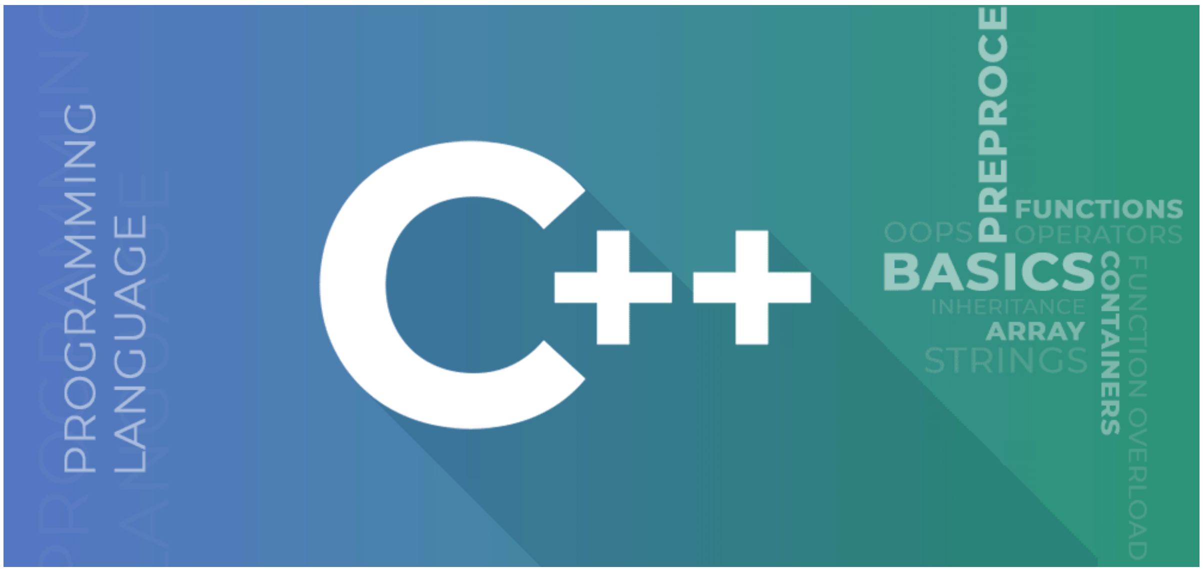
C++ Programming Language

Last Updated : 05 Mar, 2023

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac etc.



C++ Recent Articles!

C++ Interview Questions

C++ Programs

Basics, C vs C++, C++ vs Java, Input and Output, Operators, Arrays and Strings, Functions, References and Pointers, Dynamic memory allocation, Object Oriented Programming(OOP),Constructor and Destructor, Function Overloading, Operator Overloading, Virtual Functions, Exception Handling, Namespaces, Standard Template Library (STL), Inheritance, C++ Library, C++ Advanced, C++ in Competitive Programming, Puzzles, Interview Questions, Multiple Choice Questions

Basics

1. Setting up C++ Development Environment
2. Writing first C++ program(Practice)

Standard Template Library (STL)

Algorithms

1. Introduction to STL
2. Sorting

1. I/O Redirection in C++
2. Clearing The Input Buffer
3. Basic Input/Output(Practice)
4. cout << endl vs cout << “\n” in C++
5. Problem with scanf() when there is fgets()/gets()/scanf() after it
6. How to use getline() in C++ when there are blank lines in input?
7. scanf() and fscanf() in C – Simple Yet Poweful
8. Using return value of cin to take unknown number of inputs in C++
9. How to change the output of printf() in main() ?
10. Implementation of a Falling Matrix
11. What does buffer flush means in C++ ?
12. kbhit in C language
13. Code to generate the map of India

Operators

1. Operators in C++
2. Unary operators in C/C++
3. Conditionally assign a value without using conditional and arithmetic operators
4. Execution of printf with ++ operators
5. Set a variable without using Arithmetic, Relational or Conditional Operator
6. Scope Resolution Operator vs this pointer
7. Pre-increment (or pre-decrement)
8. new and delete operator in C++
9. CHAR_BIT in C
10. Casting operators | Set 1 (const_cast)

Arrays and Strings

1. Arrays in C/C++
2. Array of Strings
3. Multidimensional arrays in C/C++
4. Raw string literal
5. Counts of distinct consecutive sub-string of length two
6. Converting string to number and vice-versa
7. Find size of array in C/C++ without using sizeof

- forward_list::cbefore_begin() in C++ STL
- forward_list::unique() in C++ STL
- forward_list::before_begin() in C++ STL
- forward_list::cbefore_begin() in C++ STL
- forward_list::reverse() in C++ STL
- forward_list::max_size() in C++ STL
- forward_list::splice_after() in C++ STL
- list::empty() and list::size() in C++ STL
- list::front() and list::back() in C++ STL
- list::pop_front() and list::pop_back() in C++ STL
- list::push_front() and list::push_back() in C++ STL
- list push_front() function in C++ STL
- list pop_back() function in C++ STL
- list pop_front() function in C++ STL
- list reverse function in C++ STL
- list resize() function in C++ STL
- list size() function in C++ STL
- list max_size() function in C++ STL

4. Dequeue
5. Deque::empty() and deque::size() in C++ STL
6. Deque::pop_front() and deque::pop_back() in C++ STL
7. Deque::clear() and deque::erase() in C++ STL
8. Queue (Practice)
9. Queue::front() and queue::back() in C++ STL
10.
11. Queue::push() and queue::pop() in C++ STL
12. queue::empty() and queue::size() in C++ STL
13. Priority Queue
14. Stack (Practice)
15. Stack::push() and stack::pop() in C++ STL
16. Forward_list :: push_front() and forward_list :: pop_front() in C++ STL
17. Stack::top() in C++ STL
18. Stack::empty() and stack::size() in C++ STL
19. Set (Practice)

- Count number of unique Triangles using STL | Set 1 (Using set)
- std::istream_iterator and std::ostream_iterator in C++ STL

20. Std::next_permutation and prev_permutation in C++

8. How to quickly reverse a string in C++?
9. Tokenizing a string in C++
10. Getline() function and character array
11. Convert string to char array in C++
12. C++ string class and its applications , Set 2
13. How to create a dynamic 2D array inside a class in C++ ?
14. Lexicographically next permutation
15. Print size of array parameter
16. Split a string in C/C++, Python and Java
17. Stringstream in C++ and its applications
18. Strchr() function in C/C++
19. Isspace() in C/C++ and its application to count whitespace characters
20. Char* vs std:string vs char[] in C++
21. Std::lexicographical_compare() in C++STL
22. Std::string::at in C++
23. Std::substr() in C/C++
24. std::stol() and std::stoll() functions in C++
25. Extract all integers from string in C++
26. Strchr() function in C++ and its applications
27. Strcat() vs strncat() in C++
28. Strncat() function in C/C++
29. Strpbrk() in C
30. strcoll() in C/C++
31. Why strcpy and strncpy are not safe to use?

Functions

1. Functions in C++
2. Default Arguments
3. C function argument and return values
4. Inline Functions
5. Return from void functions
6. Returning multiple values from a function using Tuple and Pair
7. Function Call Puzzle
8. Functors
9. Ciel and floor functions in C++
10. Const member functions
11. atol(), atoll() and atof() functions in C/C++
12. swap() in C++
13. wmemmove() function in c++
14. wcscat() function in C++
15. wcscmp() function in C++ with Examples

21. Std::stoul and std::stoull in C++
22. Shuffle vs random_shuffle in C++
23. Difference between set, multiset, unordered_set, unordered_multiset
24. Check if a key is present in a C++ map or unordered_map
25. Std::stable_partition in C++
26. Valarray slice selector
27. Std::memchr in C++
28. Std::strncmp() in C++
29. Stable_sort() in C++ STL
30. Std::memcmp() in C++
31. Std::memset in C++
32. Std::bucket_count and std::bucket_size in unordered_map in C++
33. Map of pairs in STL
34. Range-based for loop in C++
35. Std::includes() in C++ STL
36. Std::set_symmetric_difference in C++
37. Std::sort_heap in C++
38. Map vs unordered_map in C++
39. Round() in C++
40. Modulus of two float or double numbers
41. Multiset
42. Map (Practice)
43. Heap using STL C++

Multimap

- Multimap in C++ Standard Template Library (STL)
- multimap::find() in C++ STL
- multimap::erase() in C++ STL
- map emplace() in C++ STL
- multimap::emplace_hint() in C++ STL
- multimap::emplace() in C++ STL
- multimap::count() in C++ STL
- multimap::find() in C++ STL
- multimap::erase() in C++ STL
- multimap::begin() and multimap::end() in C++ STL
- multimap::cbegin() and multimap::cend() in C++ STL
- map cbegin() and cend() function in C++ STL

16. wcscpy() function in C++ with Examples
17. wcslen() function in C++ with Examples
18. difftime() function in C++
19. asctime() function in C++
20. localtime() function in C++
21. scalbn() function in C++
22. isunordered() function in C++
23. isnormal() in C++
24. isinf() function in C++
25. quick_exit() function in C++ with Examples
26. ctime() Function in C/C++
27. clock() function in C/C++
28. nearbyint() function in C++
29. quick_exit() function in C++ with Examples
30. wcscmp() function in C++ with Examples
31. wcscpy() function in C++ with Examples
32. wcslen() function in C++ with Examples

Pointers and References

1. Pointers in C and C++
2. What is Array Decay in C++? How can it be prevented?
3. Opaque Pointer
4. References
5. Can references refer to invalid location?
6. Pass arguments by reference or pointer
7. Smart Pointers
8. 'this' pointer
9. Type of 'this' pointer
10. "delete this"
11. auto_ptr, unique_ptr, shared_ptr and weak_ptr
12. Dangling, Void , Null and Wild Pointers
13. Passing by pointer Vs Passing by Reference
14. NaN in C++ – What is it and how to check for it?
15. nullptr
16. Pointers vs References in C++

Dynamic memory allocation

1. new and delete operator in C++
2. malloc() vs new
3. delete() and free()
4. Std::get_temporary_buffer in C++

- multimap::crbegin() and multimap::crend() in C++ STL
- multimap size() function in C++ STL
- multimap lower_bound() function in C++ STL
- multimap swap() function in C++ STL
- multimap upper_bound() function in C++ STL
- multimap maxsize() in C++ STL
- multimap insert() in C++ STL
- multimap equal_range() in C++ STL

CPP-Math

- sinh() function in C++ STL
- cosh() function in C++ STL
- tanh() function in C++ STL
- acos() function in C++ STL
- asinh() function in C++ STL
- acosh() function in C++ STL
- atanh() function in C++ STL

More:

1. sort() in C++ STL
2. Strand sort
3. Type Inference in C++ (auto and decltype)
4. transform() in C++ STL
5. Variadic function templates in C++
6. Template Specialization
7. Implementing iterator pattern of a singly linked list
8. Binary Search functions in C++ STL
9. Descending order in Map and Multimap of C++ STL
10. Insertion and Deletion in STL Set C++
11. set::key_comp() in C++ STL
12. set value_comp() function in C++ STL
13. unordered_set get_allocator() in C++ STL with Examples

Inheritance

- What all is inherited from parent class in C++?
- Virtual Functions and Runtime Polymorphism in C++
- Multiple Inheritance in C++

Object Oriented Programming(OOP)

1. Object oriented design
2. Introduction to OOP in C++
3. Classes and Objects
4. Access Modifiers
5. Inheritance
6. Polymorphism
7. Encapsulation
8. Data Abstraction
9. Structure vs class
10. Can a C++ class have an object of self type?
11. Why is the size of an empty class not zero?
12. Static data members in C++
13. Some interesting facts about static member functions
14. Friend class and function
15. Local Class
16. Nested Classes
17. Simulating final class

Constructor and Destructor

1. Constructors
2. Copy Constructor
3. Destructors
4. Does compiler create default constructor when we write our own?
5. When should we write our own copy constructor?
6. When is copy constructor called?
7. Initialization of data members
8. Use of explicit keyword
9. When do we use Initializer List in?
10. Default Constructors
11. Private Destructor
12. Playing with Destructors
13. Copy elision
14. C++ default constructor | Built-in types
15. When does compiler create a default constructor and copy constructor?
16. Why copy constructor argument should be const in C++?
17. Advanced C++ | Virtual Constructor
18. Advanced C++ | Virtual Copy Constructor

- What happens when more restrictive access is given to a derived class method in C++?
- Object Slicing in C++
- Hiding of all overloaded methods in base class
- Inheritance and friendship
- Simulating final class

C++ Library

1. <random> file – generators and distributions
2. Array type manipulation
3. C++ programming and STL facts
4. Sqrt, sqrtl and sqrtf in C++
5. std::stod, std::stof, std::stold in C++
6. C program to demonstrate fork() and pipe()
7. Complex numbers in C++ | Set 1 Set 2
8. Inbuilt library functions for user Input
9. Rename function in C/C++
10. Chrono
11. valarray class
12. Floating Point Manipulation (fmod(), remainder(), remquo() ... in cmath) (Practice)
13. Character Classification: cctype
14. Snprintf() in C library
15. Boost::split in C++ library
16. Modulus of two float or double numbers
17. Is_trivial function in C++
18. Array sum in C++ STL
19. Div() function in C++
20. Exit() vs _Exit() in C and C++
21. Std::none_of in C++
22. Isprint() in C++
23. Iscntrl() in C++ and its application to find control characters
24. Std::partition_point in C++
25. Iterator Invalidation in C++
26. Fesetround() and fegetround() in C++ and their application
27. Rint(), rintf(), rintl() in C++
28. Hypot(), hypotf(), hypotl() in C++
29. Std::gslice | Valarray generalized slice selector
30. std::setbase, std::setw , std::setfill in C++
31. Strxfrm() in C/C++

