Procedural Programming

It is defined as a programming language derived from the structure programming and based on calling procedures. The procedures are the functions, routines, or subroutines that consist of the computational steps required to be carried. It follows a step-by-step approach in order to break down a task into a set of variables and routines via a sequence of instructions.

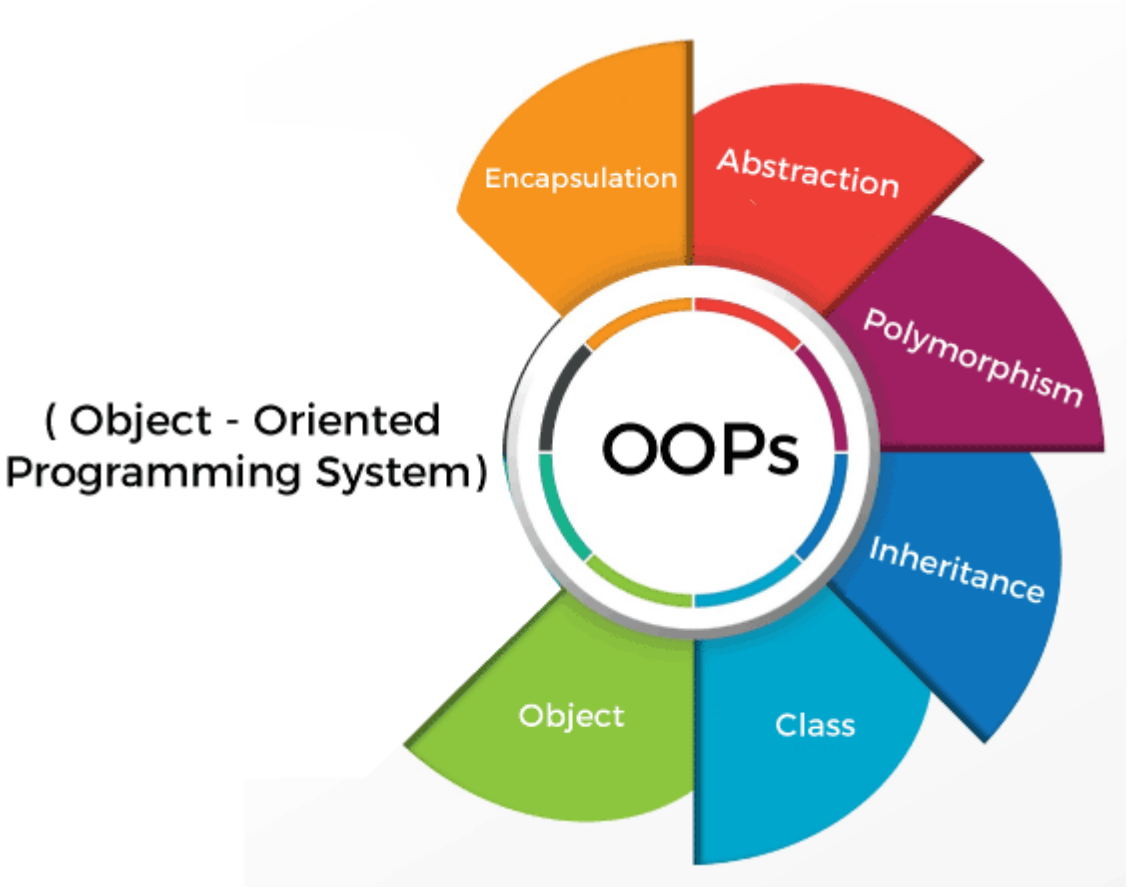
During the program's execution, a procedure can be called at any point, either by other procedures or by itself. The examples of procedural programming are ALGOL, COBOL, BASIC, PASCAL, FORTRAN, and C.

As compared to object-oriented programming, procedural programming is less secure. Procedural programming follows a top-down approach during the designing of a program. It gives importance to the concept of the function and divides the large programs into smaller parts or called as functions. Procedural programming is straightforward. Unlike object-oriented programming, there are no access modifiers introduced in procedural programming.

Object-oriented programming

Object-oriented programming is a computer programming design philosophy or methodology that organizes/ models software design around data or objects rather than functions and logic. It includes two words, "object" and "oriented". In a dictionary object is an article or entity that exists in the real world. The meaning of oriented is interested in a particular kind of thing or entity. In layman's terms, it is a programming pattern that rounds around an object or entity.

The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language. **Smalltalk** is said to be the first truly object-oriented programming language.



OOP is said to be the most popular programming model among developers. It is well suited for programs that are large, complex, and actively updated or maintained. It makes the development and maintenance of software easy by providing major concepts such as abstraction, inheritance, polymorphism, and encapsulation. These four are also the four pillars of an object-oriented programming system.

