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

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, and has 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 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 in directly manipulating storage addresses. This helps in doing low-level programming (where one might need to have explicit control over 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

[**Some interesting facts about C++**](https://www.geeksforgeeks.org/interesting-facts-about-c/)**:**   
Here are some awesome facts about C++ that may interest you:

1. The name of C++ signifies the evolutionary nature of the changes from C. “++” is the C increment operator.
2. C++ is one of the predominant languages for the development of all kinds of technical and commercial software.
3. 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.
4. C++ got the OOP features from Simula67 Programming language.
5. A function is a minimum requirement for a C++ program to run.(at least main() function)

Before explaining the basics of C++, I 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 us 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 I 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 as 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.**

Therefore, below are the basics of C++ in the format in which it will help you the most to get a headstart:

 ---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. [**Basic Syntax and First Program in C++:**](https://www.geeksforgeeks.org/writing-first-c-program-hello-world-example/) 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.

// C++ program to display "Hello World"

// Header file for input-output functions

#include <iostream>

**using** **namespace** std;

// Main() function: where the execution of program begins

**int** main()

{

    // prints hello world

    cout << "Hello World";

**return** 0;

}

**Output**

Hello World

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

**1)// C++ program to display “Hello World”:** This line is a comment line. A comment is used to display additional information about the program. A comment does not contain any programming logic. When a comment is encountered by a compiler, the compiler simply skips that line of code. Any line beginning with ‘//’ without quotes OR in between /\*…\*/ in C++ is comment. Click to know [**More about Comments.**](https://www.geeksforgeeks.org/comments-in-c-c/)

**2) #include**: In C++,  all lines that start with the pound (#) sign are called directives and are processed by a preprocessor which is a program invoked by the compiler. The **#include** directive tells the compiler to include a file and **#include<iostream>**. It tells the compiler to include the standard iostream file which contains declarations of all the standard input/output library functions. Click to Know [**More on Preprocessors.**](https://www.geeksforgeeks.org/cc-preprocessors/)

**3) using namespace std**: This is used to import the entirety of the std namespace into the current namespace of the program. The statement using namespace std is generally considered a bad practice. When we import a namespace we are essentially pulling all type definitions into the current scope. The std namespace is huge. The alternative to this statement is to specify the namespace to which the identifier belongs using the scope operator(::) each time we declare a type. Click to know [**more**](https://www.geeksforgeeks.org/using-namespace-std-considered-bad-practice/)

**4) int main()**: This line is used to declare a function named “main” which returns data of integer type. A function is a group of statements that are designed to perform a specific task. Execution of every C++ program begins with the main() function, no matter where the function is located in the program. So, every C++ program must have a main() function. Click to know [**More about the main() function.**](https://www.geeksforgeeks.org/executing-main-in-c-behind-the-scene/)

**5) { and }**: The opening braces ‘{‘indicate the beginning of the main function and the closing braces ‘}’ indicates the ending of the main function. Everything between these two comprises the body of the main function.

**6) std::cout<<“Hello World”;**:  This line tells the compiler to display the message “Hello World” on the screen. This line is called a statement in C++. Every statement is meant to perform some task. A semi-colon ‘;’ is used to end a statement. The semi-colon character at the end of the statement is used to indicate that the statement is ending there. The std::cout is used to identify the standard character output device which is usually the desktop screen. Everything followed by the character “<<” is displayed to the output device. Click to know [**More on Input/Output.**](https://www.geeksforgeeks.org/basic-input-output-c/)

**7) return 0;** This is also a statement. This statement is used to return a value from a function and indicates the finishing of a function. This statement is baed in functions to return the results of the operations performed by a function.

**8) Indentation**: As you can see the cout and the return statement have been indented or moved to the right side. This is done to make the code more readable. In a program as Hello World, it does not hold much relevance, but as the programs become more complex, it makes the code more readable, and less error-prone. Therefore, you must always use indentations and comments to make the code more readable. Must read the [**FAQ on the style of writing programs.**](https://www.geeksforgeeks.org/facts-and-question-related-to-style-of-writing-programs-in-c-c/)

**Important Points to Note while Writing a C++ Program:**

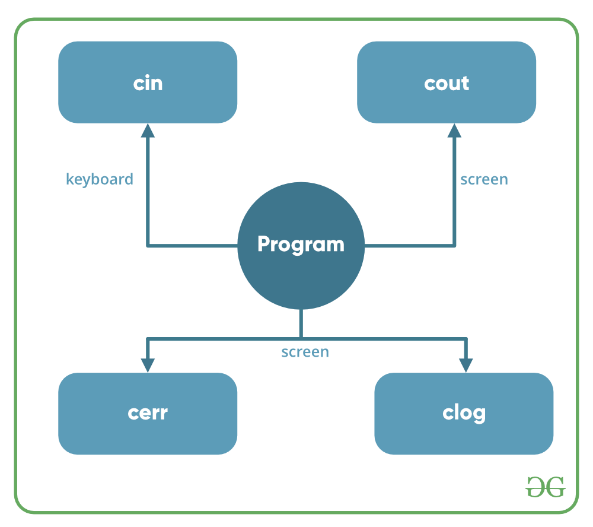
1. Always include the necessary header files for the smooth execution of functions.For example, **<iostream>** must be included to use **std::cin** and **std::cout**.
2. The execution of code begins from the **main()** function.
3. It is a good practice to use **Indentation** and **comments** in programs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

------------------------------------------------------------------------------------------------------------------------------------------------

[**Basic I/O in C++:**](https://www.geeksforgeeks.org/basic-input-output-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**.

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



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

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

The two instances **cout in C++** and **cin in C++** of the iostream class are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.

This article mainly discusses the objects defined in the header file *iostream* like the cin and cout.

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

#include <iostream>

**using** **namespace** std;

**int** main()

{

**char** sample[] = "GeeksforGeeks";

    cout << sample << " - A computer science portal for geeks";

**return** 0;

}

**Output:**

GeeksforGeeks - A computer science portal for geeks

In the above program, the insertion operator(**<<**) inserts the value of the string variable **sample** followed by the string “A computer science portal for geeks” in the standard output stream **cout** which is then displayed on the screen.

* **standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ cin statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.   
  The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.
* #include <iostream>
* **using** **namespace** std;
* **int** main()
* {
* **int** age;
* cout << "Enter your age:";
* cin >> age;
* cout << "\nYour age is: " << age;
* **return** 0;
* }

**Input :**

18

**Output:**

Enter your age:

Your age is: 18

The above program asks the user to input the age. The object cin is connected to the input device. The age entered by the user is extracted from cin using the extraction operator(**>>**) and the extracted data is then stored in the variable **age** present on the right side of the extraction operator.

* **Un-buffered standard error stream (cerr)**: The C++ cerr is the standard error stream that is used to output the errors. This is also an instance of the iostream class. As cerr in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
* The main difference between cerr and cout comes when you would like to redirect output using “cout” that gets redirected to a file if you use “cerr” the error doesn’t get stored ithe n file.(This is what un-buffered means ..It cant store the message)
* #include <iostream>
* **using** **namespace** std;
* **int** main()
* {
* cerr << "An error occurred";
* **return** 0;
* }

**Output:**

An error occurred

* **buffered standard error stream (clog)**: This is also an instance of the ostream class and used to display errors but unlike ,cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()). The error message will be displayed on the screen too.

|  |
| --- |
| * #include <iostream> * **using** **namespace** std; * **int** main() * { * clog << "An error occurred"; * **return** 0; * } |

**Output:**

An error occurred

endl and \n both seem to do the same thing but there is a subtle difference between them.

**cout << endl**inserts a new line and flushes the stream(output buffer), whereas **cout << “\n”**just inserts a new line.

Therefore, **cout << endl;**can be said equivalent to **cout << ‘\n’ << flush;**

| endl | \n |
| --- | --- |
| It is a manipulator. | It is a character. |
| It doesn’t occupy any memory. | It occupies 1-byte memory as it is a character. |
| It is a keyword and would not specify any meaning when stored in a string. | It can be stored in a string and will still convey its specific meaning of line break. |
| We cannot write ‘endl’ in between double quotations. | We can write ‘\n’ in between double quotations like cout<<“\n”; |
| It is only supported by C++. | It is supported in both C and C++. |
| It keeps flushing the queue in the output buffer throughout the process. | It flushes the output buffer only once at the end of the program |

***Note: cout << “\n”****looks performance wise better but in real****cout << endl****is much better in C++****;****As it doesn’t occupy any memory and also if flushing of stream is required.*

We can use **endl** in C++ but not in C. So **endl** runs well in C++ but if we use C, it runs an error.

|  |
| --- |
| #include <stdio.h>    **int** main() {        // code        //We cann't use endl in C  **printf**("GFG!",endl);  //So it runs error  **printf**("GFG!");  **return** 0;  }    //Code submitted by Susobhan AKhuli |

**Output:**

GFG!

GFG!

**Example 2:**

We can use **“\n”** in both C and C++ but it occupies 1-byte of memory.

* C++
* C

|  |
| --- |
| #include <stdio.h>    **int** main() {        // code  **printf**("GFG!\n");  **printf**("GFG!");  **return** 0;  }    //Code submitted by Susobhan AKhuli |

**Output:**

GFG!

GFG!

**fgets()/gets()/scanf() After scanf() in C**

**scanf()** is a library function in C. It reads standard input from stdin. **fgets()** is a library function in C. It reads a line from the specified stream and stores it into the string pointed to by the string variable. It only terminates when either:

* end-of-file is reached
* n-1 characters are read
* the newline character is read

**1)**Consider the below simple program in C. The program reads an integer using scanf(), then reads a string using fgets(),

**Input**

10

test

// C program to demonstrate the problem when

// fgets()/gets() is used after scanf()

#include <stdio.h>

**int** main()

{

**int** x;

**char** str[100];

**scanf**("%d", &x);

**fgets**(str, 100, stdin);

**printf**("x = %d, str = %s", x, str);

**return** 0;

}

**Output**

x = 10, str =

**Explanation:**The problem with the above code is scanf() reads an integer and leaves a newline character in the buffer. So fgets() only reads newline and the string “test” is ignored by the program.

**2)**The similar problem occurs when scanf() is used in a loop.

**Input:**

a

b

q

// C program to demonstrate the problem when

// scanf() is used in a loop

#include <stdio.h>

**int** main()

{

**char** c;

**printf**("Press q to quit\n");

**do** {

**printf**("Enter a character\n");

**scanf**("%c", &c);

**printf**("%c\n", c);

    } **while** (c != 'q');

**return** 0;

}

**Output**

Press q to quit

Enter a character

a

Enter a character

Enter a character

b

Enter a character

Enter a character

q

**Explanation:**We can notice that the above program prints an extra “Enter a character” followed by an extra newline. This happens because every scanf() leaves a newline character in a buffer that is read by the next scanf.

**How to Solve the Above Problem?**

* We can make**scanf()**to read a new line by using an extra **\n**, i.e., **scanf(“%d\n”, &x)**. In fact **scanf(“%d “, &x)**also works (Note the extra space).
* We can add a **getchar()**after **scanf()**to read an extra newline.

The corrected programs for the above points will be,

**1) scanf() when there is fgets() after it:**

**Input:**

10

test

|  |
| --- |
| // C program to demonstrate the problem when  // fgets()/gets() is used after scanf()  #include <stdio.h>    **int** main()  {  **int** x;  **char** str[100];  **scanf**("%d\n", &x);  **fgets**(str, 100, stdin);  **printf**("x = %d, str = %s", x, str);  **return** 0;  } |

**Output**

x = 10, str = test

**2) When scanf() is used in a loop:**

**Input:**

a

b

q

* C

|  |
| --- |
| // C program to demonstrate the problem when  // scanf() is used in a loop  #include <stdio.h>    // Driver Code  **int** main()  {  **char** c;  **printf**("Press q to quit\n");  **do** {  **printf**("Enter a character\n");  **scanf**("%c\n", &c);  **printf**("%c\n", c);      } **while** (c != 'q');  **return** 0;  } |

**Output:**Press q to quit

|  |
| --- |
| // C program to demonstrate the problem when  // fgets()/gets() is used after scanf()  #include <stdio.h>    **int** main()  {  **int** x;  **char** str[100];  **scanf**("%d\n", &x);  **fgets**(str, 100, stdin);  **printf**("x = %d, str = %s", x, str);  **return** 0;  } |

**Output**

x = 10, str = test

**2) When scanf() is used in a loop:**

**Input:**

a

b

q

* C

|  |
| --- |
| // C program to demonstrate the problem when  // scanf() is used in a loop  #include <stdio.h>    // Driver Code  **int** main()  {  **char** c;  **printf**("Press q to quit\n");  **do** {  **printf**("Enter a character\n");  **scanf**("%c\n", &c);  **printf**("%c\n", c);      } **while** (c != 'q');  **return** 0;  } |

**Output:**Press q to quit

# getline (string) in C++

# The C++ **getline()** is a standard library function that is used to read a string or a line from an input stream. It is a part of the ***<string>* header**. The getline() function extracts characters from the input stream and appends it to the string object until the delimiting character is encountered. While doing so the previously stored value in the string object *str* will be replaced by the input string if any. The getline() function can be represented in two ways:

**Syntax:**

istream& getline(istream& is,

string& str, char delim);

**2. Parameters:**

* **is:**It is an object of istream class and tells the function about the stream from which to read the input from.
* **str:** It is a string object, the input is stored in this object after being read from the stream.
* **delim:** It is the delimitation character that tells the function to stop reading further input after reaching this character.

**Example:**To demonstrate the use of delimiter in the **getline()** function

#include  <iostream>

#include  <bits/stdc++.h>

**using** **namespace** std;

//macro definitions

#define MAX\_NAME\_LEN 60  // Maximum len of your name can't be more than 60

#define MAX\_ADDRESS\_LEN 120  // Maximum len of your address can't be more than 120

#define MAX\_ABOUT\_LEN 250 // Maximum len of your profession can't be more than 250

**int** main () {

**char** y\_name[MAX\_NAME\_LEN], y\_address[MAX\_ADDRESS\_LEN], about\_y[MAX\_ABOUT\_LEN];

  cout << "Enter your name: ";

  cin.getline (y\_name, MAX\_NAME\_LEN);

  cout << "Enter your City: ";

  cin.getline (y\_address, MAX\_ADDRESS\_LEN);

  cout << "Enter your profession (press $ to complete): ";

  cin.getline (about\_y, MAX\_ABOUT\_LEN, '$');    //$ is a delimiter

  cout << "\nEntered details are:\n"<<'\n';

  cout << "Name: " << y\_name << endl;

  cout << "Address: " << y\_address << endl;

  cout << "Profession is: " << about\_y << endl;

}

**What is a buffer?**   
A temporary storage area is called a buffer. All standard input and output devices contain an input and output buffer. In standard C/C++, streams are buffered. For example, in the case of standard input, when we press the key on the keyboard, it isn’t sent to your program, instead of that  it is send to the buffer by the operating system, till the time is allotted to that program.

**How does it affect Programming?**   
On various occasions, you may need to clear the unwanted buffer so as to get the next input in the desired container and not in the buffer of the previous variable. For example, in the case of C after encountering “scanf()”, if we need to input a character array or character, and in the case of C++, after encountering the “cin” statement, we require to input a character array or a string, we require to clear the input buffer or else the desired input is occupied by a buffer of the previous variable, not by the desired container. On pressing “Enter” (carriage return) on the output screen after the first input, as the buffer of the previous variable was the space for a new container(as we didn’t clear it), the program skips the following input of the container.