19. When are static objects destroyed?
20. Is it possible to call constructor and destructor explicitly?

Function Overloading

1. Function Overloading
2. Functions that can't be overloaded
3. Function overloading and const keyword
4. Function overloading and return type
5. Does overloading work with Inheritance?
6. Can main() be overloaded
7. Function Overloading and float

Operator Overloading

1. Operator Overloading
2. Copy constructor vs assignment operator
3. When should we write our own assignment operator?
4. Operators that cannot be overloaded
5. Conversion Operators
6. Is assignment operator inherited?
7. Default Assignment Operator and References
8. Overloading stream insertion (<<) and extraction (>>) operators
9. Overloading array index operator []

Virtual Functions

1. Virtual Functions and Runtime Polymorphism
2. Default arguments and virtual function
3. Virtual functions in derived classes
4. Can static functions be virtual?
5. Virtual Destructor
6. Virtual Constructor
7. Virtual Copy Constructor
8. RTTI (Run-time type information)
9. Can virtual functions be private?
10. Inline virtual function
11. Pure Virtual Functions and Abstract Classes
12. Pure virtual destructor

Exception Handling

1. Exception Handling Basics
2. Stack Unwinding

32. Set position with seekg() in C++ language file handling
33. Strstr() in C/C++
34. Difftime() C library function
35. Socket Programming
36. Precision of floating point numbers in C++ (floor(), ceil(), trunc(), round() and setprecision())
37. <bit/stdc++.h> header file
38. std::string class in C++
39. Merge operations using STL in C++ (merge, includes, set_union, set_intersection, set_difference, ..)
40. std::partition in C++ STL
41. Ratio Manipulations in C++ | Set 1 (Arithmetic) , Set 2 (Comparison)
42. numeric header in C++ STL | Set 1 (accumulate() and partial_sum()), Set 2 (adjacent_difference(), inner_product() and iota())
43. Bind function and placeholders
44. Array class
45. Tuples
46. Regex (Regular Expression)
47. Common Subtleties in Vector STLs
48. Understanding constexpr specifier
49. unordered_multiset and its uses
50. unordered_multimap and its application
51. Populating a vector in C++ using fill() and fill_n()
52. Writing OS Independent Code in C/C++
53. C Program to display hostname and IP address
54. Database Connectivity using C/C++
55. C++ bitset and its application
56. unordered_map in STL and its applications
57. unorderd_set in STL and its applications
58. nextafter() and nexttoward()

C++ Advanced

1. User Defined Literal
2. Placement new operator
3. Advanced C++ with boost library
4. Copy-and-Swap Idiom

3. Catching base and derived classes as exceptions
4. Catch block and type conversion
5. Exception handling and object destruction

Namespace

1. Namespace in C++ | Set 1 (Introduction)
2. Set 2 (Extending namespace and Unnamed namespace)
3. Namespace in C++ | Set 3 (Accessing, creating header, nesting and aliasing)
4. Inline namespaces and usage of the “using” directive inside namespaces
5. Can namespaces be nested?

5. Zombie and Orphan Processes
6. Lambda expression
7. C++ | Signal Handling
8. Preventing Object Copy in C++
9. Command line arguments in C++

C++ in Competitive Programming

1. Writing C/C++ code efficiently in Competitive programming
2. Useful Array algorithms in C++ STL
3. searching in fork()
4. Data Type Ranges and their macros
5. Cin-Cout vs Scanf-Printf
6. getchar_unlocked() – faster input in C/C++ for Competitive Programming
7. C qsort() vs C++ sort()
8. Middle of three using minimum comparisons
9. Check for integer overflow on multiplication
10. Generating Test Cases (generate() and generate_n()

Puzzles

1. Can we call an undeclared function in?
2. Can we access global variable if there is a local variable with same name?
3. Can we use function on left side of an expression in C and C++?
4. Can we access private data members of a class without using a member or a friend function?
5. How to make a C++ class whose objects can only be dynamically allocated?
6. How to print “GeeksforGeeks” with empty main()
7. Print 1 to 100, without loop and recursion
8. C/C++ Tricky Programs
9. Print a number 100 times without using loop, recursion and macro expansion in C++
10. How to restrict dynamic allocation of objects
11. Sum of digits of a number in single statement
12. Write a URL in a C++ program
13. Zoom digits of an integer
14. Composite Design Pattern in C++
15. Assign value without any control statement

16. Printing pyramid pattern
17. How to swap two variables in one line in C/C++, Python and Java?
18. Program to shut down a computer

Interview Questions

1. Commonly Asked C++ Interview Questions | Set 1
2. Commonly Asked OOP Interview Questions | Set 1
3. **C/C++ Programs**

Quick Links:

- **Recent Articles on C++**
- **Practice Track on C++**
- C++ Output & Multiple Choice Questions

Related Articles

1. Ruby Programming Language
[https://www.geeksforgeeks.org/ruby-programming-language/?ref=rp]

2. Kotlin Programming Language
[https://www.geeksforgeeks.org/kotlin-programming-language/?ref=rp]

3. Perl Programming Language
[https://www.geeksforgeeks.org/perl-programming-language/?ref=rp]

4. Scala Programming Language
[https://www.geeksforgeeks.org/scala-programming-language/?ref=rp]

5. Learning the art of Competitive Programming
[https://www.geeksforgeeks.org/learning-art-competitive-programming/?ref=rp]

6. GATE and Programming Multiple Choice Questions with Solutions
[https://www.geeksforgeeks.org/gate-programming-multiple-choice-questions-solutions/?ref=rp]

7. Quizzes on Programming Languages
[https://www.geeksforgeeks.org/quizzes-on-programming-languages-gq/?ref=rp]

8. Articles on Programming Languages
[https://www.geeksforgeeks.org/articles-on-programming-languages-gq/?ref=rp]

Introduction to C++ Programming Language

Difficulty Level : Easy • Last Updated : 20 Feb, 2023

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

1. C++ is a high-level, general-purpose programming language designed for system and application programming. It was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C programming language. C++ is an object-oriented, multi-paradigm language that supports procedural, functional, and generic programming styles.
2. One of the key features of C++ is its ability to support low-level, system-level programming, making it suitable for developing operating systems, device drivers, and other system software. At the same time, C++ also provides a rich set of libraries and features for high-level application programming, making it a popular choice for developing desktop applications, video games, and other complex applications.
3. C++ has a large, active community of developers and users, and a wealth of resources and tools available for learning and using the language. Some of the key features of C++ include:
4. Object-Oriented Programming: C++ supports object-oriented programming, allowing developers to create classes and objects and to define methods and properties for these objects.
5. Templates: C++ templates allow developers to write generic code that can work with any data type, making it easier to write reusable and flexible code.
6. Standard Template Library (STL): The STL provides a wide range of containers and algorithms for working with data, making it easier to write efficient and effective code.
7. Exception Handling: C++ provides robust exception handling capabilities, making it easier to write code that can handle errors and unexpected situations.

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

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

1.Hello World:

C++

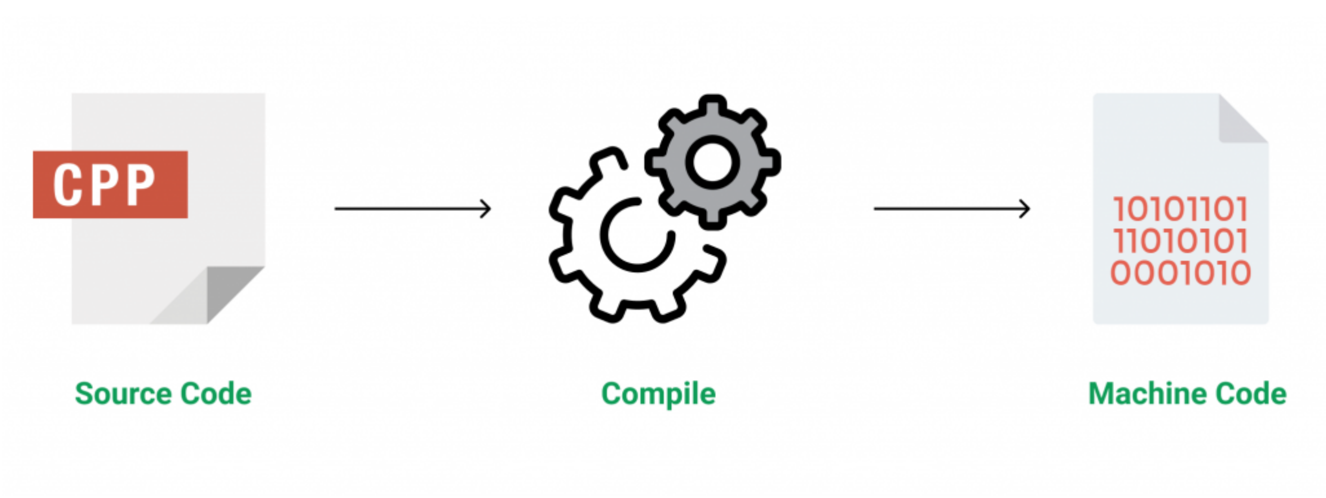
```cpp
#include <iostream>

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

Output

AD

```
Hello, World!
```



C++ is a middle-level language rendering it the advantage of programming low-level (drivers, kernels) and even higher-level applications (games, GUI, desktop apps etc.). The basic syntax and code structure of both C and C++ are the same.

Some of the ***features & key-points*** to note about the programming language are as follows:

- **Simple**: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.
- **Machine Independent but Platform Dependent**: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- **Mid-level language**: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- **Rich library support**: Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- **Speed of execution**: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.
- **Pointer and direct Memory-Access**: C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).
- **Object-Oriented**: One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and extensible programs. i.e.

Large-scale applications can be built. Procedural code becomes difficult to maintain as code-size grows.

- **Compiled Language**: C++ is a compiled language, contributing to its speed.

Applications of C++:

C++ finds varied usage in applications such as:

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

Here are some key points to keep in mind while working with C++:

1. Object-Oriented Programming: C++ is an object-oriented programming language, which means that it allows you to define classes and objects to model real-world entities and their behavior.
2. Strong Type System: C++ has a strong type system, which means that variables have a specific type and that type must be respected in all operations performed on that variable.
3. Low-level Access: C++ provides low-level access to system resources, which makes it a suitable choice for system programming and writing efficient code.
4. Standard Template Library (STL): The STL provides a large set of pre-written algorithms and data structures that can be used to simplify your code and make it more efficient.
5. Cross-platform Compatibility: C++ can be compiled and run on multiple platforms, including Windows, MacOS, and Linux, making it a versatile language for developing cross-platform applications.
6. Performance: C++ is a compiled language, which means that code is transformed into machine code before it is executed. This can result in faster execution times and improved performance compared to interpreted languages like Python.
7. Memory Management: C++ requires manual memory management, which can lead to errors if not done correctly. However, this also provides more control over the program's memory usage and can result in more efficient memory usage.
8. Syntax: C++ has a complex syntax that can be difficult to learn, especially for beginners. However, with practice and experience, it becomes easier to understand and work with.

These are some of the key points to keep in mind when working with C++. By understanding these concepts, you can make informed decisions and write effective code in this language.

Advantages of C++:

1. Performance: C++ is a compiled language, which means that its code is compiled into machine-readable code, making it one of the fastest programming languages.
2. Object-Oriented Programming: C++ supports object-oriented programming, which makes it easier to write and maintain large, complex applications.
3. Standard Template Library (STL): The STL provides a wide range of algorithms and data structures for working with data, making it easier to write efficient and effective code.
4. Platform Independent: C++ is a platform-independent language, meaning that code written in C++ can be compiled and run on a wide range of platforms, from desktop computers to mobile devices.
5. Large Community: C++ has a large, active community of developers and users, providing a wealth of resources and support for learning and using the language.

Disadvantages of C++:

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

Reference Books:

1. “The C++ Programming Language” by Bjarne Stroustrup
2. “Effective C++: 55 Specific Ways to Improve Your Programs and Designs” by Scott Meyers
3. “C++ Primer Plus” by Stephen Prata
4. “C++ For Dummies” by Stephen R. Davis
5. “Data Structures and Algorithm Analysis in C++” by Mark Allen Weiss
   Some interesting facts about C++:
   Here are some awesome facts about C++ that may interest you:
6. The name of C++ signifies the evolutionary nature of the changes from C. “++” is the C increment operator.
7. C++ is one of the predominant languages for the development of all kind of technical and commercial software.
8. C++ introduces Object-Oriented Programming, not present in C. Like other things, C++ supports the four primary features of OOP: encapsulation, polymorphism, abstraction, and inheritance.

9. C++ got the OOP features from Simula67 Programming language.
10. A function is a minimum requirement for a C++ program to run.(at least main() function)

Introduction to C++ | Sample Video for C++ Foundation Course | Geeks...

792

Related Articles

1. C Programming Language Standard
2. Tips and Tricks for Competitive Programmers | Set 2 (Language to be used for Competitive Programming)
3. Why Java Language is Slower Than CPP for Competitive Programming?
4. Introduction to Parallel Programming with OpenMP in C++
5. Why C++ is partially Object Oriented Language?
6. Convert C/C++ code to assembly language
7. kbhit in C language
8. ToDo App in C Language
9. Fast I/O for Competitive Programming
10. getchar_unlocked() – Faster Input in C/C++ For Competitive Programming

Next

Setting up C++ Development Environment

Setting up C++ Development Environment

Difficulty Level : Basic • Last Updated : 20 Mar, 2023

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features.
C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc. Before we start programming with C++. We will need an environment to be set up on our local computer to compile and run our C++ programs successfully. If you do not want to set up a local environment you can also use online IDEs for compiling your program.

Using Online IDE

IDE stands for an integrated development environment. IDE is a software application that provides facilities to a computer programmer for developing software. There are many online IDEs available that you can use to compile and run your programs easily without setting up a local development environment. The

ide.geeksforgeeks.org is one such IDE provided by GeeksforGeeks.

You can click on the Run on IDE button to run the program.

C++

