 0;
}
```

Output:

Age is: 18

Global Variables

As the name suggests, Global Variables can be accessed from any part of the program.

- They are available through out the life time of a program.
- They are declared at the top of the program outside all of the functions or blocks.
- **Declaring global variables**: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

```
#include<iostream>
using namespace std;
// global variable
int global = 5;
// global variable accessed from
// within a function
void display()
{
    cout<<global<<endl;
```

```
}
int main()
{
    display();
    // changing value of global
    // variable from main function
    global = 10;
    display();
}
```

What if there exists a local variable with the same name as that of global variable inside a function?

Let us repeat the question once again. The question is : if there is a variable inside a function with the same name as that of a global variable and if the function tries to access the variable with that name, then which variable will be given precedence? Local variable or Global variable? Look at the below program to understand the question:

```
// CPP program to illustrate
// scope of local variables
// and global variables together
#include<iostream>
using namespace std;
// global variable
int global = 5;
// main function
int main()
{
    // local variable with same
    // name as that of global variable
    int global = 2;
    cout << global << endl;
}
```

Look at the above program. The variable “global” declared at the top is global and stores the value 5 where as that declared within main function is local and stores a value 2. So, the question is when the value stored in the variable named “global” is printed from the main function then what will be the output? 2 or 5?

- Usually when two variable with same name are defined then the compiler produces a compile time error. But if the variables are defined in different scopes then the compiler allows it.
- Whenever there is a local variable defined with same name as that of a global variable then the **compiler will give precedence to the local variable**

How to access a global variable when there is a local variable with same name?

What if we want to do the opposite of above task. What if we want to access global variable when there is a local variable with same name?

To solve this problem we will need to use the **scope resolution operator**. Below program explains how to do this with the help of scope resolution operator.

```
// C++ program to show that we can access a global
// variable using scope resolution operator :: when
// there is a local variable with same name
#include<iostream>
using namespace std;
// Global x
int x = 0;
int main()
{
  // Local x
  int x = 10;
  cout << "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
  return 0;
}
```

Output:

Value of global x is 0

Value of local x is 10

C++ Storage Classes

C++ Storage Classes are used to describe the characteristics of a variable/function. It determines the lifetime, visibility, default value, and storage location which helps us to trace the existence of a particular variable during the runtime of a program. Storage class specifiers are used to specify the storage class for a variable.

Syntax

To specify the storage class for a variable, the following syntax is to be followed:

storage_class var_data_type *var_name*;

C++ uses 6 storage classes, which are as follows:

1. auto Storage Class
2. register Storage Class
3. extern Storage Class
4. static Storage Class
5. mutable Storage Class
6. thread_local Storage Class

Below is a detailed explanation of each storage class:

1. auto Storage Class

The **auto storage class** is the default class of all the variables declared inside a block. The auto stands for automatic and all the local variables that are declared in a block automatically belong to this class.

Properties of auto Storage Class Objects

- **Scope:** Local
- **Default Value:** Garbage Value
- **Memory Location:** RAM
- **Lifetime:** Till the end of its scope

Example of auto Storage Class

```
// C++ Program to illustrate the auto storage class
#include <iostream>
using namespace std;
```

```
void autoStorageClass()
{
   cout << "Demonstrating auto class\n";
   // Declaring an auto variable
   int a = 32;
   float b = 3.2;
   char* c = "sk";
   char d = 'G';
   // printing the auto variables
   cout << a << " \n";
   cout << b << " \n";
   cout << c << " \n";
   cout << d << " \n";
}
int main()
{
   // To demonstrate auto Storage Class
   autoStorageClass();
   return 0;
}
```

Note: *Earlier in C++, we could use the* ***auto keyword*** *to declare the auto variables explicitly but after C++11, the meaning of* ***auto*** *keyword is changed and we could no longer use it to define the auto variables.*

2. extern Storage Class

The **extern storage class** simply tells us that the variable is defined elsewhere and not within the same block where it is used (i.e. external linkage). Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. An extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere.

A normal global variable can be made extern as well by placing the **'extern' keyword** before its declaration/definition in any function/block. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

Properties of extern Storage Class Objects

- **Scope:** Global
- **Default Value:** Zero
- **Memory Location:** RAM
- **Lifetime:** Till the end of the program.

Example of extern Storage Class

```
#include <iostream>
using namespace std;
int x;
void externStorageClass()
{
    cout << "Demonstrating extern class\n";
    extern int x;
    // printing the extern variables 'x'
    cout << "Value of the variable 'x'"<< "declared, as extern: " << x << "\n";
    // value of extern variable x modified
    x = 2;
    // printing the modified values of
    // extern variables 'x'
    cout << "Modified value of the variable 'x'"
         << " declared as extern: \n"
         << x;
}
int main()
{
    // To demonstrate extern Storage Class
    externStorageClass();
    return 0;
}
```

Output

Demonstrating extern class

Value of the variable 'x'declared, as extern: 0

Modified value of the variable 'x' declared as extern: 2

3. Static Storage Class

The **static storage class** is used to declare static variables. Static variables have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

We can say that they are initialized only once and exist until the termination of the program. Thus, no new memory is allocated because they are not re-declared. Global static variables can be accessed anywhere in the program.

Properties of static Storage Class

- **Scope:** Local
- **Default Value:** Zero
- **Memory Location:** RAM
- **Lifetime:** Till the end of the program

```
// C++ program to illustrate the static storage class
// objects
#include <iostream>
using namespace std;
// Function containing static variables
// memory is retained during execution
int staticFun()
{
    cout << "For static variables: ";
    static int count = 0;
    count++;
    return count;
}
// Function containing non-static variables
// memory is destroyed
int nonStaticFun()
{
    cout << "For Non-Static variables: ";
    int count = 0;
```

```
    count++;
    return count;
}
int main()
{
    // Calling the static parts
    cout << staticFun() << "\n";
    cout << staticFun() << "\n";
    // Calling the non-static parts
    cout << nonStaticFun() << "\n";
    cout << nonStaticFun() << "\n";
    return 0;
}
```

Output

For static variables: 1

For static variables: 2

For Non-Static variables: 1

For Non-Static variables: 1

4. Register Storage Class

The **register storage class** declares register variables using the '**register' keyword** which has the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only.

An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

Properties of register Storage Class Objects

- **Scope:** Local
- **Default Value:** Garbage Value
- **Memory Location:** Register in CPU or RAM

- **Lifetime:** Till the end of its scope

Example of register Storage Class

```
// C++ Program to illustrate the use of register variables
#include <iostream>
using namespace std;
void registerStorageClass()
{
    cout << "Demonstrating register class\n";
    // declaring a register variable
    register char b = 'G';
    // printing the register variable 'b'
    cout << "Value of the variable 'b'"
         << " declared as register: " << b;
}
int main()
{
    registerStorageClass();
    return 0;
}
```

Output

Demonstrating register class

Value of the variable 'b' declared as register: G

5. Mutable Storage Class

Sometimes there is a requirement to modify one or more data members of class/struct through the const function even though you don't want the function to update other members of class/struct. This task can be easily performed by using the mutable keyword. The keyword mutable is mainly used to allow a particular data member of a const object to be modified. When we declare a function as const, this pointer passed to the function becomes const. Adding a mutable to a variable allows a const pointer to change members.

Properties of mutable Storage Class

The mutable specifier does not affect the linkage or lifetime of the object. It will be the same as the normal object declared in that place.

```
#include <iostream>
using std::cout;
class Test {
public:
   int x;
   mutable int y; // defining mutable variable y now this can be modified

   Test()
   {
      x = 4;
      y = 10;
   }
};
int main()
{
   // t1 is set to constant
   const Test t1;
   // trying to change the value
   t1.y = 20;
   cout << t1.y;
   // Uncommenting below lines
   // will throw error
   // t1.x = 8;
   // cout << t1.x;
   return 0;
}
```

Output: 20

6. Thread_local Storage Class

The thread_local Storage Class is the new storage class that was added in C++11. We can use the **thread_local** storage class specifier to define the object as thread_local. The thread_local variable can be combined with other storage specifiers like static or extern and the properties of the thread_local object changes accordingly.

Properties of thread_local Storage Class

- **Memory Location:** RAM

- **Lifetime:** Till the end of its thread

Operator

An **operator** is a symbol that operates on a value to perform specific mathematical or logical computations. They form the foundation of any programming language. In C++, we have built-in operators to provide the required functionality. An operator operates the **operands**. For example,

int c = a + b;

Here, '+' is the addition operator. 'a' and 'b' are the operands that are being 'added'.

Operators in C++ can be classified into 6 types:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Ternary or Conditional Operators

1) Arithmetic Operators

These operators are used to perform arithmetic or mathematical operations on the operands. For example, '+' is used for addition, '-' is used for subtraction '*' is used for multiplication, etc.

Arithmetic Operators can be classified into 2 Types:

A) Unary Operators: These operators operate or work with a single operand. For example: Increment(++) and Decrement(--) Operators.

Name	Symbol	Description	Example
Increment Operator	++	Increases the integer value of the variable by one	int a = 5; a++; // returns 6
Decrement Operator	--	Decreases the integer value of the variable by one	int a = 5; a--; // returns 4

Note: *++**a** and **a**++, both are increment operators, however, both are slightly different. In ++**a**, the value of the variable is incremented first and then It is used in the program. In **a**++, the value of the variable is assigned first and then It is incremented. Similarly happens for the decrement operator.*

B) Binary Operators: These operators operate or work with two operands. For example: Addition(+), Subtraction(-), etc.

Name	Symbol	Description	Example
Addition	+	Adds two operands	int a = 3, b = 6; int c = a+b; // c = 9
Subtraction	–	Subtracts second operand from the first	int a = 9, b = 6; int c = a-b; // c = 3
Multiplication	*	Multiplies two operands	int a = 3, b = 6; int c = a*b; // c = 18
Division	/	Divides first operand by the second operand	int a = 12, b = 6; int c = a/b; // c = 2
Modulo Operation	%	Returns the remainder an integer division	int a = 8, b = 6; int c = a%b; // c = 2

Note: *The Modulo operator(%) operator should only be used with integers.*

Example:

```
// CPP Program to demonstrate the Binary Operators
#include <iostream>
using namespace std;
int main()
{
    int a = 8, b = 3;

    cout << "a + b = " << (a + b) << endl;
                                                    // Addition operator

    cout << "a - b = " << (a - b) << endl;
                                                    // Subtraction operator

                                                    // Multiplication operator
```

```
    cout << "a * b = " << (a * b) << endl;
                                                    // Division operator
    cout << "a / b = " << (a / b) << endl;
                                                    // Modulo operator
    cout << "a % b = " << (a % b) << endl;
    return 0;
}
```

Output

a + b = 11
a - b = 5
a * b = 24
a / b = 2
a % b = 2

2) Relational Operators

These operators are used for the comparison of the values of two operands. For example, '>' checks if one operand is greater than the other operand or not, etc. The result returns a Boolean value, i.e., **true** or **false.**

Name	Symbol	Description	Example

Is Equal To	==	Checks if both operands are equal	int a = 3, b = 6;
			a==b;
			// returns false

Greater Than	>	Checks if first operand is greater than the second operand	int a = 3, b = 6;
			a>b;
			// returns false

Greater Than or Equal To	>=	Checks if first operand is greater than or equal to the second operand	int a = 3, b = 6;
			a>=b;
			// returns false

Less Than	<	Checks if first operand is lesser than the second operand	int a = 3, b = 6;
			a<b;
			// returns true

Less Than or Equal To	<=	Checks if first operand is lesser than or equal to the second operand	int a = 3, b = 6;
			a<=b;
			// returns true

Not Equal To	!=	Checks if both operands are not equal	int a = 3, b = 6; a!=b;

Name	Symbol	Description	Example
			// returns true

Example:

```
#include <iostream>
using namespace std;
 int main()
{
    int a = 6, b = 4;
                                                    // Equal to operator
    cout << "a == b is " << (a == b) << endl;
                                                   // Greater than operator
    cout << "a > b is " << (a > b) << endl;
                                                  // Greater than or Equal to operator
    cout << "a >= b is " << (a >= b) << endl;
                                                    //  Lesser than operator
    cout << "a < b is " << (a < b) << endl;
                                                     // Lesser than or Equal to operator
    cout << "a <= b is " << (a <= b) << endl;
                                                      // true
    cout << "a != b is " << (a != b) << endl;
    return 0; }
```

Output

```
a == b is 0
a > b is 1
a >= b is 1
a < b is 0
a <= b is 0
a != b is 1
```

3) Logical Operators

These operators are used to combine two or more conditions or constraints or to complement the evaluation of the original condition in consideration. The result returns a Boolean value, i.e., **true** or **false**.

Name	Symbol	Description	Example

Logical AND	&&	Returns true only if all the operands are true or non-zero	int a = 3, b = 6;
			a&&b;
			// returns true

Logical OR	||	Returns true if either of the operands is true or non-zero	int a = 3, b = 6;
			a||b;
			// returns true

Logical NOT	!	Returns true if the operand is false or zero	int a = 3;
			!a;
			// returns false

Example:

```
#include <iostream>
using namespace std;
int main()
{
   int a = 6, b = 4;
                                                    // Logical AND operator
   cout << "a && b is " << (a && b) << endl;
                                                   // Logical OR operator
   cout << "a ||b is " << (a || b) << endl;
                                                  // Logical NOT operator
   cout << "!b is " << (!b) << endl;
   return 0;
}
```

Output

```
a && b is 1
a || b is 1
!b is 0
```

4) Bitwise Operators

These operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

Name	Symbol	Description	Example
Binary AND	&	Copies a bit to the evaluated result if it exists in both operands	int a = 2, b = 3; (a & b); //returns 2
Binary OR	|	Copies a bit to the evaluated result if it exists in any of the operand	int a = 2, b = 3; (a | b); //returns 3
Binary XOR	^	Copies the bit to the evaluated result if it is present in either of the operands but not both	int a = 2, b = 3; (a ^ b); //returns 1
Left Shift	<<	Shifts the value to left by the number of bits specified by the right operand.	int a = 2, b = 3; (a << 1); //returns 4
Right Shift	>>	Shifts the value to right by the number of bits specified by the right operand.	int a = 2, b = 3; (a >> 1); //returns 1
One's Complement	~	Changes binary digits 1 to 0 and 0 to 1	int b = 3; (~b);

Note: *Only* ***char*** *and* ***int*** *data types can be used with Bitwise Operators.*

Example:

```
  #include <iostream>
  using namespace std;
int main()
{
   int a = 6, b = 4;
                                                  // Binary AND operator
   cout << "a & b is " << (a & b) << endl;
                                                  // Binary OR operator
   cout << "a | b is " << (a | b) << endl;
                                                 // Binary XOR operator
   cout << "a ^ b is " << (a ^ b) << endl;
                                                  // Left Shift operator
   cout << "a<<1 is " << (a << 1) << endl;
                                                 // Right Shift operator
   cout << "a>>1 is " << (a >> 1) << endl;
                                                // One's Complement operator
   cout << "~(a) is " << ~(a) << endl;
   return 0;
}
```

Output

```
a & b is 4
a | b is 6
a ^ b is 2
a<<1 is 12
a>>1 is 3
```

5) Assignment Operators

These operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

Namemultiply	Symbol	Description	Example
Assignment Operator	=	Assigns the value on the right to the variable on the left	int a = 2; // a = 2
Add and Assignment Operator	+=	First adds the current value of the variable on left to the value on	int a = 2, b = 4; a+=b; // a = 6

Namemultiply	Symbol	Description	Example
		the right and then assigns the result to the variable on the left	
Subtract and Assignment Operator	- =	First subtracts the value on the right from the current value of the variable on left and then assign the result to the variable on the left	int a = 2, b = 4; a-=b; // a = -2
Multiply and Assignment Operator	*=	First multiplies the current value of the variable on left to the value on the right and then assign the result to the variable on the left	int a = 2, b = 4; a*=b; // a = 8
Divide and Assignment Operator	/=	First divides the current value of the variable on left by the value on the right and then assign the result to the variable on the left	int a = 4, b = 2; a /=b; // a = 2

Example:

```
#include <iostream>
using namespace std;
 int main()
{
    int a = 6, b = 4;
    // Assignment Operator
    cout << "a = " << a << endl;
    //  Add and Assignment Operator
    cout << "a += b is " << (a += b) << endl;
    // Subtract and Assignment Operator
    cout << "a -= b is " << (a -= b) << endl;
    //  Multiply and Assignment Operator
    cout << "a *= b is " << (a *= b) << endl;
    //  Divide and Assignment Operator
    cout << "a /= b is " << (a /= b) << endl;
    return 0;
```

}

Output

a = 6

a += b is 10

a -= b is 6

a *= b is 24

a /= b is 6

6) Ternary or Conditional Operators (?:)

This operator returns the value based on the condition.

Expression1? Expression2: Expression3

The ternary operator determines the answer on the basis of the evaluation of **Expression1**. If it is **true**, then **Expression2** gets evaluated and is used as the answer for the expression. If **Expression1** is **false**, then **Expression3** gets evaluated and is used as the answer for the expression. This operator takes **three operands**, therefore it is known as a Ternary Operator.

Example:

```
// CPP Program to demonstrate the Conditional Operators
#include <iostream>
using namespace std;
int main()
{
    int a = 3, b = 4;
    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number is " << result << endl;
    return 0;
}
```

7) There are some other common operators available in C++ besides the operators discussed above. Following is a list of these operators discussed in detail:

A) sizeof Operator: This unary operator is used to compute the size of its operand or variable.

```
sizeof(char); // returns 1
```

B) Comma Operator(,): This binary operator (represented by the token) is used to evaluate its first operand and discards the result, it then evaluates the second operand and returns this value (and type). It is used to combine various expressions together.

```
int a = 6;

int b = (a+1, a-2, a+5);        // b = 11
```

C) -> Operator: This operator is used to access the variables of classes or structures.

```
cout<<emp->first_name;
```

D) Cast Operator: This unary operator is used to convert one data type into another.

```
float a = 11.567;

int c = (int) a; // returns 11
```

E) Dot Operator(.): This operator is used to access members of structure variables or class objects in C++.

```
cout<<emp.first_name;
```

F) & Operator: This is a pointer operator and is used to represent the memory address of an operand.

G) * Operator: This is an Indirection Operator

```
#include <iostream>
using namespace std;
int main()
{
   int a = 6;
   int* b;
   int c;
   b = &a;                    // & Operator
   c = *b;                    // * Operator
   cout << " a = " << a << endl;
   cout << " b = " << b << endl;
   cout << " c = " << c << endl;
   return 0;
}
```

Output