// C++ Code to explain why

// not clearing the input

// buffer causes undesired

// outputs

#include<iostream>

#include<vector>

**using** **namespace** std;

**int** main()

{

**int** a;

**char** ch[80];

    // Enter input from user

    // - 4 for example

    cin >> a;

    // Get input from user -

    // "GeeksforGeeks" for example

    cin.getline(ch,80);

    // Prints 4

    cout << a << endl;

    // Printing string : This does

    // not print string

    cout << ch << endl;

**return** 0;

}

**How can it be resolved?**

**In the** **case of C :**

* + 1. **Using “ while ((getchar()) != ‘\n’); ”**: Typing “while ((getchar()) != ‘\n’);” reads the buffer characters till the end and discards them(including newline) and using it after the “scanf()” statement clears the input buffer and allows the input in the desired container.

|  |
| --- |
| * // flushes the input buffer * #include<stdio.h> * **int** main() * { * **char** str[80], ch; * // scan input from user - * // GeeksforGeeks for example * **scanf**("%s", str); * // flushes the standard input * // (clears the input buffer) * **while** ((**getchar**()) != '\n'); * // scan character from user - * // 'a' for example * ch = **getchar**(); * // Printing character array, * // prints “GeeksforGeeks”) * **printf**("%s\n", str); * // Printing character a: It * // will print 'a' this time * **printf**("%c", ch); * **return** 0; * } |

Input:

GeeksforGeeks

a

Output:

GeeksforGeeks

a

**Time Complexity: O(n),**where n is the size of the string.

**Using “ fflush(stdin) ”**: Typing “fflush(stdin)” after “scanf()” statement, also clears the input buffer but generally it’s use is avoided and is termed to be “undefined” for input stream as per the C++11 standards.

* + 1. **Using “ cin.ignore(numeric\_limits::max(),’\n’); ”**:- Typing “cin.ignore(numeric\_limits::max(),’\n’);” after the “cin” statement discards everything in the input stream including the newline.

|  |
| --- |
| // C++ Code to explain how  // "cin.ignore(numeric\_limits  // <streamsize>::max(),'\n');"  // discards the input buffer  #include<iostream>    // for <streamsize>  #include<ios>    // for numeric\_limits  #include<limits>  **using** **namespace** std;    **int** main()  {  **int** a;  **char** str[80];        // Enter input from user      // - 4 for example      cin >> a;        // discards the input buffer      cin.ignore(numeric\_limits<streamsize>::max(),'\n');        // Get input from user -      // GeeksforGeeks for example      cin.getline(str, 80);        // Prints 4      cout << a << endl;        // Printing string : This      // will print string now      cout << str << endl;    **return** 0;  } |

Input:

4

GeeksforGeeks

Output:

4

GeeksforGeeks

**Time Complexity: O(1)**

**2. Using “ cin.sync() ”:**Typing “cin.sync()” after the “cin” statement discards all that is left in the buffer. Though “cin.sync()” **does not work** in all implementations (According to C++11 and above standards).

|  |
| --- |
| // C++ Code to explain how " cin.sync();"  // discards the input buffer  #include<iostream>  #include<ios>  #include<limits>  **using** **namespace** std;    **int** main()  {  **int** a;  **char** str[80];        // Enter input from user      // - 4 for example      cin >> a;        // Discards the input buffer      cin.sync();        // Get input from user -      // GeeksforGeeks for example      cin.getline(str, 80);        // Prints 4      cout << a << endl;        // Printing string - this      // will print string now      cout << str << endl;    **return** 0;  } |

Input:

4

GeeksforGeeks

Output:

4

**Time Complexity: O(1)**

**3. Using “ cin >> ws ”:** Typing “cin>>ws” after “cin” statement tells the compiler to ignore buffer and also to discard all the whitespaces before the actual content of string or character array.

* C++

|  |
| --- |
| // C++ Code to explain how "cin >> ws"  // discards the input buffer along with  // initial white spaces of string    #include<iostream>  #include<vector>  **using** **namespace** std;    **int** main()  {  **int** a;      string s;        // Enter input from user -      // 4 for example      cin >> a;        // Discards the input buffer and      // initial white spaces of string      cin >> ws;        // Get input from user -      // GeeksforGeeks for example      getline(cin, s);        // Prints 4 and GeeksforGeeks :      // will execute print a and s      cout << a << endl;      cout << s << endl;    **return** 0;  } |

Input:

4

GeeksforGeeks

Output:

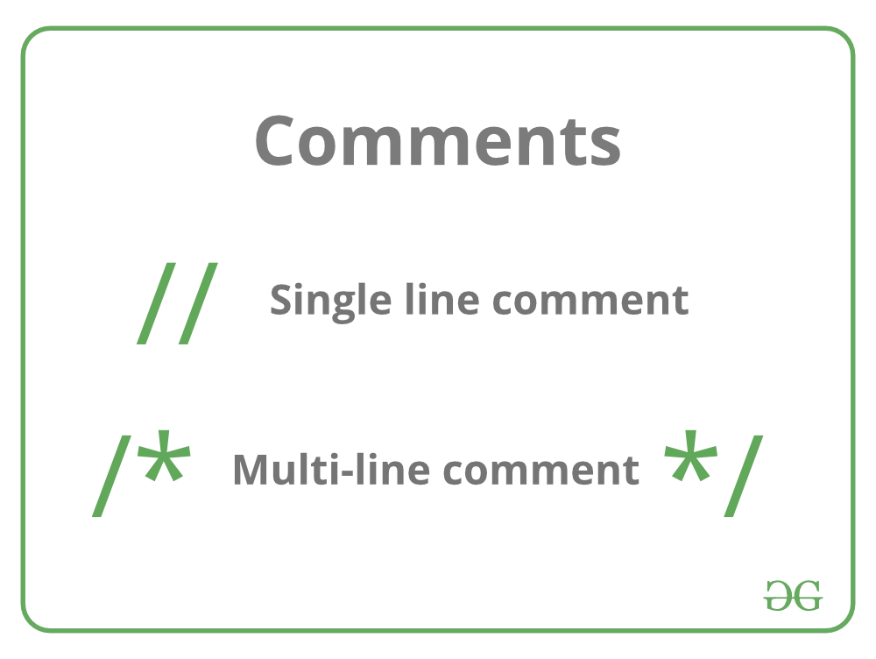
4

GeeksforGeeks

**Time Complexity: O(1)**

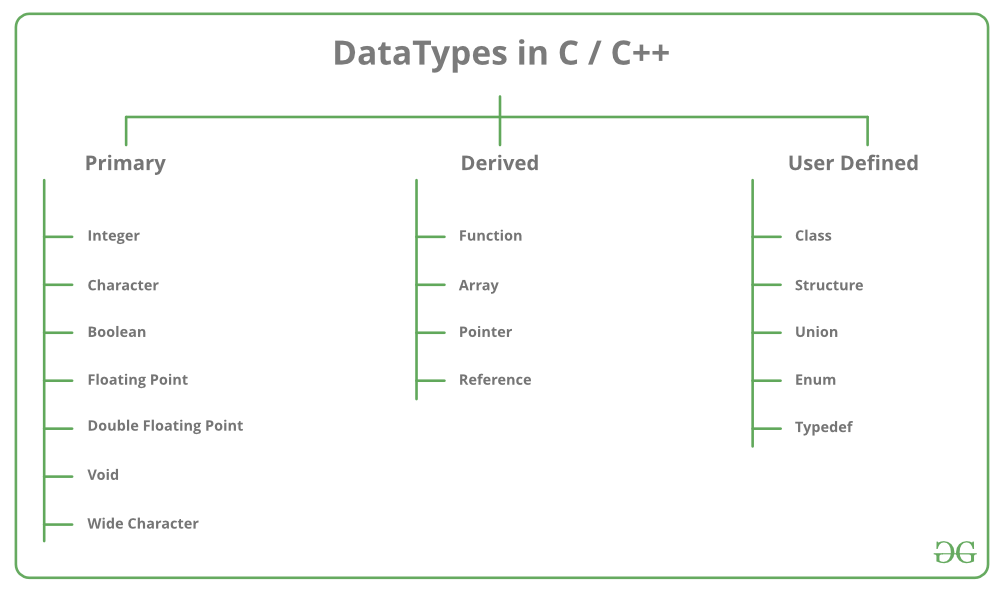
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

[**Comments in C++:**](https://www.geeksforgeeks.org/comments-in-c-c/) A well-documented program is a good practice as 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.



----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

1. [**Data Types and Modifiers in C++:**](https://www.geeksforgeeks.org/c-data-types/) 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 it 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.



**Data types in C++ are mainly divided into three 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:**](https://www.geeksforgeeks.org/derived-data-types-in-c/) The 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**](https://www.geeksforgeeks.org/user-defined-derived-data-types-in-c/): These data types are defined by the user itself. Like, as defining a class in C++ or a structure. C++ provides the following user-defined datatypes:

* Class
* Structure
* Union
* Enumeration
* Typedef defined Datatype

This article discusses **primitive data types** available in C++.

* **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**](https://www.geeksforgeeks.org/wide-char-and-library-functions-in-c/): Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit datatype. Represented by **wchar\_t**. It is generally 2 or 4 bytes long.

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

sizeof operator — sizeof operator is used to find the number of bytes occupied by a variable/data type in computer memory. Eg:     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”.

// Following is the example, which will produce 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

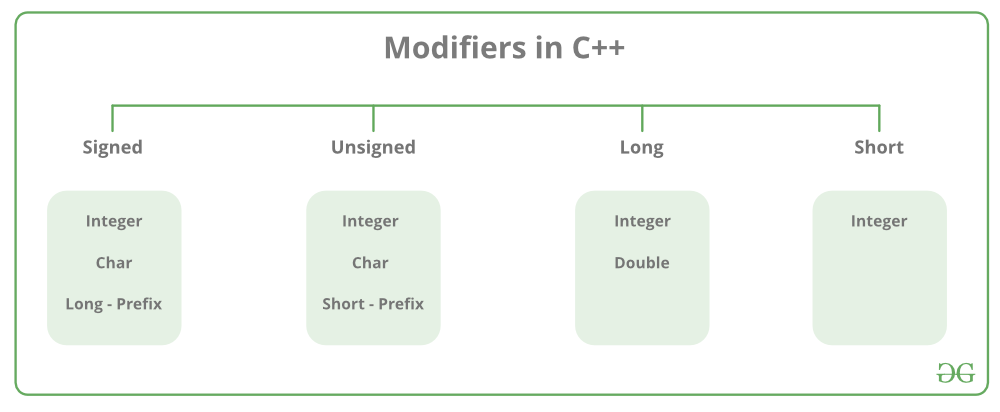
Size of double : 8

### Time Complexity: O(1)

### **Space Complexity: O(1)**

### **Datatype Modifiers**

As the name suggests, datatype modifiers are used with the 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 |
| 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 |  |
| double | 8 |  |
| long double | 12 |  |
| wchar\_t | 2 or 4 | 1 wide character |

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 |
| 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 |  |
| double | 8 |  |
| long double | 12 |  |
| wchar\_t | 2 or 4 | 1 wide character |

**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.

CHAR\_MIN                                                              Minimum value for an object of type char

CHAR\_MAX                                                                Maximum value for an object of type char

SCHAR\_MIN                                                             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                                                                 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                                                                 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                                                                 Minimum value for an object of type long int

LONG\_MAX                                                                 Maximum value for an object of type long int

ULONG\_MAX                                                                 Maximum value for an object of type Unsigned long int

LLONG\_MIN                                                                 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

|  |
| --- |
| // C++ program to 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

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)**

PRIMITIVE DATA TYPES

\*/

// number system

#include <iostream>

int main()

{

    int number1 = 15;         // Decimal

    int number2 = 017;        // Octal

    int number3 = 0x0F;       // Hexadecimal

    int number4 = 0b00001111; // Binary

    std::cout << "number1 : " << number1 << std::endl;

    std::cout << "number2 : " << number2 << std::endl;

    std::cout << "number3 : " << number3 << std::endl;

    std::cout << "number4 : " << number4 << std::endl;

    // ==========================================================================================

    // initialization

    // Braced initializers

    // Variable may contain random garbage value . WARNING

    int elephant\_count;

    int lion\_count{}; // Initializes to zero

    int dog\_count{10}; // Initializes to 10

    int cat\_count{15}; // Initializes to 15

    // Can use expression as initializer

    int domesticated\_animals{dog\_count + cat\_count};

    //

    // int new\_number{doesnt\_exist};

    // int narrowing\_conversion {2.9};//Compiler error

    std::cout << "Elephant count : " << elephant\_count << std::endl;

    std::cout << "Lion count : " << lion\_count << std::endl;

    std::cout << "Dog count : " << dog\_count << std::endl;

    std::cout << "Cat count : " << cat\_count << std::endl;

    std::cout << "Domesticated animal count : " << domesticated\_animals << std::endl;

    // Functional Initialization

    /\*

     int apple\_count(5);

     int orange\_count(10);

     int fruit\_count (apple\_count + orange\_count);

     //int bad\_initialization ( doesnt\_exist3 + doesnt\_exist4 );

     //Information lost. less safe than braced initializers

     int narrowing\_conversion\_functional (2.9);

     std::cout << "Apple count : " << apple\_count << std::endl;

     std::cout << "Orange count : " << orange\_count << std::endl;

     std::cout << "Fruit count : " << fruit\_count << std::endl;

     std::cout << "Narrowing conversion : " << narrowing\_conversion\_functional << std::endl;//Will loose info

     \*/

    // Assignment notation

    int bike\_count = 2;

    int truck\_count = 7;

    int vehicle\_count = bike\_count + truck\_count;

    int narrowing\_conversion\_assignment = 2.9;

    std::cout << "Bike count : " << bike\_count << std::endl;

    std::cout << "Truck count : " << truck\_count << std::endl;

    std::cout << "Vehicle count : " << vehicle\_count << std::endl;

    std::cout << "Narrowing conversion : " << narrowing\_conversion\_assignment << std::endl;

    // Check the size with sizeof

    std::cout << "sizeof int : " << sizeof(int) << std::endl;

    std::cout << "sizeof truck\_count : " << sizeof(truck\_count) << std::endl;

    // summary of initialization

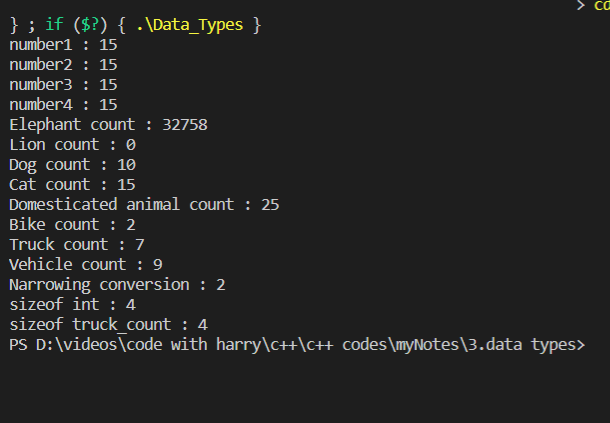
    // braced : int num{23};

    // note: only in braced initialization num{} is equal to zero and in all cases.

    // functional value(34);

    // assignment int value =89;

    // ====================================================================================================



**// INT**

 signed int value1{10};

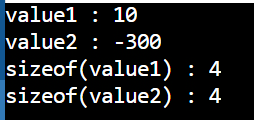
    signed int value2{-300};

    std::cout << "value1 : " << value1 << std::endl;

    std::cout << "value2 : " << value2 << std::endl;

    std::cout << "sizeof(value1) : " << sizeof(value1) << std::endl;

    std::cout << "sizeof(value2) : " << sizeof(value2) << std::endl;



    unsigned int value3{4};

    // unsigned int value4{-5}; // Compiler error.

