Ch-1 Principles of object-oriented programming Tokens, expressions and control statements

What is Procedural Programming?

Procedural Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables through a sequence of instructions. In procedural oriented programming, each step is executed in a systematic manner so that the computer can understand what to do.

The programming model of the procedural oriented programming is derived from structural programming. The concept followed in the procedural oriented programming is called the "procedure". These procedures consist several computational steps that are carried out during the execution of a program. Examples of procedural oriented programming language include – C, Pascal, ALGOL, COBOL, BASIC, etc.

What is Object Oriented Programming?

Object-oriented Programming is a programming language that uses classes and objects to create models based on the real-world environment. These objects contain data in the form of attributes and program codes in the form of methods or functions.

In OOP, the computer programs are designed by using the concept of objects that can interact with the real-world entities.

We have several types of object-oriented programming languages, but the most popular is one among all is class-based language.

the objects are the instances of the classes that determine their types. Examples of some object-oriented programming languages are – Jave, C++, C#, Python, PHP, Swift, etc.

Differences between Procedural and Object Oriented Programming

Parameter	Object Oriented Programming	Procedural Programming
Definition	Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment.	Procedural Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or

	In OOPs, it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones.	subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do.
Approach	In OOPs concept of objects and classes is introduced and hence the program is divided into small chunks called objects which are instances of classes.	In procedural programming, the main program is divided into small parts based on the functions and is treated as separate program for individual smaller program.
Access modifiers	In OOPs access modifiers are introduced namely as Private, Public, and Protected.	No such modifiers are introduced in procedural programming.
Security	Due to abstraction in OOPs data hiding is possible and hence it is more secure than POP.	Procedural programming is less secure as compare to OOPs.
Complexity	OOPs due to modularity in its programs is less complex and hence new data objects can be created easily from existing objects making object-oriented programs easy to modify	There is no simple process to add data in procedural programming, at least not without revising the whole program.
Program division	OOP divides a program into small parts and these parts are referred to as objects.	Procedural programming divides a program into small programs and each small program is referred to as a function.
Importance	OOP gives importance to data rather than functions or procedures.	Procedural programming does not give importance to data. In POP, functions along with sequence of actions are followed.
Inheritance	OOP provides inheritance in three modes i.e. protected, private, and public	Procedural programming does not provide any inheritance.
Examples	C++, C#, Java, Python, etc. are the examples of OOP languages.	C, BASIC, COBOL, Pascal, etc. are the examples POP languages.

Basic Concepts of Object Oriented Programming using C++

Object oriented programming is a type of programming which uses objects and classes its functioning. The object oriented programming is based on real world entities like inheritance, polymorphism, data hiding, etc.

Some basic concepts of object oriented programming are −

- CLASS
- OBJECTS
- ENCAPSULATION
- POLYMORPHISM
- INHERITANCE
- ABSTRACTION

Class − A class is a data-type that has its own members i.e. data members and member functions. It is the blueprint for an object in object oriented programming language. It is the basic building block of object oriented programming in c++.

The members of a class are accessed in programming language by creating an instance of the class.

- **Class** is a user-defined data-type.
- A class contains members like data members and member functions.
- **Data members** are variables of the class.
- **Member functions** are the methods that are used to manipulate data members.
  - Data members define the properties of the class whereas the member functions define the behaviour of the class.

A class can have multiple objects which have properties and behaviour that in common for all of them.

Syntax

```
class class_name {
  data_type data_name;
  return_type method_name(parameters);
}
```

Object − An object is an instance of a class. It is an entity with characteristics and behaviour that are used in the object oriented programming. An object is the

entity that is created to allocate memory. A class when defined does not have memory chunk itself which will be allocated as soon as objects are created.

Syntax

```
class_name object_name;
```

Example

```
#include<iostream>
using namespace std;
class calculator {
   int number1;
   int number2;
   char symbol;
   public :
   void add() {
      cout<<"The sum is "<<number1 + number2 ;
   }
   void sub() {
      cout<<"The subtraction is "<<number1 - number2 ;
   }
   void mult() {
      cout<<"The multiplication is "<<number1 * number2 ;
   }
   void div() {
      cout<<"The division is "<<number1 / number2 ;
   }
   calculator (int a , int b , char sym) {
      number1 = a;
      number2 = b;
      symbol = sym;
      switch(symbol){
```

```
        case '+' : add();
          break;
        case '-' : add();
          break;
        case '*' : mult();
          break;
        case '/' : div();
          break;
        default : cout<<"Wrong operator";
      }
    }
};
int main() {
  calculator c1(12 , 34 , '+');
}
```

Encapsulation In object oriented programming, encapsulation is the concept of wrapping together of data and information in a single unit. A formale defination of encapsulation would be: encapsulation is binding togather the data and related function that can manipulate the data.

Real life example,

In our colleges, we have departments for each course like computer science, information tech. , electronics, etc. each of these departments have their own students and subjects that are kept track of and being taught. let's think of each department as a class that encapsulates the data about students of that department and the subjects that are to be taught. Also a department has some fixed rules and guidelines that are to be followed by the students that course like timings, methods used while learning, etc. this is encapsulation in real life, there are data and there are ways to manipulate data.

Due to the concept of encapsulation in object oriented programming another very important concept is possible, it is data abstraction or Data Hiding. it is possible as encapsulating hides the data at show only the information that is required to be displayed.

Polymorphism The name defines polymorphism is multiple forms. which means polymorphism is the ability of object oriented programming to do some

work using multiple forms. The behaviour of the method is dependent on the type or the situation in which the method is called.

Let’s take a real life example, A person can have more than one behaviour depending upon the situation. like a woman a mother, manager and a daughterAnd this define her behaviour. This is from where the concept of polymorphism came from.

In c++ programming language, polymorphism is achieved using two ways. They are operator overloading and function overloading.

Operator overloading In operator overloading and operator can have multiple behaviour in different instances of usage.

Function overloading Functions with the same name that can do multiple types based on some condition.

Inheritance it is the capability of a class to inherit or derive properties or characteristics other class. it is very important and object oriented program as it allows reusability i.e. using a method defined in another class by using inheritance. The class that derives properties from other class is known as child class or subclass and the class from which the properties are inherited is base class or parent class.

C plus plus programming language supports the following types of inheritance

- single inheritance
- multiple inheritance
- multi level inheritance
- Hierarchical inheritance
- hybrid inheritance

Abstraction Data abstraction or Data Hiding is the concept of hiding data and showing only relevant data to the final user. It is also an important part object oriented programing.

let's take real life example to understand concept better, when we ride a bike we only know that pressing the brake will stop the bike and rotating the throttle will accelerate but you don't know how it works and it is also not think we should know that's why this is done from the same as a concept data abstraction.