```
// Using online ide of C++
#include <iostream>
using namespace std;

int main()
{
    cout << "Learning C++ at GeekforGeeks";
    return 0;
}
```

Output

```
Learning C++ at GeekforGeeks
```

Time Complexity: O(1)

Auxiliary Space: O(1)

Setting up a Local Environment

For setting up a C++ Integrated Development Environment (IDE) on your local machine you need to install two important software:

1. C++ Compiler
2. Text Editor

1. C++ Compiler

Once you have installed the text editor and saved your program in a file with the '.cpp' extension, you will need a C++ compiler to compile this file. A compiler is a computer program that converts high-level language into machine-understandable low-level language. In other words, we can say that it converts the source code written in a programming language into another computer language that the computer understands. For compiling a C++ program we will need a C++ compiler that will convert the source code written in C++ into machine codes. Below are the details about setting up compilers on different platforms.

Installing GNU GCC on Linux
We will install the GNU GCC compiler on Linux. To install and work with the GCC compiler on your Linux machine, proceed according to the below steps:

A. You have to first run the below two commands from your Linux terminal window:

```
sudo apt-get update
sudo apt-get install gcc
sudo apt-get install g++
```

B. This command will install the GCC compiler on your system. You may also run the below command:

```
sudo apt-get install build-essential
```

This command will install all the libraries which are required to compile and run a C++ program.

C. After completing the above step, you should check whether the GCC compiler is installed in your system correctly or not. To do this you have to run the below-given command from the Linux terminal:

```
g++ --version
```

D. If you have completed the above two steps without any errors, then your Linux environment is set up and ready to be used to compile C++ programs. In further steps, we will learn how to compile and run a C++ program on Linux using the GCC compiler.

E. Write your program in a text file and save it with any file name and.CPP extension. We have written a program to display “Hello World” and saved it in a file with the filename “helloworld.cpp” on the desktop.

F. Now you have to open the Linux terminal and move to the directory where you have saved your file. Then you have to run the below command to compile your file:

```
g++ filename.cpp -o any-name
```

G. *filename.cpp* is the name of your source code file. In our case, the name is “helloworld.cpp” and *any-name* can be any name of your choice. This name will be assigned

to the executable file which is created by the compiler after compilation. In our case, we choose *any-name* to be "hello".
We will run the above command as:

```
g++ helloworld.cpp -o hello
```

H. After executing the above command, you will see a new file is created automatically in the same directory where you have saved the source file and the name of this file is the name you chose as *any-name*.
Now to run your program you have to run the below command:

```
./hello
```

I. This command will run your program in the terminal windows.

2. Text Editor

Text Editors are the type of programs used to edit or write texts. We will use text editors to type our C++ programs. The normal extension of a text file is (.txt) but a text file containing a C++ program should be saved with a '.cpp' or '.c' extension. Files ending with the extension '.CPP' and '.C' are called source code files and they are supposed to contain source code written in C++ programming language. These extension helps the compiler to identify that the file contains a C++ program.
Before beginning programming with C++, one must have a text editor installed to write programs. Follow the below instructions to install popular code editors like VS Code and Code::Block on different Operating Systems like windows, Mac OS, etc.

1. Code::Blocks Installation

There are lots of IDE available that you can use to work easily with the C++ programming language. One of the popular IDE is **Code::Blocks**.

- To download Code::Blocks, select the setup package based on your OS from this link – Code::Blocks Setup Packages.
- Once you have downloaded the setup file of Code::Blocks from the given link open it and follow the instruction to install.
- After successfully installing Code::Blocks, go to *File* menu -> Select *New* and *create an Empty* file.
- Now write your C++ program in this empty file and save the file with a '.cpp' extension.
- After saving the file with the '.cpp' extension, go to the *Build* menu and choose the *Build and Run* option.

2. XCode Mac OS X Installation

If you are a Mac user, you can download Xcode as a code editor.

- To download Xcode you have to visit the apple website or you can search for it on the apple app store. You may follow the link – Xcode for MacOS to download Xcode. You will find all the necessary installation instructions there.
- After successfully installing Xcode, open the Xcode application.
- To create a new project. Go to File menu -> select New -> select Project. This will create a new project for you.
- Now in the next window, you have to choose a template for your project. To choose a C++ template choose the *Application* option which is under the *OS X* section on the left sidebar. Now choose *command-line tools* from available options and hit the *Next* button.
- On the next window provide all the necessary details like 'name of organization', 'Product Name, etc. But make sure to choose the language as C++. After filling in the details hit the next button to proceed to further steps.
- Choose the location where you want to save your project. After this choose the *main.cpp* file from the directory list on the left sidebar.
- Now after opening the main.cpp file, you will see a pre-written c++ program or template provided. You may change this program as per your requirement. To run your C++ program you have to go to the *Product* menu and choose the *Run* option from the dropdown.

Another very easy-to-use and most popular IDE nowadays, is **VSC(Visual Studio Code),** for both Windows and Mac OS.

3. Installing VS Code on Windows

Start with installing Visual Studio Code as per your windows. Open the downloaded file and click Run -> (Accept the agreement) Next -> Next -> Next -> (check all the options) -> Next ->Install->Finish.
Now you'll be able to see the Visual Studio Code icon on your desktop.

- Download the MinGW from the link.
- After Install, "Continue". Check all the Packages (Right Click -> Mark for Installation). Now, Click on Installation (left corner) -> Apply Changes. (This may take time)
- Open This PC -> C Drive -> MinGW -> Bin. (Copy this path)
- Right, Click on "This PC" -> Properties -> Advanced System Setting -> Environment variables -> (Select PATH in System variables) -> Edit -> New -> Paste the path here and OK.
- Go to Visual Studio Code, and Install some useful extensions (from the right sidebar, last icon(probably))-
  - C/C++
  - Code Runner

- Now, Go to Setting -> Settings -> Search for Terminal -> Go to the end of this page -> Check [Code-runner: Run In Terminal]

Yay! You are good to go now. Open any folder, create new files and Save them with the extension ".cpp".

4. Installing VS Code on Mac OS

First of all, Install Visual Studio Code for Mac OS using this link. Then We'll install the compiler MinGW. For this, we first need to install Homebrew.

To install Homebrew, Open Terminal (cmd + space). Write Terminal and hit Enter. In cmd copy the given command

```
arch -x86_64 ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install)" <
/dev/null 2> /dev/null
```

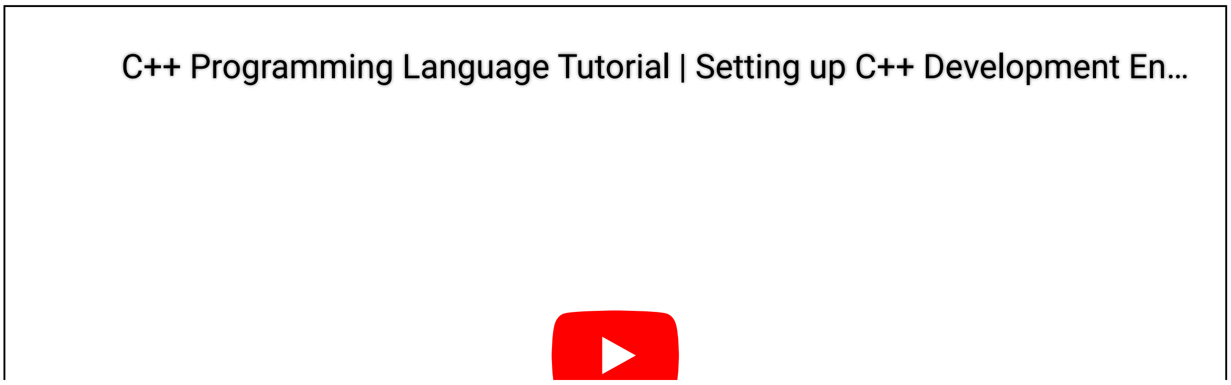
This will download and install HomeBrew on your Mac system. This process may take time.

Now We'll install the MinGW compiler on Mac OS. Paste the given command in the terminal and press Enter.

```
arch -x86_64 brew install MinGW-w64
```

- This is also time taking process so be patient!
- Go to Visual Studio Code, and Install some useful extensions (from the right sidebar, last icon(probably))-
  - C/C++
  - Code Runner
- Now, Go to Setting -> Settings -> Search for Terminal -> Go to the end of this page -> Check [Code-runner: Run In Terminal]

Yay! You are good to go now. Now open any folder, create new files, and Save them with the extension ".cpp".


C++ Programming Language Tutorial | Setting up C++ Development En...

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

914

Related Articles

1. Setting up a C++ Competitive Programming Environment
2. Setting up Sublime Text for C++ Competitive Programming Environment
3. Different Ways to Setting Up Environment For C++ Programming in Mac
4. Installing MinGW Tools for C/C++ and Changing Environment Variable
5. Top 10 Programming Languages for Blockchain Development
6. Setting Up Sublime Text For Competitive Coding in C++14 on Ubuntu
7. Setting up Sublime Text For Competitive Programming (C++) Using Fast Olympic Coding Plugin
8. C++ Error - Does not name a type
9. Execution Policy of STL Algorithms in Modern C++
10. C++ Program To Print Pyramid Patterns

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Basic

C++ Programming Basics

Difficulty Level : Easy • Last Updated : 11 Mar, 2023

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented, and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac, etc.

Before explaining the basics of C++, we would like to clarify two more ideas: **low-level** and **high-level**. To make it easy to understand, let's consider this scenario – when we go to the Google search engine and search for some queries, Google displays some websites according to our question. Google does this for us at a very high level. We don't know what's happening at the low level until we look into Google servers (at a low level) and further to the level where the data is in the form of 0s/1s. The point we want to make here is that a low level means nearest to the hardware, and a high level means farther from the hardware with a lot of layers of abstraction. **C ++ is considered a low-level language** as it is closer to hardware than most general-purpose programming languages. However to become proficient in any programming language, one Firstly needs to understand the basics of that language.

Basics of C++ Programming

1. Basic Syntax and First Program in C++

Learning C++ programming can be simplified into writing your program in a text editor and saving it with the correct extension(.CPP, C, CP), and compiling your program using a compiler or online IDE. The "Hello World" program is the first step toward learning any programming language and is also one of the simplest programs you will learn.

We can learn more about C++ Basic Syntax here – C++ Basic Syntax

2. Basic Input and Output in C++

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. The two methods **cin** and **cout** are used very often for taking inputs and printing outputs respectively. These two are the most basic methods of taking input and output in C++.

To learn more about basic input and output in C++, refer to this article – Basic Input/Output in C++

3. Comments in C++

A well-documented program is a good practice for a programmer. It makes a program more readable and error finding becomes easier. One important part of good documentation is Comments. In computer programming, a comment is a programmer-readable explanation or annotation in the source code of a computer program. These are statements that are not executed by the compiler and interpreter.

We can learn more about C++ comments in this article – C++ Comments

4. Data Types and Modifiers in C++

All variables use data type during declaration to restrict the type of data to be stored. Therefore, 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. We can change this by using data type modifiers.

To know more about data types, refer to this article – C++ Data Types
To know more about type modifiers, refer to this article – C++ Type Modifiers

5. Variables in C++

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

To know more about C++ variables, refer to this article – C++ Variables

6. Variable Scope in C++

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

To know more about variable scope in C++, refer to this article – Scope of Variable in C++

7. Uninitialized Variable in C++

"One of the things that have kept C++ viable is the zero-overhead rule: What you don't use, you don't pay for." -Stroustrup. The overhead of initializing a stack variable is costly as it hampers the speed of execution, therefore these variables can contain indeterminate values. It is considered a best practice to initialize a primitive data type variable before using it in code.

To know more about what happens to the uninitialized variables, refer to this article – Uninitialized primitive data types in C/C++

8. Constants and Literals in C++

As the name suggests the name constants are given to such variables or values in C++ programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any type of constant like integer, float, octal, hexadecimal, character constants, etc. Every constant has some range. The integers that are too big to fit into an int will be taken as long. Now there are various ranges that differ from unsigned to signed bits. Under the signed bit, the range of an int varies from -128 to +127, and under the unsigned bit, the int varies from 0 to 255. Literals are a kind of constant and both terms are used interchangeably in C++.

To know more about constants in C++, refer to this article – Constants in C++

To know more about literals in C++, refer to this article – Literals in C++

9. Operators in C++

Operators are the foundation of any programming language. Thus the functionality of the C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.

To know more about C++ operators, refer to this article – Operators in C++

10. Loops in C++

Loops in programming come into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print "Hello World" 10 times. This can be done in two ways, the Iterative method and using Loops.

To know more about loops in C++, refer to this article – C++ Loops

11. Decision-Making in C++

There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. Decision-making statements in programming languages decide the direction of the flow of program execution.

To know more about decision-making statements in C++, refer to this article – Decision Making in C++

12. Classes and Objects in C++

Classes and Objects are used to provide Object Oriented Programming in C++. A class is a user-defined datatype that is a blueprint of an object that contains data members (attribute) and member methods that works on that data. Objects are the instances of a class that are required to use class methods and data as we cannot use them directly.

To know more about classes and objects, refer to this article – C++ Classes and Objects

13. Access Modifiers in C++

Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding. Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by outside functions.

To know more about access modifiers in C++, refer to this article – Access Modifiers in C++

14. Storage Classes in C++

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility, and lifetime which help us to trace the existence of a particular variable during the runtime of a program.

To know more about storage classes in C++, refer to this article – Storage Classes in C++

15. Forward declarations in C++

It refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program). In C++, Forward declarations are usually used for Classes. In this, the class is pre-defined before its use so that it can be called and used by other classes that are defined before this.

To know more about forward declarations in C++, refer to this article – What are forward declarations in C++?

16. Errors in C++

An error is an illegal operation performed by the user which results in the abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

To know more about C++ errors, refer to this article – Errors in C++

17. Undefined Behaviour in C++

If a user starts learning in a C/C++ environment and is unclear about the concept of undefined behavior then that can bring plenty of problems in the future while debugging someone else code might be actually difficult in tracing the root to the undefined error.

To know more about undefined behavior, refer to this article – Undefined Behavior in C and C++

549

C++ Data Types

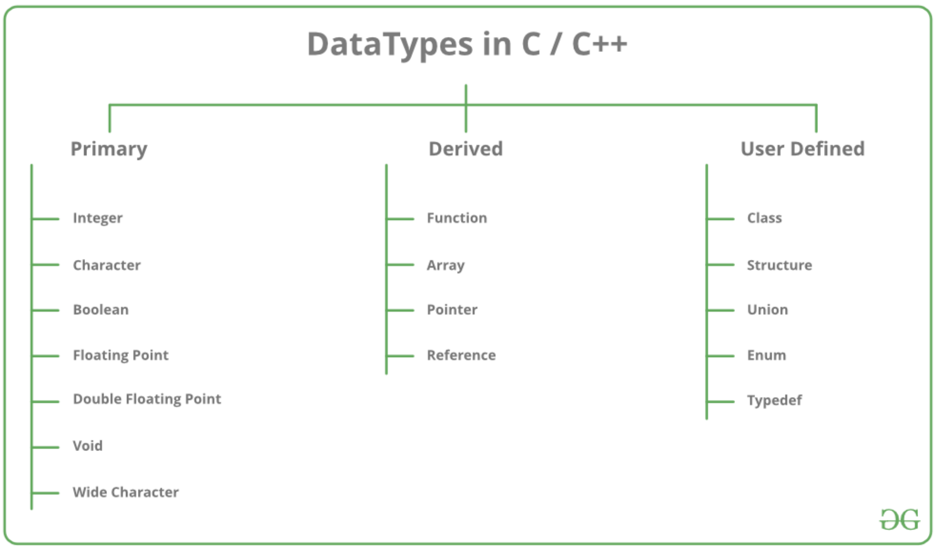
Difficulty Level : Basic • Last Updated : 18 Mar, 2023

All variables use data type during declaration to restrict the type of data to be stored. Therefore, 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**



AD

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

Primitive Data Types

- **Integer**: The keyword used for integer data types is **int**. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- **Character**: Character data type is used for storing characters. The keyword used for the character data type is **char**. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- **Boolean**: Boolean data type is used for storing Boolean or logical values. A Boolean variable can store either *true* or *false*. The keyword used for the Boolean data type is **bool**.
- **Floating Point**: Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is **float**. Float variables typically require 4 bytes of memory space.
- **Double Floating Point**: Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is **double**. Double variables typically require 8 bytes of memory space.
- **void**: Void means without any value. void data type represents a valueless entity. A void data type is used for those function which does not return a value.
- **Wide Character**: Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit data type. Represented by **wchar_t**. It is generally 2 or 4 bytes long.
- **sizeof() operator:** sizeof() operator is used to find the number of bytes occupied by a variable/data type in computer memory.

Example:

int m , x[50];

cout<<sizeof(m); //returns 4 which is the number of bytes occupied by the integer variable “m”.

cout<<sizeof(x); //returns 200 which is the number of bytes occupied by the integer array variable "x".

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

C++