OOPs, provide the ability to simulate real-world events much more effectively. We can provide the solution to real-world problems if we are using the Object-Oriented Programming language. OOPs, provide data hiding, whereas, in Procedure-oriented programming language, global data can be accessed from anywhere.

The **examples** of OOPs are - C#, Python, C++, Java, PHP, Scala, Perl, etc.

Procedural programming v/s Object-oriented programming

Now, let's see the comparison between Procedural programming and object-oriented programming. We are comparing both terms on the basis of some characteristics. The difference between both languages are tabulated as follows -

S. no.	On the basis of	Procedural Programming	Object-oriented programming
1.	**Definition**	It is a programming language that is derived from structure programming and based upon the concept of calling procedures. It follows a step-by-step approach in order to break down a task into a set of variables and routines via a sequence of instructions.	Object-oriented programming is a computer programming design philosophy or methodology that organizes/ models software design around data or objects rather than functions and logic.
2.	**Security**	It is less secure than OOPs.	Data hiding is possible in object-oriented programming due to abstraction. So, it is more secure than procedural programming.

3.	**Approach**	It follows a top-down approach.	It follows a bottom-up approach.
4.	**Data movement**	In procedural programming, data moves freely within the system from one function to another.	In OOP, objects can move and communicate with each other via member functions.
5.	**Orientation**	It is structure/procedure-oriented.	It is object-oriented.
6.	**Access modifiers**	There are no access modifiers in procedural programming.	The access modifiers in OOP are named as private, public, and protected.
7.	**Inheritance**	Procedural programming does not have the concept of inheritance.	There is a feature of inheritance in object-oriented programming.
8.	**Code reusability**	There is no code reusability present in procedural programming.	It offers code reusability by using the feature of inheritance.
9.	**Overloading**	Overloading is not possible in procedural programming.	In OOP, there is a concept of function overloading and operator overloading.
10.	**Importance**	It gives importance to functions over data.	It gives importance to data over functions.
11.	**Virtual class**	In procedural programming, there are no virtual classes.	In OOP, there is an appearance of virtual classes in inheritance.
12.	**Complex problems**	It is not appropriate for complex problems.	It is appropriate for complex problems.
13.	**Data hiding**	There is not any proper way for data hiding.	There is a possibility of data hiding.
14.	**Program division**	In Procedural programming, a program is divided into small programs that are referred to as functions.	In OOP, a program is divided into small parts that are referred to as objects.
15.	**Examples**	Examples of Procedural programming include C, Fortran, Pascal, and VB.	The examples of object-oriented programming are - .NET, C#, Python, Java, VB.NET, and C++.

Difference between C an C++

C++ is often viewed as a superset of C. C++ is also known as a “C with class”. 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++.

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++ supports polymorphism, encapsulation, and inheritance because it is an object oriented programming language.
C is (mostly) a subset of C++.	C++ is (mostly) a superset of C.

Number of keywords in C: * C90: 32 * C99: 37 * C11: 44 * C23: 59	Number of keywords in C++: * C++98: 63 * C++11: 73 * C++17: 73 * C++20: 81
For the development of code, C supports procedural programming.	C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.
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++.
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

Type punning with unions is allows (C99 and later)	Type punning with unions is undefined behavior (except in very specific circumstances)
Named initializers may appear out of order	Named initializers must match the data layout of the struct
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++

Basic Concepts of OOP.

Object

An object is a real-world element in an object–oriented environment that may have a physical or a conceptual existence. Each object has −

- Identity that distinguishes it from other objects in the system.
- State that determines characteristic properties of an object as well as values of properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects can be modeled according to the needs of the application. An object may have a physical existence, like a customer, a car, etc.; or an intangible conceptual existence, like a project, a process, etc.

Class

A class represents a collection of objects having same characteristic properties that exhibit common behavior. It gives the blueprint or the description of the objects that can be created from it. Creation of an object as a member of a class is called instantiation. Thus, an object is an instance of a class.

The constituents of a class are −

- A set of attributes for the objects that are to be instantiated from the class. Generally, different objects of a class have some difference in the values of the attributes.
- A set of operations that portray the behavior of the objects of the class. Operations are also referred as functions or methods.

Example

Let us consider a simple class, Circle, that represents the geometrical figure circle in a two–dimensional space. The attributes of this class can be identified as follows −

- x–coord, to denote x–coordinate of the center

y–coord, to denote y–coordinate of the center
a, to denote the radius of the circle

Some of its operations can be defined as follows –

findArea(), a method to calculate area
findCircumference(), a method to calculate circumference
scale(), a method to increase or decrease the radius

Encapsulation

Encapsulation is the process of binding both attributes and methods together within a class. Through encapsulation, the internal details of a class can be hidden from outside. It permits the elements of the class to be accessed from outside only through the interface provided by the class.