    // short and long

    short short\_var{-32768};                 //  2 Bytes

    short int short\_int{455};                //

    signed short signed\_short{122};          //

    signed short int signed\_short\_int{-456}; //

    unsigned short int unsigned\_short\_int{456};

    int int\_var{55};           // 4 bytes

    signed signed\_var{66};     //

    signed int signed\_int{77}; //

    unsigned int unsigned\_int{77};

    long long\_var{88}; // 4 OR 8 Bytes

    long int long\_int{33};

    signed long signed\_long{44};

    signed long int signed\_long\_int{44};

    unsigned long int unsigned\_long\_int{44};

    long long long\_long{888}; // 8 Bytes

    long long int long\_long\_int{999};

    signed long long signed\_long\_long{444};

    signed long long int signed\_long\_long\_int{1234};

    unsigned long long int unsigned\_long\_long\_int{1234};

    std::cout << "Short variable : " << short\_var << " , size : "

              << sizeof(short) << " bytes" << std::endl;

    std::cout << "Short Int : " << short\_int << " , size : "

              << sizeof(short int) << " bytes" << std::endl;

    std::cout << "Signed short : " << signed\_short

              << " , size : " << sizeof(signed short) << " bytes" << std::endl;

    std::cout << "Signed short int :  " << signed\_short\_int

              << " , size : " << sizeof(signed short int) << " bytes" << std::endl;

    std::cout << "unsigned short int :  " << unsigned\_short\_int

              << " , size : " << sizeof(unsigned short int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    std::cout << "Int variable :  " << int\_var << " , size : "

              << sizeof(int) << " bytes" << std::endl;

    std::cout << "Signed variable " << signed\_var << " , size : "

              << sizeof(signed) << " bytes" << std::endl;

    std::cout << "Signed int :  " << signed\_int << " , size : "

              << sizeof(signed int) << " bytes" << std::endl;

    std::cout << "unsigned int :  " << unsigned\_int << " , size : "

              << sizeof(unsigned int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    std::cout << "Long variable :  " << long\_var << " , size : "

              << sizeof(long) << " bytes" << std::endl;

    std::cout << "Long int :  " << long\_int << " , size : "

              << sizeof(long int) << " bytes" << std::endl;

    std::cout << "Signed long :  " << signed\_long << " , size : "

              << sizeof(signed long) << " bytes" << std::endl;

    std::cout << "Signed long int : " << signed\_long\_int << " , size : "

              << sizeof(signed long int) << " bytes" << std::endl;

    std::cout << "unsigned long int : " << unsigned\_long\_int << " , size : "

              << sizeof(unsigned long int) << " bytes" << std::endl;

    std::cout << "---------------------------" << std::endl;

    std::cout << "Long long : " << long\_long << " , size : "

              << sizeof(long long) << " bytes" << std::endl;

    std::cout << "Long long int : " << long\_long\_int << " , size : "

              << sizeof(long long int) << " bytes" << std::endl;

    std::cout << "Signed long long : " << signed\_long\_long << " , size : "

              << sizeof(signed long long) << " bytes" << std::endl;

    std::cout << "Signed long long int : " << signed\_long\_long\_int << " , size : "

              << sizeof(signed long long int) << " bytes" << std::endl;

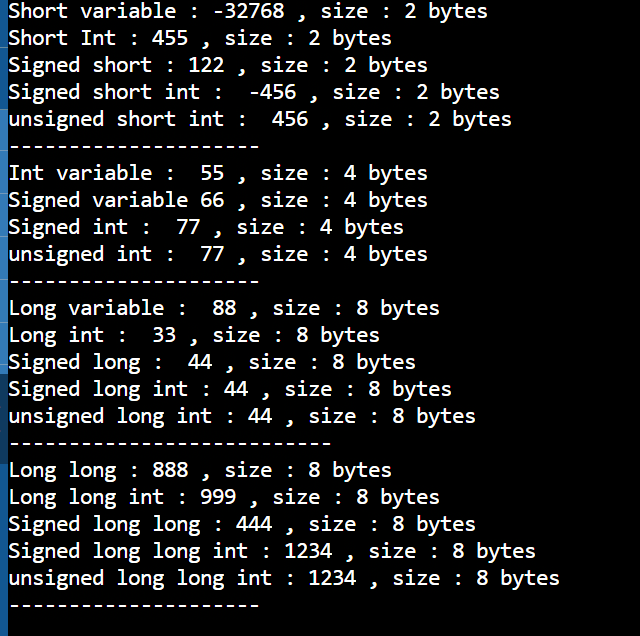
    std::cout << "unsigned long long int : " << unsigned\_long\_long\_int << " , size : "

              << sizeof(unsigned long long int) << " bytes" << std::endl;

    std::cout << "---------------------" << std::endl;

    // =================================================================

**output:**



**// FLOATS**

    // Declare and initialize the variables

    float number1{1.12345678901234567890f}; // Precision : 7

    double number2{1.12345678901234567890}; // Precision : 15

    long double number3{1.12345678901234567890L};

    // Print out the sizes

    std::cout << "sizeof float : " << sizeof(float) << std::endl;

    std::cout << "sizeof double : " << sizeof(double) << std::endl;

    std::cout << "sizeof long double : " << sizeof(long double) << std::endl;

    // Precision

    //  std::cout << std::setprecision(20); // Control the precision from std::cout.

    std::cout << "number1 is : " << number1 << std::endl; // 7 digits

    std::cout << "number2 is : " << number2 << std::endl; // 15'ish digits

    std::cout << "number3 is : " << number3 << std::endl; // 15+ digits

    // Float problems : The precision is usually too limited

    // for a lot of applications

    float number4 = 192400023.0f; // Error : narrowing conversion

    std::cout << "number4 : " << number4 << std::endl;

    // Scientific notation

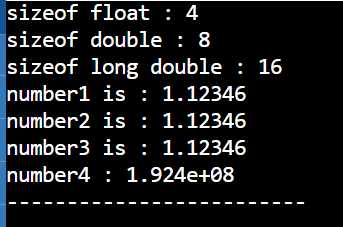
    // What we have seen so far in terms of floating point types

    // is fixed notation. There is another notation, scientific

    // that is handy if you have really huge numbers or small numbers

    // to represent

    std::cout << "-------------------------" << std::endl;



**DOUBLES:**

double number5{192400023};

    double number6{1.92400023e8};

    double number7{1.924e8}; // Can ommit the lower 00023

                             // for simplicity if our application allows that.

    double number8{0.00000000003498};

    double number9{3.498e-11}; // multiply with 10 exp(-11)

    std::cout << "number5 is : " << number5 << std::endl;

    std::cout << "number6 is : " << number6 << std::endl;

    std::cout << "number7 is : " << number7 << std::endl;

    std::cout << "number8 is : " << number8 << std::endl;

    std::cout << "number9 is : " << number9 << std::endl;

    // Infinity and Nan

    std::cout << std::endl;

    std::cout << "Infinity and NaN" << std::endl;

    double number10{-5.6};

    double number11{}; // Initialized to 0

    double number12{}; // Initialized to 0

    // Infinity

    double result{number10 / number11};

    std::cout << number10 << "/" << number11 << "  yields " << result << std::endl;

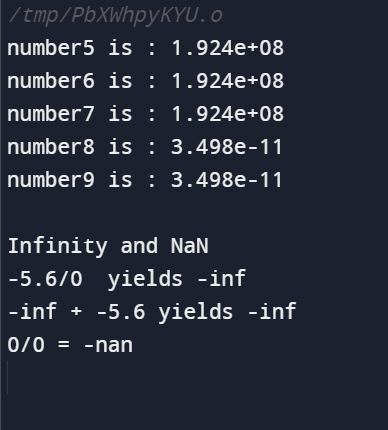
    std::cout << result << " + " << number10 << " yields " << result + number10 << std::endl;

    // NaN

    result = number11 / number12;

    std::cout << number11 << "/" << number12 << " = " << result << std::endl;

    // ==========================================================================================================



**// BOOLEAN**

 bool red\_light{false};

    bool green\_light{true};

    if (red\_light == true)

    {

        std::cout << "Stop!" << std::endl;

    }

    else

    {

        std::cout << "Go through!" << std::endl;

    }

    if (green\_light)

    {

        std::cout << "The light is green!" << std::endl;

    }

    else

    {

        std::cout << "The light is NOT green!" << std::endl;

    }

    // sizeof()

    std::cout << "sizeof(bool) : " << sizeof(bool) << std::endl;

    // Printing out a bool

    // 1 -->> true

    // 0 -->> false

    std::cout << std::endl;

    std::cout << "red\_light : " << red\_light << std::endl;

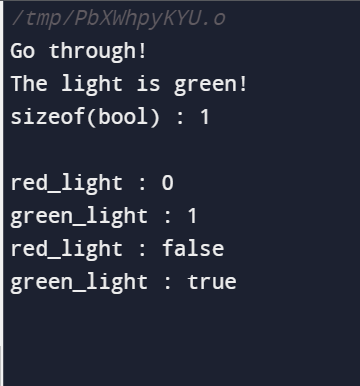
    std::cout << "green\_light : " << green\_light << std::endl;

    std::cout << std::boolalpha;

    std::cout << "red\_light : " << red\_light << std::endl;

    std::cout << "green\_light : " << green\_light << std::endl;

    //    =========================================================================================================



**// CHAR**

   char character1{'a'};

    char character2{'r'};

    char character3{'r'};

    char character4{'o'};

    char character5{'w'};

    std::cout << character1 << std::endl;

    std::cout << character2 << std::endl;

    std::cout << character3 << std::endl;

    std::cout << character4 << std::endl;

    std::cout << character5 << std::endl;

    // One byte in memory : 2^8 = 256 different values (0 ~ 255)

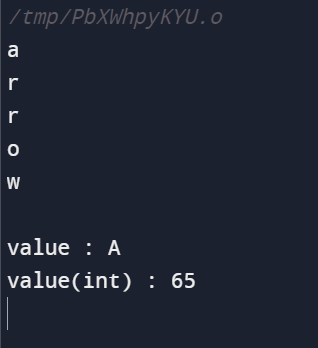
    std::cout << std::endl;

    char value = 65;                               // ASCII character code for 'A'

    std::cout << "value : " << value << std::endl; // A

    std::cout << "value(int) : " << static\_cast<int>(value) << std::endl;

    //  ============================================================================================



**//AUTO**

auto var1{12};

    auto var2{13.0};

    auto var3{14.0f};

    auto var4{15.0l};

    auto var5{'e'};

    // int modifier suffixes

    auto var6{123u};  // unsigned

    auto var7{123ul}; // unsigned long

    auto var8{123ll}; // long long

    std::cout << "var1 occupies : " << sizeof(var1) << " bytes" << std::endl;

    std::cout << "var2 occupies : " << sizeof(var2) << " bytes" << std::endl;

    std::cout << "var3 occupies : " << sizeof(var3) << " bytes" << std::endl;

    std::cout << "var4 occupies : " << sizeof(var4) << " bytes" << std::endl;

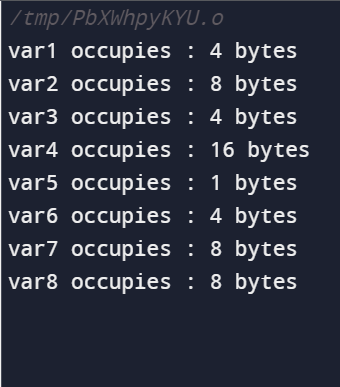
    std::cout << "var5 occupies : " << sizeof(var5) << " bytes" << std::endl;

    std::cout << "var6 occupies : " << sizeof(var6) << " bytes" << std::endl;

    std::cout << "var7 occupies : " << sizeof(var7) << " bytes" << std::endl;

    std::cout << "var8 occupies : " << sizeof(var8) << " bytes" << std::endl;

    //    ==========================================================================



**//  VOID**

    /\*

   VOID is a data used for the function declarations

   when a function is meant to give no return value only perform the operation then the void data types is used

   example: void main (){}

   example : void sorting(int arr[],int size){}

   \*/

# Uninitialized primitive data types in C/C++

**What do you think happens when you use an uninitialized primitive data type?**   
Well you may assume that the compiler should assign your primitive type variable with meaningful values like 0 for int, 0.0 for float. What about char data type?

Let’s find the answer to that by running the code in the IDE.

#include <iostream>

**using** **namespace** std;

**int** main()

{

    // The following primitive data type variables will not

    // be initialized with any default values

**char** ch;

**float** f;

**int** i;

**double** d;

**long** l;

    cout << ch << endl;

    cout << f << endl;

    cout << i << endl;

    cout << d << endl;

    cout << l << endl;

**return** 0;

}

Output in GFGs IDE: 

5.88052e-39

0

6.9529e-310

0

Output in Codechef IDE: 

0

0

0

0

**Why C/C++ compiler does not initialize variables with default values?**   
“One of the things that has 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 or garbage values as memory space is provided when we define a data type. It is considered a best practice to initialize a primitive data type variable before using it in code.

# Undefined Behavior in C and C++

When we run a code, sometimes we see absurd results instead of expected output. So, in C/C++ programming, undefined behavior means when the program fails to compile, or it may execute incorrectly, either crashes or generates incorrect results, or when it may fortuitously do exactly what the programmer intended. Whenever the result of an executing program is unpredictable, it is said to have undefined behavior.

As a C programmer, understanding undefined behavior is very important for optimal coding and for the program to yield a good efficiency, especially when it comes to there are C codes embedded in system design.

### Examples:

**Division By Zero**

int val = 5;

return val / 0; // undefined behavior

// C++ Program to demonstrate

// division by 0

#include <iostream>

**using** **namespace** std;

**int** main()

{

**int** x = 25, y = 0;

**int** z = x / y;

    cout << z;

**return** 0;

}

// This code is contributed by sarajadhav12052009

Out put:

/bin/bash: line 1: 16 Floating point exceptiontimeout 15s ./192d8cee-32a1-40b3-99e4-dd4b8848babb < 192d8cee-32a1-40b3-99e4-dd4b8848babb.in

Error:::

// C++ Program to demonstrate

// Uninitialized variables

#include <iostream>

**using** **namespace** std;

**int** main()

{

**bool** val;

**if** (val)

**printf**("TRUE");

**else**

**printf**("FALSE");

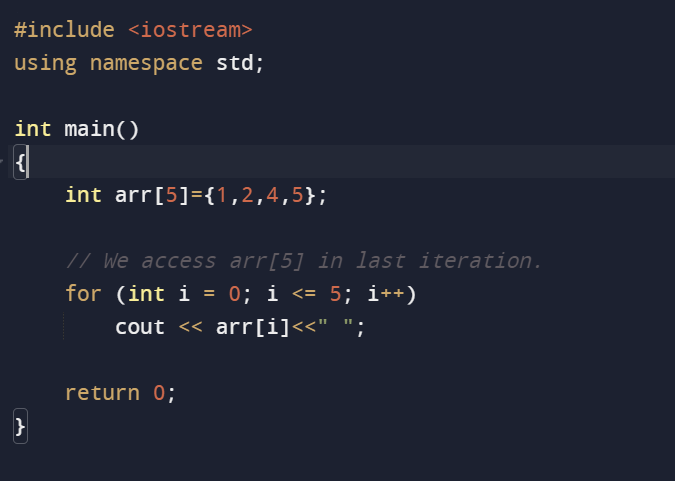
}

// This is contributed by sarajadhav12052009

**Memory accesses outside of array bounds**

int arr[4] = {0, 1, 2, 3};

return arr[5]; // undefined behavior for indexing out of bounds

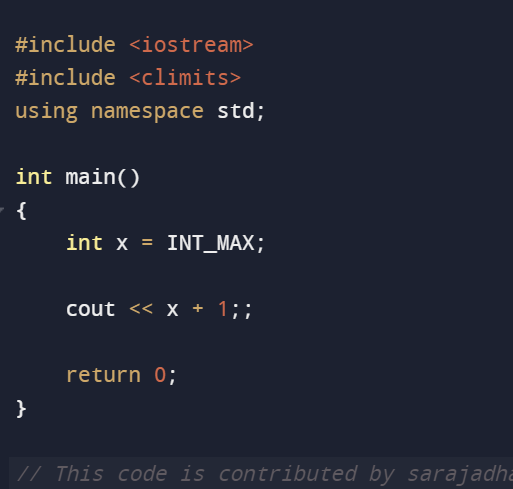


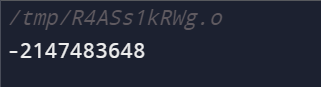


**Signed integer overflow**

int x = INT\_MAX;

printf("%d", x + 1); // undefined behavior





**Null pointer dereference**

val = 0;

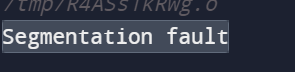
int ptr = \*val; // undefined behavior for dereferencing a null pointer

**Modification of string literal**

char\* s = "geeksforgeeks";

s[0] = 'e'; // undefined behavior





**Accessing a NULL Pointer, etc.**

int\* ptr = NULL;

printf("%d", \*ptr); // undefined behavior for accessing NULL Pointer

// C++ Program to demonstrate

// accessing value of NULL pointer

#include <iostream>

**using** **namespace** std;

**int** main()

{

**int**\* ptr = NULL;

    cout << \*ptr;

**return** 0;

}

// This code is contributed by sarajadhav12052009

// C++ Program to demonstrate modifying a variable

// multiple times before a defined sequence point

#include <iostream>

**using** **namespace** std;

**int** main()

{

**int** i = 8;

**int** p = i++ \* i++;

    cout << p;

}

// This code is contributed by sarajadhav12052009

**Output**

72

Sometimes compilers may diagnose simple errors, however, sometimes they are not designed to diagnose the undefined behavior.