```cpp
// 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;
}
```

Output

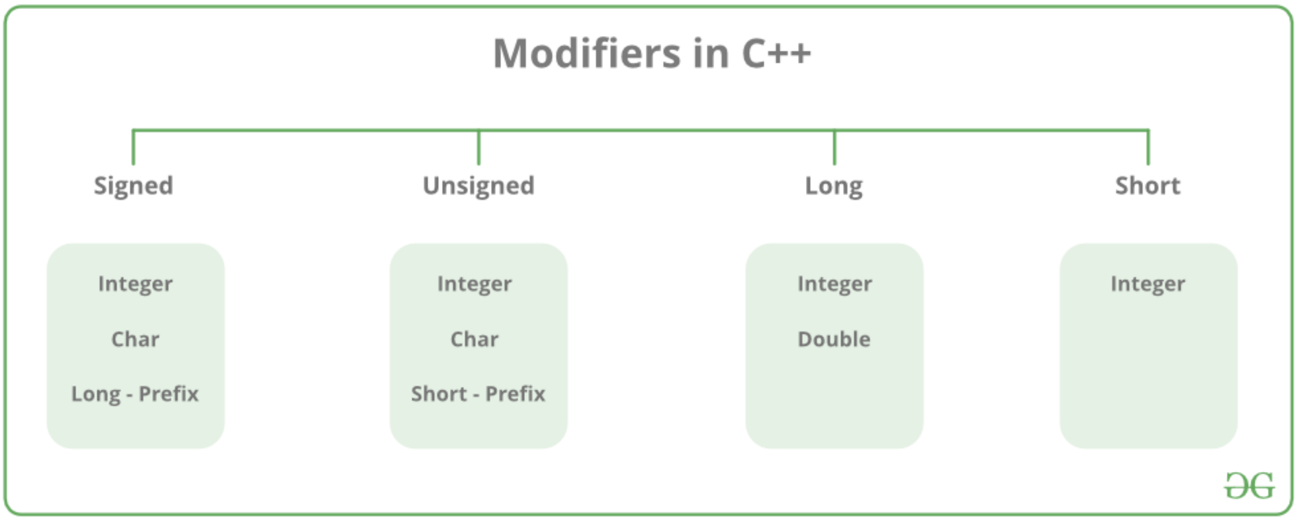
```
Size of char : 1
Size of int : 4
Size of long : 8
Size of float : 4
Size of double : 8
```

Time Complexity: O(1)

Space Complexity: O(1)

Datatype Modifiers

As the name suggests, datatype modifiers are used with built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

- **Signed**
- **Unsigned**
- **Short**
- **Long**

The below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1

Data Type	Size (in bytes)	Range
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	-3.4×10^38 to 3.4×10^38
double	8	-1.7×10^308 to1.7×10^308
long double	12	-1.1×10^4932 to1.1×10^4932
wchar_t	2 or 4	1 wide character

***Note**: Above values may vary from compiler to compiler. In the above example, we have considered GCC 32 bit.*
We can display the size of all the data types by using the sizeof() operator and passing the keyword of the datatype, as an argument to this function as shown below:

Now to get the range of data types refer to the following chart

***Note: syntax<limits.h>** header file is defined to find the range of fundamental data-types. Unsigned modifiers have minimum value is zero. So, no macro constants are defined for the unsigned minimum value.*

Macro Constants

Name	Expresses
CHAR_MIN	The minimum value for an object of type char
CHAR_MAX	Maximum value for an object of type char
SCHAR_MIN	The minimum value for an object of type Signed char
SCHAR_MAX	Maximum value for an object of type Signed char
UCHAR_MAX	Maximum value for an object of type Unsigned char
CHAR_BIT	Number of bits in a char object
MB_LEN_MAX	Maximum number of bytes in a multi-byte character
SHRT_MIN	The minimum value for an object of type short int
SHRT_MAX	Maximum value for an object of type short int
USHRT_MAX	Maximum value for an object of type Unsigned short int
INT_MIN	The minimum value for an object of type int
INT_MAX	Maximum value for an object of type int
UINT_MAX	Maximum value for an object of type Unsigned int
LONG_MIN	The minimum value for an object of type long int
LONG_MAX	Maximum value for an object of type long int

Name	Expresses
ULONG_MAX	Maximum value for an object of type Unsigned long int
LLONG_MIN	The minimum value for an object of type long long int
LLONG_MAX	Maximum value for an object of type long long int
ULLONG_MAX	Maximum value for an object of type Unsigned long long int

The actual value depends on the particular system and library implementation but shall reflect the limits of these types in the target platform. LLONG_MIN, LLONG_MAX, and ULLONG_MAX are defined for libraries complying with the C standard of 1999 or later (which only includes the C++ standard since 2011: C++11).

C++ Program to Find the Range of Data Types using Macro Constants

Example:

C++

```
// C++ program to Demonstrate the sizes of data types
#include <iostream>
#include <limits.h>
using namespace std;

int main()
{
    cout << "Size of char : " << sizeof(char) << " byte"
         << endl;

    cout << "char minimum value: " << CHAR_MIN << endl;

    cout << "char maximum value: " << CHAR_MAX << endl;

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

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

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

    cout << "Size of signed long int : "
         << sizeof(signed long int) << " bytes" << endl;
    cout << "Size of unsigned long int : "
         << sizeof(unsigned long int) << " bytes" << endl;

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

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

    cout << "Size of wchar_t : " << sizeof(wchar_t)
         << " bytes" << endl;

    return 0;
}
```

Output

```
Size of char : 1 byte
char minimum value: -128
char maximum value: 127
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes
```

Time Complexity: O(1)

Space Complexity: O(1)

C++

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

int main() {
  // Integer data types
  int a = 10;
  short b = 20;
  long c = 30;
  long long d = 40;
  cout << "Integer data types: " << endl;
  cout << "int: " << a << endl;
  cout << "short: " << b << endl;
  cout << "long: " << c << endl;
  cout << "long long: " << d << endl;
```

```
    // Floating-point data types
    float e = 3.14f;
    double f = 3.141592;
    long double g = 3.14159265358979L;
    cout << "Floating-point data types: " << endl;
    cout << "float: " << e << endl;
    cout << "double: " << f << endl;
    cout << "long double: " << g << endl;

    // Character data types
    char h = 'a';
    wchar_t i = L'b';
    char16_t j = u'c';
    char32_t k = U'd';
    cout << "Character data types: " << endl;
    cout << "char: " << h << endl;
    wcout << "wchar_t: " << i << endl;
    cout << "char16_t: " << j << endl;
    cout << "char32_t: " << k << endl;

    // Boolean data type
    bool l = true;
    bool m = false;
    cout << "Boolean data type: " << endl;
    cout << "true: " << l << endl;
    cout << "false: " << m << endl;

    // String data type
    string n = "Hello, world!";
    cout << "String data type: " << endl;
    cout << n << endl;

    return 0;
}
```

Output

```
Integer data types:
int: 10
short: 20
long: 30
long long: 40
Floating-point data types:
float: 3.14
double: 3.14159
long double: 3.14159
Character data types:
char: a
wchar_t: b
char16_t: 99
```

```
char32_t: 100
Boolean data type:
true: 1
false: 0
String data type:
Hello, world!
```

This program declares variables of various data types, assigns values to them, and then prints out their values.

The integer data types include int, short, long, and long long. These data types represent whole numbers of varying sizes.

The floating-point data types include float, double, and long double. These data types represent real numbers with varying levels of precision.

The character data types include char, wchar_t, char16_t, and char32_t. These data types represent individual characters of varying sizes.

The boolean data type is a simple data type that can only have one of two values: true or false.

The string data type is a sequence of characters. In this program, we use the string class to declare a string variable and assign it a value.

Advantages:

Data types provide a way to categorize and organize data in a program, making it easier to understand and manage.
Each data type has a specific range of values it can hold, allowing for more precise control over the type of data being stored.
Data types help prevent errors and bugs in a program by enforcing strict rules about how data can be used and manipulated.
C++ provides a wide range of data types, allowing developers to choose the best type for a specific task.

Disadvantages:

Using the wrong data type can result in unexpected behavior and errors in a program.
Some data types, such as long doubles or char arrays, can take up a large amount of memory and impact performance if used excessively.
C++'s complex type system can make it difficult for beginners to learn and use the language effectively.
The use of data types can add additional complexity and verbosity to a program, making it harder to read and understand.

1.12k

Related Articles

1. Difference between fundamental data types and derived data types
2. What Happen When We Exceed Valid Range of Built-in Data Types in C++?
3. Calculate range of data types using C++
4. Uninitialized primitive data types in C/C++
5. User defined Data Types in C++
6. Different types of Coding Schemes to represent data
7. Derived Data Types in C++
8. Data types that supports std::numeric_limits() in C++
9. Data Communication - Definition, Components, Types, Channels
10. C++ Char Data Types

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Basic

C++ Variables

Difficulty Level : Easy • Last Updated : 16 Mar, 2023

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

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

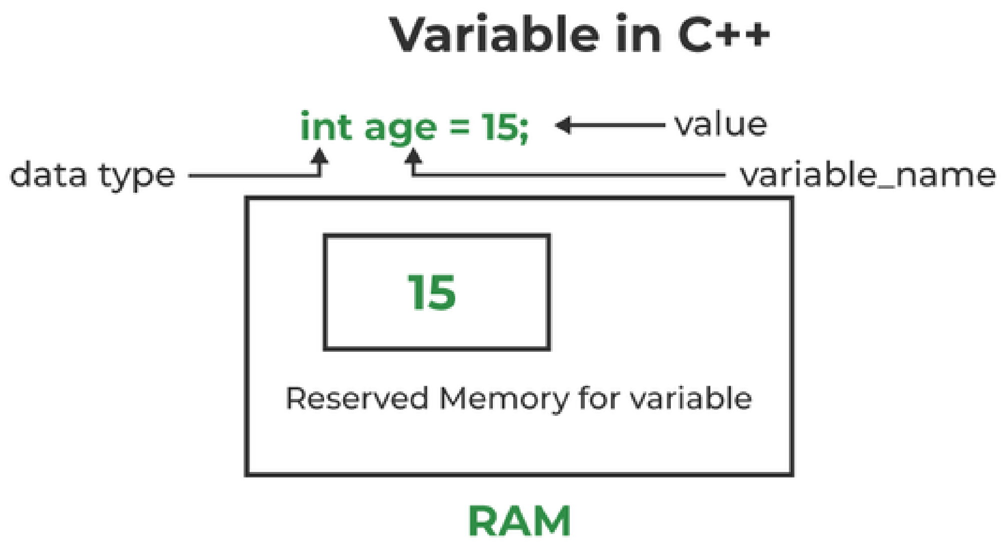
How to Declare Variables?