In C plus plus programming language write two ways using which we can accomplish data abstraction −

- using class
- using header file

Extra Link for C++ OOP:

https://www.javatpoint.com/cpp-oops-concepts

Benefits of object oriented programming

Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the C++ code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

Class	Objects		
Fruit	Apple	Mango	
Car	Volvo	Audi	Toyota

There are several advantages of OOP in C++. Some of them are as follows:

1. Easier Troubleshooting

One of the major benefits of using object-oriented programming when troubleshooting applications in C++ is its ability to help identify and isolate problems. By isolating objects, developers can more easily trace the source of an issue and determine which part of the code needs to be changed or improved.

2. Code Reusability

Save time and money by avoiding having to write new sections of code each time a project needs an update or feature change.

Reduce development costs while also improving program efficiency because with OOPs, entire blocks of code can be written once and then reused multiple times throughout the application.

Implement changes quickly without having to worry about breaking existing functionality or duplicating efforts unnecessarily.

3. Data Redundancy

Data redundancy is a common issue that can occur when working with databases. It occurs when multiple copies of the same piece of data are stored in different locations, resulting in unused disk space and likely conflicts between records. Although the term "redundancy" is disagreeable in this context, it is however seen as one of the benefits of OOP in C++ because it minimizes the repetition of tasks.

4. Code Flexibility

When an application is written with OOPs, developers will find that editing existing code becomes much easier as changes need to be made within one isolated area without affecting any other parts of the program. This means that tasks like bug fixing or adding new features can be done quickly and easily, requiring minimal effort on behalf of the developer.

5. Polymorphism Flexibility

Polymorphism is an incredibly powerful tool when it comes to coding applications in C++, allowing developers to easily create flexible and extensible programs that can be adapted for multiple purposes. In its simplest form, polymorphism is the ability of a single object or class to take on different forms depending on how it's used. This means that you can use the same codebase but change certain aspects of its behaviour based on user input or other environmental factors.

6. Better Productivity

In order to ensure better productivity when coding applications in C++, it is important for developers to employ a consistent methodology throughout the entire project. This means taking the time to plan out their program's design before actually jumping into writing code, ensuring that all components are properly identified and structured ahead of time. This will help reduce debugging efforts later on down the line.

7. Security

When encrypting applications in C++, security is of extreme importance. One-way developers can ensure data protection and privacy within their software solutions is by utilizing encapsulation principles which limit access rights between objects based on user permission levels inside these objects – ultimately making it harder for outside sources (such as hackers) from gaining unauthorized access or manipulating data stored within the system itself.

Extra Benefits of OOP

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
- OOP language allows to break the program into the bit-sized problems that can be solved easily (one object at a time).
- The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
- OOP systems can be easily upgraded from small to large systems.
- It is possible that multiple instances of objects co-exist without any interference,
- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in problem domain to those in the program.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.
- Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- The data-cantered design approach enables us to capture more details of model in an implementable form.

While it is possible to incorporate all these features in an OOP, their importance depends upon the type of project and preference of the programmer. These technology is still developing and current products may be superseded quickly.

Developing a software is easy to use makes it hard to build.

Disadvantages of OOP

- The length of the programmes developed using OOP language is much larger than the procedural approach. Since the programme becomes larger in size, it requires more time to be executed that leads to slower execution of the programme.
- We can not apply OOP everywhere as it is not a universal language. It is applied only when it is required. It is not suitable for all types of problems.
- Programmers need to have brilliant designing skill and programming skill along with proper planning because using OOP is little bit tricky.
- OOPs take time to get used to it. The thought process involved in object-oriented programming may not be natural for some people.
- Everything is treated as object in OOP so before applying it we need to have excellent thinking in terms of objects.

Applications of OOP in C++

As OOP is faster and easier to execute it becomes more powerful than procedural languages. OOPs is the most important and flexible paradigm of modern programming. It is specifically useful in modelling real-world problems. Below are some applications of OOPs:

Real-Time System design: Real-time system inherits complexities and makes it difficult to build them. OOP techniques make it easier to handle those complexities.

Hypertext and Hypermedia: Hypertext is similar to regular text as it can be stored, searched, and edited easily. Hypermedia on the other hand is a superset of hypertext. OOP also helps in laying the framework for hypertext and hypermedia.

AI Expert System: These are computer application that is developed to solve complex problems which are far beyond the human brain. OOP helps to develop such an AI expert System

Office automation System: These include formal as well as informal electronic systems that primarily concerned with information sharing and communication to and from people inside and outside the organization. OOP also help in making office automation principle.

Neural networking and parallel programming: It addresses the problem of prediction and approximation of complex-time varying systems. OOP simplifies the entire process by simplifying the approximation and prediction ability of the network.

Stimulation and modelling system: It is difficult to model complex systems due to varying specifications of variables. Stimulating complex systems require modeling and understanding interaction explicitly. OOP provides an appropriate approach for simplifying these complex models.

Object-oriented database: The databases try to maintain a direct correspondence between the real world and database object in order to let the object retain it identity and integrity.

Client-server system: Object-oriented client-server system provides the IT infrastructure creating object-oriented server internet(**OCSI**) applications.

CIM/CAD/CAM systems: OOP can also be used in manufacturing and designing applications as it allows people to reduce the efforts involved. For instance, it can be used while designing blueprints and flowcharts. So it makes it possible to produce these flowcharts and blueprint accurately.

What is C++?

C++ is a cross-platform language that can be used to create high-performance applications.

C++ gives programmers a high level of control over system resources and memory.

C++ was developed by Bjarne Stroustrup(बज़्ने स्ट्रॉस्ट्रप), as an extension to the C language.

C++ is one of the world's most popular programming languages.

C++ can be found in today's operating systems, Graphical User Interfaces, and embedded systems.

C++ is an object-oriented programming language which gives a clear structure to programs and allows code to be reused, lowering development costs.

C++ is portable and can be used to develop applications that can be adapted to multiple platforms.

C++ was developed as an extension of C, and both languages have almost the same syntax.

The main difference between C and C++ is that C++ support classes and objects, while C does not.

A compiler, like GCC, to translate the C++ code into a language that the computer will understand

Simple Example:

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

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

Example explained

Line 1: #include <iostream> is a **header file library** that lets us work with input and output objects, such as cout (used in line 5). Header files add functionality to C++ programs.

Line 2: using namespace std means that we can use names for objects and variables from the standard library.

Don't worry if you don't understand how #include <iostream> and using namespace std works. Just think of it as something that (almost) always appears in your program.

Line 3: A blank line. C++ ignores white space. But we use it to make the code more readable.

Line 4: Another thing that always appear in a C++ program, is int main(). This is called a **function**. Any code inside its curly brackets {} will be executed.

Line 5: cout (pronounced "see-out") is an **object** used together with the *insertion operator* (<<) to output/print text. In our example it will output "Hello World!".

Note: Every C++ statement ends with a semicolon ;.

Note: The body of int main() could also been written as:
int main () { cout << "Hello World! "; return 0; }

Line 6: return 0 ends the main function.