**Explanation:** The program produces 72 as output in most of the compilers, but implementing software based on this assumption is not a good idea.

The output of all of the above programs is **unpredictable** (or **undefined**). The compilers (implementing the C/C++ standard) are free to do anything as these are undefined by the C and C++ standards.   
Language like Java, trap errors as soon as they are found but languages like C and C++ in a few cases keep on executing the code in a faulty manner which may result in unpredictable results. The program can crash with any type of error message, or it can unknowingly corrupt the data which is a grave issue to deal with.

**Importance of knowing about Undefined Behaviour:**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’s code might be actually difficult in tracing the root to the undefined error.

**Risks and Disadvantages of Undefined Behaviour**

* The programmers sometimes rely on a particular implementation (or compiler) of undefined behavior which may cause problems when the compiler is changed/upgraded. For example, the last program produces 72 as output in most of the compilers, but implementing software based on this assumption is not a good idea.
* Undefined behaviors may also cause security vulnerabilities, especially due to the cases when an array out of bound is not checked (causes buffer overflow attack).

**Advantages of Undefined Behaviour**

* C and C++ have undefined behaviours because it allows compilers to avoid lots of checks. Suppose a set of code with a greater performing array need not keep a look at the bounds, which avoids the need for a complex optimization pass to check such conditions outside loops. The tightly bound loops and speed up the program from thirty to fifty percent when it gains an advantage of the undefined nature of signed overflow, which is generally offered by the C compiler.
* We also have another advantage of this as it allows us to store a variable’s value in a processor register and manipulate it over time that is larger than the variable in the source code.
* It also helps in wrap-around then compile-time checks which would not be possible without the greater knowledge of the undefined behaviour in the C/C++ compiler.

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# 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.

1.The name of the variable contain**letters**, **digits**, and **underscores**.

2.The name of the variable are **case sensitive (**ex**Arr** and **arr** both are different variable).

3.The name of the variable does not contain any whitespace and**special characters**(ex**#,$,%,\*** etc).

4.All the variable name must begin with a letter of the **alphabet** or an **underscore**(\_).

5.We cannot used**C++** **keyword**(ex float,double,class)as a variable name.

**How to declare variables?**

A typical variable declaration is of the form: 

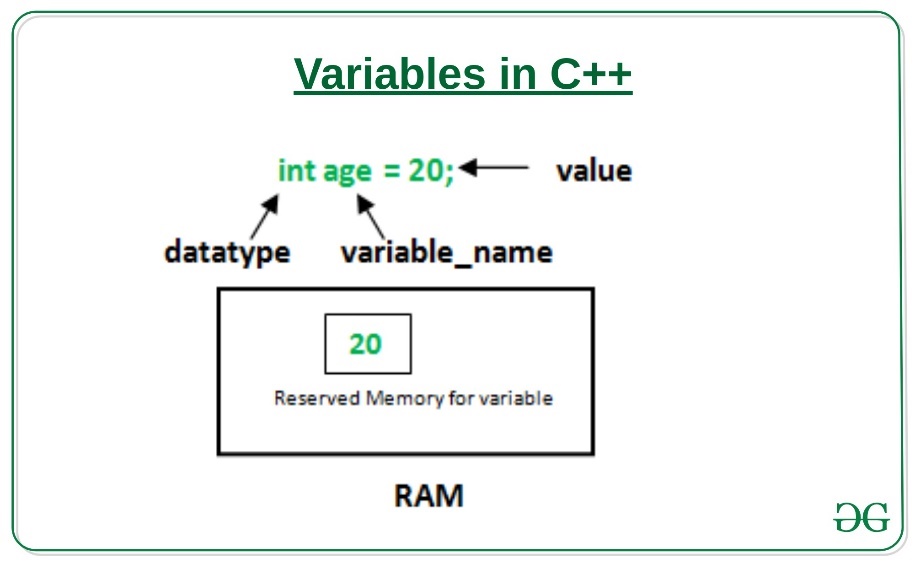
// Declaring a single variable

type variable\_name;

// Declaring multiple variables:

type variable1\_name, variable2\_name, variable3\_name;

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



**Examples**: 

// Declaring float variable

float simpleInterest;

// Declaring integer variable

int time, speed;

// Declaring character variable

char var;

**Difference between variable declaration and definition**

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

// C++ program to show difference b/w 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

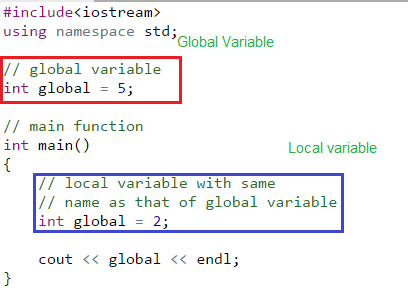
**Types of variables**

There are three types of variables based on [the scope of variables in C++](https://www.geeksforgeeks.org/scope-of-variables-in-c/):

# Scope of Variables in C++

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

1. Local Variables
2. Global Variables



### Local Variables

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

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

// CPP program to illustrate

// usage of local variables

#include<iostream>

**using** **namespace** std;

**void** func()

{

    // this variable is local to the

    // function func() and cannot be

    // accessed outside this function

**int** age=18;

}

**int** main()

{

    cout<<"Age is: "<<age;

**return** 0;

}

Output:

Error: age was not declared in this scope

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

### Global Variables

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

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

* Local Variables
* Instance Variables
* Static Variables

/ CPP program to illustrate

// usage of global variables

#include<iostream>

**using** **namespace** std;

// global variable

**int** global = 5;

// global variable accessed from

// within a function

**void** display()

{

    cout<<global<<endl;

}

// main function

**int** main()

{

    display();

    // changing value of global

    // variable from main function

    global = 10;

    display();

}

Output:

5

10

In the program, the variable “global” is declared at the top of the program outside all of the functions so it is a global variable and can be accessed or updated from anywhere in the program.

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

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

// CPP program to illustrate

// scope of local variables

// and global variables together

#include<iostream>

**using** **namespace** std;

// global variable

**int** global = 5;

// main function

**int** main()

{

    // local variable with same

    // name as that of global variable

**int** global = 2;

    cout << global << endl;

}

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

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

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

What if we want to do the opposite of above task. What if we want to access global variable when there is a local variable with same name?   
To solve this problem we will need to use the [**scope resolution operator**](https://www.geeksforgeeks.org/scope-resolution-operator-in-c/). Below program explains how to do this with the help of scope resolution operator.

Scope resolution operator:

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

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

// C++ program to show that we can access a global variable

// using scope resolution operator :: when there is a local

// variable with same name

#include<iostream>

**using** **namespace** std;

**int** x;  // Global x

**int** main()

{

**int** x = 10; // Local x

  cout << "Value of global x is " << ::x;

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

**return** 0;

}

Output:

Value of global x is 0

Value of local x is 10

**2) To define a function outside a class.**

|  |
| --- |
| // C++ program to show that scope resolution operator :: is used  // to define a function outside a class  #include<iostream>  **using** **namespace** std;    **class** A  {  **public**:       // Only declaration  **void** fun();  };    // Definition outside class using ::  **void** A::fun()  {     cout << "fun() called";  }    **int** main()  {     A a;     a.fun();  **return** 0;  } |

Output:

fun() called

**To access a class’s static variables.**

|  |
| --- |
| * #include<iostream> * **using** **namespace** std; * **class** Test * { * **static** **int** x; * **public**: * **static** **int** y; * // Local parameter 'a' hides class member * // 'a', but we can access it using :: * **void** func(**int** x) * { * // We can access class's static variable * // even if there is a local variable * cout << "Value of static x is " << Test::x; * cout << "\nValue of local x is " << x; * } * }; * // In C++, static members must be explicitly defined * // like this * **int** Test::x = 1; * **int** Test::y = 2; * **int** main() * { * Test obj; * **int** x = 3 ; * obj.func(x); * cout << "\nTest::y = " << Test::y; * **return** 0; * } |

Output:

Value of static x is 1

Value of local x is 3

Test::y = 2;

**4) In case of multiple Inheritance:**  
If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

|  |
| --- |
| // Use of scope resolution operator in multiple inheritance.  #include<iostream>  **using** **namespace** std;    **class** A  {  **protected**:  **int** x;  **public**:      A() { x = 10; }  };    **class** B  {  **protected**:  **int** x;  **public**:      B() { x = 20; }  };    **class** C: **public** A, **public** B  {  **public**:  **void** fun()     {        cout << "A's x is " << A::x;        cout << "\nB's x is " << B::x;     }  };    **int** main()  {      C c;      c.fun();  **return** 0;  } |

Output:

A's x is 10

B's x is 20

**5) For namespace**  
If a class having the same name exists inside two namespace we can use the namespace name with the scope resolution operator to refer that class without any conflicts

|  |
| --- |
| // Use of scope resolution operator for namespace.  #include<iostream>      **int** main(){      std::cout << "Hello" << std::endl;    } |

Here, cout and endl belong to the std namespace.

**6) Refer to a class inside another class:**  
If a class exists inside another class we can use the nesting class to refer the nested class using the scope resolution operator

// Use of scope resolution class inside another class.

#include<iostream>

**using** **namespace** std;

**class** outside

{

**public**:

**int** x;

**class** inside

      {

**public**:

**int** x;

**static** **int** y;

**int** foo();

      };

};

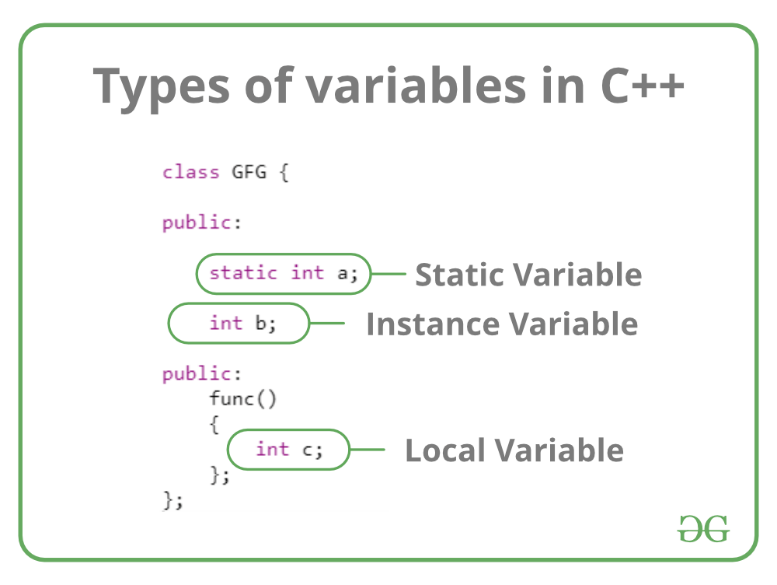
**int** outside::inside::y = 5;

**int** main(){

    outside A;

    outside::inside B;

}



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](https://www.geeksforgeeks.org/static-keyword-cpp/) 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.**
* Syntax for static and instance variables:

class Example

{

static int a; // static variable

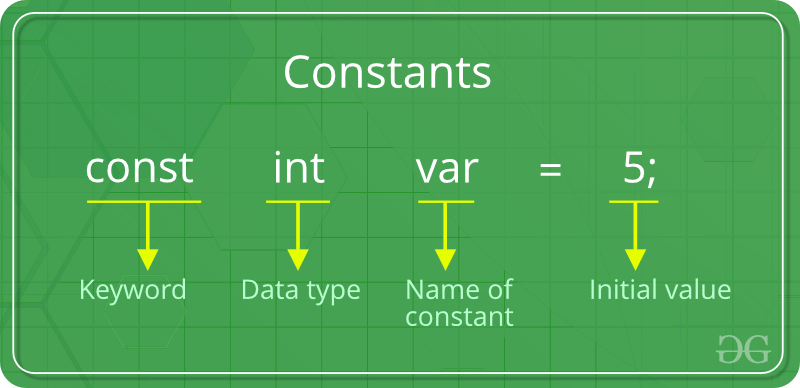
int b; // instance variable

}

------------------------------------------------------------------------------------------------------------------------------------------------

# Constants in C/C++

As the name suggests the name constants are given to such variables or values in C/C++ programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any types of constants 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, int varies from 0 to 255.



**Defining Constants:**

In C/C++ program we can define constants in two ways as shown below: 

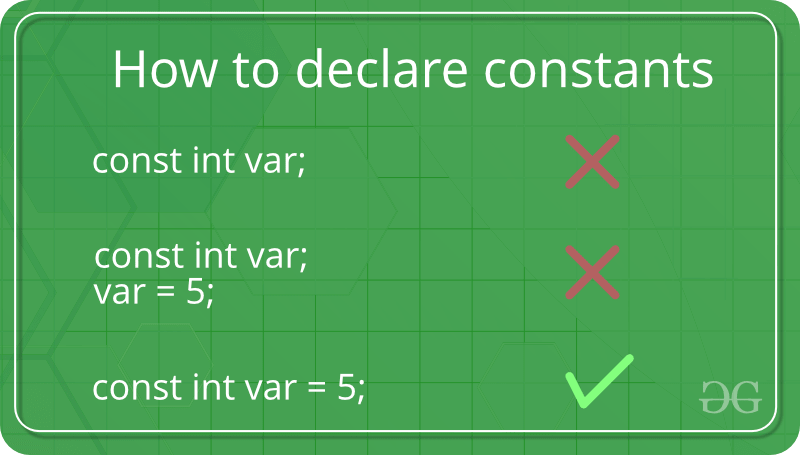
1. Using *#define* preprocessor directive
2. Using a *const* keyword

**Using *#define* preprocessor directive:** This directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:

#define identifierName value

* **identifierName:** It is the name given to constant.
* **value:** This refers to any value assigned to identifierName.