```
 a = 6
 b = 0x7ffe8e8681bc
 c = 6
```

H) << Operator: It is called the insertion operator. It is used with cout to print the output.

I) >> Operator: It is called the extraction operator. It is used with cin to get the input.

int a;

cin>>a;

cout<<a;

Operator Precedence Chart

Precedence	Operator	Description	Associativity
1.	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	
	.	Member selection via object name	
	->	Member selection via a pointer	
	++/–	Postfix increment/decrement	
2.	++/–	Prefix increment/decrement	right-to-left
	+/-	Unary plus/minus	
	!~	Logical negation/bitwise complement	
	(type)	Cast (convert value to temporary value of type)	
	*	Dereference	
	&	Address (of operand)	
	sizeof	Determine size in bytes on this implementation	
3.	*,/,%	Multiplication/division/modulus	left-to-right
4.	+/-	Addition/subtraction	left-to-right

Precedence	Operator	Description	Associativity
5.	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
6.	< , <=	Relational less than/less than or equal to	left-to-right
	> , >=	Relational greater than/greater than or equal to	left-to-right
7.	== , !=	Relational is equal to/is not equal to	left-to-right
8.	&	Bitwise AND	left-to-right
9.	^	Bitwise exclusive OR	left-to-right
10.	|	Bitwise inclusive OR	left-to-right
11.	&&	Logical AND	left-to-right
12.	||	Logical OR	left-to-right
13.	?:	Ternary conditional	right-to-left
14.	=	Assignment	right-to-left
	+= , -=	Addition/subtraction assignment	
	*= , /=	Multiplication/division assignment	
	%= , &=	Modulus/bitwise AND assignment	
	^= , |=	Bitwise exclusive/inclusive OR assignment	
	<>=	Bitwise shift left/right assignment	
15.	,	expression separator	left-to-right

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages. To perform any task, we can create function. A function can be called many times. It provides modularity and code reusability.

Advantage of functions in C: There are many advantages of functions.

1) Code Reusability: By creating functions in C++, you can call it many times. So we don't need to write the same code again and again.

2) Code optimization: It makes the code optimized, we don't need to write much code. Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times. So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions: There are two types of functions in C programming:

1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.

Declaration of a function: The syntax of creating function in C++ language is given below:

```
return_type function_name (data_type parameter...)
{
//code to be executed
}
```

C++ Function Example

```
#include <iostream>
using namespace std;
void func() {
  static int i=0; //static variable
  int j=0; //local variable
```

```
    i++;
    j++;
    cout<<"i=" << i<<" and j=" <<j<<endl;
}
int main()
{
 func();
 func();
 func();
}
```

OUTPUT: i= 1 and j= 1
i= 2 and j= 1
i= 3 and j= 1

Call by value and call by reference in C++

There are two ways to pass value or data to function in C++ language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

```
#include <iostream>
using namespace std;
void change(int data);
int main()
{
int data = 3;
change(data);
```

```
cout << "Value of the data is: " << data<< endl;
return 0;
}
void change(int data)
{
data = 5;
}
```

Output: Value of the data is: 3

Call by reference in C++

In call by reference, original value is modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

```
#include<iostream>
using namespace std;
void swap(int *x, int *y)
{
 int swap;
 swap=*x;
 *x=*y;
 *y=swap;
}
int main()
{
 int x=500, y=100;
 swap(&x, &y);  // passing value to function
 cout<<"Value of x is: "<<x<<endl;
 cout<<"Value of y is: "<<y<<endl;
 return 0;
}
```

Output: Value of x is: 100, Value of y is: 500

Difference between call by value and call by reference in C++

No.	Call by value	Call by reference
1	A copy of value is passed to the function	An address of value is passed to the function
2	Changes made inside the function is not reflected on other functions	Changes made inside the function is reflected outside the function also
3	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in same memory location

Inline Functions

C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

Syntax:
inline return-type function-name(parameters)

{

// function code

}

```
#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;
}
```

Output: The cube of 3 is: 27
Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

The compiler may not perform inlining in such circumstances as:

1. If a function contains a loop. (*for, while and do-while*)
2. If a function contains static variables.

3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn't exist in a function body.
5. If a function contains a switch or goto statement.

Why Inline Functions are Used?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

Inline functions Advantages:

1. Function call overhead doesn't occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.
5. An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.

Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation: In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures: Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
∗ (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer: The pointer in C++ language can be declared using ∗ (asterisk symbol).

- **int** ∗ a; //pointer to int
- **char** ∗ c; //pointer to char

Pointer Example: Let's see the simple example of using pointers printing the address and value.

- #include <iostream>
- **using namespace** std;
- **int** main()
- {
- **int** number=30;
- **int** ∗ p;
- p=&number;//stores the address of number variable
- cout<<"Address of number variable is:"<<&number<<endl;
- cout<<"Address of p variable is:"<<p<<endl;
- cout<<"Value of p variable is:"<<*p<<endl;
- **return** 0;
- }

Output:

Address of number variable is: 0x7ffccc8724c4

Address of p variable is: 0x7ffccc8724c4

Value of p variable is: 30

Pointer Program to swap 2 numbers without using 3rd variable

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
  return 0;
}
```

Output:

Before swap: *p1=20 *p2=10

After swap: *p1=10 *p2=20

sizeof() operator in C++

The sizeof() is an operator that evaluates the size of data type, constants, variable. It is a compile-time operator as it returns the size of any variable or a constant at the compilation time. The size, which is calculated by the sizeof() operator, is the amount of RAM occupied in the computer.

Syntax of the sizeof() operator is given below:

sizeof(data_type);

In the above syntax, the data_type can be the data type of the data, variables, constants, unions, structures, or any other user-defined data type.

The sizeof () operator can be applied to the following operand types:

When an operand is of data type

If the parameter of a **sizeof()** operator contains the data type of a variable, then the **sizeof()** operator will return the size of the data type.

Let's understand this scenario through an example.

```
#include <iostream>
using namespace std;
int main()
{
  // Determining the space in bytes occupied by each data type.
  std::cout << "Size of integer data type : " <<sizeof(int)<< std::endl;
  std::cout << "Size of float data type : " <<sizeof(float)<< std::endl;
  std::cout << "Size of double data type : " <<sizeof(double)<< std::endl;
  std::cout << "Size of char data type : " <<sizeof(char)<< std::endl;
  return 0;
}
```

C++ References

Till now, we have read that C++ supports two types of variables:

- An ordinary variable is a variable that contains the value of some type. For example, we create a variable of type int, which means that the variable can hold the value of type integer.
- A pointer is a variable that stores the address of another variable. It can be dereferenced to retrieve the value to which this pointer points to.
- There is another variable that C++ supports, i.e., references. It is a variable that behaves as an alias for another variable.

How to create a reference?

Reference can be created by simply using an ampersand (&) operator. When we create a variable, then it occupies some memory location. We can create a reference of the variable; therefore, we can access the original variable by using either name of the variable or reference. For example,

```
int a=10;
```

Now, we create the reference variable of the above variable.

```
int &ref=a;
```

The above statement means that 'ref' is a reference variable of 'a', i.e., we can use the 'ref' variable in place of 'a' variable.

C++ provides two types of references:

- References to non-const values
- References as aliases

References to non-const values

It can be declared by using & operator with the reference type variable.

```
#include <iostream>
using namespace std;
int main()
{
int a=10;
int &value=a;
std::cout << value << std::endl;
return 0;
}
```

Output: 10

References as aliases: References as aliases is another name of the variable which is being referenced.

```
#include <iostream>
using namespace std;
int main()
{
int a=70; // variable initialization
int &b=a;
int &c=a;
std::cout << "Value of a is :" <<a<< std::endl;
std::cout << "Value of b is :" <<b<< std::endl;
std::cout << "Value of c is :" <<c<< std::endl;
```

- **return** 0;}

In the above code, we create a variable 'a' which contains a value '70'. We have declared two reference variables, i.e., b and c, and both are referring to the same variable 'a'. Therefore, we can say that 'a' variable can be accessed by 'b' and 'c' variable.

Output

```
Value of a is :70
Value of b is :70
Value of c is :70
```

Reassignment: It cannot be reassigned means that the reference variable cannot be modified.

- #include <iostream>
- **using namespace** std;
- **int** main()
- {
- **int** x=11; // variable initialization
- **int** z=67;
- **int** &y=x; // y reference to x
- **int** &y=z; // y reference to z, but throws a compile-time error.
- **return** 0;}

In the above code, 'y' reference variable is referring to 'x' variable, and then 'z' is assigned to 'y'. But this reassignment is not possible with the reference variable, so it throws a compile-time error.

Memory Management

Memory management is a process of managing computer memory, assigning the memory space to the programs to improve the overall system performance.

Memory Management Operators

In C language, we use the **malloc()** or **calloc()** functions to allocate the memory dynamically at run time, and free() function is used to deallocate the dynamically allocated memory.

C++ also supports these functions, but C++ also defines unary operators such as **new** and **delete** to perform the same tasks, i.e., allocating and freeing the memory.

New operator

A **new** operator is used to create the object while a **delete** operator is used to delete the object. When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object. Therefore, we can say that the lifetime of the object is not related to the block structure of the program.

Syntax: pointer_variable = **new** data-type

The above syntax is used to create the object using the new operator. In the above syntax, **'pointer_variable'** is the name of the pointer variable, **'new'** is the operator, and **'data-type'** defines the type of the data.

Example 1:

- **int** *p;
- p = **new int**;

In the above example, 'p' is a pointer of type int.

In the above case, the declaration of pointers and their assignments are done separately. We can also combine these two statements as follows:

- **int** *p = **new int**;
- **float** *q = **new float**;

Two ways of assigning values to the newly created object:

(1). we can assign the value to the newly created object by simply using the assignment operator. In the above case, we have created two pointers 'p' and 'q' of type int and float, respectively. Now, we assign the values as follows:

- *p = 45;
- *q = 9.8;

We assign 45 to the newly created int object and 9.8 to the newly created float object.

(2). we can also assign the values by using new operator which can be done as follows:

pointer_variable = **new** data-type(value);

- **int** *p = **new int**(45);
- **float** *p = **new float**(9.8);

How to create a single dimensional array

As we know that new operator is used to create memory space for any data-type or even user-defined data type such as an array, structures, unions, etc., so the syntax for creating a one-dimensional array is given below:

Pointer-variable = **new** data-type [size];

Examples:

int *a1 = **new int**[8];

In the above statement, we have created an array of type int having a size equal to 8 where p[0] refers first element, p[1] refers the first element, and so on.

Delete operator

When memory is no longer required, then it needs to be deallocated so that the memory can be used for another purpose. This can be achieved by using the delete operator, as shown below:

delete pointer_variable;

In the above statement, '**delete**' is the operator used to delete the existing object, and '**pointer_variable**' is the name of the pointer variable.