Polymorphism

Polymorphism is originally a Greek word that means the ability to take multiple forms. In object-oriented paradigm, polymorphism implies using operations in different ways, depending upon the instances they are operating upon. Polymorphism allows objects with different internal structures to have a common external interface. Polymorphism is particularly effective while implementing inheritance.

Example

Let us consider two classes, Circle and Square, each with a method findArea(). Though the name and purpose of the methods in the classes are same, the internal implementation, i.e., the procedure of calculating an area is different for each class. When an object of class Circle invokes its findArea() method, the operation finds the area of the circle without any conflict with the findArea() method of the Square class.

Relationships

In order to describe a system, both dynamic (behavioral) and static (logical) specification of a system must be provided. The dynamic specification describes the relationships among objects e.g. message passing. And static specification describe the relationships among classes, e.g. aggregation, association, and inheritance.

Message Passing

Any application requires a number of objects interacting in a harmonious manner. Objects in a system may communicate with each other by using message passing. Suppose a system has two objects – obj1 and obj2. The object obj1 sends a message to object obj2, if obj1 wants obj2 to execute one of its methods.

Composition or Aggregation

Aggregation or composition is a relationship among classes by which a class can be made up of any combination of objects of other classes. It allows objects to be placed directly within the body of other classes. Aggregation is referred as a "part–of" or "has–a" relationship, with the ability to navigate from the whole to its parts. An aggregate object is an object that is composed of one or more other objects.

Association

Association is a group of links having common structure and common behavior. Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association. The Degree of an association denotes the number of classes involved in a connection. The degree may be unary, binary, or ternary.

- A unary relationship connects objects of the same class.
- A binary relationship connects objects of two classes.
- A ternary relationship connects objects of three or more classes.

Inheritance

It is a mechanism that permits new classes to be created out of existing classes by extending and refining its capabilities. The existing classes are called the base classes/parent classes/super-classes, and the new classes are called the derived classes/child classes/subclasses.

The subclass can inherit or derive the attributes and methods of the super-class (es) provided that the super-class allows so. Besides, the subclass may add its own attributes and methods and may modify any of the super-class methods. Inheritance defines a "is – a" relationship.

Example

From a class Mammal, a number of classes can be derived such as Human, Cat, Dog, Cow, etc. Humans, cats, dogs, and cows all have the distinct characteristics of mammals. In addition, each has its own particular characteristics. It can be said that a cow "is – a" mammal.

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and ignoring the details. Data abstraction refers to providing only essential information about the data to the outside world, ignoring unnecessary details or implementation.

Consider a ***real-life example of a man driving a car***. The man only knows that pressing the accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car.

Types of Abstraction:

1. **Data abstraction –** This type only shows the required information about the data and ignores unnecessary details.

2. **Control Abstraction –** This type only shows the required information about the implementation and ignores unnecessary details.

Abstraction using Classes

We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.

Abstraction in Header files

One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

Abstraction using Access Specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of the code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that define the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be

marked as public. And these public members can access the private members as they are inside the class.

OO Analysis

In object-oriented analysis phase of software development, the system requirements are determined, the classes are identified, and the relationships among classes are acknowledged. The aim of OO analysis is to understand the application domain and specific requirements of the system. The result of this phase is requirement specification and initial analysis of logical structure and feasibility of a system.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modeling, dynamic modeling, and functional modeling.

Object Modeling

Object modeling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modeling can be visualized in the following steps −

- Identify objects and group into classes
- Identify the relationships among classes
- Create a user object model diagram
- Define a user object attributes
- Define the operations that should be performed on the classes

Applications of OOPs

OOPs is the most important and flexible paradigm of modern programming. It is specifically useful in modeling real-world problems. Below are some applications of OOPs:

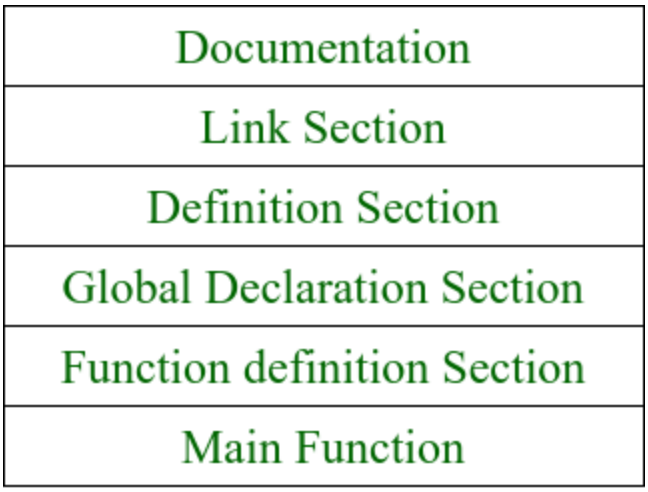
- **Real-Time System design:** Real-time system inherits complexities and makes it difficult to build them. OOP techniques make it easier to handle those complexities.
- **Hypertext and Hypermedia:** Hypertext is similar to regular text as it can be stored, searched, and edited easily. Hypermedia on 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 modeling 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.