**using a *const* keyword**: Using *const* keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a *const* keyword.



#include <iostream>

**using** **namespace** std;

**int** main() {

    // int constant

**const** **int** intVal = 10;

    // Real constant

**const** **float** floatVal = 4.14;

    // char constant

**const** **char** charVal = 'A';

    // string constant

**const** string stringVal = "ABC";

    cout << "Integer Constant: " << intVal << "\n";

    cout << "Floating point Constant: " << floatVal << "\n";

    cout << "Character Constant: "<< charVal << "\n";

    cout << "String Constant: "<< stringVal << "\n";

**return** 0;

}

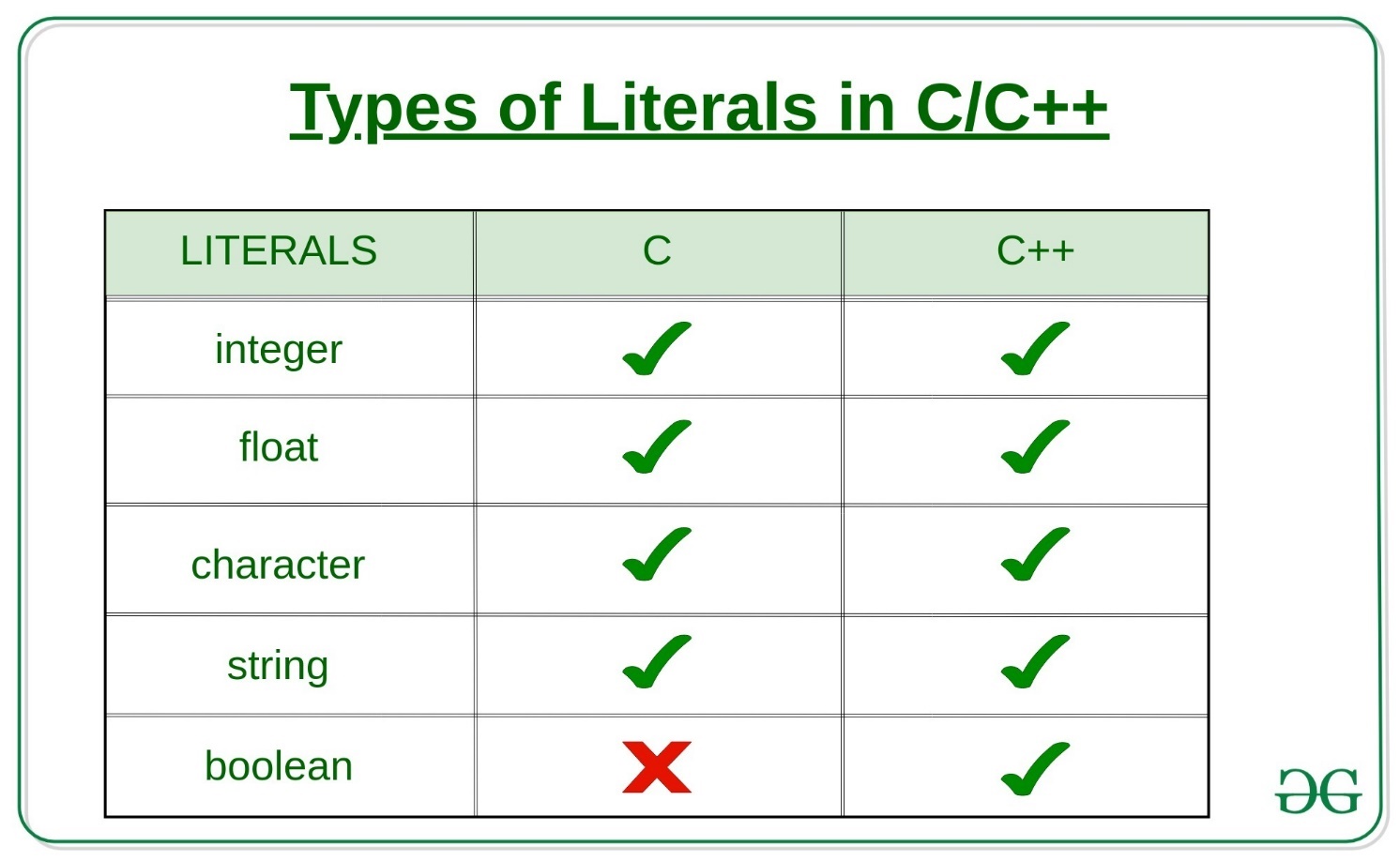
**Output:**

Integer constant: 10

Floating point constant: 4.14

Character constant: A

String constant: ABC

**Literals:** 

The values assigned to each constant variable are referred to as the **literals**. Generally, both terms, constants, and literals are used interchangeably.

For example, “const int = 5;“, is a constant expression and the value 5 is referred to as a constant integer literal. There are 4 types of literal in C and five types of literal in C++.

### 1) [**Integer Literals**](https://www.geeksforgeeks.org/integer-literal-in-c-cpp-prefixes-suffixes/)

These are used to represent and store the integer values. Integer literals are expressed in two types i.e.

**A) Prefixes:** The Prefix of the integer literal indicates the base in which it is to be read.

**For Example:**

***0x10****= 16*

*Because 0x prefix represents a HexaDecimal base. So 10 in HexaDecimal is 16 in Decimal. Hence the value 16.*

There are basically represented in 4 types.

**a. Decimal-literal(base 10):** A **non-zero decimal digit** followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

**Example:**

56, 78

**b. Octal-literal(base 8):** a **0** followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7).

**Example:**

045, 076, 06210

**c. Hex-literal(base 16):** **0x** or **0X** followed by one or more hexadecimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).

**Example:**

0x23A, 0Xb4C, 0xFEA

**d. Binary-literal(base 2):** **0b** or **0B** followed by one or more binary digits(0, 1).

**Example:**

0b101, 0B111

**B) Suffixes:** The Prefix of the integer literal indicates the type in which it is to be read.

**For example:**

*12345678901234LL*

*indicates a long long integer value 12345678901234 because of the suffix LL*

These are represented in many ways according to their data types.

* **int:** No suffix is required because integer constant is by default assigned as an int data type.
* **unsigned int:** character u or U at the end of an integer constant.
* **long int:** character l or L at the end of an integer constant.
* **unsigned long int:** character ul or UL at the end of an integer constant.
* **long long int:** character ll or LL at the end of an integer constant.
* **unsigned long long int:** character ull or ULL at the end of an integer constant.

**Example:**

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {      // constant integer literal  **const** **int** intVal = 10;      cout << "Integer Literal: "           << intVal << "\n";    **return** 0;  } |

**Output:**

Integer Literal:10

### 2) Floating-Point Literals

These are used to represent and store real numbers. The real number has an integer part, real part, fractional part, and exponential part. The floating-point literals can be stored either in decimal form or exponential form. While representing the floating-point decimals one must keep two things in mind to produce valid literal:

* In the decimal form, one must include the decimal point, exponent part, or both, otherwise, it will lead to an error.
* In the exponential form, one must include the integer part, fractional part, or both, otherwise, it will lead to an error.

Few floating-point literal representations are shown below:

**Valid Floating Literals:**

10.125

1.215-10L

10.5E-3

**Invalid Floating Literals:**

123E

1250f

0.e879

**Example:**

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {      // Real literal  **const** **float** floatVal = 4.14;        cout << "Floating-point literal: "           << floatVal << "\n";  **return** 0;  } |

**Output:**

Floating point literal: 4.14

### **3) Character Literal**

This refers to the literal that is used to store a single character within a single quote. To store multiple characters, one needs to use a character array. Storing more than one character within a single quote will throw a warning and display just the last character of the literal. It gives rise to the following two representations:

**A. char type:** This is used to store normal character literal or the narrow-character literals. This is supported by both C and C++.

**Example:**

// **For C**

char chr = 'G';

// **For C++**

char chr = 'G';

**B. wchar\_t type:** This literal is supported only in C++ and **not in C**. If the character is followed by L, then the literal needs to be stored in wchar\_t. This represents a wide-character literal.

**Example:**

// **Not Supported For C**

// **For C++**

wchar\_t chr = L'G';

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {      // constant char literal  **const** **char** charVal = 'A';        // wide char literal  **const** **wchar\_t** charVal1 = L'A';        cout << "Character Literal: "           << charVal << "\n";      cout << "Wide\_Character Literal: "           << charVal1 << "\n";    **return** 0;  }    // output  // Character Literal: A  // Wide\_Character Literal: 65 |

**Output:**

Character Literal: A

[**Escape Sequences**](https://www.geeksforgeeks.org/escape-sequences-c/)**:** There are various special characters that one can use to perform various operations.

### 4) String Literals

String literals are similar to that character literals, except that they can store multiple characters and uses a double quote to store the same. It can also accommodate the special characters and escape sequences mentioned in the table above.

**Example:**

// For C

char stringVal[] = "GeeksforGeeks";

// For C++

string stringVal = "GeeksforGeeks"

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **const** string str          = "Welcome\nTo\nGeeks\tFor\tGeeks";      cout << str;  **return** 0;  } |

**Output:**

Welcome

To

Geeks For Geeks

### 5) Boolean Literals

This literal is provided **only in C++** and **not in C**. They are used to represent the boolean datatypes. These can carry two values:

* **true:** To represent True value. This must not be considered equal to int 1.
* **false:** To represent False value. This must not be considered equal to int 0.

**Example:**

|  |
| --- |
| // C++ program to show Boolean literals    #include <iostream>  **using** **namespace** std;    **int** main()  {  **const** **bool** isTrue = **true**;  **const** **bool** isFalse = **false**;        cout << "isTrue? "          << isTrue << "\n";      cout << "isFalse? "          << isFalse << "\n";    **return** 0;  } |

**Output:**

isTrue? 1

isFalse? 0

#include <iostream>

int main(){

    //Literal types : u and l combinations for unsigned and long.

    unsigned char unsigned\_char {53u};// 555U would fail because of narrowing

    //2 Bytes

    short short\_var {-32768} ; //  No special literal type for short)

    short int short\_int {455} ; //  No special literal type for short

    signed short signed\_short {122}; // No special literal type for short

    signed short int signed\_short\_int {-456}; // No special literal type for short

    unsigned short int unsigned\_short\_int {5678U };

    // 4 Bytes

    const int int\_var {55} ;            //

    signed signed\_var {66};             //

    signed int signed\_int {77};         //

    unsigned int unsigned\_int {555U};       //

    //4 or 8 Bytes

    long long\_var {88L}; // 4 OR 8 Bytes

    long int long\_int {33L};

    signed long signed\_long {44l};

    signed long int signed\_long\_int {44l};

    unsigned long int unsigned\_long\_int {555ul};

    long long long\_long {888ll};// 8 Bytes

    long long int long\_long\_int {999ll};

    signed long long signed\_long\_long {444ll};

    signed long long int signed\_long\_long\_int{1234ll};

    //Grouping Numbers : C++14 and onwards

    unsigned int prize  {1'500'00'0u};

    std::cout << "The prize is : " << prize << std::endl;

    std::cout << " signed\_long\_long\_int : " << signed\_long\_long\_int << std::endl;

    //Narrowing errors

    //Possible narrowing errors are cought by the braced initializer method.

    //Assignment and functional don't catch that.

    //unsigned char distance {555u}; //Error [0~255]

    //unsigned int game\_score {-20}; //Error

    //std::cout << "game\_score : " << game\_score << std::endl;

    //With number systems - Hex : prefix with 0x

    unsigned int hex\_number{ 0x22BU}; // Dec 555

    int hex\_number2 {0x400};// Dec 1024

    std::cout << std::hex <<  "The hex number is : " << hex\_number << std::endl;

    std::cout << std::dec <<  "The hex number2 is : " << hex\_number2 << std::endl;

    //Representing colors with hex

    int black\_color {0xffffff};

    std::cout << "Black color is : " << std::dec << black\_color << std::endl;

    //Octal literals : prefix with 0

    int octal\_number {0777u}; // 511 Dec

    std::cout << "The octal number is : " << octal\_number << std::endl;

    //!!BE CAREFUL NOT TO PREFIX YOUR INTEGERS WITH 0 IF YOU MEAN DEC

    int error\_octal {055};// This is not 55 in memory , it is 45 dec

    std::cout << "The erronous octal number is : " << error\_octal << std::endl;

    //Binary literals

    unsigned int binary\_literal {0b11111111u};// 255 dec

    std::cout << "The binary literal is : " << binary\_literal << std::endl;

    // Other literals. This is just an example and we will learn

    // more about  strings as we progress in the course.

    char char\_literal {'c'};

    int number\_literal {15};

    float fractional\_literal {1.5f};

    std::string string\_literal {"Hit the road"};

    std::cout << "The char literal is : " << char\_literal << std::endl;

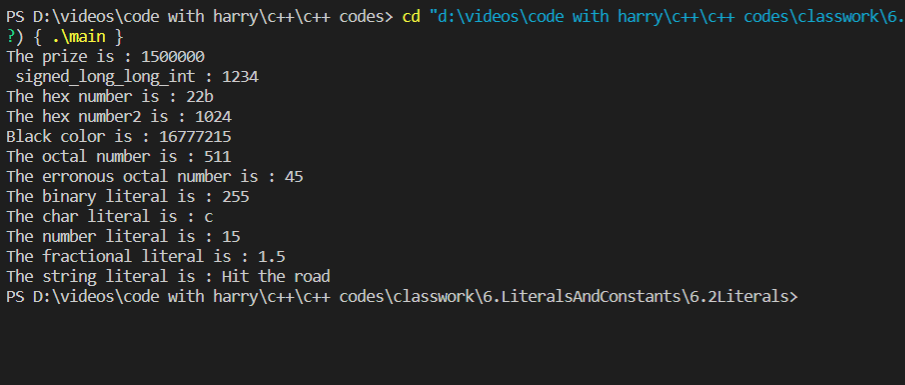
    std::cout << "The number literal is : " << number\_literal << std::endl;

    std::cout << "The fractional literal is : " << fractional\_literal << std::endl;

    std::cout << "The string literal is : " << string\_literal << std::endl;

    return 0;

}



**constexpression:**

#include <iostream>

int main(){

    constexpr int SOME\_LIB\_MAJOR\_VERSION {1237};

    constexpr int eye\_count {2};

    constexpr double PI {3.14};

    //eye\_count = 4;

    std::cout << "eye count : " << eye\_count << std::endl;

    std::cout << "PI : " << PI << std::endl;

    //int leg\_count {2}; // Non constexpr

                        // leg\_count is not known at compile time

    //constexpr int arm\_count{leg\_count}; // Error

    constexpr int room\_count{10};

    constexpr int door\_count{room\_count};// OK

    const int table\_count{5};

    constexpr int chair\_count{ table\_count \* 5};// Works

   // static\_assert( SOME\_LIB\_MAJOR\_VERSION == 123);

   // int age = 5;

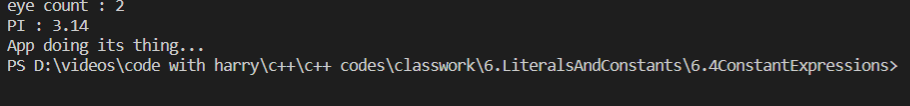
   // static\_assert( age == 5);

    std::cout << "App doing its thing..." << std::endl;

    return 0;

}

Output:



Data conversion:

Implicit conversion:

#include <iostream>

int main(){

    //      . The compiler applies implicit conversions

    //          when types are different in

    //          an expression

    //      . Conversions are always done from the smallest

    //          to the largest type in this case int is

    //          transformed to double before the expression

    //          is evaluated. Unless we are doing an assignment

    double price { 45.6 };

    int units {10};

    auto total\_price = price \* units; // units will be implicitly converted to double

    std::cout << "Total price : " << total\_price << std::endl;

    std::cout << "sizeof total\_price : " << sizeof(total\_price) << std::endl;