Line 7: Do not forget to add the closing curly bracket } to actually end the main function.

Omitting Namespace

You might see some C++ programs that runs without the standard namespace library. The using namespace std line can be omitted and replaced with the std keyword, followed by the :: operator for some objects:

Example:

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

New Lines

To insert a new line, you can use the \n character

```
#include <iostream>
using namespace std;
int main()
{
 cout << "Hello World! \n";
        cout << "I am learning C++";
 return 0;
}
```

Another way to insert a new line, is with the endl manipulator:

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

int main()
{
 cout << "Hello World!" << endl;
        cout << "I am learning C++";
 return 0;
}
```

Applications Of C++:

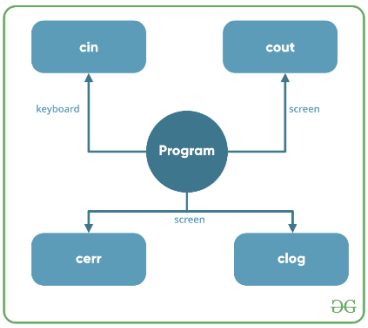
There are several uses or benefits of C++ for developing applications. For example, the applications that are based on the graphic user interface(GUI) like adobe photoshop and others. It is popular among students as a beginning language. Some of the major applications that are used in C++ by major software vendors, sellers Etc.

- **Google**
- **Mozilla**
- **Microsoft**
- **Rockstar Games**
- **MongoDB**
- **Games and Animations**
- **Media Access**
- **Compilers**
- **Scanning**
-

Input / Output in C++0

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.



Header files available in C++ for Input/Output operations are:

1. **iostream**: iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
2. **iomanip**: iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.
3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

Standard output stream (cout): Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the ostream class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(<<).

```
#include <iostream>
using namespace std;
int main()
{
   char sample[] = "Harivandana";
   cout << sample << " – College Of computer science ";
   return 0;
}
```

standard input stream (cin): Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.

The extraction operator(>>) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.

```
#include <iostream>
using namespace std;
int main()
{
   int age;

   cout << "Enter your age:";
   cin >> age;
   cout << "\nYour age is: " << age;
   return 0;
}
```

Structure of C++ Program

The C++ program is written using a specific template structure. The structure of the program written in C++ language is as follows:

Documentation Section:

- This section comes first and is used to document the logic of the program that the programmer going to code.
- It can be also used to write for purpose of the program.
- Whatever written in the documentation section is the comment and is not compiled by the compiler.
- Documentation Section is optional since the program can execute without them. Below is the snippet of the same:

For Example:

```
/*      This is a C++ program to find the factorial of a number

        The basic requirement for writing this program is to have knowledge of loops

        To find the factorial of number iterate over range from number to one

*/
```

Linking Section:
The linking section contains two parts:

Header Files:

Generally, a program includes various programming elements like built-in functions, classes, keywords, operators, etc. that are already defined in the standard C++ library.

In order to use such pre-defined elements in a program, an appropriate header must be included in the program.

Standard headers are specified in a program through the preprocessor directive #include. In Figure, the iostream header is used. When the compiler processes the instruction #include<iostream>, it includes the contents of the stream in the program.

This enables the programmer to use standard input, output, and error facilities that are provided only through the standard streams defined in <iostream>.

Namespaces:

- A namespace permits grouping of various entities like classes, objects, functions, and various C++ tokens, etc. under a single name.
- Any user can create separate namespaces of its own and can use them in any other program.

namespace std contains declarations for cout, cin, endl, etc. statements.

using namespace std;

Namespaces can be accessed in multiple ways:

1. using namespace std;
2. using std :: cout;

Definition Section:

It is used to declare some constants and assign them some value.

In this section, anyone can define your own datatype using primitive data types.

Global Declaration Section:

- Here, the variables and the class definitions which are going to be used in the program are declared to make them global.
- The scope of the variable declared in this section lasts until the entire program terminates.
- These variables are accessible within the user-defined functions also.

Function Declaration Section:

- It contains all the functions which our main functions need.
- Usually, this section contains the User-defined functions.

Main Function:

- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.

- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces {} which encloses the body of the main function.
- Once all instructions from the main function are executed, control comes out of the main function and the program terminates and no further execution occur.

Example:

```
// C++ program to print name  as output
#include <iostream>
using namespace std;
// Driver code
int main()
{
  string name;
  cout << "Enter the name: ";
  cin >> name;

  cout << "Entered name is: " <<
          name;
    return 0;
}
```

```
// C++ program to swap two numbers using 3rd variable
#include<bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    int a = 2, b = 3;

    cout << "Before swapping a = " <<
            a << " , b = " << b << endl;
    int temp;
    temp = a;
    a = b;
    b = temp;
    cout << "After swapping a = " << a <<
           " , b = " << b << endl;
    return 0;
```

```
}
// C++ program to swap two numbers without using 3rd variable
#include<bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
   int a = 2, b = 3;

   cout << "Before swapping a = " <<
         a << " , b = " << b << endl;

   b = a + b;
   a = b - a;
   b = b - a;

   cout << "After swapping a = " << a <<
        " , b = " << b << endl;
   return 0;
}
```

Namespace in C++

Namespace provide the space where we can define or declare identifier i.e. variable, method, classes.

Using namespace, you can define the space or context in which identifiers are defined i.e. variable, method, classes. In essence, a namespace defines a scope.

Advantage of Namespace

you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries.

The best example of namespace scope is the C++ standard library (std) where all the classes, methods and templates are declared. Hence while writing a C++ program we usually include the directive using namespace std;

Defining a Namespace:

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace  namespace_name

{

   // code declarations i.e. variable  (int a;)

   method (void add();)

   classes ( class student{};)

}
```

The using directive:

You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.

The namespace is thus implied for the following code:

```
#include <iostream>
using namespace std;
// first name space
namespace first_space
{
void func()
{
        cout << "Inside first_space" << endl;
}
}

// second name space
namespace second_space
{
void func()
{
        cout << "Inside second_space" << endl;
}
}
using namespace first_space;
```

```
int main ()
{
// This calls function from first name space.
func();
return 0;
}
```

A namespace is a feature added in C++ and is not present in C.
A namespace is a declarative region that provides a scope to the identifiers (names of functions, variables or other user-defined data types) inside it.
Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

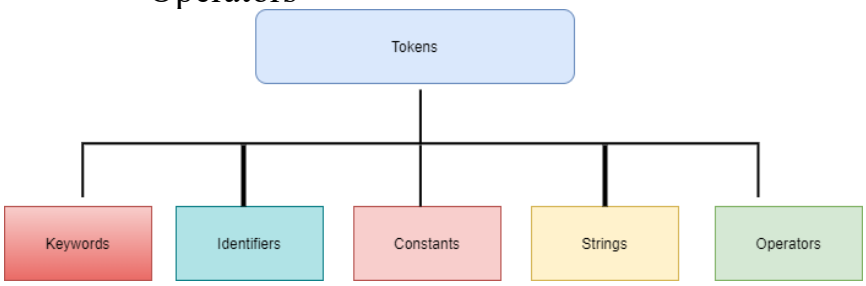
Tokens in C++:

C++ is a powerful language. In C++, we can write structured programs and object-oriented programs also. C++ is a superset of C and therefore most constructs of C are legal in C++ with their meaning unchanged. However, there are some exceptions and additions.