A typical variable declaration is of the form:

```
// Declaring a single variable
type variable_name;

// Declaring multiple variables:
type variable1_name, variable2_name, variable3_name;
```

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



Initialization of a variable in C++

In the above diagram,

AD

> ***datatype***: *Type of data that can be stored in this variable.*
> ***variable_name***: *Name given to the variable.*
> ***value***: *It is the initial value stored in the variable.*

Examples:

```
// Declaring float variable
float simpleInterest;

// Declaring integer variable
int time, speed;

// Declaring character variable
char var;
```

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

```
int a=50,b=100;  //declaring 2 variable of integer type
float f=50.8;  //declaring 1 variable of float type
char c='Z';    //declaring 1 variable of char type
```

Rules For Declaring Variable

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

Valid variable names:

```
int x;   //can be letters
int _yz; //can be underscores
int z40;//can be letters
```

Invalid variable names:

```
int 89; Should not be a number
int a b; //Should not contain any whitespace
int double;// C++ keyword CAN NOT BE USED
```

Difference Between Variable Declaration and Definition

The **variable declaration** refers to the part where a variable is first declared or introduced before its first use. A **variable definition** is a part where the variable is assigned a memory location and a value. Most of the time, variable declaration and definition are done together.
See the following C++ program for better clarification:

C++

```
// 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;
}
```

Output

```
a
```

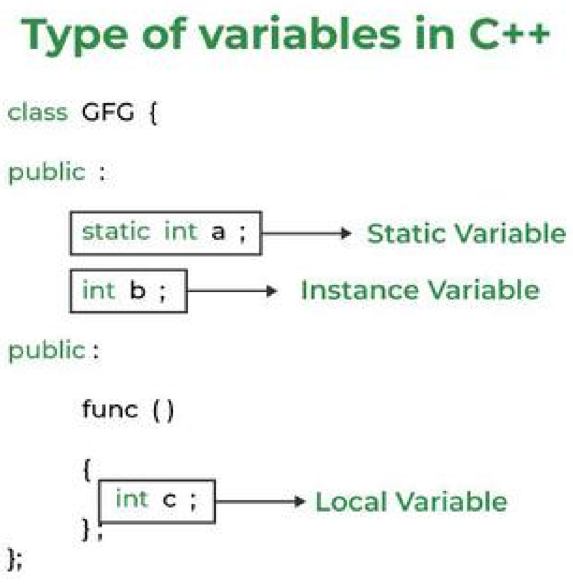
Time Complexity: O(1)

Space Complexity: O(1)

Types of Variables

There are three types of variables based on the scope of variables in C++

- **Local Variables**
- **Instance Variables**
- **Static Variables**



Types of Variables in C++

Let us now learn about each one of these variables in detail.

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

2. **Instance Variables**: Instance variables are non-static variables and are declared in a class outside any method, constructor, or block.
   - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
   - Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
   - Initialization of Instance Variable is not Mandatory.
   - Instance Variable can be accessed only by creating objects.

3. **Static Variables**: Static variables are also known as Class variables.
   - These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
   - Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
   - Static variables are created at the start of program execution and destroyed automatically when execution ends.

- Initialization of Static Variable is not Mandatory. Its default value is 0
- If we access the static variable like the Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access the static variable without the class name, the Compiler will automatically append the class name.

Instance Variable Vs Static Variable

- Each object will have its **own copy** of the instance variable whereas We can only have **one copy** of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of the instance variable. In the case of static, changes **will be reflected** in other objects as static variables are common to all objects of a class.
- We can access instance variables **through object references** and Static Variables can be accessed **directly using the class name.**
- The syntax for static and instance variables:

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

384

Related Articles

1. Templates and Static variables in C++
2. Swap Two Variables in One Line
3. Scope of Variables in C++
4. Can Global Variables be dangerous ?
5. C++ 17 | New ways to Assign values to Variables
6. Why do we need reference variables if we have pointers

C++ Loops

Difficulty Level : Easy • Last Updated : 18 Mar, 2023

In Programming, sometimes there is a need to perform some operation **more than once** or (say) **n number** of times. Loops come into use when we need to repeatedly execute a block of statements.

For example: Suppose we want to print "Hello World" 10 times. This can be done in two ways as shown below:

Manual Method (Iterative Method)

Manually we have to write ***cout*** for the C++ statement 10 times. Let's say you have to write it 20 times (it would surely take more time to write 20 statements) now imagine you have to write it 100 times, it would be really hectic to re-write the same statement again and again. So, here loops have their role.

C++

```
// C++ program to Demonstrate the need of loops
#include <iostream>
using namespace std;

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

Output

AD

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Time complexity: O(1)

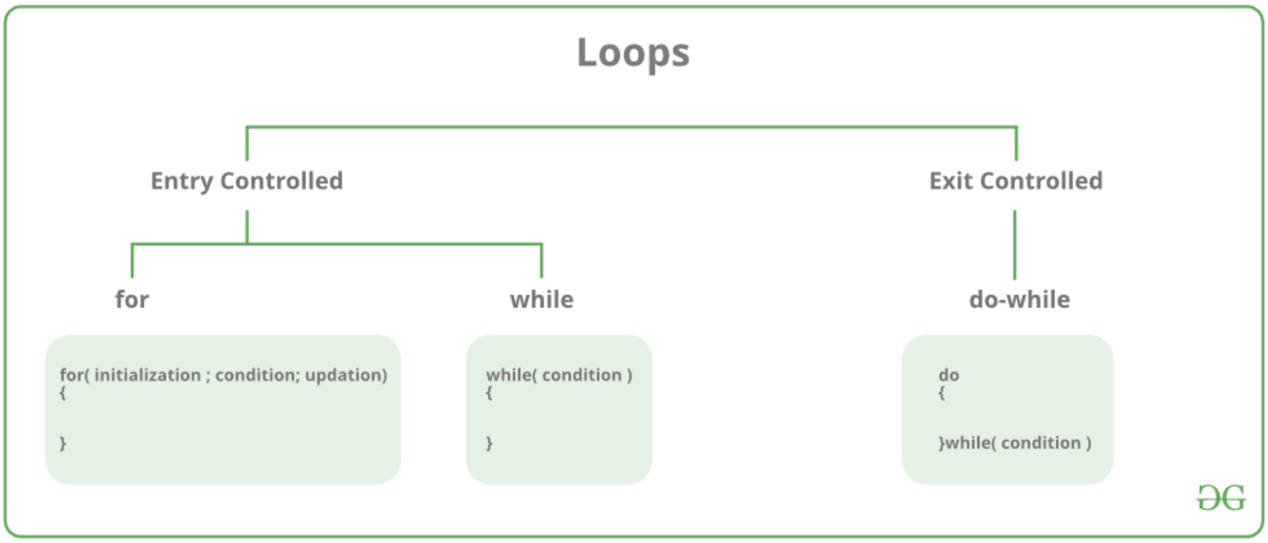
Space complexity: O(1)

Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below. In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

There are mainly two types of loops:

1. **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.
2. **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.



S.No.	Loop Type and Description
1.	**while loop** – First checks the condition, then executes the body.
2.	**for loop** – firstly initializes, then, condition check, execute body, update.
3.	**do-while loop** – firstly, execute the body then condition check

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.)

NOTES:

- The initialization and increment statements can perform operations unrelated to the condition statement, or nothing at all – if you wish to do. But the good practice is to only perform operations directly relevant to the loop.
- A variable declared in the initialization statement is visible only inside the scope of the for loop and will be released out of the loop.
- Don't forget that the variable which was declared in the initialization statement can be modified during the loop, as well as the variable checked in the condition.

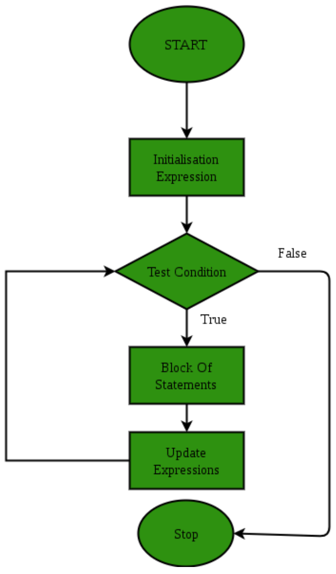
Example1:

```
for(int i = 0; i < n; i++)
{
    // BODY
}
```

Example2:

```
for(auto element:arr)
{
    //BODY
}
```

Flow Diagram of for loop:



Example1:

C++

```
// 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;
}
```

Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Time complexity: O(1)

Space complexity: O(1)

Example2:

C++

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

int main() {

int arr[] {40, 50, 60, 70, 80, 90, 100};
 for (auto element: arr){
   cout << element << " ";
 }
return 0;

}
```

Output

```
40 50 60 70 80 90 100
```

Time complexity: O(n) n is the size of array.

Space complexity: O(n) n is the size of array.

***For loop* can also be valid in the given form:-**

C++

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

int main()
{
    for (int i = 0, j = 10, k = 20; (i + j + k) < 100;
         j++, k--, i += k) {
        cout << i << " " << j << " " << k << "\n";
    }
    return 0;
}
```

Output

```
0 10 20
19 11 19
37 12 18
54 13 17
```

Time complexity: O(1)

Space complexity: O(1)

Example of hiding declared variables before a loop is:

C++

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

int main()
{
    int i = 99;
    for (int i = 0; i < 5; i++) {
        cout << i << "\t";
    }
    cout << "\n" << i;
    return 0;
}
```

Output

```
0    1    2    3    4
99
```

Time complexity: O(1)

Space complexity: O(1)

But if you want to use the already declared variable and not hide it, then must not redeclare that variable.

For Example:

C++

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

int main()
{
    int i = 99;
    for (i = 0; i < 5; i++) {
        cout << i << " ";
    }
    cout << "\n" << i;
    return 0;
}
```

Output

```
0 1 2 3 4
5
```

Time complexity: O(1)

Space complexity: O(1)

The For loop can be used to iterating through the elements in the STL container(e.g., Vector, etc). here we have to use iterator.

For Example:

C++

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

int main()
{
    vector<int> v = { 1, 2, 3, 4, 5 };
    for (vector<int>::iterator it = v.begin();
         it != v.end(); it++) {
        cout << *it << "\t";
    }
    return 0;
}
```

Output

```
1	2	3	4	5
```

Time complexity: O(n) n is the size of vector.

Space complexity: O(n) n is the size of vector.

While Loop-

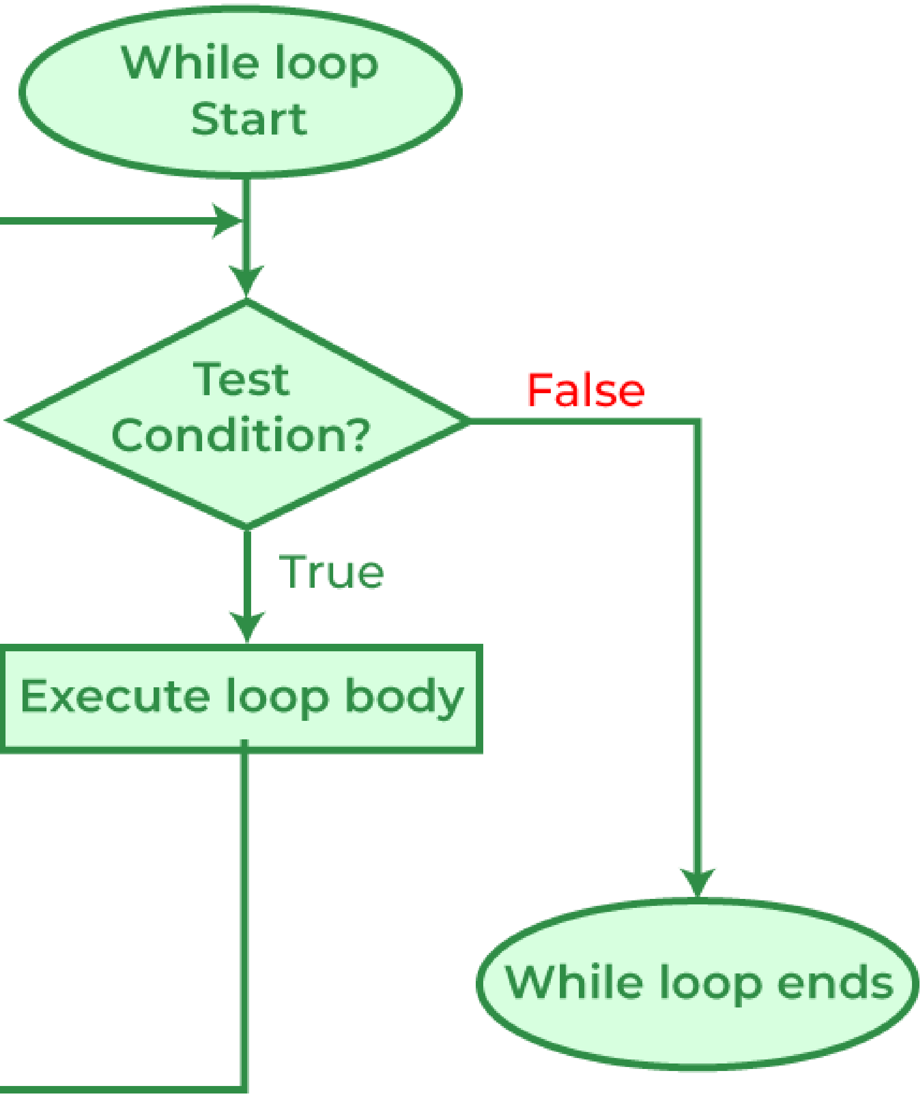
While studying **for loop** we have seen that the number of iterations is ***known beforehand***, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where **we do not know** the exact number of iterations of the loop **beforehand**. The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