    //Implicit conversions in assignments

    // The assignment operation is going to cause an implicit

    // narrowing conversion , y is converted to int before assignment

    int x;

    double y {45.44};

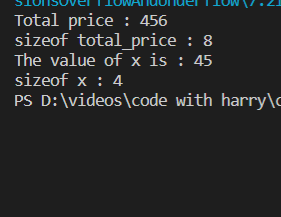
    x = y; // double to int

    std::cout << "The value of x is : " << x << std::endl; // 45

    std::cout << "sizeof x : " << sizeof(x) << std::endl;// 4

    return 0;

}



Explicit training:

#include <iostream>

int main(){

    //Implicit cast will add up the doubles,

    //then turn result into int for assignment

    double x { 12.5 };

    double y { 34.6};

    int sum = x + y;

    std::cout << "The sum  is : " << sum << std::endl;

    //Explicity cast : cast then sum up

    sum = static\_cast<int>(x) + static\_cast<int>(y) ;

    std::cout << "The sum  is : " << sum << std::endl;

    //Explicit cast : sum up then cast, same thing as implicit cast

    sum =  static\_cast<int> (x + y);

    std::cout << "Sum up then cast, result : " << sum << std::endl;

    //Old style C-cast

    double PI {3.14};

    //int int\_pi = (int)(PI);

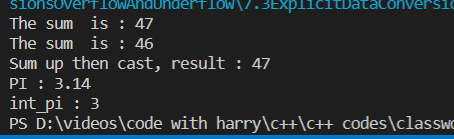
    int int\_pi = static\_cast<int>(PI);

    std::cout << "PI : " << PI << std::endl;

    std::cout << "int\_pi : " << int\_pi << std::endl;

    return 0;

}



Overflow and underflow:

#include <iostream>

int main(){

    //Overflow

    unsigned char data {250};

    ++data;

    std::cout << "data : " << static\_cast<int>(data) << std::endl;

    ++data;

    std::cout << "data : " << static\_cast<int>(data) << std::endl;

    ++data;

    std::cout << "data : " << static\_cast<int>(data) << std::endl;

    ++data;

    std::cout << "data : " << static\_cast<int>(data) << std::endl;

    ++data;

    std::cout << "data : " << std::hex <<  static\_cast<int>(data) << std::endl; // 255

    ++data;  // Overflow

    std::cout << "data : " << static\_cast<int>(data) << std::endl; // 256

    std::cout << std::dec ;

    data = 1;

    --data;

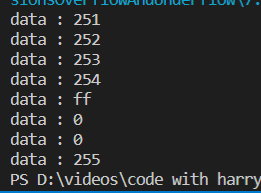
    std::cout << "data : " << static\_cast<int>(data) << std::endl;

    --data;

    std::cout << "data : " << static\_cast<int>(data) << std::endl;

    return 0;

}



# Storage Classes in C++ with Examples

# **Storage Classes** are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program. To specify the storage class for a variable, the following syntax is to be followed:

**Syntax:**

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

C++ uses 5 storage classes, namely:

1. auto
2. register
3. extern
4. static
5. mutable

# 

[**auto**](https://www.geeksforgeeks.org/type-inference-in-c-auto-and-decltype/): The auto keyword provides type inference capabilities, using which automatic deduction of the data type of an expression in a programming language can be done. This consumes less time having to write out things the compiler already knows. As all the types are deduced in compiler phase only, the time for compilation increases slightly but it does not affect the run time of the program. This feature also extends to functions and non-type template parameters. Since C++14 for functions, the return type will be deduced from its return statements. Since C++17, for non-type template parameters, the type will be deduced from the argument.

#include <iostream>

**using** **namespace** std;

**void** autoStorageClass()

{

    cout << "Demonstrating auto class\n";

    // Declaring an auto variable

    // No data-type declaration needed

**auto** a = 32;

**auto** b = 3.2;

**auto** c = "GeeksforGeeks";

**auto** d = 'G';

    // printing the auto variables

    cout << a << " \n";

    cout << b << " \n";

    cout << c << " \n";

    cout << d << " \n";

}

**int** main()

{

    // To demonstrate auto Storage Class

    autoStorageClass();

**return** 0;

}

**Output:**

Demonstrating auto class

32

3.2

GeeksforGeeks

G

[**extern**](https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/): Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well. So an extern variable is nothing but a global variable initialized with a legal value where it is declared in order to be used elsewhere. It can be accessed within any function/block. Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block. This basically signifies that we are not initializing a new variable but instead we are using/accessing the global variable only. The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program. For more information on how extern variables work, have a look at this [link](https://www.geeksforgeeks.org/understanding-extern-keyword-in-c/).

#include <iostream>

**using** **namespace** std;

// declaring the variable which is to

// be made extern an initial value can

// also be initialized to x

**int** x;

**void** externStorageClass()

{

    cout << "Demonstrating extern class\n";

    // telling the compiler that the variable

    // x is an extern variable and has been

    // defined elsewhere (above the main

    // function)

**extern** **int** x;

    // printing the extern variables 'x'

    cout << "Value of the variable 'x'"

         << "declared, as extern: " << x << "\n";

    // value of extern variable x modified

    x = 2;

    // printing the modified values of

    // extern variables 'x'

    cout

        << "Modified value of the variable 'x'"

        << " declared as extern: \n"

        << x;

}

**int** main()

{

    // To demonstrate extern Storage Class

    externStorageClass();

**return** 0;

}

**Output:**

Demonstrating extern class

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

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

2

.

[**static**](https://www.geeksforgeeks.org/static-variables-in-c/): This storage class is used to declare static variables which are popularly used while writing programs in C++ language. Static variables have a property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope. So we can say that they are initialized only once and exist until the termination of the program. Thus, no new memory is allocated because they are not re-declared. Their scope is local to the function to which they were defined. Global static variables can be accessed anywhere in the program. By default, they are assigned the value 0 by the compiler.

#include <iostream>

**using** **namespace** std;

// Function containing static variables

// memory is retained during execution

**int** staticFun()

{

    cout << "For static variables: ";

**static** **int** count = 0;

    count++;

**return** count;

}

// Function containing non-static variables

// memory is destroyed

**int** nonStaticFun()

{

    cout << "For Non-Static variables: ";

**int** count = 0;

    count++;

**return** count;

}

**int** main()

{

    // Calling the static parts

    cout << staticFun() << "\n";

    cout << staticFun() << "\n";

    ;

    // Calling the non-static parts

    cout << nonStaticFun() << "\n";

    ;

    cout << nonStaticFun() << "\n";

    ;

**return** 0;

}

**Output:**

For static variables: 1

For static variables: 2

For Non-Static variables: 1

For Non-Static variables: 1

[**register**](https://www.geeksforgeeks.org/understanding-register-keyword/): This storage class declares register variables which have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available. This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program. If a free register is not available, these are then stored in the memory only. Usually, a few variables which are to be accessed very frequently in a program are declared with the register keyword which improves the running time of the program. An important and interesting point to be noted here is that we cannot obtain the address of a register variable using pointers.

#include <iostream>

**using** **namespace** std;

**void** registerStorageClass()

{

    cout << "Demonstrating register class\n";

    // declaring a register variable

**register** **char** b = 'G';

    // printing the register variable 'b'

    cout << "Value of the variable 'b'"

         << " declared as register: " << b;

}

**int** main()

{

    // To demonstrate register Storage Class

    registerStorageClass();

**return** 0;

}

**Output:**

Demonstrating register class

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

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

#include <iostream>

**using** std::cout;

**class** Test {

**public**:

**int** x;

    // defining mutable variable y

    // now this can be modified

**mutable** **int** y;

    Test()

    {

        x = 4;

        y = 10;

    }

};

**int** main()

{

    // t1 is set to constant

**const** Test t1;

    // trying to change the value

    t1.y = 20;

    cout << t1.y;

    // Uncommenting below lines

    // will throw error

    // t1.x = 8;

    // cout << t1.x;

**return** 0;

}

**Output:**

20

# Operators in C / C++

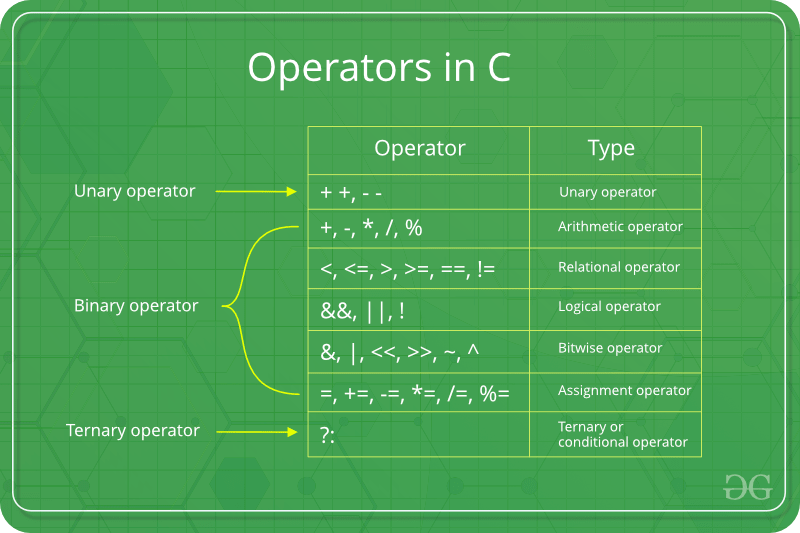
**Operators** are the foundation of any programming language. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands. For example, ‘+’ is an operator used for addition, as shown below:

c = a + b;

Here, ‘+’ is the operator known as the addition operator and ‘a’ and ‘b’ are operands. The addition operator tells the compiler to add both of the operands ‘a’ and ‘b’.

***C/C++ has many built-in operators and can be classified into 6 types:***

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Other Operators



**1. Arithmetic Operators:**

These operators are used to perform arithmetic/mathematical operations on operands. Examples: (+, -, \*, /, %,++,–). Arithmetic operators are of two types:

**a) Unary Operators**: Operators that operate or work with a single operand are unary operators. For example: Increment(++) and Decrement(–) Operators

int val = 5;

++val; // 6

**b) Binary Operators**: Operators that operate or work with two operands are binary operators. For example: Addition(+), Subtraction(-), multiplication(\*), Division(/) operators

int a = 7;

int b = 2;

cout<<a+b; // 9

**2. Relational Operators:**

These are used for the comparison of the values of two operands. For example, checking if one operand is equal to the other operand or not, whether an operand is greater than the other operand or not, etc. Some of the relational operators are (==, >= , <= )(See [this](https://www.geeksforgeeks.org/operators-in-c-set-2-relational-and-logical-operators/) article for more reference).

int a = 3;

int b = 5;

a < b;

// operator to check if a is smaller than b

**3. Logical Operators:**

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either **true** or **false**.

For example, the **logical AND** represented as **‘&&’ operator in C or C++** returns true when both the conditions under consideration are satisfied. Otherwise, it returns false. Therefore, a && b returns true when both a and b are true (i.e. non-zero)(See [this](https://www.geeksforgeeks.org/operators-in-c-set-2-relational-and-logical-operators/) article for more reference).

(4 != 5) && (4 < 5); // true

**4. Bitwise Operators:**

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit-level for faster processing. For example, the **bitwise AND** represented as **& operator in C or C++** takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. (See [this](https://www.geeksforgeeks.org/interesting-facts-bitwise-operators-c/) article for more reference).

int a = 5, b = 9; // a = 5(00000101), b = 9(00001001)

cout << (a ^ b); // 00001100

cout <<(~a); // 11111010

**5. Assignment Operators:**

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

Different types of assignment operators are shown below:   
**a. “=”:** This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.   
For example:

a = 10;

b = 20;

ch = 'y';

**b. “+=”**: This operator is combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.   
For example:

(a += b) can be written as (a = a + b)

If initially value stored in a is 5. Then (a += 6) = 11.

**c. “-=”**: This operator is a combination of ‘-‘ and ‘=’ operators. This operator first subtracts the value on the right from the current value of the variable on left and then assigns the result to the variable on the left.   
For example:

(a -= b) can be written as (a = a - b)

If initially value stored in a is 8. Then (a -= 6) = 2.

**d. “\*=”**: This operator is a combination of ‘\*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.   
For example:

(a \*= b) can be written as (a = a \* b)

If initially, the value stored in a is 5. Then (a \*= 6) = 30.

**e. “/=”**: This operator is a combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.   
For example:

(a /= b) can be written as (a = a / b)

If initially, the value stored in a is 6. Then (a /= 2) = 3.

**6. Other Operators**:

Apart from the above operators, there are some other operators available in C or C++ used to perform some specific tasks. Some of them are discussed here:

**a.**[**sizeof operator**](https://www.geeksforgeeks.org/sizeof-operator-c/):

* sizeof is much used in the C/C++ programming language.
* It is a compile-time unary operator which can be used to compute the size of its operand.
* The result of sizeof is of the unsigned integral type which is usually denoted by size\_t.
* Basically, the sizeof the operator is used to compute the size of the variable.(See [this](https://www.geeksforgeeks.org/sizeof-operator-c/) article for reference)

**b.**[**Comma Operator**](https://www.geeksforgeeks.org/comna-in-c-and-c/):

* The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
* The comma operator has the lowest precedence of any C operator.
* Comma acts as both operator and separator. (See [this](https://www.geeksforgeeks.org/comna-in-c-and-c/) article for reference)

**c.**[**Conditional Operator**](https://www.geeksforgeeks.org/cc-ternary-operator-some-interesting-observations/):

* The conditional operator is of the form *Expression1? Expression2: Expression3*.
* Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
* We may replace the use of if..else statements with conditional operators. (See [this](https://www.geeksforgeeks.org/cc-ternary-operator-some-interesting-observations/) article for reference)

**d.** [**dot (.)**](https://www.geeksforgeeks.org/dot-operator-in-c-c/)**and**[**arrow (->)**](https://www.geeksforgeeks.org/arrow-operator-in-c-c-with-examples/)**Operators:**

* Member operators are used to reference individual members of classes, structures, and unions.
* The dot operator is applied to the actual object. (See [this](https://www.geeksforgeeks.org/dot-operator-in-c-c/) article for reference)
* The arrow operator is used with a pointer to an object. (See [this](https://www.geeksforgeeks.org/arrow-operator-in-c-c-with-examples/) article for reference)

**e.**[**Cast Operator:**](https://www.geeksforgeeks.org/typecasting-in-c/)

* Casting operators convert one data type to another. For example, int(2.2000) would return 2.
* A cast is a special operator that forces one data type to be converted into another.
* The most general cast supported by most of the C++ compilers is as follows −  **[ (type) expression ]**. (See [this](https://www.geeksforgeeks.org/typecasting-in-c/) article for reference)