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

double new switch auto else operator break enum private this

case extern protected throw catch float public try

char for register typedef class 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 A,Name .
- 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.

C++ Data Types

All variables use data type during declaration to restrict the type of data to be stored. we can say that data types are used to tell the variables the type of data they can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared.

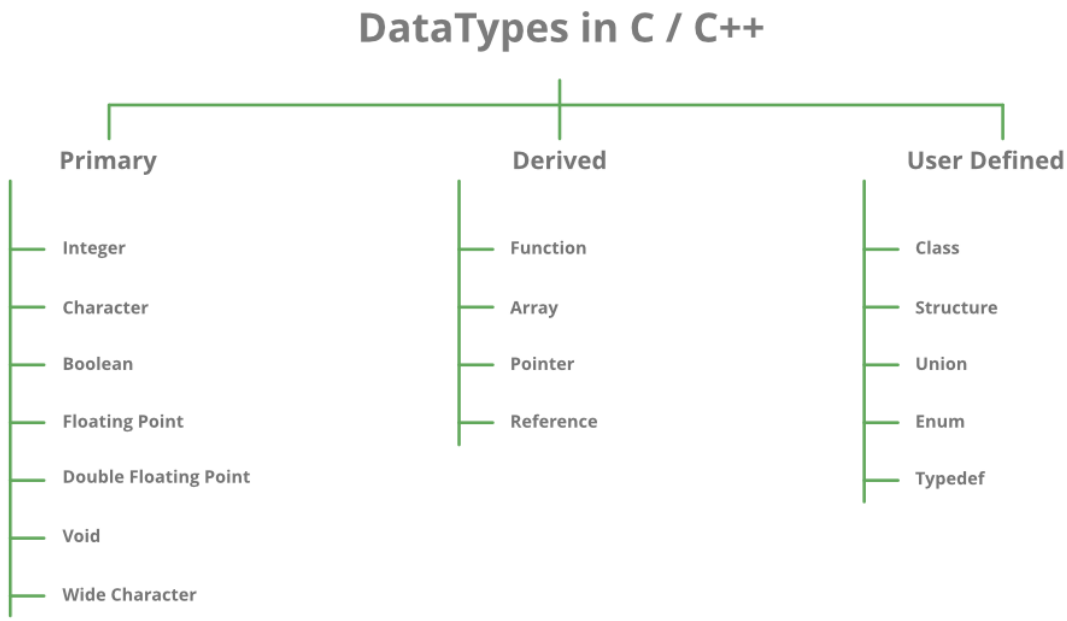
Every data type requires a different amount of memory.

C++ supports a wide variety of data types and the programmer can select the data type appropriate to the needs of the application.

Data types specify the size and types of values to be stored. However, storage representation and machine instructions to manipulate each data type differ from machine to machine, although C++ instructions are identical on all machines.

C++ supports the following data types:

1. **Primary** or **Built-in** or **Fundamental data type**
2. **Derived data types**
3. **User-defined data types**



Data Types in C++ are Mainly Divided into 3 Types:

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

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

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

- Function
- Array
- Pointer
- Reference

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

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

Simple Example:

// C++ Program to Demonstrate the correct size of various data types on your computer.

```
#include <iostream>

using namespace std;

int main()

{

    cout << "Size of char : " << sizeof(char) << endl;

    cout << "Size of int : " << sizeof(int) << endl;

    cout << "Size of long : " << sizeof(long) << endl;

    cout << "Size of float : " << sizeof(float) << endl;

    cout << "Size of double : " << sizeof(double) << endl;

    return 0;

}
```

User-Defined DataTypes:

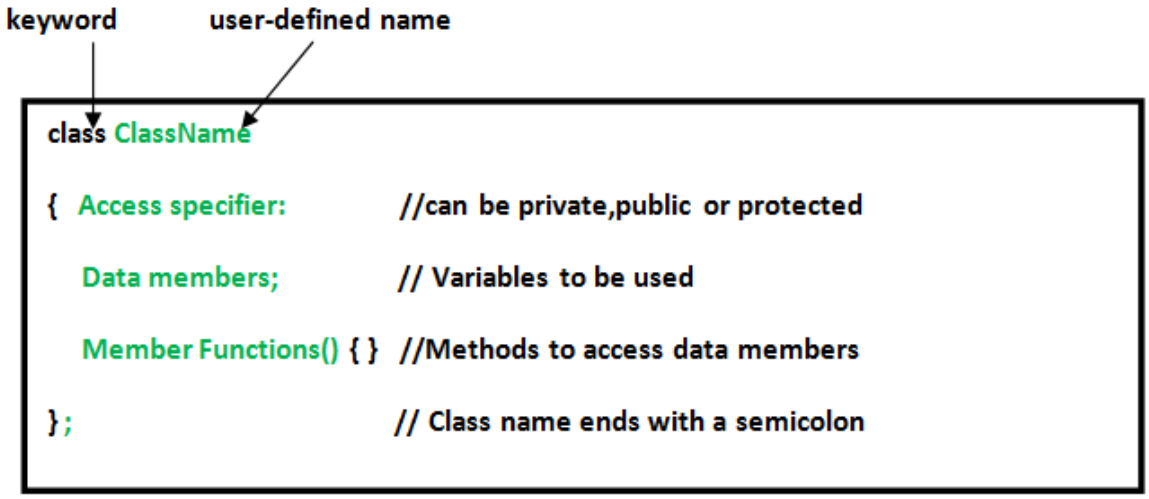
The data types that are defined by the user are called the derived datatype or user-defined derived data type.
These types include:

- Class
- Structure
- Union
- Enumeration
- Typedef defined DataType

Below is the detailed description of the following types:

Class:
The building block of C++ that leads to Object-Oriented programming is a **Class**. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

Syntax:



Example:

```
// C++ program to demonstrate Class
#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 = 1;
```

```
    s1.name = "Shree Ram";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

Structure:

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

Syntax:

```
struct address {
    char name[50];
    char street[100];
    char city[50];
    char state[20];
    int pin;
};
```

For Example:

```
// C++ program to demonstrate Structures in C++
#include <iostream>
using namespace std;
struct Point {
        int x, y;
};
int main()
{
```

```
    // Create an array of structures
    struct Point arr[10];
    // Access array members
    arr[0].x = 10;
    arr[0].y = 20;
    cout << arr[0].x << ", " << arr[0].y;
    return 0;
}
```

Union:

Like Structures, union is a user defined data type. In union, all members share the same memory location.

For example, in the following C program, both x and y share the same location. If we change x, we can see the changes being reflected in y.