Flow Diagram of while loop:



Example:

C++

```
// 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;
}
```

Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
```

Time complexity: O(1)

Space complexity: O(1)

It's explanation is same as that of the *for loop*.

Do-while loop

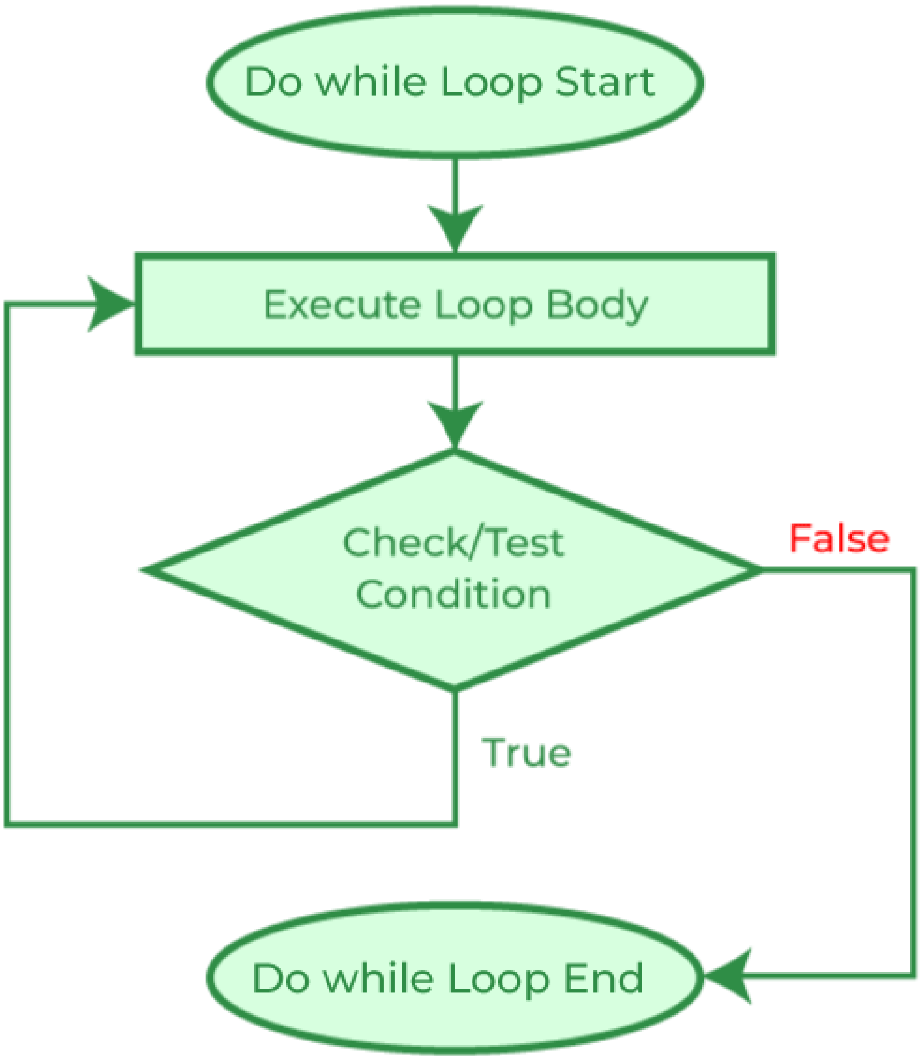
In Do-while loops also the loop execution is terminated on the basis of test conditions. The main difference between a do-while loop and the while loop is in the do-while loop the condition is tested at the end of the loop body, i.e do-while loop is exit controlled whereas the other two loops are entry-controlled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

Note: *Notice the semi – colon(";")in the end of loop.*

Flow Diagram of the do-while loop:



Example:

C++

```
// 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;
}
```

Output

```
Hello World
```

Time complexity: O(1)

Space complexity: O(1)

In the above program, the test condition (i<1) evaluates to false. But still, as the loop is an exit – controlled the loop body will execute once.

What about an Infinite Loop?

An infinite loop (sometimes called an endless loop) is a piece of coding that lacks a **functional exit** so that it repeats indefinitely. An infinite loop occurs when a condition is always evaluated to be true. Usually, this is an error.

Using For loop:

C++

```
// C++ program to demonstrate infinite loops
// using for and while loop

// Uncomment the  sections to see the output
#include <iostream>
using namespace std;
int main()
{
    int i;

    // This is an infinite for loop as the condition
    // expression is blank
    for (;;) {
        cout << "This loop will run forever.\n";
    }

    // This is an infinite for loop as the condition
    // given in while loop will keep repeating infinitely
    /*
    while (i != 0)
    {
        i-- ;
        cout << "This loop will run forever.\n";
    }
    */

    // This is an infinite for loop as the condition
    // given in while loop is "true"
    /*
    while (true)
    {
        cout << "This loop will run forever.\n";
    }
    */
}
```

Output

Output:

```
This loop will run forever.
This loop will run forever.
...................
```

Time complexity: O(infinity) as the loop will run forever.

Space complexity: O(1)

Using While loop:

C++

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

int main()
{

    while (1)
        cout << "This loop will run forever.\n";
    return 0;
}
```

Output

Output:

```
This loop will run forever.
This loop will run forever.
...................
```

Time complexity: O(infinity) as the loop will run forever.

Space complexity: O(1)

Using the Do-While loop:

C++

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

int main()
{

    do {
        cout << "This loop will run forever.\n";
    } while (1);

    return 0;
}
```

Output

Output:

```
This loop will run forever.
This loop will run forever.
...................
```

Time complexity: O(infinity) as the loop will run forever.

Space complexity: O(1)

Now let us take a look at decrementing loops.

Sometimes we need to decrement a variable with a looping condition.

Using for loop

C++

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

int main() {

    for(int i=5;i>=0;i--){
      cout<<i<<" ";
    }
    return 0;
}
```

Output

```
5 4 3 2 1 0
```

Time complexity: O(1)

Space complexity: O(1)

Using while loop.

C++

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

int main() {
    //first way is to decrement in the condition itself
    int i=5;
      while(i--){
      cout<<i<<" ";
    }
      cout<<endl;
      //second way is to decrement inside the loop till i is 0
      i=5;
      while(i){
      cout<<i<<" ";
      i--;
    }
    return 0;
}
```

Output

```
4 3 2 1 0
5 4 3 2 1
```

Time complexity: O(1)

Space complexity: O(1)

Using do-while loop

C++

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

int main() {
    int i=5;
   do{
       cout<<i<<" ";
   }while(i--);
}
```

Output

```
5 4 3 2 1 0
```

Time complexity: O(1)

Space complexity: O(1)

C++

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

int main() {
  // For loop
  for (int i = 1; i <= 5; i++) {
    cout << "For loop: The value of i is: " << i << endl;
  }

  // While loop
  int j = 1;
```

```
    while (j <= 5) {
      cout << "While loop: The value of j is: " << j << endl;
      j++;
    }

    // Do-while loop
    int k = 1;
    do {
      cout << "Do-while loop: The value of k is: " << k << endl;
      k++;
    } while (k <= 5);

    // Range-based for loop
    vector<int> myVector = {1, 2, 3, 4, 5};
    for (int element : myVector) {
      cout << "Range-based for loop: The value of element is: " << element << endl;
    }

    return 0;
  }
```

Output

```
For loop: The value of i is: 1
For loop: The value of i is: 2
For loop: The value of i is: 3
For loop: The value of i is: 4
For loop: The value of i is: 5
While loop: The value of j is: 1
While loop: The value of j is: 2
While loop: The value of j is: 3
While loop: The value of j is: 4
While loop: The value of j is: 5
Do-while loop: The value of k is: 1
Do-while loop: The value of k is: 2
Do-while loop: The value of k is: 3
Do-while loop: The value of k is: 4
Do-while loop: The value of k is: 5
Range-based for loop: The value of element is: 1
Range-based for loop: The value of element is: 2
Range-based for loop: The value of element is: 3
Range-based for loop: The value of element is: 4
Range-based for loop: The value of element is: 5
```

C++

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

int main() {

    // for loop example
    cout << "For loop:" << endl;
    for(int i = 0; i < 5; i++) {
        cout << i << endl;
    }

    // while loop example
    cout << "While loop:" << endl;
    int j = 0;
    while(j < 5) {
        cout << j << endl;
        j++;
    }

    // do-while loop example
    cout << "Do-while loop:" << endl;
    int k = 0;
    do {
        cout << k << endl;
        k++;
    } while(k < 5);

    return 0;
}
```

Output

```
For loop:
0
1
2
3
4
While loop:
0
1
2
3
4
Do-while loop:
0
1
2
```

```
3
4
```

Advantages :

1. **High performance**: C++ is a compiled language that can produce efficient and high-performance code. It allows low-level memory manipulation and direct access to system resources, making it ideal for applications that require high performance, such as game development, operating systems, and scientific computing.
2. **Object-oriented programming:** C++ supports object-oriented programming, allowing developers to write modular, reusable, and maintainable code. It provides features such as inheritance, polymorphism, encapsulation, and abstraction that make code easier to understand and modify.
3. **Wide range of applications:** C++ is a versatile language that can be used for a wide range of applications, including desktop applications, games, mobile apps, embedded systems, and web development. It is also used extensively in the development of operating systems, system software, and device drivers.
4. **Standardized language:** C++ is a standardized language, with a specification maintained by the ISO (International Organization for Standardization). This ensures that C++ code written on one platform can be easily ported to another platform, making it a popular choice for cross-platform development.
5. **Large community and resources:** C++ has a large and active community of developers and users, with many resources available online, including documentation, tutorials, libraries, and frameworks. This makes it easy to find help and support when needed.
6. **Interoperability with other languages:** C++ can be easily integrated with other programming languages, such as C, Python, and Java, allowing developers to leverage the strengths of different languages in their applications.

Overall, C++ is a powerful and flexible language that offers many advantages for developers who need to create high-performance, reliable, and scalable applications.

More Advanced Looping Techniques

- Range-Based for Loop in C++
- for each Loop in C++

Important Points

- Use for a loop when a number of iterations are known beforehand, i.e. the number of times the loop body is needed to be executed is known.

- Use while loops, where an exact number of iterations is not known but the loop termination condition, is known.
- Use do while loop if the code needs to be executed at least once like in Menu-driven programs

Related Articles:

- What happens if loop till Maximum of Signed and Unsigned in C/C++?
- Quiz on Loops

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

635

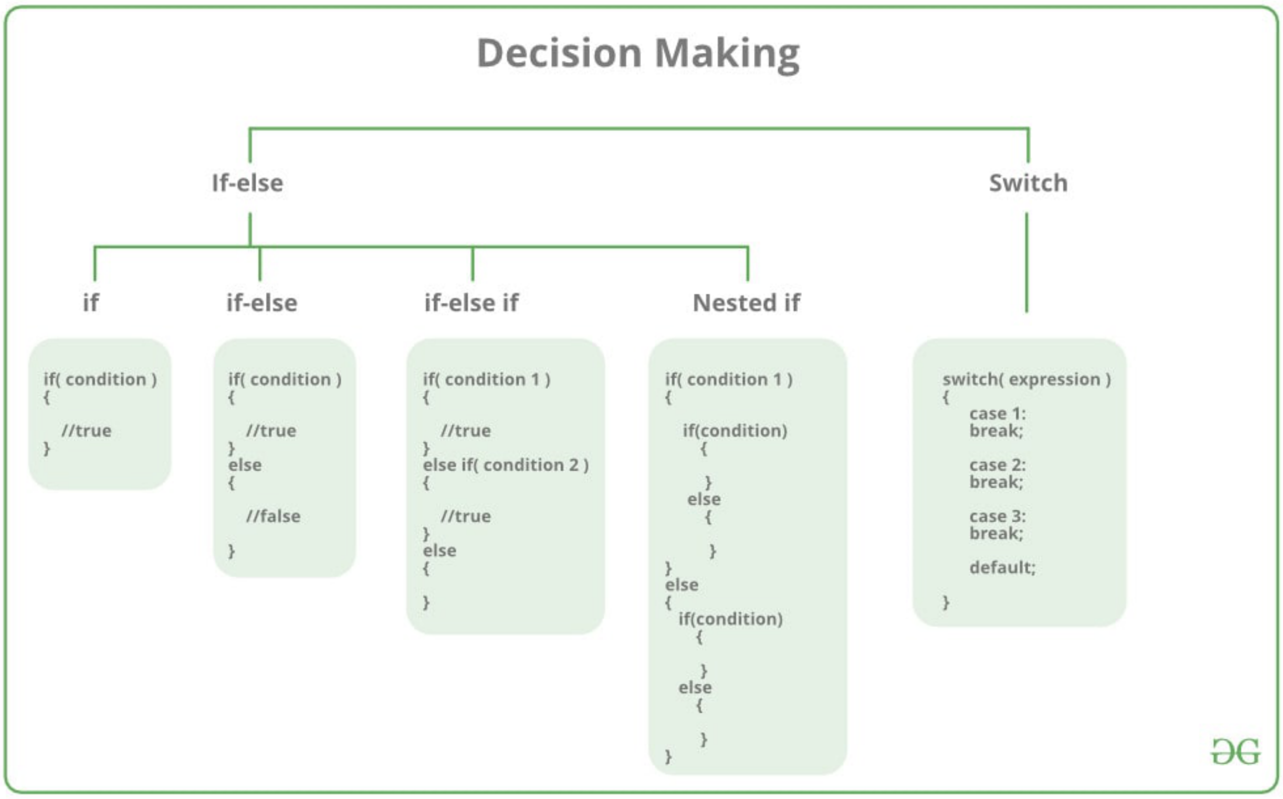
Related Articles

1. Print 1 to 100 in C++ Without Loops and Recursion
2. Understanding for loops in Java
3. Nested Loops in C++ with Examples
4. Loops in Java
5. How to print N times without using loops or recursion ?
6. Sum of array Elements without using loops and recursion
7. Loops and Control Statements (continue, break and pass) in Python
8. main Function in C
9. printf in C
10. C++ Error - Does not name a type

Decision Making in C / C++ (if , if..else, Nested if, if-else-if)

Difficulty Level : Easy • Last Updated : 16 Jan, 2023

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code.
For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions. The Decision Making Statements are used to evaluate the one or more conditions and make the decision whether to execute set of statement or not.