**f.**[**&,\* Operator:**](https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/)

* Pointer operator & returns the address of a variable. For example &a; will give the actual address of the variable.
* Pointer operator \* is a pointer to a variable. For example \*var; will pointer to a variable var. (See [this](https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/) article for reference

|  |
| --- |
| // Operators in C++  #include<iostream>  **using** **namespace** std;    **int** main(){  **int** a=10, b=5;      // Arithmetic operators      cout<<"Following are the Arithmetic operators in C++"<<endl;      cout<<"The value of a + b is "<<a+b<<endl;      cout<<"The value of a - b is "<<a-b<<endl;      cout<<"The value of a \* b is "<<a\*b<<endl;      cout<<"The value of a / b is "<<a/b<<endl;      cout<<"The value of a % b is "<<a%b<<endl;      cout<<"The value of a++ is "<<a++<<endl; // First print (a) and then increment it by 1      cout<<"The value of a-- is "<<a--<<endl; // First print (a+1) and then decrease it by 1      cout<<"The value of ++a is "<<++a<<endl; // Increment (a) by (a+1) and then print      cout<<"The value of --a is "<<--a<<endl; // Decrement (a+1) by (a) and then print      cout<<endl;        // Assignment Operators --> used to assign values to variables      // int a =3, b=9;      // char d='d';        // Comparison operators      // Output of all these comparison operators will be (1) if it is true and (0) if it is false      cout<<"Following are the comparison operators in C++"<<endl;      cout<<"The value of a == b is "<<(a==b)<<endl;      cout<<"The value of a != b is "<<(a!=b)<<endl;      cout<<"The value of a >= b is "<<(a>=b)<<endl;      cout<<"The value of a <= b is "<<(a<=b)<<endl;      cout<<"The value of a > b is "<<(a>b)<<endl;      cout<<"The value of a < b is "<<(a<b)<<endl;      cout<<endl;      // Logical operators      cout<<"Following are the logical operators in C++"<<endl;      cout<<"The value of this logical and operator ((a==b) && (a<b)) is:"<<((a==b) && (a<b))<<endl;      cout<<"The value of this logical or operator ((a==b) || (a<b)) is:"<<((a==b) || (a<b))<<endl;      cout<<"The value of this logical not operator (!(a==b)) is:"<<(!(a==b))<<endl;      **return** 0;  }   // This code is contributed by Suruchi Kumari |

**Output**

Following are the Arithmetic operators in C++

The value of a + b is 15

The value of a - b is 5

The value of a \* b is 50

The value of a / b is 2

The value of a % b is 0

The value of a++ is 10

The value of a-- is 11

The value of ++a is 11

The value of --a is 10

Following are the comparison operators in C++

The value of a == b is 0

The value of a != b is 1

The value of a >= b is 1

The value of a <= b is 0

The value of a > b is 1

The value of a < b is 0

Following are the logical operators in C++

The value of this logical and operator ((a==b) && (a<b)) is:0

The value of this logical or operator ((a==b) || (a<b)) is:0

The value of this logical not operator (!(a==b)) is:1

### **Operator Precedence Chart**

The below table describes the precedence order and associativity of operators in C / C++. The precedence of the operator decreases from top to bottom.

| Precedence | Operator | Description | Associativity |
| --- | --- | --- | --- |
| 1 | () | Parentheses (function call) | left-to-right |
| [] | Brackets (array subscript) | left-to-right |
| . | Member selection via object name | left-to-right |
| -> | Member selection via a pointer | left-to-right |
| a++/a– | Postfix increment/decrement (a is a variable) | left-to-right |
| 2 | ++a/–a | Prefix increment/decrement (a is a variable) | right-to-left |
| +/- | Unary plus/minus | right-to-left |
| !~ | Logical negation/bitwise complement | right-to-left |
| (type) | Cast (convert value to temporary value of type) | right-to-left |
| \* | Dereference | right-to-left |
| & | Address (of operand) | right-to-left |
| sizeof | Determine size in bytes on this implementation | right-to-left |
| 3 | \*,/,% | Multiplication/division/modulus | left-to-right |
| 4 | +/- | Addition/subtraction | left-to-right |
| 5 | << , >> | Bitwise shift left, Bitwise shift right | left-to-right |
| 6 | < , <= | Relational less than/less than or equal to | left-to-right |
| > , >= | Relational greater than/greater than or equal to | left-to-right |
| 7 | == , != | Relational is equal to/is not equal to | left-to-right |
| 8 | & | Bitwise AND | left-to-right |
| 9 | ^ | Bitwise exclusive OR | left-to-right |
| 10 | | | Bitwise inclusive OR | left-to-right |
| 11 | && | Logical AND | left-to-right |
| 12 | || | Logical OR | left-to-right |
| 13 | ?: | Ternary conditional | right-to-left |
| 14 | = | Assignment | right-to-left |
| += , -= | Addition/subtraction assignment | right-to-left |
| \*= , /= | Multiplication/division assignment | right-to-left |
| %= , &= | Modulus/bitwise AND assignment | right-to-left |
| ^= , |= | Bitwise exclusive/inclusive OR assignment | right-to-left |
| <>= | Bitwise shift left/right assignment | right-to-left |
| 15 | , | expression separator | left-to-right |

Arithmetic operator :

#include <iostream>

int main(){

   //Addition

   int number1{2};

   int number2{7};

   int result = number1 + number2;

   std::cout << "result : " << result << std::endl;

    //Subtraction

    result = number2 - number1 ;

   std::cout << "result : " << result << std::endl;

   result = number1 - number2;

   std::cout << "result : " << result << std::endl;

   //Multiplication

   result = number1 \* number2;

   std::cout << "result : " << result << std::endl;

   //Division

   result = number2 / number1;

   std::cout << "result : " << result << std::endl;

   //Modulus

   result = number2 % number1; // 7 % 2

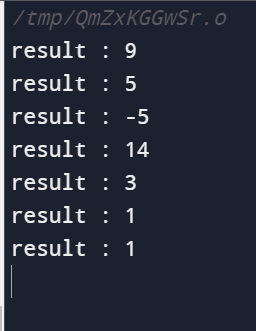
   std::cout << "result : " << result << std::endl; // 1

   result = 31 % 10;

   std::cout << "result : " << result << std::endl; // 1

    return 0;

}



Operator precedence:

#include <iostream>

int main(){

    int a {6};

    int b {3};

    int c {8};

    int d {9};

    int e {3};

    int f {2};

    int g {5};

    int result = a + b \* c -d/e -f + g; //  6 +  24  -   3 - 2 + 5

    std::cout << "result : " << result << std::endl;

    result = a/b\*c +d - e + f;  //   16 + 9 - 3 + 2

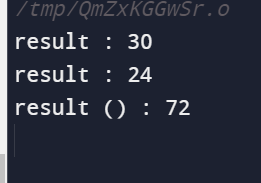
    std::cout << "result : " << result << std::endl;

    result = (a + b) \* c -d/e -f + g; // 72-3-2+5

    std::cout << "result () : " << result << std::endl;

    return 0;

}



Prefix and post fix:

#include <iostream>

int main(){

    int value { 5 };

    //Increment by one

    value = value + 1; //6

    std::cout << "The value is : " << value << std::endl; // 6

    value = 5; // Reset value to 5

    //Decrement by one

    value = value - 1; // 4

    std::cout << "The value is : " << value << std::endl; //4

    //===================================================================

    std::cout << "======Postfix increment and decrement======"<< std::endl;

    //Reset value to 5

    value = 5;

    std::cout << "The value is (incrementing) : " << value++ << std::endl; // 5

    std::cout << "The value is : " << value << std::endl; // 6

    std::cout << std::endl;

    //Decrement with postfix

    //Reset value to 5

    value = 5;

    std::cout << "The value is (decrementing) : " << value-- << std::endl; //5

    std::cout << "The value is : " << value << std::endl; // 4

    //===================================================================

    std::cout << "======Prefix increment and decrement======"<< std::endl;

    //Reset value to 5

    value = 5;

    ++value;

    std::cout << "The value is (prefix++) : " << value << std::endl; // 6

    //Reset value to 5

    value = 5;

    std::cout << "The value is (prefix++ in place) : " << ++value << std::endl; // 6

    std::cout << std::endl;

    //Prefix : Decrementing

    //Reset value to 5;

    value = 5;

    --value;

    std::cout << "The value is (prefix--) : " << value << std::endl; // 4

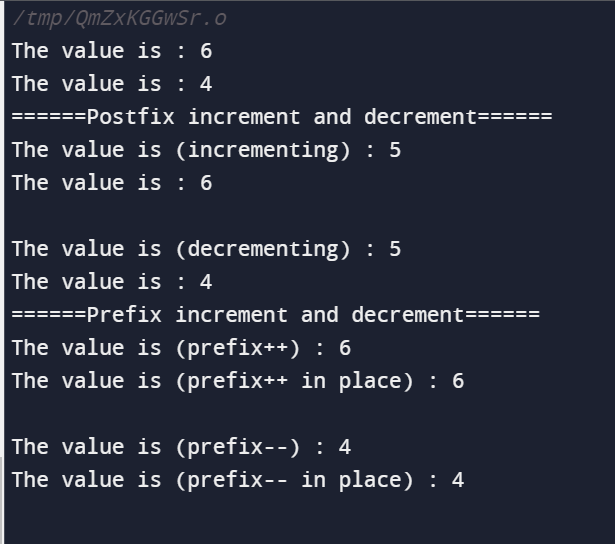
    //Reset value to 5;

    value = 5;

    std::cout << "The value is (prefix-- in place) : " << --value << std::endl;//4

    return 0;

}



Compound assignment:

#include <iostream>

int main(){

    int value {45};

    std::cout << "The value is : " << value << std::endl;

    std::cout << std::endl;

    value = value + 5;

    //value +=5; // equivalent to value = value + 5

    std::cout << "The value is (after +=5) : " << value << std::endl; // 50

    std::cout << std::endl;

    value -=5; // equivalent to value = value - 5

    std::cout << "The value is (after -=5) : " << value << std::endl; // 45

    std::cout << std::endl;

    value \*=2;

    std::cout << "The value is (after \*=2) : " << value << std::endl; // 90

    std::cout << std::endl;

    value /= 3;

    std::cout << "The value is (after /=3) : " << value << std::endl; // 30

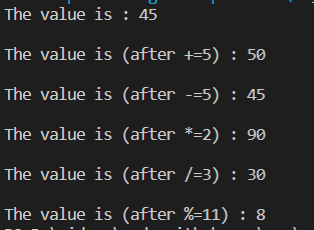
    std::cout << std::endl;

    value %= 11;

    std::cout << "The value is (after %=11) : " << value << std::endl;// 8

    return 0;

}



Relational operator:

#include <iostream>

int main(){

    int number1 {20};

    int number2 {20};

    std::cout << "number1 : " << number1 << std::endl;

    std::cout << "number2 : " << number2 << std::endl;

    std::cout << std::endl;

    std::cout << "Comparing variables" << std::endl;

    std::cout << std::boolalpha ; // Make bool show up as true/false instead of 1/0

    //Stress the need for parentheses here

    std::cout << "number1 < number2 : " << (number1 < number2) << std::endl;

    std::cout << "number1 <= number2 : " << (number1 <= number2) << std::endl;

    std::cout << "number1 > number2 : " << (number1 > number2) << std::endl;

    std::cout << "number1 >= number2 : " << (number1 >= number2) << std::endl;

    std::cout << "number1 == number2 : " << (number1 == number2) << std::endl;

    std::cout << "number1 != number2 : " << (number1 != number2) << std::endl;

    std::cout << std::endl;

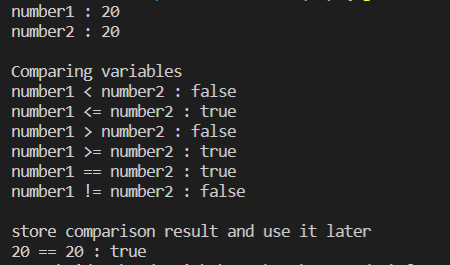
    std::cout << "store comparison result and use it later" << std::endl;

    bool result = (number1 == number2);

    std::cout <<  number1 << " == " << number2 << " : " << result <<  std::endl;

    return 0;

}



#include <iostream>

int main(){

    bool a {true};

    bool b {false};

    bool c {true};

    std::cout << std::boolalpha; // true / false

    std::cout << "a : " << a << std::endl;

    std::cout << "b : " << b << std::endl;

    std::cout << "c : " << c << std::endl;

    //AND : Evaluates to true when all operands are true.

    //      A single false operand will drag

    //      the entire expression to evaluating false.

    std::cout << std::endl;

    std::cout << "Basic AND operations" << std::endl;

    std::cout << " a && b : " << (a && b) << std::endl; // false

    std::cout << " a && c : " << (a && c ) << std::endl; // true

    std::cout << " a && b && c :" << (a && b && c) << std::endl; // false

    //OR : Evaluates to true when at least one operand true.

    //      A single true operand will push

    //      the entire expression to evaluating true.

    std::cout << std::endl;

    std::cout << "Basic OR operations" << std::endl;

    std::cout << " a || b : " << (a || b) << std::endl;

    std::cout << " a || c : " << (a || c ) << std::endl;

    std::cout << " a ||b || c :" << (a || b || c) << std::endl;

    //NOT : Negates whateve operand you put it with

    std::cout << std::endl;

    std::cout << "Basic NOT operations" << std::endl;

    std::cout << "!a : " << !a << std::endl;

    std::cout << "!b : " << !b << std::endl;

    std::cout << "!c : " << !c << std::endl;

    //Combinations of all these operators

    std::cout << std::endl;

    std::cout << "Combining logical operators" << std::endl;

    // !(a &&b) || c

    std::cout << "!(a &&b) || c : " << (!(a &&b) || c) << std::endl; // true

    //Combining logical operators with relational operators

  //relational operator:

  int d{45};

    int e{20};

    int f{11};

    std::cout << std::endl;

    std::cout << "Relational and logic operations on integers" << std::endl;

    std::cout << "d : " << d << std::endl;

    std::cout << "e : " << e << std::endl;

    std::cout << "f : " << f << std::endl;

    std::cout << std::endl;

    std::cout << "(d > e) && (d > f) : " << ((d > e) && (d > f)) << std::endl; // true

    std::cout << "(d==e) || (e <= f ) : " << ((d==e) || (e <= f ) ) << std::endl;

    std::cout << "(d < e) || (d > f) : " << ((d < e) || (d > f)) << std::endl;

    std::cout << "(f > e) || (d < f) : " << ((f > e) || (d < f)) << std::endl;

    std::cout << "(d > f) && (f <= d) : " << ((d > f) && (f <= d)) << std::endl;

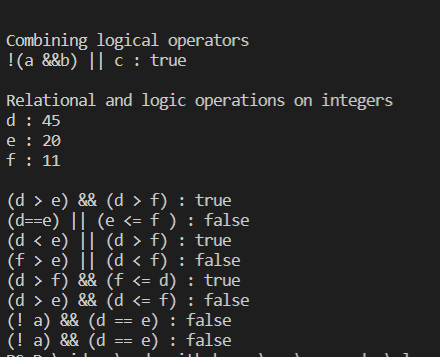
    std::cout << "(d > e) && (d <= f) : " << ((d > e) && (d <= f)) << std::endl;

    std::cout << "(! a) && (d == e) : " << ((! a) && (d == e)) << std::endl;

    std::cout << "(! a) && (d == e) : " << ((! a) && (d == e)) << std::endl;

    return 0;

}



Output formatting:

Std::endl;

    //std::endl : places a new line character on the output stream.

    //       This is identical to placing '\n' on the output stream.

    std::cout << "Hello";

    std::cout << "World";

    std::cout << std::endl;

    std::cout << "-------------" << std::endl;

    std::cout << "Hello" << std::endl;

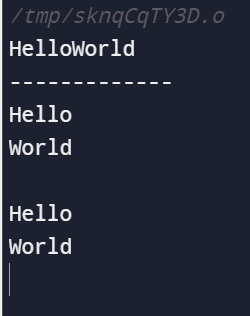
    std::cout << "World" << std::endl;

    std::cout << std::endl;

    std::cout << "Hello\n";

    std::cout << "World\n";

    //===================================================================



Std::flush;

 //===================================================================

    std::cout << std::endl;

    //std::flush : flushes the output buffer to its final destination.

    std::cout << "This is a nice message...." << std::endl << std::flush;

    //After this std::flush, we're sure that at this line, the message has been sent

    //to the stream. This may be important in some applications.

    //===================================================================

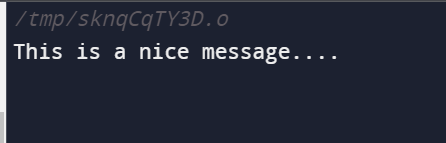


Table formatting :

 //===================================================================

    std::cout << std::endl;

    //std::setw() : Adjusts the field with for the item about to be printed.