```
#include <iostream>
using namespace std;
// Declaration of union is same as the structures
union test {
    int x, y;
};
int main()
{
    // A union variable t
    union test t;
    // t.y also gets value 2
```

```
    t.x = 2;
    cout << "After making x = 2:"
        << endl
        << "x = " << t.x
        << ", y = " << t.y
        << endl;
    // t.x is also updated to 10
    t.y = 10;
    cout << "After making Y = 10:"
        << endl
        << "x = " << t.x
        << ", y = " << t.y
        << endl;
    return 0;
}
```

Enumeration:

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Syntax:
enum State {Working = 1, Failed = 0};

For Example:

```
// Program to demonstrate working of enum in C++
#include <iostream>
using namespace std;
```

```
enum week { Mon,
            Tue,
            Wed,
            Thur,
            Fri,
            Sat,
            Sun };
int main()
{
    enum week day;
    day = Wed;
    cout << day;
    return 0;
}
```

Typedef:

C++ allows you to define explicitly new data type names by using the keyword typedef. Using typedef does not actually create a new data class, rather it defines a name for an existing type. This can increase the portability(the ability of a program to be used across different types of machines.

Syntax:
typedef type name;

// C++ program to demonstrate typedef

```
#include <iostream>

using namespace std;

// After this line BYTE can be used in place of unsigned char
```

```
typedef unsigned char BYTE;

int main()

{

        BYTE b1, b2;

        b1 = 'c';

        cout << " " << b1;

        return 0;

}
```

Constants in C++

Constants in C++ are the tokens which are defined at the time of initialization and the assigned value cannot be altered or updated after that. There are two methods to define constants in C++.

#define preprocessor directive method

'const' keyword method

Define preprocessor directive method:

This preprocessor directive provides an alias or a reference name to any variable or value. It is used to define constants in C++ by giving alias names to the value. This method defines constants globally.

Syntax:

#define constantName value

ConstantName: It is the identifier through which the value is going to be referred in the code. Value: It is the value whose reference is being created.

Example:

```
#include <iostream>
```

```
using namespace std;
#define PI 3.14
//function to calculate area of circle
float circle(float radius){
    return PI*radius*radius;
}
//function to calculate area of cylinder
float cylinder(float radius,float height){
    return (2*PI*radius*height)+(2*PI*radius*radius);
}
//function to calculate area of cone
float cone(float radius,float height){
    return PI*radius*(radius+(height*height)+(radius*radius));
}
//driver code
int main(){
    float radius=4,height=5;
    cout<<"Area of circle: "<<circle(radius)<<"\n";
    cout<<"Area of cylinder: "<<cylinder(radius,height)<<"\n";
    cout<<"Area of cone: "<<cone(radius,height)<<"\n";
}
```

'const' keyword method:

The definition of constants in C++ is very similar to the definition of variables in C++ but the definition is started with a const keyword. The definition follows a

particular pattern, starting with a const keyword followed by a datatype, an identifier, an assignment operator, and the value. Constants can be defined locally or globally through this method.

Syntax:

const datatype constantName = value

constantName: It is the identifier in which the value is being stored. value: It is the value that is being stored in the constantName.

For Example:

```
#include <iostream>

using namespace std;

//global constant definition

const int PI=3.14;

//function to calculate area of circle

float circle(float radius){

   return PI*radius*radius;

}

//function to calculate area of cylinder

float cylinder(float radius,float height){

   return (2*PI*radius*height)+(2*PI*radius*radius);

}

//function to calculate area of cone

float cone(float radius){

   //local constant definition

   const int height=6;

   return PI*radius*(radius+(height*height)+(radius*radius));
```

```
}

//driver code

int main(){

   float radius=4,height=5;

   cout<<"Area of circle: "<<circle(radius)<<"\n";

   cout<<"Area of cylinder: "<<cylinder(radius,height)<<"\n";

   cout<<"Area of cone: "<<cone(radius)<<"\n";

}
```

C++ Type compatibility

c++ is very strict with the type compatibility as compared to C language. for instance, c++ language defines int, short int, and long int as three different types, although each of these has size of one byte.

In C++ language the types of values must be the same for complete compatibility, or a cast must be applied.

These restrictions in c++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments

C++ is very strict with regard to type compatibility as compared to C programming. Type compatibility is very close to implicit or automatic type conversion. The type compatibility is being able to use two types being able to substitute one for the other without modification and together without modification.

The type compatibility is categorized into following three types by the compiler:

1. Assignment compatibility

In assignment compatibility, if the one type of variable assigned to another type variable is different It will results into loss of value of assigned variable if the size of the assigned variable is large than the size of variable to which it is assigned.

For example

Float f1=12.5;
int i1=f1;
This assigning of variable f1 float value to i1 int type will result in loss of decimal value of f1. However, this type type compatibility will not show any type error but it might give a warning “possible loss of data”.

2. Expression compatibility

Consider following example
int num=5/2;
cout<< num;
Here in the above example the result will be 2 because the actual result of 5/2 is 2.5 but because of incompatibility there will be loss of decimal value.

3. Parameter compatibility

Due to incompatibility when we pass the values from calling to called function in type of actual parameter and formal parameters loss of data occurs.

For example

```
void show(int num)

{

cout << ”num=” << num;

}

void main()

{

show(5.2);

}
```

For Example:

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

```
int main()
{
int num; // declare int type variable
double num2 = 15.25; // declare and assign the double variable

// use implicit type conversion to assign a double value to int variable
num = num2;
cout << " The value of the int variable is: " << num << endl;
cout << " The value of the double variable is: " << num2 << endl;
return 0;
}
```

Declaration of variables

Variables are containers for storing data values.

In C++, there are different **types** of variables (defined with different keywords), for example:

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

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

For Example:

// 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;

}
```

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.

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.

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

Dynamic Initialization of Variables

The process of initializing a variable at the moment it is declared at runtime is called dynamic initialization of the variable. Thus, during the dynamic initialization of a variable, a value is assigned to execution when it is declared.

Example:

```
main()

{

Int a;

 cout<<“Enter Value of a”;

cin>>a;

int cube = a * a * a;

}
```

In the above example, the variable cube is initialized at run time using expression a * a * an at the time of its declaration.

Reference Variables

A reference variable allows us to create an alternative name for the already defined variable. It is introduced in C++. Once you define a reference variable that references an already defined variable then you can use any of them alternatively in the program. Both refer to the same memory location. Thus, if you change the value of any of the variables it will affect both variables because one variable references another variable.

The general syntax for creating a reference variable is as follows:

Data-type & Referace_Name = Variable_Name;

The reference variable must be initialized during the declaration.