- #include <iostream>
- **using namespace** std
- **int** main()
- {
- **int** size; // variable declaration
- **int** *arr = **new int**[size];   // creating an array
- cout<<"Enter the size of the array : ";
- std::cin >> size;   //

```
cout<<"\nEnter the element : ";
for(int i=0;i<size;i++)   // for loop
{
cin>>arr[i];
}
cout<<"\nThe elements that you have entered are :";
for(int i=0;i<size;i++)    // for loop
{
cout<<arr[i]<<",";
}
delete arr;  // deleting an existing array.
return 0;
}
```

In the above code, we have created an array using the new operator. The above program will take the user input for the size of an array at the run time. When the program completes all the operations, then it deletes the object by using the statement **delete arr**.

UNIT-2

C++ OOPs Concepts: The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language. Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.** The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical. In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality. Object is a runtime entity, it is created at runtime. Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1;  //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

Class

Collection of objects is called class. It is a logical entity.

A Class in C++ is the foundational element that leads to Object-Oriented programming. A class instance must be created in order to access and use the user-defined data type's data members and member functions. An object's class acts as its blueprint. Take the class of cars as an example. Even if different names and brands may be used for different cars, all of them will have some characteristics in common, such as four wheels, a speed limit, a range of miles, etc. In this case, the class of car is represented by the wheels, the speed limitations, and the mileage.

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

An example of C++ class that has three fields only.

```
class Student
 {
    public:
    int id;  //field or data member
    float salary; //field or data member
    String name;//field or data member
Int getdata()        // data member function
{ ----------; function body }
Int putdata()       // data member function
{  ---------;function body }
 }
```

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

1. Super class - The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.

2. Sub class - Subclass or Derived Class refers to a class that receives properties from another class.
3. Reusability - As a result, when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class thanks to inheritance. This allows us to utilize the fields and methods of the pre-existing class.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

Different situations may cause an operation to behave differently. The type of data utilized in the operation determines the behavior.

Abstraction

Hiding internal details and showing functionality is known as abstraction. Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information.For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Encapsulation is typically understood as the grouping of related pieces of information and data into a single entity. Encapsulation is the process of tying together data and the functions that work with it in object-oriented programming.

Take a look at a practical illustration of encapsulation: at a company, there are various divisions, including the sales division, the finance division, and the accounts division. All financial transactions are handled by the finance sector, which also maintains records of all financial data. In a similar vein, the sales section is in charge of all tasks relating to sales and

maintains a record of each sale. Now, a scenario could occur when, for some reason, a financial official requires all the information on sales for a specific month. Under the umbrella term "sales section," all of the employees who can influence the sales section's data are grouped together. Data abstraction or concealing is another side effect of encapsulation. In the same way that encapsulation hides the data. In the aforementioned example, any other area cannot access any of the data from any of the sections, such as sales, finance, or accounts.

Dynamic Binding - In dynamic binding, a decision is made at runtime regarding the code that will be run in response to a function call. For this, C++ supports virtual functions.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

Why is C++ a partial oop?

The object-oriented features of the C language were the primary motivation behind the construction of the C++ language.

The C++ programming language is categorized as a partial object-oriented programming language despite the fact that it supports OOP concepts, including classes, objects, inheritance, encapsulation, abstraction, and polymorphism.

1) The main function must always be outside the class in C++ and is required. This means that we may do without classes and objects and have a single main function in the application.

It is expressed as an object in this case, which is the first time Pure OOP has been violated.

2) Global variables are a feature of the C++ programming language that can be accessed by any other object within the program and are defined outside of it. Encapsulation is broken here.

Even though C++ encourages encapsulation for classes and objects, it ignores it for global variables.

Object and Class Example

Let's see an example of class that has two fields: id and name. It creates instance of the class, initializes the object and prints the object value.

```
#include <iostream>
using namespace std;
class Student {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
};
int main() {
   Student s1; //creating an object of Student
   s1.id = 201;
   s1.name = "Hari";
   cout<<s1.id<<endl;
   cout<<s1.name<<endl;
   return 0;
}
```

Output: 201 Hari

C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
#include <iostream>
using namespace std;
class Student {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
```

```
        void insert(int i, string n)
        {
            id = i;
            name = n;
        }
        void display()
        {
            cout<<id<<"  "<<name<<endl;
        }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

Output:
201 Sonoo
202 Nakul

Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
#include <iostream>
using namespace std;
class Employee {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    float salary;
```

```
    void insert(int i, string n, float s)
    {
        id = i;
        name = n;
        salary = s;
    }
    void display()
    {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
    }
};
int main(void) {
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    e1.insert(201, "Sonoo",990000);
    e2.insert(202, "Nakul", 29000);
    e1.display();
    e2.display();
    return 0;
}
```

Output:

201 Sonoo 990000

202 Nakul 29000

Access Specifiers:

In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes.

In the following example, we demonstrate the differences between public and private members:

Example

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

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

If you try to access a private member, an error occurs: error: y is private

Note: It is possible to access private members of a class using a public method inside the same class.

By default, all members of a class are private if you don't specify an access specifier:

Encapsulation

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

Access Private Members: To access a private attribute, use public "get" and "set" methods:

Example

```
#include <iostream>
using namespace std;
class Employee {
  private:                    // Private attribute
    int salary;
  public:
                                  // Setter
    void setSalary(int s) {
      salary = s;
    }
                              // Getter
    int getSalary() {
      return salary;
    }
};
```

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

Example explained

The salary attribute is private, which have restricted access.

The public setSalary() method takes a parameter (s) and assigns it to the salary attribute (salary = s).

The public getSalary() method returns the value of the private salary attribute.

Inside main(), we create an object of the Employee class. Now we can use the setSalary() method to set the value of the private attribute to 50000. Then we call the getSalary() method on the object to return the value.

Member function: A **member function** of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

```
class Box {
   public:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
      double getVolume(void) {
         return length * breadth * height;
      }
};
```

If you like, you can define the same function outside the class using the **scope resolution operator** (::) as follows −

```
double Box::getVolume(void) {
   return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (.) on a object where it will manipulate data related to that object only as follows −

```
Box myBox;                // Create an object
myBox.getVolume();   // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class −

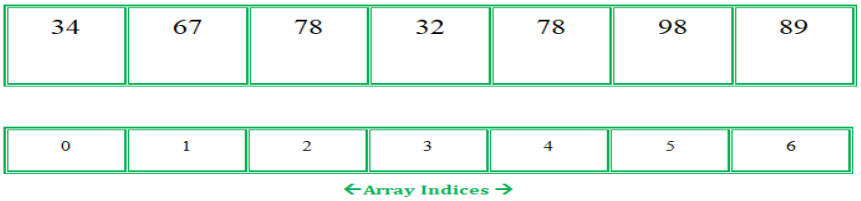
```
#include <iostream>
using namespace std;
class Box {
  public:
    double length;        // Length of a box
    double breadth;       // Breadth of a box
    double height;        // Height of a box
                                  // Member functions declaration
    double getVolume(void);
    void setLength( double l);
    void setBreadth( double b);
    void setHeight( double h);
};

// Member functions definitions
double Box::getVolume(void) {
  return length * breadth * height;
}
void Box::setLength( double l) {
  length = l;
}
void Box::setBreadth( double b) {
  breadth = b;
}
void Box::setHeight( double h) {
  height = h;
}
// Main function for the program
int main() {
  Box Box1;              // Declare Box1 of type Box
  Box Box2;              // Declare Box2 of type Box
  double volume = 0.0;     // Store the volume of a box here
                          // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);
                               // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);
                                        // volume of box 1
  volume = Box1.getVolume();
  cout << "Volume of Box1 : " << volume <<endl;
                                         // volume of box 2
  volume = Box2.getVolume();
  cout << "Volume of Box2 : " << volume <<endl;
  return 0;
}
```

When the above code is compiled and executed, it produces the following result −
Volume of Box1 : 210
Volume of Box2 : 1560

Array of Objects in C++ with Examples

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store the collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as structures, pointers, etc. Given below is the picture representation of an array.

Example: Let's consider an example of taking random integers from the user.



Array of Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax: ClassName ObjectName[number of objects];

The Array of Objects stores *objects*. An array of a class type is also known as an array of objects.

Example#1:

Storing more than one Employee data. Let's assume there is an array of objects for storing employee data emp[50].

Below is the C++ program for storing data of one Employee:

```
// C++ program to implement
#include<iostream>
using namespace std;
class Employee
{
  int id;
  char name[30];
  public:
  void getdata();//Declaration of function
  void putdata();//Declaration of function
};
void Employee::getdata(){//Defining of function
  cout<<"Enter Id : ";
  cin>>id;
  cout<<"Enter Name : ";
  cin>>name;
}
void Employee::putdata(){//Defining of function
```

```
  cout<<id<<" ";
  cout<<name<<" ";
  cout<<endl;
}
int main(){
  Employee emp; //One member
  emp.getdata();//Accessing the function
  emp.putdata();//Accessing the function
  return 0;
}
```

Let's understand the above example –

- In the above example, a class named Employee with id and name is being considered.
- The two functions are declared-
    - **getdata():** Taking user input for id and name.
    - **putdata():** Showing the data on the console screen.

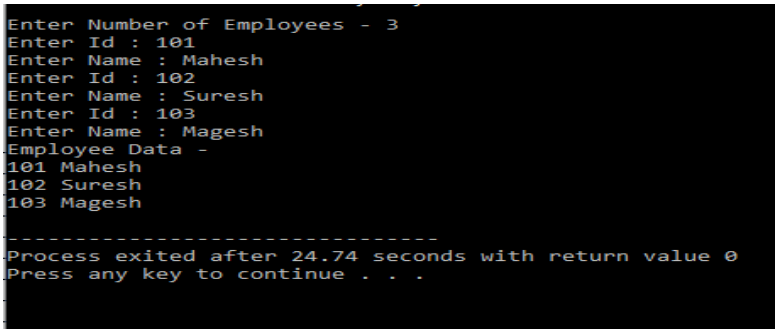
This program can take the data of only one Employee. What if there is a requirement to add data of more than one Employee. Here comes the answer Array of Objects. An array of objects can be used if there is a need to store data of more than one employee. Below is the C++ program to implement the above approach.

// C++ program to implement