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:



Skeleton of C Program

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:**

C++

```
/*    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, constants, 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>. These standard streams

process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

#include<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.
- In the below snippets, namespace std contains declarations for cout, cin, endl, etc. statements.

```
using namespace std;
```

- Namespaces can be accessed in multiple ways:
    - using namespace std;
    - 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.

- In #define is a compiler directive which tells the compiler whenever the message is found to replace it with “Factorial\n”.
- typedef int INTEGER; this statement tells the compiler that whenever you will encounter INTEGER replace it by int and as you have declared INTEGER as datatype you cannot use it as an identifier.

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.
- This part of the program can be written after the main function but for this, write the function prototype in this section for the function which for you are going to write code after the main function.

// Documentation Section

```
/* 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 a number
        iterate over the range from number to 1
*/
// Linking Section
#include <iostream>
using namespace std;

// Definition Section
#define msg "FACTORIAL\n"
typedef int INTEGER;

// Global Declaration Section
INTEGER num = 0, fact = 1, storeFactorial = 0;

// Function Section
INTEGER factorial(INTEGER num)
{
        // Iterate over the loop from
        // num to one
        for (INTEGER i = 1; i <= num; i++) {
                fact *= i;
        }

        // Return the factorial
        return fact;
}

// Main Function
INTEGER main()
{
```

```
        // Given number Num
        INTEGER Num = 5;

        // Function Call
        storeFactorial = factorial(Num);
        cout << msg;

        // Print the factorial
        cout << Num << "! = "
                << storeFactorial << endl;

        return 0;
}
```

Main Function:

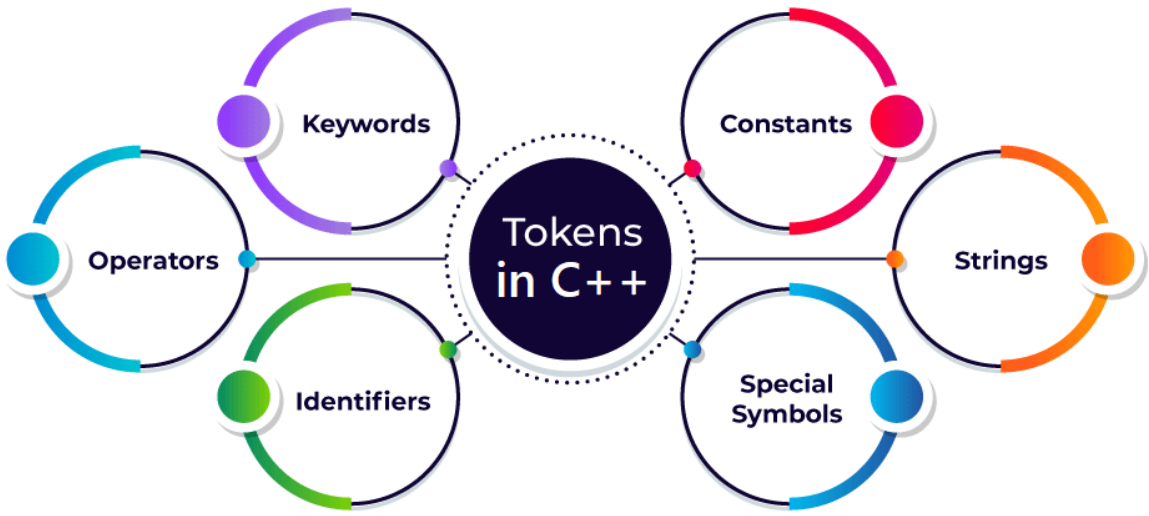
- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.
- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces {} 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.

Tokens

In C++, tokens can be defined as the smallest building block of C++ programs that the compiler understands. Every word in a C++ source code can be considered a token

We have several types of tokens each of which serves a specific purpose in the syntax and semantics of C++. Below are the main types of tokens in C++:

1. Identifiers
2. Keywords
3. Constants
4. Strings
5. Special Symbols
6. Operators



1. Identifiers

In C++, entities like variables, functions, classes, or structs must be given unique names within the program so that they can be uniquely identified. The unique names given to these entities are known as identifiers.

It is recommended to choose valid and relevant names of identifiers to write readable and maintainable programs. Keywords cannot be used as an identifier because they are reserved words to do specific tasks. In the below example, “first_name’ is an identifier.