Decision Making
If-else
Switch
if
if-else
if-else if
Nested if
if(condition) { //true }
if(condition) { //true } else { //false }
if(condition 1) { //true } else if(condition 2) { //true } else { }
if(condition 1) { if(condition) { } else { } } else { if(condition) { } else { } }
switch(expression) { case 1: break; case 2: break; case 3: break; default; }

Decision-making statements in programming languages decide the direction of the flow of program execution. Decision-making statements available in C or C++ are:

1. if statement
2. if-else statements
3. nested if statements
4. if-else-if ladder
5. switch statements
6. Jump Statements:
   - break
   - continue
   - goto
   - return

1. if statement in C/C++

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 statement is executed otherwise not.

Syntax:

AD

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

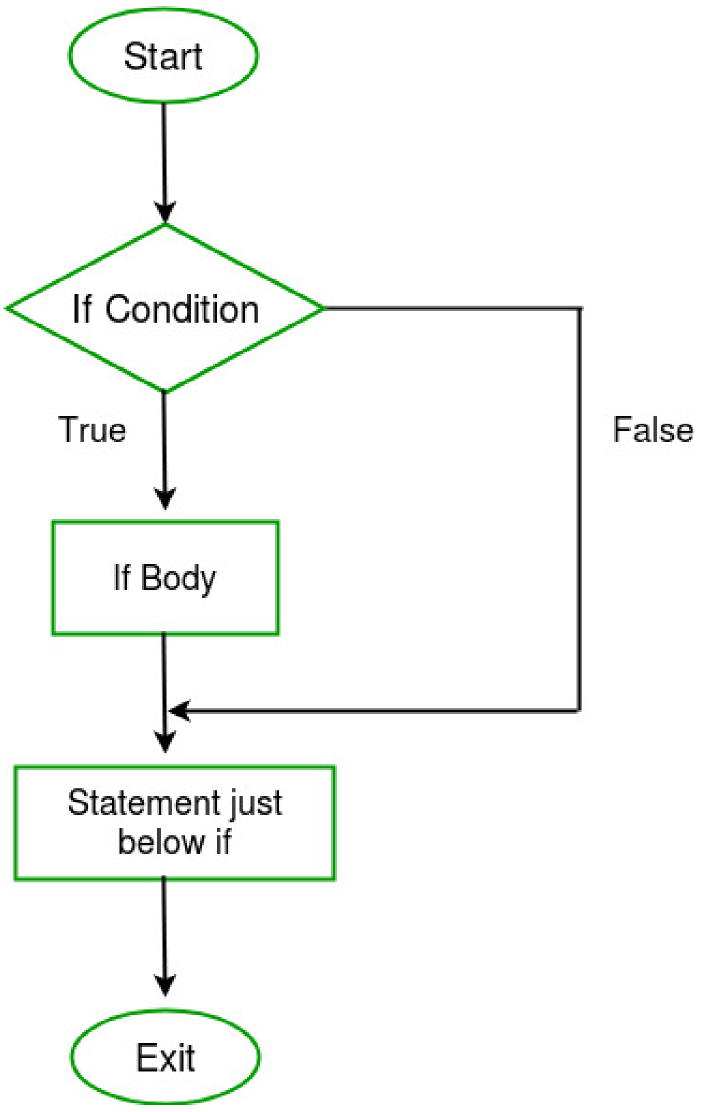
Example:

If Ram can having 100 GeekBits then he can redeem these GeekBits and get the GFG T-shirt .

```
if(condition)
   statement1;
   statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

Flowchart



C

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
    int i = 10;

    if (i > 15) {
        printf("10 is greater than 15");
    }

    printf("I am Not in if");
}
```

C++

```
// 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";
}
```

Output:

```
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

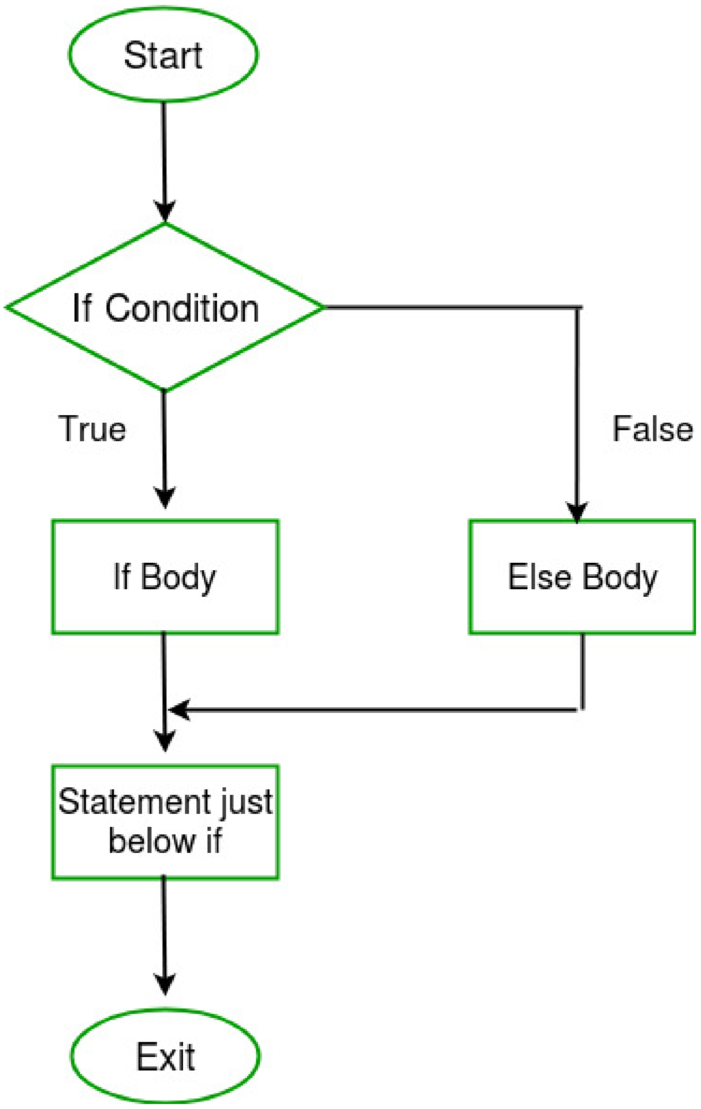
2. if-else in C/C++

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 if 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.

Syntax:

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

Flowchart:



Example:

The person who having correct 50 Geek Bits is redeem the gifts otherwise they can't redeem.

C

```
// C program to illustrate If statement
#include <stdio.h>

int main()
{
    int i = 20;

    if (i < 15) {

        printf("i is smaller than 15");
    }
    else {

        printf("i is greater than 15");
    }
    return 0;
}
```

C++

```
// 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;
}
```

Output:

```
i is greater than 15
```

The block of code following the *else* statement is executed as the condition present in the *if* statement is false.

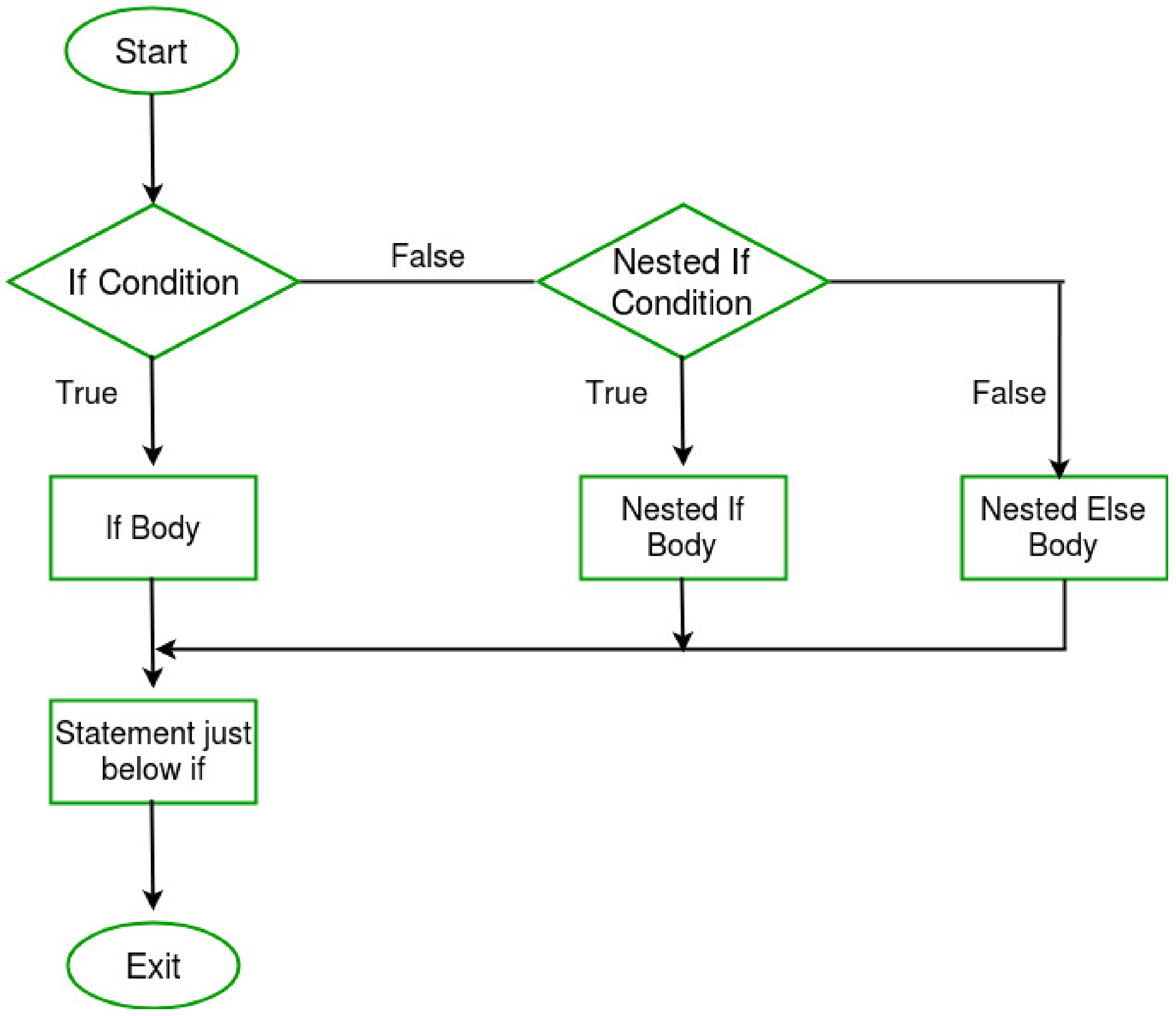
3. nested-if in C/C++

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:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Flowchart



Example:

If the person having more than 50 Geek Bits and less than 150 Geek Bits he won the GFG T-shirt.

C

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 10;

    if (i == 10) {
        // First if statement
        if (i < 15)
            printf("i is smaller than 15\n");
        // Nested - if statement
        // Will only be executed if statement above
        // is true
        if (i < 12)
            printf("i is smaller than 12 too\n");
        else
            printf("i is greater than 15");
    }

    return 0;
}
```