```
#include<iostream>

using namespace std;

class Employee

{

  int id;

  char name[30];

  public:

  // Declaration of function

  void getdata();

  // Declaration of function

  void putdata();

};

// Defining the function outside

// the class

void Employee::getdata()

{

  cout << "Enter Id : ";

  cin >> id;
```

```
  cout << "Enter Name : ";
  cin >> name;
}
// Defining the function outside
// the class
void Employee::putdata()
{
  cout << id << " ";
  cout << name << " ";
  cout << endl;
}
int main()
{
  // This is an array of objects having
  // maximum limit of 30 Employees
  Employee emp[30];
  int n, i;
  cout << "Enter Number of Employees - ";
  cin >> n;
  // Accessing the function
  for(i = 0; i < n; i++)
    emp[i].getdata();
  cout << "Employee Data - " << endl;
  // Accessing the function
  for(i = 0; i < n; i++)
    emp[i].putdata();
}
```

Output:



Explanation:

In this example, more than one Employee's details with an Employee id and name can be stored.

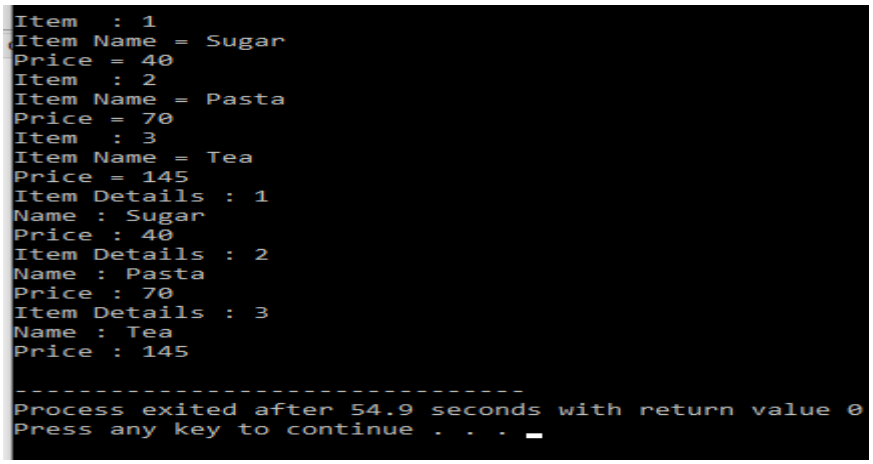
- Employee emp[30] – This is an array of objects having a maximum limit of 30 Employees.
- Two for loops are being used-
    - First one to take the input from user by calling emp[i].getdata() function.
    - Second one to print the data of Employee by calling the function emp[i]. putdata() function.

Example#2:

```
// C++ program to implement
// the above approach
#include<iostream>
using namespace std;
class item
{
  char name[30];
  int price;
  public:
  void getitem();
  void printitem();
};
// Function to get item details
void item::getitem()
{
  cout << "Item Name = ";
  cin >> name;
  cout << "Price = ";
  cin >> price;
}
void item ::printitem()                    // Function to print item
{
  cout << "Name : " << name <<
          "\n";
  cout << "Price : " << price <<
          "\n";
}
 const int size = 3;
 int main()
{
  item t[size];
```

```
  for(int i = 0; i < size; i++)
  {
    cout << "Item  : " <<
          (i + 1) << "\n";
    t[i].getitem();
  }
  for(int i = 0; i < size; i++)
  {
    cout << "Item Details : " <<
           (i + 1) << "\n";
    t[i].printitem();
  }
}
```

Output:



C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

In brief, A particular procedure called a constructor is called automatically when an object is created in C++. In general, it is employed to create the data members of new things. In C++, the class or structure name also serves as the constructor name. When an object is completed, the constructor is called. Because it creates the values or gives data for the thing, it is known as a constructor.

The Constructors prototype looks like this:
<**class**-name> (list-of-parameters);

The following syntax is used to define the class's constructor:

<class-name> (list-of-parameters) { // constructor definition }

The following syntax is used to define a constructor outside of a class:

<class-name>: :<class-name> (list-of-parameters){ // constructor definition}

Constructors lack a return type since they don't have a return value.
There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Program:

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    return 0;
}
```

Output: Default Constructor Invoked

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

```
#include <iostream>
using namespace std;
class Employee {
  public:
    int id;//data member (also instance variable)
    string name;//data member(also instance variable)
    float salary;
    Employee(int i, string n, float s)
     {
        id = i;
        name = n;
        salary = s;
     }
    void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000);
   e1.display();
   e2.display();
   return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

What distinguishes constructors from a typical member function?

1. Constructor's name is the same as the class's
2. Default There isn't an input argument for constructors. However, input arguments are available for copy and parameterized constructors.

3. There is no return type for constructors.
4. An object's constructor is invoked automatically upon creation.
5. The C++ compiler creates a default constructor for the object if a constructor is not specified (expects any parameters and has an empty body).

What are the characteristics of a constructor?

1. The constructor has the same name as the class it belongs to.
2. Although it is possible, constructors are typically declared in the class's public section. However, this is not a must.
3. Because constructors don't return values, they lack a return type.
4. When we create a class object, the constructor is immediately invoked.
5. Overloaded constructors are possible.
6. Declaring a constructor virtual is not permitted.
7. One cannot inherit a constructor.
8. Constructor addresses cannot be referenced to.
9. When allocating memory, the constructor makes implicit calls to the new and delete operators.

Copy constructor

A member function known as a copy constructor initializes an item using another object from the same class-an in-depth discussion on Copy Constructors.

Every time we specify one or more non-default constructors (with parameters) for a class, we also need to include a default constructor (without parameters), as the compiler won't supply one in this circumstance. The best practice is to always declare a default constructor, even though it is not required.

A reference to an object belonging to the same class is required by the copy constructor.

- Sample(Sample &t)
- {
- id=t.id;
- }

What is a destructor in C++?

An equivalent special member function to a constructor is a destructor. The constructor creates class objects, which are destroyed by the destructor. The word "destructor," followed by the tilde () symbol, is the same as the class name. You can only define one destructor at a time. One method of destroying an object made by a constructor is to use a destructor. Destructors cannot be overloaded as a result. Destructors don't take any arguments and don't give anything back. As soon as the item leaves the scope, it is immediately called. Destructors free up the memory used by the objects the constructor generated. Destructor reverses the process of creating things by destroying them.

The language used to define the class's destructor

- ~ <**class**-name>()
- {
- }

The language used to define the class's destructor outside of it

<**class**-name>: : ~ <**class**-name>(){}

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

> ***Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.***

C++ Constructor and Destructor Example

An example of constructor and destructor in C++ which is called automatically.

- #include <iostream>
- **using namespace** std;
- **class** Employee

```
{
  public:
     Employee()
     {
        cout<<"Constructor Invoked"<<endl;
     }
     ~Employee()
     {
        cout<<"Destructor Invoked"<<endl;
     }
};
int main(void)
{
   Employee e1; //creating an object of Employee
   return 0;
}
```

Output:

```
Constructor Invoked
Destructor Invoked
```

This Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
```

```
        Employee(int id, string name, float salary)
        {
            this->id = id;
           this->name = name;
           this->salary = salary;
        }
       void display()
        {
           cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee

    e1.display();
    return 0;
}
```

Output: 101 Sonoo 890000

C++ static

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

static field example

```
#include <iostream>
using namespace std;
class Account {
  public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
     {
         this->accno = accno;
        this->name = name;
     }
    void display()
     {
        cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
     }
};
float Account::rateOfInterest=6.5;
int main(void) {
   Account a1 =Account(201, "Sanjay"); //creating an object of Employee
   a1.display();
   return 0;
}
```

Output: 201 Sanjay 6.5

Static field example: Counting Objects

Another example of static keyword in C++ which counts the objects.

```
#include <iostream>
using namespace std;
class Account {
```

```
    public:
        int accno; //data member (also instance variable)
        string name;
        static int count;
        Account(int accno, string name)
        {
            this->accno = accno;
           this->name = name;
            count++;
        }
        void display()
        {
            cout<<accno<<" "<<name<<endl;
        }
};
int Account::count=0;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object of Account
    Account a2=Account(202, "Nakul");
     a1.display();
    a2.display();
    cout<<"Total Objects are: "<<Account::count;
    return 0;
}
```

Output:

201 Sanjay

202 Nakul

Total Objects are: 3

C++ Structs

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.
Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

The Syntax of Structure

```
struct structure_name
{
    // member declarations.
}
```

In the above declaration, a structure is declared by preceding the **struct keyword** followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. **Consider the following situation:**

```
struct Student
{
   char name[20];
    int id;
    int age;
}
```

In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

Student s;

Here, s is a structure variable of type **Student**. When the structure variable is created, the memory will be allocated. Student structure contains one char variable and two integer variable.

Therefore, the memory for one char variable is 1 byte and two ints will be 2*4 = 8. The total memory occupied by the s variable is 9 byte.

How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example: s.id = 4;

In the above statement, we are accessing the id field of the structure Student by using the **dot(.)** operator and assigns the value 4 to the id field.