```
string first_name = "Raju";
```

We have to follow a set of rules to define the name of identifiers as follows:

1. An identifier can only begin with a letter or an underscore(_).
2. An identifier can consist of letters (A-Z or a-z), digits (0-9), and underscores (_). White spaces and Special characters can not be used as the name of an identifier.
3. Keywords cannot be used as an identifier because they are reserved words to do specific tasks. For example, string, int, class, struct, etc.
4. Identifier must be unique in its namespace.
5. As C++ is a case-sensitive language so identifiers such as ‘first_name’ and ‘First_name’ are different entities.

Here are some examples of valid and invalid identifiers in C++:

Valid Identifiers	Invalid Identifiers

_name	#name (Cannot start with special character except '_')
Number89	2num (Cannot start with a digit)
first_name	first name (Cannot include space)
_last_name_	string (Cannot be same as a keyword)

2. Keywords

Keywords in C++ are the tokens that are the reserved words in programming languages that have their specific meaning and functionalities within a program. Keywords cannot be used as an identifier to name any variables.

For example, a variable or function cannot be named as 'int' because it is reserved for declaring integer data type.

There are 95 keywords reserved in C++. Some of the main Keywords are:

break	try	catch	Char	class	const	continue
default	delete	auto	Else	friend	for	float
long	new	operator	Private	protected	public	return
short	sizeof	static	This	typedef	enum	throw
mutable	struct	case	Register	switch	and	or
namespace	static_cast	goto	Not	xor	bool	do

double	int	unsigned	Void	virtual	union	while

Constants

Constants are the tokens in C++ that are used to define variables at the time of initialization and the assigned value cannot be changed after that.

We can define the constants in C++ in two ways that are using the ‘const’ keyword and ‘#define’ preprocessor directive

```
const data_type variable_name = value;
```

#define preprocessor can be used to define constant identifiers and the identifier is replaced with the defined value throughout the program where ever it is used. It is defined globally outside the main function.

Syntax

```
// The constant_Name is replaced by its value throughout the program where ever it is used

#define constant_Name value
```

Strings

In C++, a string is not a built-in data type like ‘int’, ‘char’, or ‘float’. It is a class available in the STL library which provides the functionality to work with a sequence of characters, that represents a string of text.

When we define any variable using the ‘string’ keyword we are actually defining an object that represents a sequence of characters. We can perform

various methods on the string provided by the string class such as length(), push_backk(), and pop_back().

Syntax of declaring a string

```
string variable_name;
```

Initialize the string object with string within the double inverted commas (“).