Example:

```
int count;
count = 5;
Int & incre = count;
```

Here, we have already defined a variable named count.

Then we create a reference variable named incre that refers to the variable count. Since both variables refer to the same memory location if you change the value of variable count using the following statement:

```
count=count + 5;
```

Will modify the variable count value and increment to 10. The major use of the reference variable is in function. If we want to pass the reference of the variable at the time of function call so that the function works with the original variable, we can use the reference variable.

For Example:

```
// C++ Program to demonstrate use of references

#include <iostream>

using namespace std;

int main()

{

	int x = 10;

	// ref is a reference to x.

	int& ref = x;

	// Value of x is now changed to 20

	ref = 20;

	cout << "x = " << x << '\n';

	// Value of x is now changed to 30

	x = 30;

	cout << "ref = " << ref << '\n';

	return 0;

}
```

C++ Operators

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,
Like: int c = a + b;

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

For Example:

```
// CPP Program to demonstrate the increment and decrement operators

#include <iostream>

using namespace std;
```

```
int main()
{
    int a = 10;
    cout << "a++ is " << a++ << endl;
    cout << "++a is " << ++a << endl;
    int b = 15;
    cout << "b-- is " << b-- << endl;
    cout << "--b is " << --b << endl;
    return 0;
}
```

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;

Name	Symbol	Description	Example
			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

For Example:

// CPP Program to demonstrate the Binary Operators

```
#include <iostream>

using namespace std;

int main()

{

    int a = 8, b = 3;

    // Addition operator

    cout << "a + b = " << (a + b) << endl;

    // Subtraction operator

    cout << "a - b = " << (a - b) << endl;

    // 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;
}
```

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

Name	Symbol	Description	Example

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;
			// returns true

Example:

```
// CPP Program to demonstrate the Relational Operators
#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;

}
```

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;

Name	Symbol	Description	Example
			// returns false

Example:

```
// CPP Program to demonstrate the Logical Operators
#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;
}
```

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); //returns -4

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

For Example:

// CPP Program to demonstrate the Bitwise Operators

```
#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;

}
```

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.

Name	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 the right and then assigns the result to the variable on the left	int a = 2, b = 4; a+=b; // a = 6
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

// CPP Program to demonstrate the Assignment Operators

```
#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;
}
```

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.

// 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 = 10
```

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

// CPP Program to demonstrate the & and * Operators

```
#include <iostream>

using namespace std;

int main()

{

        int a = 6;

        int* b;

        int c;

        // & Operator

        b = &a;

        // * Operator

        c = *b;

        cout << " a = " << a << endl;

        cout << " b = " << b << endl;

        cout << " c = " << c << endl;

        return 0;

}
```

Scope resolution operator

In C++, the scope resolution operator is **::**. It is used for the following purposes.

1) To access a global variable when there is a local variable with same name:

```
// 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;
int x; // Global x
int main()
{
int x = 10; // Local x
cout << "Value of global x is " << ::x;
cout << "\nValue of local x is " << x;
return 0;
}
```

2) To define a function outside a class.

```
// C++ program to show that scope resolution operator :: is
// used to define a function outside a class
#include <iostream>
using namespace std;

class A {
public:
        // Only declaration
        void fun();
};
// Definition outside class using ::
void A::fun() { cout << "fun() called"; }
int main()
```

```
{
    A a;
    a.fun();
    return 0;
}
```

Memory management operator in c++

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

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

1. pointer_variable = **new** data-type

Example:

1. **int** *p = **new int**(45);
2. **float** *p = **new float**(9.8);

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:

1. **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.

In the previous case, we have created two pointers 'p' and 'q' by using the new operator, and can be deleted by using the following statements:

1. **delete** p;
2. **delete** q;

For Example:

```
#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;
}
```

Manipulators in C++

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

- Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream.
- Manipulators are operators that are used to format the data display.

- To access manipulators, the file iomanip.h should be included in the program.

For example, if we want to print the hexadecimal value of 100 then we can print it as:

cout<<setbase(16)<<100

Types of Manipulators There are various types of manipulators:
Manipulators without arguments: The most important manipulators defined by the **IOStream library** are provided below.

- **endl**: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
- **ws**: It is defined in istream and is used to ignore the whitespaces in the string sequence.
- **ends**: It is also defined in ostream and it inserts a null character into the output stream. It typically works with std::ostrstream, when the associated output buffer needs to be null-terminated to be processed as a C string.
- **flush**: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time.

Casting Operators in C++
Casting operators are used for type casting in C++. They are used to convert one data type to another. C++ supports four types of casts:

1. **static_cast**
2. **dynamic_cast**
3. **const_cast**
4. **reinterpret_cast**

1. static_cast
The static_cast operator is the most commonly used casting operator in C++. It performs compile-time type conversion and is mainly used for explicit conversions that are considered safe by the compiler.
Syntax of static_cast
static_cast <*new_type*> (expression);

// C++ program to illustrate the static_cast

#include <iostream>

#include <typeinfo>

using namespace std;

```
int main()
{
    int num = 10;
    // converting int to double
    double numDouble = static_cast<double>(num);
    // printing data type
    cout << typeid(num).name() << endl;
    // typecasting
    cout << typeid(static_cast<double>(num)).name() << endl;
    // printing double type t
    cout << typeid(numDouble).name() << endl;
    return 0;
}
```

we have included the **"typeinfo"** library so that we can use **typeid()** function to check the data type. We have defined an integer variable 'num' and converted it into a double using static_cast. After that, we print the data types of variables and pass **static_cast<double>(num)** in **typeid()** function to check its data type.

2. dynamic_castdivBlock

The dynamic_cast operator is mainly used to perform downcasting (converting a pointer/reference of a base class to a derived class). It ensures type safety by performing a runtime check to verify the validity of the conversion.

Syntax of dynamic_cast

dynamic_cast <*new_type*> (expression);

If the conversion is not possible, **dynamic_cast** returns a **null pointer** (for pointer conversions) or throws a **bad_cast exception** (for reference conversions).