C++ Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

```
#include <iostream>
using namespace std;
 struct Rectangle
{
   int width, height;
 };
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
   cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
 return 0;
}
```

Output: Area of Rectangle is: 40

C++ Struct Example: Using Constructor and Method

Let's see another example of struct where we are using the constructor to initialize data and method to calculate the area of rectangle.

```
#include <iostream>
using namespace std;
 struct Rectangle    {
   int width, height;
  Rectangle(int w, int h)
    {
       width = w;
       height = h;
    }
  void areaOfRectangle() {
    cout<<"Area of Rectangle is: "<<(width*height); }
 };
int main(void) {
    struct Rectangle rec=Rectangle(4,6);
    rec.areaOfRectangle();
   return 0;
}
```

Output: Area of Rectangle is: 24

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
    friend data_type function_name(argument/s);          // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

The example of C++ friend function used to print the length of a box.

```
#include <iostream>
using namespace std;
class Box
{
   private:
      int length;
   public:
      Box(): length(0) { }
      friend int printLength(Box); //friend function
};
int printLength(Box b)
{
  b.length += 10;
   return b.length;
}
int main()
{
   Box b;
   cout<<"Length of box: "<< printLength(b)<<endl;
   return 0;
}
```

Output: Length of box: 10

Let's see a simple example when the function is friendly to two classes.

```
#include <iostream>
using namespace std;
class B;          // forward declarartion.
class A
{
   int x;
   public:
   void setdata(int i)
   {
      x=i;
   }
   friend void min(A,B);        // friend function.
};
class B
{
   int y;
   public:
   void setdata(int i)
   {
      y=i;
   }
   friend void min(A,B);                // friend function
};
void min(A a,B b)
{
   if(a.x<=b.y)
   std::cout << a.x << std::endl;
   else
   std::cout << b.y << std::endl;
}
  int main()
{
```

```
    A a;
    B b;
    a.setdata(10);
    b.setdata(20);
    min(a,b);
     return 0;
  }
```

Output: 10

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

A friend class can access both private and protected members of the class in which it has been declared as friend.

```
#include <iostream>
using namespace std;
class A
{
   int x =5;
   friend class B;          // friend class.
};
class B
{
  public:
    void display(A &a)
    {
       cout<<"value of x is : "<<a.x;
    }
};
int main()
{
    A a;
```

```
    B b;
    b.display(a);
    return 0;
}
```

Output: value of x is : 5

In the above example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A. Class B can access the private members of class A.

UNIT-3

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

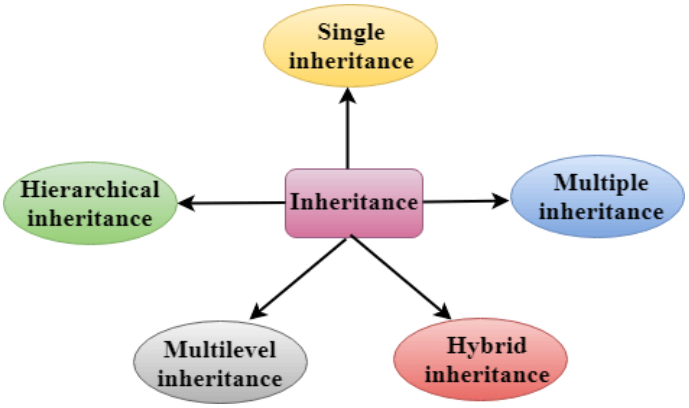
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types of Inheritance

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Derived Classes: A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

- **class** derived_class_name :: visibility-mode base_class_name
- {
- // body of the derived class.
- }

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

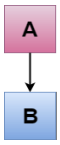
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
- When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

Note:

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

(1) Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Output:
Salary: 60000
Bonus: 5000

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

```
#include <iostream>
using namespace std;
 class Animal {
   public:
  void eat() {
     cout<<"Eating..."<<endl;
  }
    };
    class Dog: public Animal
    {
       public:
     void bark(){
    cout<<"Barking...";
     }
    };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:
Eating...
Barking...

```
#include <iostream>
using namespace std;
class A
{
```

```
    int a = 4;
    int b = 5;
    public:
    int mul()
    {
        int c = a*b;
        return c;
    }
};
class B : private A
{
    public:
    void display()
    {
        int result = mul();
        std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
    }
};
int main()
{
  B b;
  b.display();
   return 0;
}
```

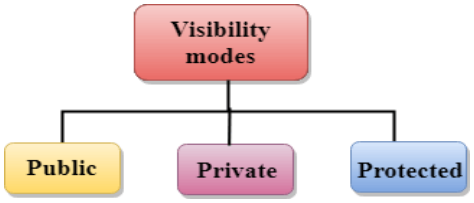
Output: Multiplication of a and b is: 20

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

Visibility modes can be classified into three categories:



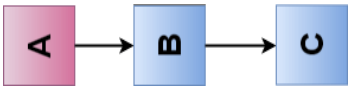
- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

(2) Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
  void eat() {
     cout<<"Eating..."<<endl;
  }
    };
    class Dog: public Animal
    {
       public:
     void bark(){
     cout<<"Barking..."<<endl;
      }
    };
    class BabyDog: public Dog
    {
       public:
     void weep() {
     cout<<"Weeping...";
      }
    };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

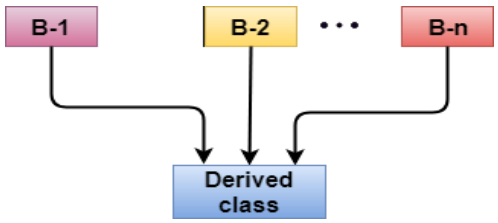
Eating...

Barking...

Weeping...

(3) Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Syntax of the Derived class:

- **class** D : visibility B-1, visibility B-2, ?
- {
- // Body of the class;
- }

Example of multiple inheritance.

- #include <iostream>
- **using namespace** std;
- **class** A
- {
- **protected**:
- **int** a;
- **public**:
- **void** get_a(**int** n)
- {
- a = n;
- }
- };
- **class** B
- {
- **protected**:
- **int** b;

```
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A, public B
{
  public:
   void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();
   return 0;
}
```

Output:
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

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

```
{
    public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
class B
{
    public:
    void display()
    {
        std::cout << "Class B" << std::endl;
    }
};
class C : public A, public B
{
    void view()
    {
        display();
    }
};
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output: error: reference to 'display' is ambiguous

display();

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
```

```
{
    void view()
    {
        A:: display();        // Calling the display() function of class A.
        B:: display();        // Calling the display() function of class B.

    }
};
```

An ambiguity can also occur in single inheritance.
Consider the following situation:

```
class A
{
  public:
void display()
{
  cout<<?Class A?;
}
} ;
class B
{
  public:
 void display()
{
 cout<<?Class B?;
}
} ;
```

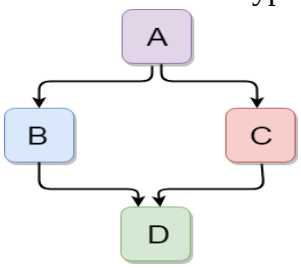
In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
    b.display();            // Calling the display() function of B class.
```

```
    b.B :: display();       // Calling the display() function defined in B class.
}
```

Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



Let's see a simple example:

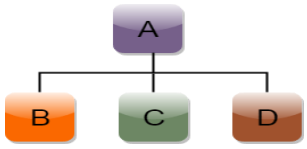
```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
     std::cout << "Enter the value of 'a' : " << std::endl;
     cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
      std::cout << "Enter the value of 'b' : " << std::endl;
```

```
        cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                // Declaration of base class.
{
   public:
   int a;
   int b;
   void get_data(int n,int m)
   {
      a= n;
      b = m;
   }
};
class Rectangle : public Shape  // inheriting Shape class
{
   public:
   int rect_area()
   {
      int result = a*b;
      return result;
   }
};
class Triangle : public Shape    // inheriting Shape class
{
   public:
   int triangle_area()
   {
      float result = 0.5*a*b;
      return result;
   }
};
int main()
{
   Rectangle r;
```

```
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    cin>>base>>height;
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : "  << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

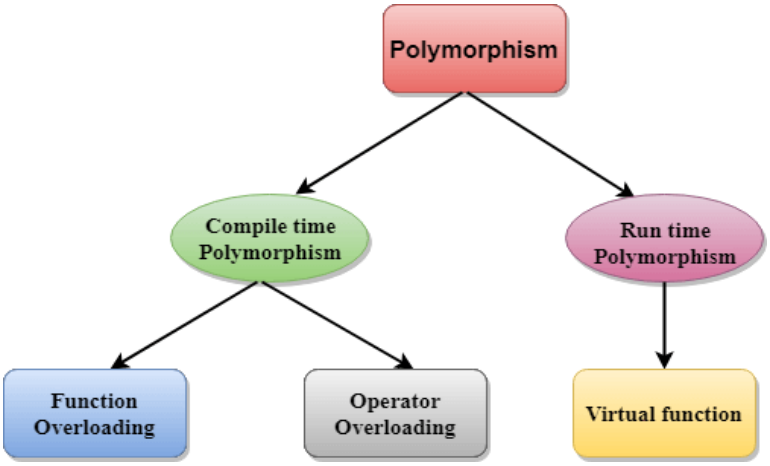
Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word.

Real Life Example of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



- **Compile time polymorphism**: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as **static binding or early binding.** Now, let's consider the case where function name and prototype is same.

```
class A                          //  base class declaration.
{
    int a;
    public:
    void display()
    {
        cout<< "Class A ";
    }
};
class B : public A                  //  derived class declaration.
{
   int b;
   public:
  void display()
  {
      cout<<"Class B";
  }
};
```

In the above case, the prototype of display() function is the same in both the **base and derived class**. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

Run time polymorphism: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

// an example without the virtual keyword.

- #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...

C++ Run time Polymorphism Example: By using two derived class

```
// an example with virtual keyword.
#include <iostream>
using namespace std;
class Shape {                          //  base class
   public:
virtual void draw(){                    // virtual function
cout<<"drawing..."<<endl;
   }
};
class Rectangle: public Shape              //  inheriting Shape class.
```

```
{
 public:
 void draw()
   {
      cout<<"drawing rectangle..."<<endl;
    }
};
class Circle: public Shape                    //  inheriting Shape class.