```
string variable_name = "This is string";
```

Special Symbols

Special symbols are the token characters having specific meanings within the syntax of the programming language. These symbols are used in a variety of functions, including ending the statements, defining control statements, separating items, and more.

Below are the most common special symbols used in C++ programming:

- Semicolon (;): It is used to terminate the statement.
- Square brackets []: They are used to store array elements.
- Curly Braces {}: They are used to define blocks of code.
- Scope resolution (::): Scope resolution operator is used to access members of namespaces, classes, etc.
- Dot (.): Dot operator also called member access operator used to access class and struct members.
- Assignment operator ‘=’: This operator is used to assign values to variables.
- Double-quote (“): It is used to enclose string literals.

- Single-quote (‘): It is used to enclose character literals

Operators

C++ operators are special symbols that are used to perform operations on operands such as variables, constants, or expressions. A wide range of operators is available in C++ to perform a specific type of operations which includes arithmetic operations, comparison operations, logical operations, and more.

For example (A+B), in which ‘A’ and ‘B’ are operands, and ‘+’ is an arithmetic operator which is used to add two operands.

Types of Operators

1. Unary Operators
2. Binary Operators
3. Ternary Operators

1. Unary Operators

Unary operators are used with single operands only. They perform the operations on a single variable. For example, increment and decrement operators.

- Increment operator (++): It is used to increment the value of an operand by 1.
- Decrement operator (–): It is used to decrement the value of an operand by 1.

2. Binary Operators

They are used with the two operands and they perform the operations between the two variables. For example (A<B), less than (<) operator compares A and B, returns true if A is less than B else returns false.

- **Arithmetic Operators: These operators perform basic arithmetic operations on operands. They include '+', '-', '*', '/', and '%'**
- **Comparison Operators: These operators are used to compare two operands and they include '==', '!=', '<', '>', '<=', and '>='.**
- **Logical Operators: These operators perform logical operations on boolean values. They include '&&', '||', and '!'.**
- **Assignment Operators: These operators are used to assign values to variables and they include 'variables', '-=', '*=', '/=', and '%='.**
- **Bitwise Operators: These operators perform bitwise operations on integers. They include'&', '|', '^', '~', '<<', and '>>'.**

3. Ternary Operator

The ternary operator is the only operator that takes three operands. It is also known as a conditional operator that is used for conditional expressions.

Syntax:

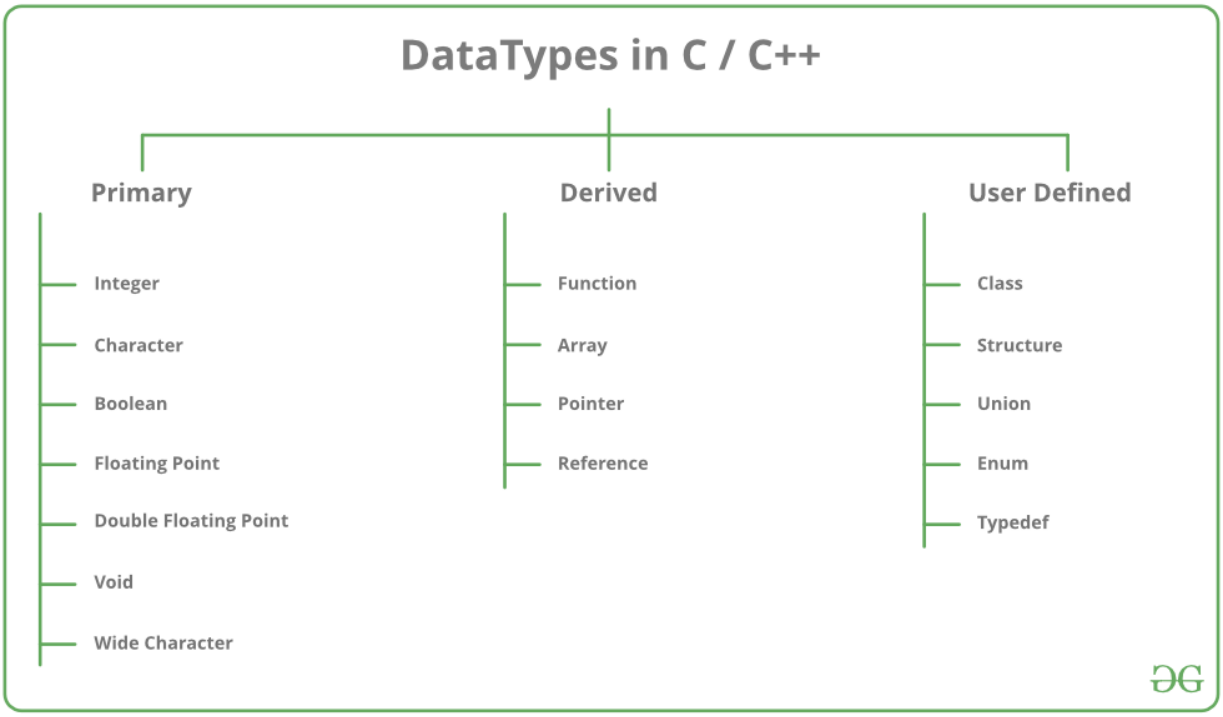
```
Expression1 ? Expression2 : Expression3;
```

If 'Expression1' became true then 'Expression2' will be executed otherwise 'Expression3' will be executed.

C++ Data types

C++ supports the following data types:

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



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 int
- Character char
- Boolean bool
- Floating Point float

- Double Floating Point double
- Valueless or Void void
- Wide Character w_chart

Wide char is similar to char data type, except that wide char take up twice the space and can take on much larger values as a result. char can take 256 values which corresponds to entries in the ASCII table. On the other hand, wide char can take on 65536 values which corresponds to UNICODE values which is a recent international standard which allows for the encoding of characters for virtually all languages and commonly used symbols.

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

- When a variable is declared as **reference**, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting '&' in the declaration.

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

int main() {
  int x = 10;

  // Reference Derived Type
  // ref is a reference to x.
  int& ref = x;

  // Value of x is now changed to 20
  ref = 20;
  cout << "x = " << x << endl;
```

```
    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << endl;

    return 0;
}
```

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

Class

A **Class** is the building block of C++'s Object-Oriented programming paradigm. 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

```
#include <bits/stdc++.h>
using namespace std;

class GfG {

    // Access specifier
```

```
public:

    // Data Member
    string name;

    // Member Function
    void printname() {
        cout << name;
    }
};

int main() {

    // Declare an object of class geeks
    GfG g;

    // Accessing data member
    g.name = "GeeksForGeeks";

    // Accessing member function
    g.printname();

    return 0;
}
```

Structure

A **Structure** is a user-defined data type like class. A structure creates a data type that can be used to group items of possibly different types into a single type.