```
// C++ program to illustrate the dynamic_cast
#include <iostream>
using namespace std;
// Base Class
class Animal {
public:
	virtual void speak() const
	{
		cout << "Animal speaks." << endl;
	}
};
// Derived Class
class Dog : public Animal {
public:
	void speak() const override
	{
		cout << "Dog barks." << endl;
	}
};
// Derived Class
class Cat : public Animal {
public:
	void speak() const override
```

```
    {
        cout << "Cat meows." << endl;
    }
};
int main()
{
    // base class pointer to derived class object
    Animal* animalPtr = new Dog();
    // downcasting
    Dog* dogPtr = dynamic_cast<Dog*>(animalPtr);
    // checking if the typecasting is successfull
    if (dogPtr) {
        dogPtr->speak();
    }
    else {
        cout << "Failed to cast to Dog." << endl;
    }
    // typecasting to other dervied class
    Cat* catPtr = dynamic_cast<Cat*>(animalPtr);
    if (catPtr) {
        catPtr->speak();
    }
    else {
```

```
        cout << "Failed to cast to Cat." << endl;
    }
    delete animalPtr;
    return 0;
}
```

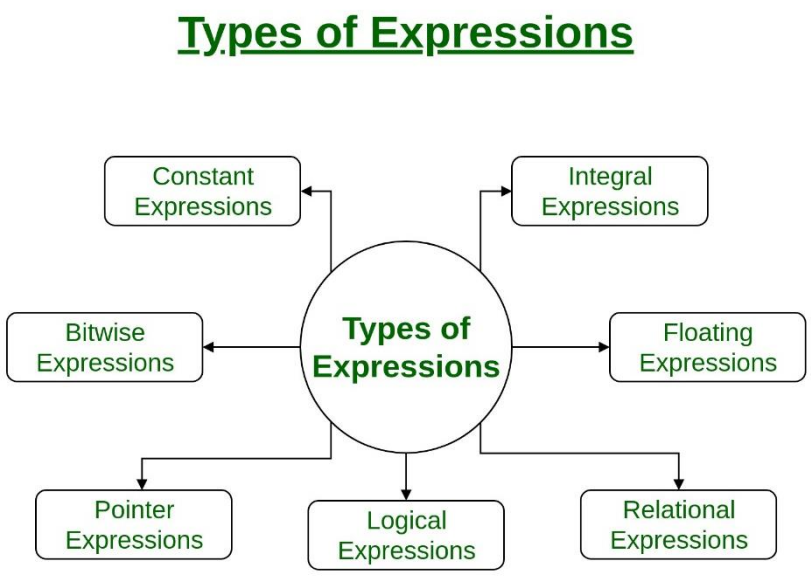
3. const_cast

The **const_cast** operator is used to modify the const or volatile qualifier of a variable. It allows programmers to temporarily remove the constancy of an object and make modifications. Caution must be exercised when using const_cast, as modifying a const object can lead to undefined behavior.

Syntax for const_cast

const_cast <*new_type*> (expression);

Expression: An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.



- **Constant expressions**: Constant Expressions consists of only constant values. A constant value is one that doesn't change.
  Examples:
  5, 10 + 5 / 6.0, 'x'

- **Integral expressions**: Integral Expressions are those which produce integer results after implementing all the automatic and explicit type conversions.
  Examples:
  x, x * y, x + int(5.0)

  where x and y are integer variables.

- **Floating expressions**: Float Expressions are which produce floating point results after implementing all the automatic and explicit type conversions.
  Examples:
  x + y, 10.75

  where x and y are floating point variables.

- **Relational expressions**: Relational Expressions yield results of type bool which takes a value true or false. When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as Boolean expressions.
  Examples:
  x <= y, x + y > 2

- **Logical expressions**: Logical Expressions combine two or more relational expressions and produces bool type results.
  Examples:
  x > y && x == 10, x == 10 || y == 5

- **Pointer expressions**: Pointer Expressions produce address values.
  Examples:
  &x, ptr, ptr++

  where x is a variable and ptr is a pointer.

- **Bitwise expressions**: Bitwise Expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits.
  Examples:
  x << 3

  shifts three bit position to left

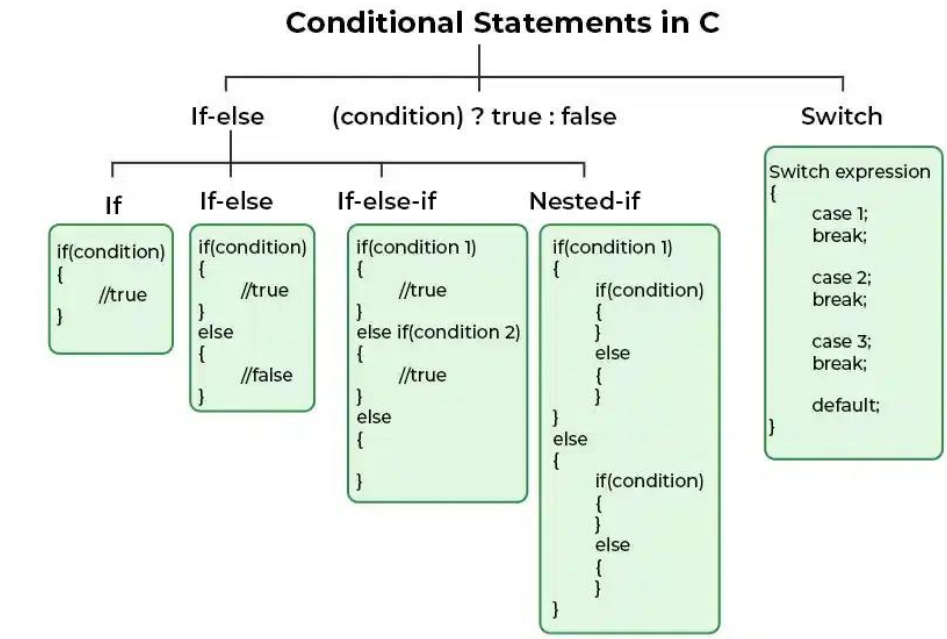
  y >> 1

  shifts one bit position to right.

  Shift operators are often used for multiplication and division by powers of two.

Conditional control structure in c++

conditional statements (also known as decision control structures) such as if, if else, switch, etc. are used for decision-making purposes in C/C++ programs. They are also known as Decision-Making Statements and are used to evaluate one or more conditions and make the decision whether to execute a set of statements or not. These decision-making statements in programming languages decide the direction of the flow of program execution.



1. if in C/C++

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax of if Statement

```
if(condition)

{

  // Statements to execute if

  // condition is true

}
```

For Example

// C++ program to illustrate If statement

```
#include <iostream>

using namespace std;

int main()

{

        int i = 10;

        if (i > 15) {

                cout << "10 is greater than 15";

        }

        cout << "I am Not in if";

}
```

2. if-else in C/C++

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else when the condition is false? Here comes the C *else* statement. We can use the *else* statement with the *if* statement to execute a block of code when the condition is false. The if-else statement consists of two blocks, one for false expression and one for true expression.

Syntax of if else in C/C++

```
if (condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
   // condition is false
}
```

For Example

```
// C++ program to illustrate if-else statement

#include <iostream>

using namespace std;

int main()

{

        int i = 20;

        if (i < 15)

                cout << "i is smaller than 15";

        else

                cout << "i is greater than 15";

        return 0;

}
```

3. Nested if-else in C/C++

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, both C and C++ allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax of Nested if-else

```
if (condition1)

{

  // Executes when condition1 is true

  if (condition2)

  {

    // Executes when condition2 is true
```

```
    }
    else
    {
        // Executes when condition2 is false
}
```

Example:

```
// C++ program to illustrate nested-if statement
#include <iostream>
using namespace std;
int main()
{
        int i = 10;
        if (i == 10) {
                // First if statement
                if (i < 15)
                        cout << "i is smaller than 15\n";
                // Nested - if statement
                // Will only be executed if
                // statement above is true
                if (i < 12)
                        cout << "i is smaller than 12 too\n";
                else
                        cout << "i is greater than 15";
```

```
    }

    return 0;

}
```

4. if-else-if Ladder in C++

The if else if statements are used when the user has to decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. if-else-if ladder is similar to the switch statement.

Syntax of if-else-if Ladder

```
if (condition)
   statement;
else if (condition)
   statement;
.
.

else
   statement;
```

Example of if-else-if Ladder

```
// C++ program to illustrate if-else-if ladder

#include <iostream>

using namespace std;



int main()

{

    int i = 20;
```

```
        if (i == 10)
                cout << "i is 10";
        else if (i == 15)
                cout << "i is 15";
        else if (i == 20)
                cout << "i is 20";
        else
                cout << "i is not present";
}
```

5. switch Statement in C/C++

The switch case statement is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

```
switch (expression) {
   case value1:
      statements;
   case value2:
      statements;
   ....
   ....
   ....
   default:
      statements;
}
```

Note: *The switch expression should evaluate to either integer or character. It cannot evaluate any other data type.*

```
// C Program to illustrate the use of switch statement

#include <iostream>

using namespace std;
```

```
// driver code
int main()
{
    // variable to be used in switch statement
    int var = 2;

    // declaring switch cases
    switch (var) {
    case 1:
        cout << "Case 1 is executed";
        break;
    case 2:
        cout << "Case 2 is executed";
        break;
    default:
        cout << "Default Case is executed";
        break;
    }
    return 0;
}
```

6. Conditional Operator in C/C++

The conditional operator is used to add conditional code in our program. It is similar to the if-else statement. It is also known as the ternary operator as it works on three operands.

Syntax of Conditional Operator

(condition) ? [true_statements] : [flase_statements];

For Example:

// C++ Program to illustrate the use of conditional operator

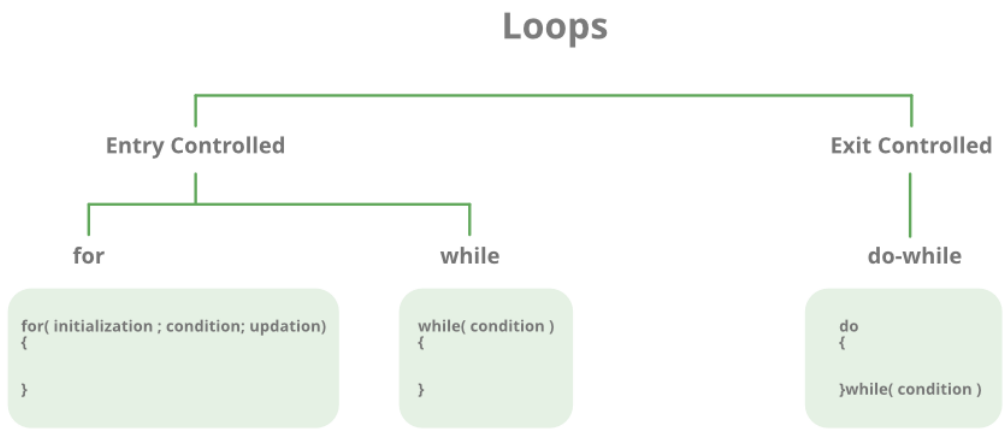
```
#include <iostream>
using namespace std;
// driver code
int main()
{
    int var;
    int flag = 0;
    // using conditional operator to assign the value to var
    // according to the value of flag
    var = flag == 0 ? 25 : -25;
    cout << "Value of var when flag is 0: " << var << endl;
    // changing the value of flag
    flag = 1;
    // again assigning the value to var using same statement
    var = flag == 0 ? 25 : -25;
    cout << "Value of var when flag is NOT 0: " << var;
    return 0;
}
```

C++ Loop Types

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

There are mainly two types of loops:
Entry Controlled loops: In this type of loop, the test condition is tested before entering the loop body. **For Loop** and **While Loop** is entry-controlled loops.
Exit Controlled Loops: In this type of loop the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. the do-while **loop** is exit controlled loop.



For Loop-
A *For loop* is a repetition control structure that allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.
Syntax:

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

Explanation of the Syntax:
Initialization statement: This statement gets executed only once, at the beginning of the for loop. You can enter a declaration of multiple variables of one type, such as int x=0, a=1, b=2. These variables are only valid in the scope

of the loop. Variable defined before the loop with the same name are hidden during execution of the loop.
Condition: This statement gets evaluated ahead of each execution of the loop body, and abort the execution if the given condition get false.
Iteration execution: This statement gets executed after the loop body, ahead of the next condition evaluated, unless the for loop is aborted in the body (by break, goto, return or an exception being thrown.)

Example:

```
// C++ program to Demonstrate for loop

#include <iostream>

using namespace std;

int main()

{

	for (int i = 1; i <= 5; i++)

	{

		cout << "Hello World\n";

	}

	return 0;

}
```

Example:

```
#include <iostream>

using namespace std;


int main() {


int arr[] {40, 50, 60, 70, 80, 90, 100};
```

```
for (auto element: arr){

cout << element << " ";

}

return 0;

}
```

While Loop-

While studying **for loop** we have seen that the number of iterations is ***known beforehand***, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where **we do not know** the exact number of iterations of the loop **beforehand**. The loop execution is terminated on the basis of the test conditions.

We have already stated that a loop mainly consists of three statements – initialization expression, test expression, and update expression. The syntax of the three loops – For, while, and do while mainly differs in the placement of these three statements.

Syntax:

```
initialization expression;
while (test_expression)
{
  // statements

  update_expression;
}
```

For Example:

```
// C++ program to Demonstrate while loop

#include <iostream>

using namespace std;

int main()

{

        // initialization expression

        int i = 1;

        // test expression

        while (i < 6) {
```

```
            cout << "Hello World\n";


            // update expression
            i++;
        }
        return 0;
}
```

Do-while loop

In Do-while loops also the loop execution is terminated on the basis of test conditions. The main difference between a do-while loop and the while loop is in the do-while loop the condition is tested at the end of the loop body, i.e do-while loop is exit controlled whereas the other two loops are entry-controlled loops.

Note: In a do-while loop, the loop body will ***execute at least once*** irrespective of the test condition.

Syntax:

```
initialization expression;
do
{
  // statements
  update_expression;
} while (test_expression);
```

For Example

```
// C++ program to Demonstrate do-while loop

#include <iostream>

using namespace std;

int main()

{

        int i = 2; // Initialization expression

        do {

                // loop body
```

```
        cout << "Hello World\n";
        // update expression
        i++;
    } while (i < 1); // test expression
    return 0;
}
```