{
 public:
 void draw()
   {
     cout<<"drawing circle..."<<endl;
   }
};
int main(void) {
    Shape *s;                        //  base class pointer.
    Shape sh;                        // base class object.
      Rectangle rec;
       Circle cir;
     s=&sh;
    s->draw();
      s=&rec;
    s->draw();
    s=?
    s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>
using namespace std;
class Animal {                          //  base class declaration.
   public:
   string color = "Black";
};
class Dog: public Animal                // inheriting Animal class.
{
 public:
   string color = "Grey";
};
int main(void) {
    Animal d= Dog();
   cout<<d.color;
}
```

Output: Black

Overloading (Function and Operator)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

- Function overloading
- Operator overloading

C++ 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++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#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

Program of function overloading with different types of arguments.

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);
int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
    std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

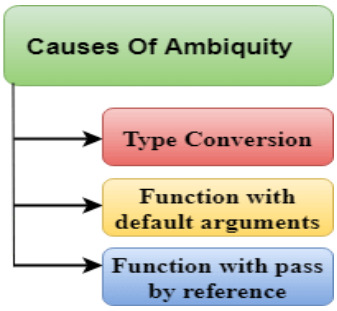
Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



Type Conversion:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
    std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
    fun(12);
    fun(1.2);
```

```
    return 0;
}
```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int

i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with pass by reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation

on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}    // initialize to the constructor the value 8
     void operator ++()        {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output: The Count is: 10

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
{
   int x;
     public:
     A(){}
   A(int i)
   {
     x=i;
   }
   void operator+(A);
   void display();
};

void A :: operator+(A a)
{
   int m = x+a.x;
   cout<<"The result of the addition of two objects is : "<<m;
}
int main()
{
   A a1(5);
   A a2(4);
   a1+a2;
   return 0;
}
```

Output: The result of the addition of two objects is: 9

Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#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...

Virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- 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.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
```

```
    int x=5;
    public:
    void display()
    {
        std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
        std::cout << "Value of y is : " <<y<< std::endl;
    }
};
int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
```

Output: Value of x is : 5

In the above example, *** a is the base class pointer. The pointer can only access the base class members but not the members of the derived class.** Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

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>
Class A
{
 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

Pure Virtual Function

A virtual function is not used for performing any task. It only serves as a placeholder. When the function has no definition, such function is known as "**do-nothing**" function. The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function

declared in the base class that has no definition relative to the base class. A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes. The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

Output: Derived class is derived from the base class.

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>
using namespace std;
class Address {
   public:
  string addressLine, city, state;
    Address(string addressLine, string city, string state)
    {
       this->addressLine = addressLine;
       this->city = city;
       this->state = state;
    }
};
class Employee
    {
       private:
       Address* address;  //Employee HAS-A Address
       public:
       int id;
       string name;
```

```
    Employee(int id, string name, Address* address)
    {
       this->id = id;
       this->name = name;
       this->address = address;
    }
   void display()
    {
       cout<<id <<" "<<name<< " "<<
        address->addressLine<< " "<< address->city<< " "<<address->state<<endl;
    }
  };
int main(void) {
   Address a1= Address("C-146, Sec-15","Noida","UP");
   Employee e1 = Employee(101,"Nakul",&a1);
        e1.display();
  return 0;
}
```

Output: **101 Nakul** C-146, Sec-15 Noida UP

Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. **Abstract class**
2. **Interface**

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract class

In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

Let's see an example of abstract class in C++ which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
#include <iostream>
using namespace std;
 class Shape
{
   public:
   virtual void draw()=0;
};
 class Rectangle : Shape
{
   public:
    void draw()
    {
       cout < <"drawing rectangle..." < <endl;
    }
};
class Circle : Shape
{
   public:
    void draw()
    {
       cout <<"drawing circle..." < <endl;
    }
};
int main( )
{
   Rectangle rec;
   Circle cir;
```

```
    rec.draw();
    cir.draw();
    return 0;
}
```

Output: drawing rectangle... drawing circle...

Data Abstraction

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.
- Data Abstraction is a programming technique that depends on the seperation of the interface and implementation details of the program.
- Let's take a real life example of AC, which can be turned ON or OFF, change the temperature, change the mode, and other external components such as fan, swing. But, we don't know the internal details of the AC, i.e., how it works internally. Thus, we can say that AC seperates the implementation details from the external interface.
- C++ provides a great level of abstraction. For example, pow() function is used to calculate the power of a number without knowing the algorithm the function follows.

In C++ program if we implement class with private and public members then it is an example of data abstraction.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.

Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which

algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Access Specifiers Implement Abstraction:

- **Public specifier:** When the members are declared as public, members can be accessed anywhere from the program.
- **Private specifier:** When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

// program to calculate the power of a number.

```
#include <iostream>
#include<math.h>
using namespace std;
int main()
{
 int n = 4;
   int power = 3;
   int result = pow(n,power);        // pow(n,power) is the  power function
   std::cout << "Cube of n is : " <<result<< std::endl;
   return 0;
}
```

Output: Cube of n is : 64

Let's see a simple example of data abstraction using classes.

```
#include <iostream>
using namespace std;
 class Sum
{
private: int x, y, z; // private variables
public:
```

```
void add()
{
cout<<"Enter two numbers: ";
cin>>x>>y;
z= x+y;
cout<<"Sum of two number is: "<<z<<endl;
}
};
int main()
{
Sum sm;
sm.add();
return 0;
}
```

Output:

```
Enter two numbers:
3
6
Sum of two number is: 9
```

Advantages of Abstraction:

- Implementation details of the class are protected from the inadvertent user level errors.
- A programmer does not need to write the low level code.
- Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.
- The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.
- Internal implementation can be changed without affecting the user level code.

C++ Namespaces

Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application.

For accessing the class of a namespace, we need to use namespacename::classname. We can use **using** keyword so that we don't have to use complete name all the time.

In C++, global namespace is the root namespace. The global::std will always refer to the namespace "std" of C++ Framework.

C++ namespace Example

The example of namespace which include variable and functions.

```
#include <iostream>
using namespace std;
namespace First {
    void sayHello() {
        cout<<"Hello First Namespace"<<endl;
    }
}
namespace Second  {
        void sayHello() {
            cout<<"Hello Second Namespace"<<endl;
        }
}
int main()
{
 First::sayHello();
 Second::sayHello();
return 0;
}
```

Output: Hello First Namespace Hello Second Namespace

C++ namespace example: by using keyword

Let's see another example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

```
#include <iostream>
using namespace std;
namespace First{
  void sayHello(){
    cout << "Hello First Namespace" << endl;
  }
}
namespace Second{
  void sayHello(){
    cout << "Hello Second Namespace" << endl;
  }
}
using namespace First;
int main () {
  sayHello();
  return 0;
}
```

Output: Hello First Namespace

Strings

In C++, string is an object of **std::string** class that represents sequence of characters. We can perform many operations on strings such as concatenation, comparison, conversion etc.

Example

```
#include <iostream>
using namespace std;
int main( ) {
   string s1 = "Hello";
      char ch[] = { 'C', '+', '+'};
      string s2 = string(ch);
```

```
        cout<<s1<<endl;
        cout<<s2<<endl;
}
```

Output: Hello C++

String Compare Example

Let's see the simple example of string comparison using strcmp() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
  char key[] = "mango";
  char buffer[50];
  do {
     cout<<"What is my favourite fruit? ";
     cin>>buffer;
  } while (strcmp (key,buffer) != 0);
 cout<<"Answer is correct!!"<<endl;
  return 0;
}
```

Output:

What is my favourite fruit? apple
What is my favourite fruit? banana
What is my favourite fruit? mango
Answer is correct!!

String Concat Example

Let's see the simple example of string concatenation using strcat() function.

```
#include <iostream>
#include <cstring>
```

```
using namespace std;
int main()
{
    char key[25], buffer[25];
    cout << "Enter the key string: ";
    cin.getline(key, 25);
    cout << "Enter the buffer string: ";
     cin.getline(buffer, 25);
    strcat(key, buffer);
    cout << "Key = " << key << endl;
    cout << "Buffer = " << buffer<<endl;
    return 0;
}
```

Output:

```
Enter the key string: Welcome to
Enter the buffer string:  C++ Programming.
Key = Welcome to C++ Programming.
Buffer =  C++ Programming.
```

String Copy Example

Let's see the simple example of copy the string using strcpy() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char key[25], buffer[25];
    cout << "Enter the key string: ";
    cin.getline(key, 25);
    strcpy(buffer, key);
    cout << "Key = "<< key << endl;
    cout << "Buffer = "<< buffer<<endl;
```

```
    return 0;
}
```

C++ String Length Example

Let's see the simple example of finding the string length using strlen() function.