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

```
// Declaring structure
struct A {
   int i;
   char c;
};

int main() {

   // Create an instance of structure
   A a;

   // Access array members
   a.i = 65;
   a.c = 'A';

   cout << a.c << ": " << a.i;

   return 0;
}
```

Union

Like structures , **union** is also user-defined data type used to group data of different type into a single type. But in union, all members share the same memory location

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

// Declaration of union is same as the structures
union A {
   int i;
     char c;
};
```

```
int main() {
    // A union variable t
    A a;

      // Assigning value to c, i will also
      // assigned the same
    a.c = 'A';

    cout << "a.i: " << a.i << endl;
      cout << "a.c: " << a.c;

    return 0;
}
```

Scope reso;ution 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;
}
```

3) To access a class's static variables.

```
// C++ program to show that :: can be used to access static
// members when there is a local variable with same name
#include<iostream>
using namespace std;

class Test
```

```
{
        static int x;
public:
        static int y;

        // Local parameter 'x' hides class member
        // 'x', but we can access it using ::
        void func(int x)
        {
        // We can access class's static variable
        // even if there is a local variable
        cout << "Value of static x is " << Test::x;

        cout << "\nValue of local x is " << x;
        }
};

// In C++, static members must be explicitly defined
// like this
int Test::x = 1;
int Test::y = 2;
```

```
int main()
{
        Test obj;
        int x = 3 ;
        obj.func(x);

        cout << "\nTest::y = " << Test::y;

        return 0;
```

```
}
```

new and delete Operators in C++ For Dynamic Memory

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap,** and non-static and local variables get memory allocated on **Stack**

How is it different from memory allocated to normal variables?

For normal variables like “int a”, “char str[10]”, etc, memory is automatically allocated and deallocated. For dynamically allocated memory like “int *p = new int[10]”, it is the programmer’s responsibility to deallocate memory when no longer needed. If the programmer doesn’t deallocate memory, it causes a memory leak (memory is not deallocated until the program terminates).
How is memory allocated/deallocated in C++?
C uses the malloc() and calloc() function to allocate memory dynamically at run time and uses a free() function to free dynamically allocated memory. C++ supports these functions and also has two operators **new** and **delete,** that perform the task of allocating and freeing the memory in a better and easier way.

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator

```
pointer-variable = new data-type;
```

Here, the pointer variable is the pointer of type data-type. Data type could be any built-in data type including array or any user-defined data type including structure and class.

Example:

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;

            OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

Allocate a block of memory: a new operator is also used to allocate a block(an array) of memory of type *data type*.

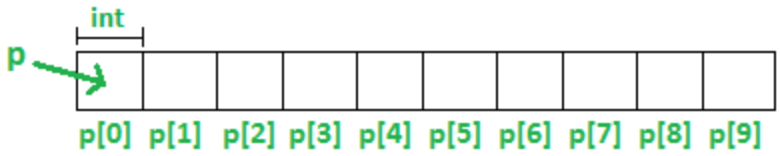
```
pointer-variable = new data-type[size];
```

where size(a variable) specifies the number of elements in an array.

Example:

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned top(a pointer). p[0] refers to the first element, p[1] refers to the second element, and so on.



Normal Array Declaration vs Using new
There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, that normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates

What if enough memory is not available during runtime?
If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type std::bad_alloc, unless "nothrow" is used with the new operator, in which case it returns a NULL pointer (scroll to section "Exception handling of new operator" in this article). Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

```
int *p = new(nothrow) int;
if (!p)
{
   cout << "Memory allocation failed\n";
}
```

delete operator
Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.
Syntax:

```
// Release memory pointed by pointer-variable
```

```
delete pointer-variable;
```

Here, the pointer variable is the pointer that points to the data object created by ***new***.

Examples:

```
delete p;
delete q;
```

To free the dynamically allocated array pointed by pointer variable, use the following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;
```

Example:

```
// It will free the entire array
// pointed by p.
delete[] p;
```

C++ Arrays

In C++, an array is a data structure that is used to store multiple values of similar data types in a contiguous memory location.

Properties of Arrays in C++

- An Array is a collection of data of the same data type, stored at a contiguous memory location.
- Indexing of an array starts from 0. It means the first element is stored at the 0th index, the second at 1st, and so on.
- Elements of an array can be accessed using their indices.
- Once an array is declared its size remains constant throughout the program.
- An array can have multiple dimensions.
- The number of elements in an array can be determined using the sizeof operator.

In C++, we can declare an array by simply specifying the data type first and then the name of an array with its size.

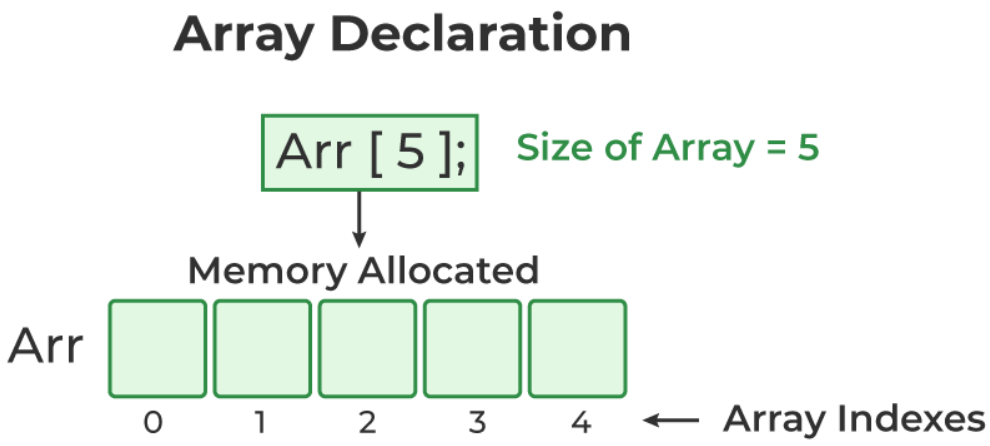
```
data_type array_name[Size_of_array];
```

Example