C++

```
// 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;
}
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

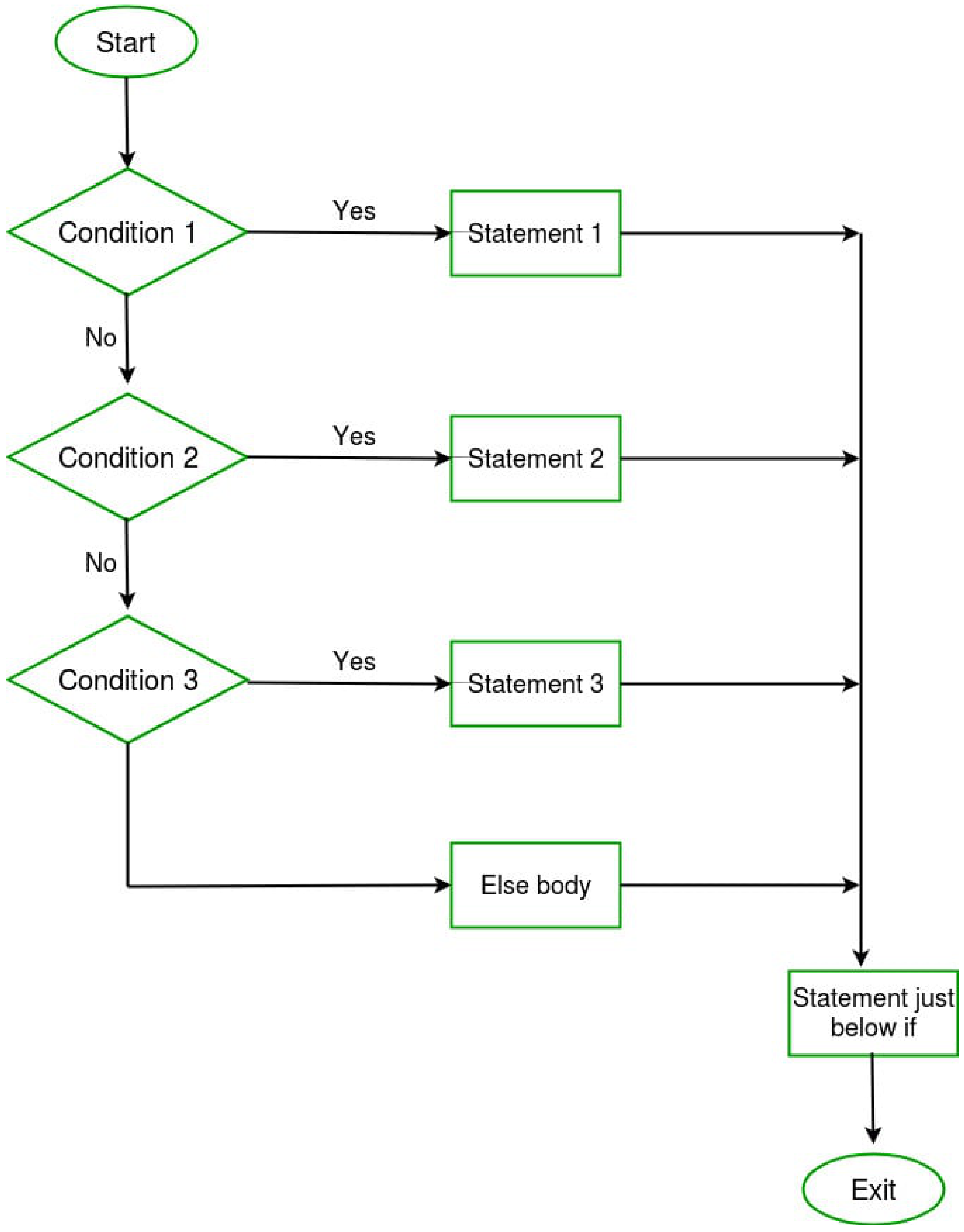
4. if-else-if ladder in C/C++

Here, a user can 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 switch statement.

Syntax:

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

else
    statement;
```



Example:

If the person having the 50 Geek Bits then he get the GfG Course Coupon otherwise he will having 100 Geek Bits then he get the GfG T-shirt otherwise he will having the 200 Geek Bits then he get the GfG Bag otherwise he will having less than 50 he cant get anything.

C

```
// C program to illustrate nested-if statement
#include <stdio.h>

int main()
{
    int i = 20;

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

C++

```
// 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";
}
```

Output:

```
 i is 20
```

5. Jump Statements in C/C++

These statements are used in C or C++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

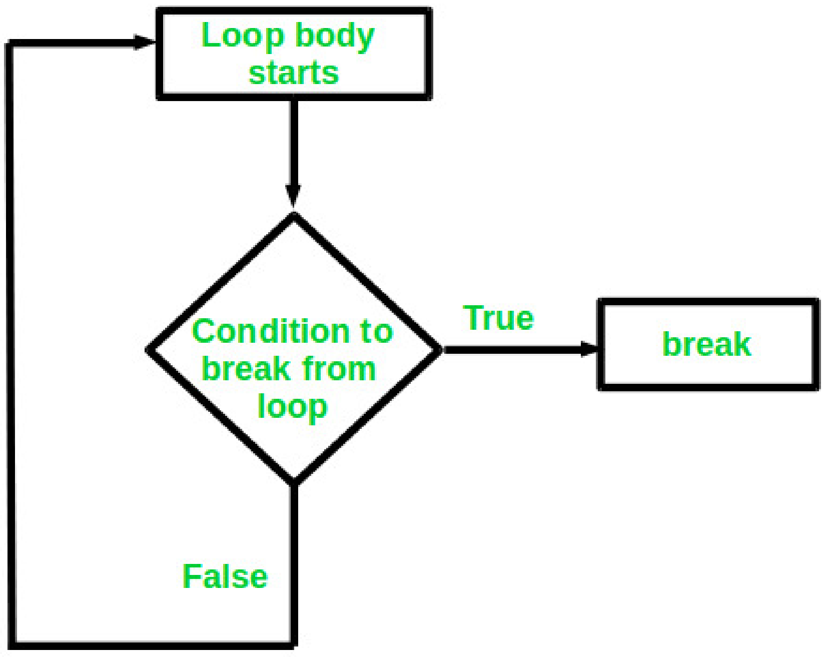
A) break

This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Syntax:

```
break;
```

Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.



Example:

C

```
// C program to illustrate
// to show usage of break
// statement
#include <stdio.h>

void findElement(int arr[], int size, int key)
{
    // loop to traverse array and search for key
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            printf("Element found at position: %d",
                   (i + 1));
            break;
        }
    }
}

int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6 };

    // no of elements
    int n = 6;

    // key to be searched
    int key = 3;

    // Calling function to find the key
    findElement(arr, n, key);

    return 0;
}
```

C++

```
// C++ program to illustrate
// to show usage of break
// statement
#include <iostream>
using namespace std;

void findElement(int arr[], int size, int key)
{
    // loop to traverse array and search for key
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            cout << "Element found at position: "
                 << (i + 1);
            break;
        }
    }
}

// Driver program to test above function
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    int n = 6; // no of elements
    int key = 3; // key to be searched

    // Calling function to find the key
    findElement(arr, n, key);

    return 0;
}
```

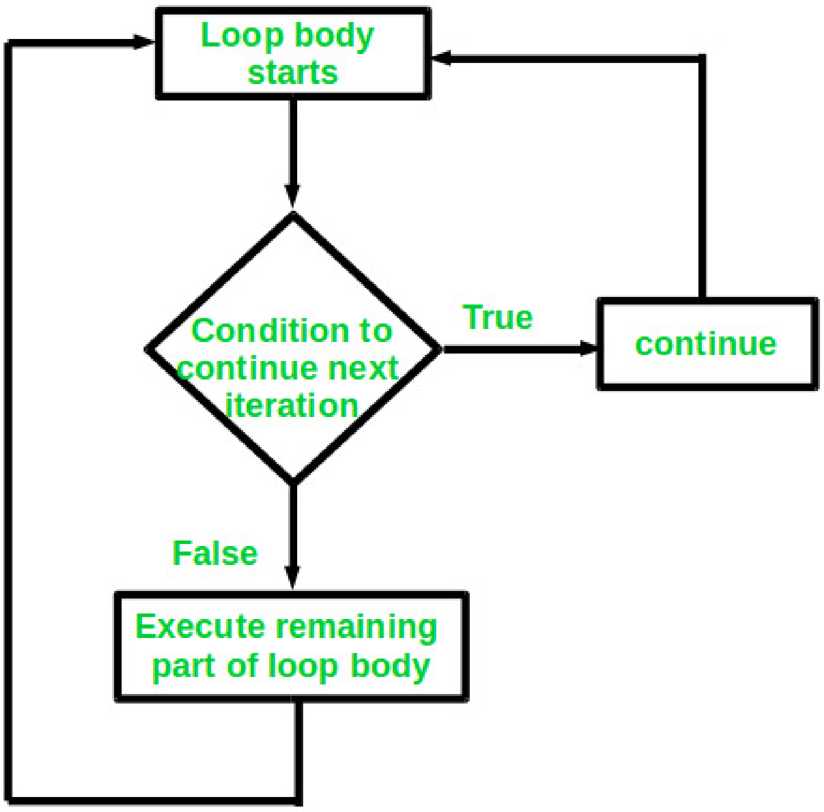
Output:

```
Element found at position: 3
```

B) continue

This loop control statement is just like the break statement. The *continue* statement is opposite to that of the break *statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.
As the name suggests the continue statement forces the loop to continue or execute the next iteration. When the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped and the next iteration of the loop will begin.
Syntax:

```
continue;
```


Loop body starts
Condition to continue next iteration
True
continue
False
Execute remaining part of loop body

Example:

C

```
// C program to explain the use
// of continue statement
#include <stdio.h>
```

```
int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            printf("%d ", i);
    }

    return 0;
}
```

C++

```
// C++ program to explain the use
// of continue statement

#include <iostream>
using namespace std;

int main()
{
    // loop from 1 to 10
    for (int i = 1; i <= 10; i++) {

        // If i is equals to 6,
        // continue to next iteration
        // without printing
        if (i == 6)
            continue;

        else
            // otherwise print the value of i
            cout << i << " ";
    }

    return 0;
}
```

Output:

```
1 2 3 4 5 7 8 9 10
```

If you create a variable in if-else in C/C++, it will be local to that if/else block only. You can use global variables inside the if/else block. If the name of the variable you created in if/else is as same as any global variable then priority will be given to `local variable`.

C

```
#include <stdio.h>

int main()
{

    int gfg = 0; // local variable for main
    printf("Before if-else block %d\n", gfg);
    if (1) {
        int gfg = 100; // new local variable of if block
        printf("if block %d\n", gfg);
    }
    printf("After if block %d", gfg);
    return 0;
}
```

C++

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

int main()
{
    int gfg = 0; // local variable for main
    cout << "Before if-else block " << gfg << endl;
    if (1) {
        int gfg = 100; // new local variable of if block
        cout << "if block " << gfg << endl;
    }
    cout << "After if block " << gfg << endl;
    return 0;
}
/*
    Before if-else block 0
    if block 100
    After if block 0
*/
```

Output:

```
Before if-else block 0
if block 100
After if block 0
```

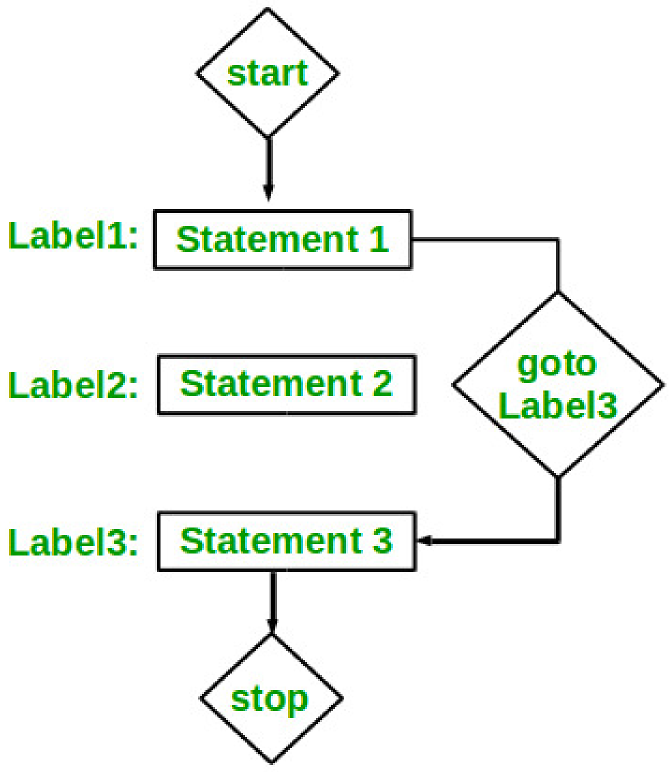
C) goto

The goto statement in C/C++ also referred to as the unconditional jump statement can be used to jump from one point to another within a function.

Syntax:

```
Syntax1      |   Syntax2
----------------------------
goto label;  |    label:
.            |    .
.            |    .
.            |    .
label:       |    goto label;
```

In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here, a label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.



Examples:

C

```
// C program to print numbers
// from 1 to 10 using goto
// statement
#include <stdio.h>
```

```
// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    printf("%d ", n);
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
int main()
{
    printNumbers();
    return 0;
}
```

C++

```
// C++ program to print numbers
// from 1 to 10 using goto
// statement
#include <iostream>
using namespace std;

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    cout << n << " ";
    n++;
    if (n <= 10)
        goto label;
}

// Driver program to test above function
int main()
{
    printNumbers();
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

D) return

The return in C or C++ returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and returns the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value must be returned.

Syntax:

```
return[expression];
```

Example:

C

```
// C code to illustrate return
// statement
#include <stdio.h>

// non-void return type
// function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;
    return s1;
}

// returns void
// function to print
void Print(int s2)
{
    printf("The sum is %d", s2);
    return;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    Print(sum_of);
    return 0;
}
```

C++

```
// C++ code to illustrate return
// statement
#include <iostream>
using namespace std;
```

```
// non-void return type
// function to calculate sum
int SUM(int a, int b)
{
    int s1 = a + b;
    return s1;
}

// returns void
// function to print
void Print(int s2)
{
    cout << "The sum is " << s2;
    return;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    Print(sum_of);
    return 0;
}
```

Output:

```
The sum is 20
```

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using write.geeksforgeeks.org or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

363

Related Articles

1. C program to invert (making negative) an image content in PGM format
2. Making your own Linux Shell in C
3. Publicly inherit a base class but making some of public method as private
4. Nested Classes in C++