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{
    char ary[] = "Welcome to C++ Programming";
    cout << "Length of String = " << strlen(ary)<<endl;
    return 0;
}
```

Output: Length of String = 26

C++ String Functions

Function	Description
int compare(const string& str)	It is used to compare two string objects.
int length()	It is used to find the length of the string.
void swap(string& str)	It is used to swap the values of two string objects.
string substr(int pos,int n)	It creates a new string object of n characters.
int size()	It returns the length of the string in terms of bytes.
void resize(int n)	It is used to resize the length of the string up to n characters.

string& replace(int pos,int len,string& str)	It replaces portion of the string that begins at character position pos and spans len characters.
string& append(const string& str)	It adds new characters at the end of another string object.
char& at(int pos)	It is used to access an individual character at specified position pos.
int find(string& str,int pos,int n)	It is used to find the string specified in the parameter.
int find_first_of(string& str,int pos,int n)	It is used to find the first occurrence of the specified sequence.
int find_first_not_of(string& str,int pos,int n)	It is used to search the string for the first character that does not match with any of the characters specified in the string.
int find_last_of(string& str,int pos,int n)	It is used to search the string for the last character of specified sequence.
int find_last_not_of(string& str,int pos)	It searches for the last character that does not match with the specified sequence.
string& insert()	It inserts a new character before the character indicated by the position pos.
int max_size()	It finds the maximum length of the string.
void push_back(char ch)	It adds a new character ch at the end of the string.
void pop_back()	It removes a last character of the string.
string& assign()	It assigns new value to the string.

int copy(string& str)	It copies the contents of string into another.
char& back()	It returns the reference of last character.
Iterator begin()	It returns the reference of first character.
int capacity()	It returns the allocated space for the string.
const_iterator cbegin()	It points to the first element of the string.
const_iterator cend()	It points to the last element of the string.
void clear()	It removes all the elements from the string.
const_reverse_iterator crbegin()	It points to the last character of the string.
const_char* data()	It copies the characters of string into an array.
bool empty()	It checks whether the string is empty or not.
string& erase()	It removes the characters as specified.
char& front()	It returns a reference of the first character.
string& operator+=()	It appends a new character at the end of the string.
string& operator=()	It assigns a new value to the string.
char operator[](pos)	It retrieves a character at specified position pos.
int rfind()	It searches for the last occurrence of the string.

iterator end()	It references the last character of the string.
reverse_iterator rend()	It points to the first character of the string.
void shrink_to_fit()	It reduces the capacity and makes it equal to the size of the string.
char* c_str()	It returns pointer to an array that contains null terminated sequence of characters.
const_reverse_iterator crend()	It references the first character of the string.
reverse_iterator rbegin()	It reference the last character of the string.
void reserve(inr len)	It requests a change in capacity.
allocator_type get_allocator();	It returns the allocated object associated with the string.

Unit

Exception Handling

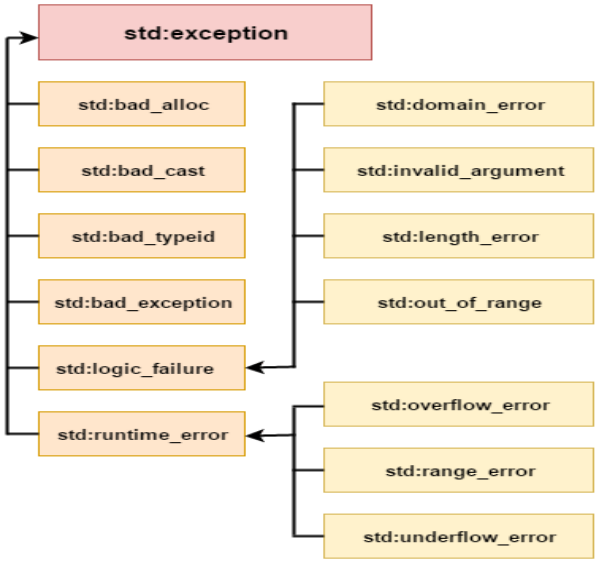
Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage: It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

Exception	Description
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.

std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement. The C++ **try block** is used to place the code that may occur exception. The **catch block** is used to handle the exception.

example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
   return (x/y);
}
int main () {
   int i = 50;
   int j = 0;
   float k = 0;
      k = division(i, j);
```

```
        cout << k << endl;
    return 0;
}
```

Output: Floating point exception (core dumped)

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  if( y == 0 ) {
    throw "Attempted to divide by zero!";
  }
  return (x/y);
}
int main () {
  int i = 25;
  int j = 0;
  float k = 0;
  try {
    k = division(i, j);
    cout << k << endl;
  }catch (const char* e) {
    cerr << e << endl;
  }
  return 0;
}
```

Output: Attempted to divide by zero!

1. #include <iostream>

```
using namespace std;
int main()
{
  int x = -1
```

```
  // Some code
  cout << "Before try \n";
  try {
    cout << "Inside try \n";
    if (x < 0)
    {
      throw x;
      cout << "After throw (Never executed) \n";
    }
  }
  catch (int x ) {
    cout << "Exception Caught \n";
  }
  cout << "After catch (Will be executed) \n";
  return 0;
}
```

Output:

Before try
Inside try
Exception Caught
After catch (Will be executed)

2) There is a special catch block called the 'catch all' block, written as catch(…), that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(…) block will be executed.

```
#include <iostream>
using namespace std;
int main()
{
   try  {
     throw 10;
   }
   catch (char *excp)  {
      cout << "Caught " << excp;
```

```
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output: Default Exception

3) Implicit type conversion doesn’t happen for primitive types. For example, in the following program, ‘a’ is not implicitly converted to int.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
      throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output: Default Exception

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch the char.

```
#include <iostream>
using namespace std;
int main()
{
    try  {
      throw 'a';
```

```
    }
    catch (int x)  {
        cout << "Caught ";
    }
    return 0;
}
```

Output:

terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
#include <iostream>
#include <exception>
using namespace std;
class MyException : public exception{
   public:
      const char * what() const throw()
      {
         return "Attempted to divide by zero!\n";
      }
};
int main()
{
   try
   {
      int x, y;
```

```
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0)
        {
            MyException z;
            throw z;
        }
        else
        {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(exception& e)
    {
        cout << e.what();
    }
}
```

Output: Enter the two numbers :

10

2

x / y = 5

Note: In above example what() is a public method provided by the exception class. It is used to return the cause of an exception.

C++ Templates

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

- Function templates
- Class templates

Function Templates:

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

Class Template:

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

Function Template

- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Syntax of Function Template

- **template** < **class** Ttype> ret_type func_name(parameter_list)
- {
- // body of function.
- }

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

class: A class keyword is used to specify a generic type in a template declaration.

- #include <iostream>

```
using namespace std;
template<class T> T add(T &a,T &b)
{
    T result = a+b;
    return result;
}
int main()
{
 int i =2;
 int j =3;
 float m = 2.3;
 float n = 1.2;
 cout<<"Addition of i and j is :"<<add(i,j);
 cout<<'\n';
 cout<<"Addition of m and n is :"<<add(m,n);
 return 0;
}
```

Output: Addition of i and j is :5 Addition of m and n is :3.5

In the above example, we create the function template which can perform the addition operation on any type either it can be integer, float or double.

Function Templates with Multiple Parameters

We can use more than one generic type in the template function by using the comma to separate the list.

Syntax

```
template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
    // body of function.
}
```

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
  fun(15,12.3);

  return 0;
}
```

Output: Value of a is : 15 Value of b is : 12.3

In the above example, we use two generic types in the template function, i.e., X and Y.

Overloading a Function Template

We can overload the generic function means that the overloaded template functions can differ in the parameter list.

```
#include <iostream>
using namespace std;
template<class X> void fun(X a)
{
    std::cout << "Value of a is : " <<a<< std::endl;
}
template<class X,class Y> void fun(X b ,Y c)
{
    std::cout << "Value of b is : " <<b<< std::endl;
```

```
    std::cout << "Value of c is : " <<c<< std::endl;
}
int main()
{
   fun(10);
   fun(20,30.5);
   return 0;
}
```

Output:

```
Value of a is : 10
Value of b is : 20
Value of c is : 30.5
```

In the above example, template of fun() function is overloaded.

Restrictions of Generic Functions

Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

```
#include <iostream>
using namespace std;
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}
void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
```

```
    else
    {
        cout<<"Number is odd";
    }
}
int main()
{
  fun(4.6);
  fun(6);
  return 0;
}
```

Output: value of a is : 4.6 Number is even

In the above example, we overload the ordinary functions. We cannot overload the generic functions as both the functions have different functionalities. First one is displaying the value and the second one determines whether the number is even or not.

CLASS TEMPLATE

Class Template can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

Syntax

```
template<class Ttype>
class class_name
{
  .
  .
}
```

Ttype is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

class_name<type> ob;

where class_name: It is the name of the class.

type: It is the type of the data that the class is operating on.

ob: It is the name of the object.

```
#include <iostream>
using namespace std;
template<class T>
class A
{
   public:
   T num1 = 5;
   T num2 = 6;
   void add()
   {
      std::cout << "Addition of num1 and num2 : " << num1+num2<<std::endl;
   }
};
int main()
{
   A<int> d;
   d.add();
   return 0;
}
```

Output: Addition of num1 and num2 : 11

In the above example, we create a template for class A. Inside the main() method, we create the instance of class A named as, 'd'.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

Syntax

- **template**<**class** T1, **class** T2, ...>
- **class** class_name
- {
- // Body of the class.
- }

Let's see a simple example when class template contains two generic data types.

```
#include <iostream>
    using namespace std;
    template<class T1, class T2>
   class A
   {
      T1 a;
      T2 b;
      public:
     A(T1 x,T2 y)
     {
        a = x;
        b = y;
     }
        void display()
       {
            std::cout << "Values of a and b are : " << a<<" ,"<<b<<std::endl;
        }
    };

    int main()
```

```
{
    A<int,float> d(5,6.5);
    d.display();
    return 0;
}
```

Output: Values of a and b are : 5,6.5

Nontype Template Arguments

The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types. **Let' s see the following example:**

```
template<class T, int size>
class array
{
    T arr[size];          // automatic array initialization.
};
```

In the above case, the nontype template argument is size and therefore, template supplies the size of the array as an argument.

Arguments are specified when the objects of a class are created:

```
array<int, 15> t1;                  // array of 15 integers.
array<float, 10> t2;                // array of 10 floats.
array<char, 4> t3;                  // array of 4 chars.
#include <iostream>
using namespace std;
template<class T, int size>
class A
{
  public:
  T arr[size];
  void insert()
```

```
    {
       int i =1;
       for (int j=0;j<size;j++)
       {
          arr[j] = i;
          i++;
       }
    }

    void display()
    {
       for(int i=0;i<size;i++)
       {
          std::cout << arr[i] << " ";
       }
    }
};
int main()
{
    A<int,10> t1;
    t1.insert();
    t1.display();
    return 0;
}
```

Output: 1 2 3 4 5 6 7 8 9 10

In the above example, the class template is created which contains the nontype template argument, i.e., size. It is specified when the object of class 'A' is created.

Points to Remember

- C++ supports a powerful feature known as a template to implement the concept of generic programming.

- A template allows us to create a family of classes or family of functions to handle different data types.
- Template classes and functions eliminate the code duplication of different data types and thus makes the development easier and faster.
- Multiple parameters can be used in both class and function template.
- Template functions can also be overloaded.
- We can also use nontype arguments such as built-in or derived data types as template arguments.