```
int arr[5];
```

Here,

- int: It is the type of data to be stored in the array. We can also use other data types such as char, float, and double.
- arr: It is the name of the array.
- 5: It is the size of the array which means only 5 elements can be stored in the array.



Initialization of Array in C++

In C++, we can initialize an array in many ways but we will discuss some most common ways to initialize an array. We can initialize an array at the time of declaration or after declaration.

1. Initialize Array with Values in C++

We have initialized the array with values. The values enclosed in curly braces '{}' are assigned to the array. Here, 1 is stored in arr[0], 2 in arr[1], and so on. Here the size of the array is 5.

```
int arr[5] = {1, 2, 3, 4, 5};
```

2. Initialize Array with Values and without Size in C++

We have initialized the array with values but we have not declared the length of the array, therefore, the length of an array is equal to the number of elements inside curly braces.

```
int arr[] = {1, 2, 3, 4, 5};
```

3. Initialize Array after Declaration (Using Loops)

We have initialized the array using a loop after declaring the array. This method is generally used when we want to take input from the user or we cant to assign elements one by one to each index of the array. We can modify the loop conditions or change the initialization values according to requirements.

```
for (int i = 0; i < N; i++) {
   arr[i] = value;
}
```

4. Initialize an array partially in C++

Here, we have declared an array 'partialArray' with size '5' and with values '1' and '2' only. So, these values are stored at the first two indices, and at the rest of the indices '0' is stored.

```
int partialArray[5] = {1, 2};
```

5. Initialize the array with zero in C++

We can initialize the array with all elements as '0' by specifying '0' inside the curly braces. This will happen in case of zero only if we try to initialize the array with a different value say '2' using this method then '2' is stored at the 0th index only.

```
int zero_array[5] = {0};
```

Accessing an Element of an Array in C++

Elements of an array can be accessed by specifying the name of the array, then the index of the element enclosed in the array subscript operator []. For example, arr[i].

Example 1: The C++ Program to Illustrate How to Access Array Elements

```
// C++ Program to Illustrate How to Access Array Elements

#include <iostream>

using namespace std;


int main()

{

        int arr[3];

        // Inserting elements in an array

        arr[0] = 10;

        arr[1] = 20;

        arr[2] = 30;

        // Accessing and printing elements of the array

        cout << "arr[0]: " << arr[0] << endl;

        cout << "arr[1]: " << arr[1] << endl;

        cout << "arr[2]: " << arr[2] << endl;

        return 0;
}
```

Update Array Element

To update an element in an array, we can use the index which we want to update enclosed within the array subscript operator and assign the new value.

```
arr[i] = new_value;
```

Traverse an Array in C++

We can traverse over the array with the help of a loop using indexing in C++. First, we have initialized an array 'table_of_two' with a multiple of 2. After that, we run a for loop from 0 to 9 because in an array indexing starts from zero. Therefore, using the indices we print all values stored in an array.

Example 2: The C++ Program to Illustrate How to Traverse an Array

```
// C++ Program to Illustrate How to Traverse an Array

#include <iostream>

using namespace std;

int main()

{

        // Initialize the array

        int table_of_two[10]

                = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };


        // Traverse the array using for loop

        for (int i = 0; i < 10; i++) {
```

```
            // Print the array elements using indexing
            cout << table_of_two[i] << " ";
        }
        return 0;
}
```

Size of an Array in C++

In C++, we do not have the length function as in Java to find array size but we can calculate the size of an array using sizeof() operator trick. First, we find the size occupied by the whole array in the memory and then divide it by the size of the type of element stored in the array. This will give us the number of elements stored in the array.

```
data_type size = sizeof(Array_name) / sizeof(Array_name[index]);
```

Example 3: The C++ Program to Illustrate How to Find the Size of an Array

```
// C++ Program to Illustrate How to Find the Size of an
// Array
#include <iostream>
using namespace std;
int main()
```

```
{
        int arr[] = { 1, 2, 3, 4, 5 };
        // Size of one element of an array
        cout << "Size of arr[0]: " << sizeof(arr[0]) << endl;
        // Size of array 'arr'
        cout << "Size of arr: " << sizeof(arr) << endl;
        // Length of an array
        int n = sizeof(arr) / sizeof(arr[0]);
        cout << "Length of an array: " << n << endl;
        return 0;
}
```