    //The setw() manipulator only affects the next value to be printed.

    std::cout << "Unformatted table : " << std::endl;

    std::cout << "Daniel" << " " << "Gray" << " 25" << std::endl;

    std::cout << "Stanley" <<" "  << "Woods" << " 33" << std::endl;

    std::cout << "Jordan" << " "  << "Parker" << " 45" << std::endl;

    std::cout << "Joe" << " " << "Ball" << " 21" << std::endl;

    std::cout << "Josh" << " " << "Carr" << " 27" << std::endl;

    std::cout << "Izaiah" << " " << "Robinson" << " 29" << std::endl;

    std::cout << std::endl;

    std::cout << "Formatted table : " << std::endl;

    std::cout << std::setw(10) <<  "Lastname"  << std::setw(10) << "Firstname" << std::setw(5) << "Age" << std::endl;

    std::cout << std::setw(10) << "Daniel"  << std::setw(10) << "Gray" << std::setw(5) << "25" << std::endl;

    std::cout << std::setw(10) << "Stanley" << std::setw(10)  << "Woods" << std::setw(5) <<  "33" << std::endl;

    std::cout << std::setw(10) <<  "Jordan" << std::setw(10)  << "Parker" << std::setw(5) << "45" << std::endl;

    std::cout << std::setw(10) <<  "Joe" << std::setw(10) << "Ball" << std::setw(5) << "21" << std::endl;

    std::cout << std::setw(10) << "Josh" << std::setw(10) << "Carr" << std::setw(5) <<"27" << std::endl;

    std::cout << std::setw(10) << "Izaiah" << std::setw(10) << "Robinson" << std::setw(5) << "29" << std::endl;

    std::cout << std::endl;

    std::cout << "Formatted table with variables: " << std::endl;

    int col\_width{14};

    std::cout << std::setw(col\_width) <<  "Lastname"  << std::setw(col\_width) << "Firstname" << std::setw(col\_width/2) << "Age" << std::endl;

    std::cout << std::setw(col\_width) << "Daniel"  << std::setw(col\_width) << "Gray" << std::setw(col\_width/2) << "25" << std::endl;

    std::cout << std::setw(col\_width) << "Stanley" << std::setw(col\_width)  << "Woods" << std::setw(col\_width/2) <<  "33" << std::endl;

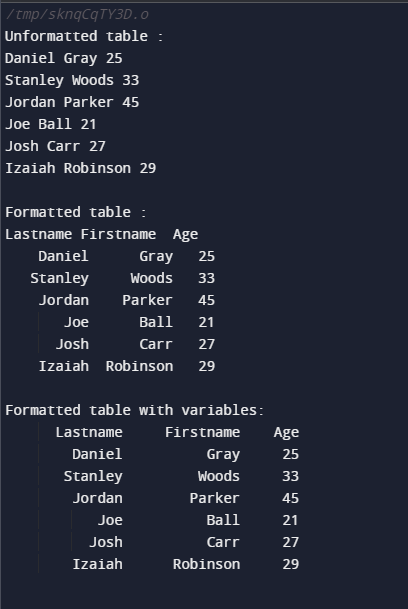
    std::cout << std::setw(col\_width) <<  "Jordan" << std::setw(col\_width)  << "Parker" << std::setw(col\_width/2) << "45" << std::endl;

    std::cout << std::setw(col\_width) <<  "Joe" << std::setw(col\_width) << "Ball" << std::setw(col\_width/2) << "21" << std::endl;

    std::cout << std::setw(col\_width) << "Josh" << std::setw(col\_width) << "Carr" << std::setw(col\_width/2) <<"27" << std::endl;

    std::cout << std::setw(col\_width) << "Izaiah" << std::setw(col\_width) << "Robinson" << std::setw(col\_width/2) << "29" << std::endl;

    //===================================================================



Justified tables:

 //===================================================================

    std::cout << std::endl;

    //Justify : Values can be justified in their fields. There are three manipulators

    //          for adjusting the justification: left, right, and internal.

    //right justified

    std::cout << std::endl;

    std::cout << "Right justified table(default) :  " << std::endl;

    col\_width = 20;

    std::cout << std::right;

    std::cout << std::setw(col\_width) <<  "Lastname"  << std::setw(col\_width) << "Firstname" << std::setw(col\_width/2) << "Age" << std::endl;

    std::cout << std::setw(col\_width) << "Daniel"  << std::setw(col\_width) << "Gray" << std::setw(col\_width/2) << "25" << std::endl;

    std::cout << std::setw(col\_width) << "Stanley" << std::setw(col\_width)  << "Woods" << std::setw(col\_width/2) <<  "33" << std::endl;

    std::cout << std::setw(col\_width) <<  "Jordan" << std::setw(col\_width)  << "Parker" << std::setw(col\_width/2) << "45" << std::endl;

    std::cout << std::setw(col\_width) <<  "Joe" << std::setw(col\_width) << "Ball" << std::setw(col\_width/2) << "21" << std::endl;

    std::cout << std::setw(col\_width) << "Josh" << std::setw(col\_width) << "Carr" << std::setw(col\_width/2) <<"27" << std::endl;

    std::cout << std::setw(col\_width) << "Izaiah" << std::setw(col\_width) << "Robinson" << std::setw(col\_width/2) << "29" << std::endl;

    //Left justified

    std::cout << std::endl;

    std::cout << "Left justified table :  " << std::endl;

    col\_width = 20;

    std::cout << std::left;

    std::cout << std::setw(col\_width) <<  "Lastname"  << std::setw(col\_width) << "Firstname" << std::setw(col\_width/2) << "Age" << std::endl;

    std::cout << std::setw(col\_width) << "Daniel"  << std::setw(col\_width) << "Gray" << std::setw(col\_width/2) << "25" << std::endl;

    std::cout << std::setw(col\_width) << "Stanley" << std::setw(col\_width)  << "Woods" << std::setw(col\_width/2) <<  "33" << std::endl;

    std::cout << std::setw(col\_width) <<  "Jordan" << std::setw(col\_width)  << "Parker" << std::setw(col\_width/2) << "45" << std::endl;

    std::cout << std::setw(col\_width) <<  "Joe" << std::setw(col\_width) << "Ball" << std::setw(col\_width/2) << "21" << std::endl;

    std::cout << std::setw(col\_width) << "Josh" << std::setw(col\_width) << "Carr" << std::setw(col\_width/2) <<"27" << std::endl;

    std::cout << std::setw(col\_width) << "Izaiah" << std::setw(col\_width) << "Robinson" << std::setw(col\_width/2) << "29" << std::endl;

    //Internal justified : sign is left justified , data is right justified

    std::cout << std::endl;

    std::cout << "Internal justified : " << std::endl;

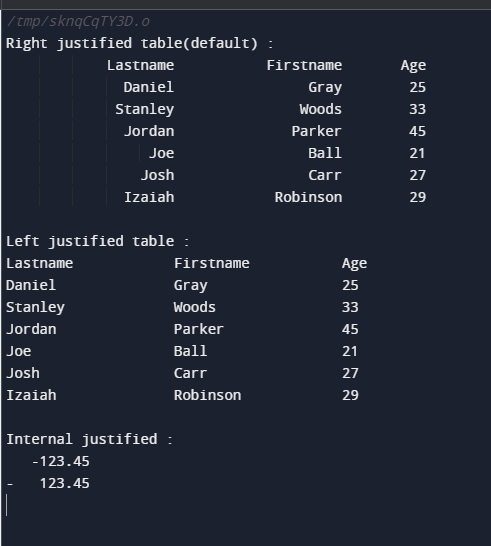
    std::cout << std::right;

    std::cout << std::setw(10) << -123.45 << std::endl;

    std::cout << std::internal;

    std::cout << std::setw(10) << -123.45 << std::endl;

    //===================================================================



std::setfill(“.”) :

 //===================================================================

    std::cout << std::endl;

    //setfill

    std::cout << std::endl;

    std::cout << "Table with fill characters :  " << std::endl;

    col\_width = 20;

    std::cout << std::left;

    std::cout << std::setfill('\*'); // The fill character

    std::cout << std::setw(col\_width) <<  "Lastname"  << std::setw(col\_width) << "Firstname" << std::setw(col\_width/2) << "Age" << std::endl;

    std::cout << std::setw(col\_width) << "Daniel"  << std::setw(col\_width) << "Gray" << std::setw(col\_width/2) << "25" << std::endl;

    std::cout << std::setw(col\_width) << "Stanley" << std::setw(col\_width)  << "Woods" << std::setw(col\_width/2) <<  "33" << std::endl;

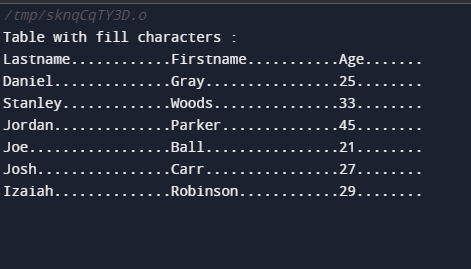
    std::cout << std::setw(col\_width) <<  "Jordan" << std::setw(col\_width)  << "Parker" << std::setw(col\_width/2) << "45" << std::endl;

    std::cout << std::setw(col\_width) <<  "Joe" << std::setw(col\_width) << "Ball" << std::setw(col\_width/2) << "21" << std::endl;

    std::cout << std::setw(col\_width) << "Josh" << std::setw(col\_width) << "Carr" << std::setw(col\_width/2) <<"27" << std::endl;

    std::cout << std::setw(col\_width) << "Izaiah" << std::setw(col\_width) << "Robinson" << std::setw(col\_width/2) << "29" << std::endl;

    //===================================================================



Printing true or false: boolplaha;

 //===================================================================

    std::cout << std::endl;

    //boolalpha and noboolapha : control bool output format : 1/0 or true/false

    bool condition {true};

    bool other\_condition {false};

    std::cout << "condition : " << condition << std::endl;

    std::cout << "other\_condition : " << other\_condition << std::endl;

    std::cout << std::endl;

    std::cout << std::boolalpha;

    std::cout << "condition : " << condition << std::endl;

    std::cout << "other\_condition : " << other\_condition << std::endl;

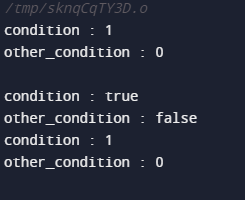
    std::cout << std::endl;

    std::cout << std::noboolalpha;

    std::cout << "condition : " << condition << std::endl;

    std::cout << "other\_condition : " << other\_condition << std::endl;

    //===================================================================



Showpos,noshowpos;

//===================================================================

    std::cout << std::endl;

    //showpos and noshowpos : show or hide the +  sign for positive numbers

    int pos\_num {34};

    int neg\_num {-45};

    std::cout << "pos\_num : " << pos\_num << std::endl;

    std::cout << "neg\_num : " << neg\_num << std::endl;

    std::cout << std::endl;

    std::cout << std::showpos;

    std::cout << "pos\_num : " << pos\_num << std::endl;

    std::cout << "neg\_num : " << neg\_num << std::endl;

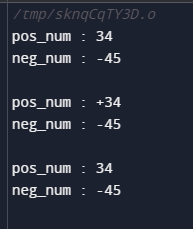
    std::cout << std::endl;

    std::cout << std::noshowpos;

    std::cout << "pos\_num : " << pos\_num << std::endl;

    std::cout << "neg\_num : " << neg\_num << std::endl;

    //===================================================================



Number in different base:

 //===================================================================

    std::cout << std::endl;

    //different number systems : std::dec, std::hex, std::oct

    int pos\_int {717171};

    int neg\_int {-47347};

    double double\_var {498.32};

    std::cout << std::endl;

    std::cout << "default base format : " << std::endl;

    std::cout << "pos\_int : " << pos\_int << std::endl;

    std::cout << "neg\_int : " << neg\_int << std::endl;

    std::cout << "double\_var : " << double\_var << std::endl;

    std::cout << std::endl;

    std::cout << "pos\_int in different bases : " << std::endl;

    std::cout << "pos\_int (dec) : " << std::dec << pos\_int << std::endl;

    std::cout << "pos\_int (hex) : " << std::hex << pos\_int << std::endl;

    std::cout << "pos\_int (oct) : " << std::oct << pos\_int << std::endl;

    std::cout << std::endl;

    std::cout << "neg\_int in different bases : " << std::endl;

    std::cout << "neg\_int (dec) : " << std::dec << neg\_int << std::endl;

    std::cout << "neg\_int (hex) : " << std::hex << neg\_int << std::endl;

    std::cout << "neg\_int (oct) : " << std::oct << neg\_int << std::endl;

    std::cout << std::endl;

    std::cout << "double\_var in different bases : " << std::endl;

    std::cout << "double\_var (dec) : " << std::dec << double\_var << std::endl;

    std::cout << "double\_var (hex) : " << std::hex << double\_var << std::endl;

    std::cout << "double\_var (oct) : " << std::oct << double\_var << std::endl;

    //===================================================================

    std::cout << std::endl;

    //uppercase and nouppercase

    pos\_int = 717171;

    std::cout << "pos\_int (nouppercase : default) : " << std::endl;

    std::cout << "pos\_int (dec) : " << std::dec << pos\_int << std::endl;

    std::cout << "pos\_int (hex) : " << std::hex << pos\_int << std::endl;

    std::cout << "pos\_int (oct) : " << std::oct << pos\_int << std::endl;

    std::cout << std::endl;

    std::cout << "pos\_int (uppercase) : " << std::endl;

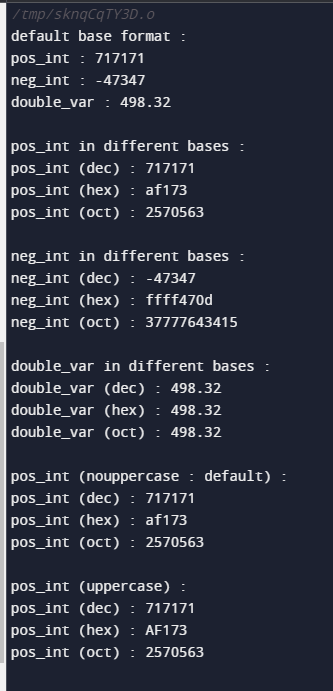
    std::cout << std::uppercase;

    std::cout << "pos\_int (dec) : " << std::dec << pos\_int << std::endl;

    std::cout << "pos\_int (hex) : " << std::hex << pos\_int << std::endl;

    std::cout << "pos\_int (oct) : " << std::oct << pos\_int << std::endl;

    //===================================================================



Scientific notations:

 //===================================================================

    std::cout << std::endl;

    //fixed and scientific : for floating point values

    double a{ 3.1415926535897932384626433832795 };

    double b{ 2006.0 };

    double c{ 1.34e-10 };

    std::cout << std::endl;

    std::cout << "double values (default : use scientific where necessary) : " << std::endl;

    std::cout << "a : " << a << std::endl;

    std::cout << "b : " << b << std::endl;

    std::cout << "c : " << c << std::endl;

    std::cout << std::endl;

    std::cout << "double values (fixed) : " << std::endl;

    std::cout << std::fixed;

    std::cout << "a : " << a << std::endl;

    std::cout << "b : " << b << std::endl;

    std::cout << "c : " << c << std::endl;

    std::cout << std::endl;

    std::cout << "double values (scientific) : " << std::endl;

    std::cout << std::scientific;

    std::cout << "a : " << a << std::endl;

    std::cout << "b : " << b << std::endl;

    std::cout << "c : " << c << std::endl;

    std::cout << std::endl;

    std::cout << "double values (back to defaults) : " << std::endl;

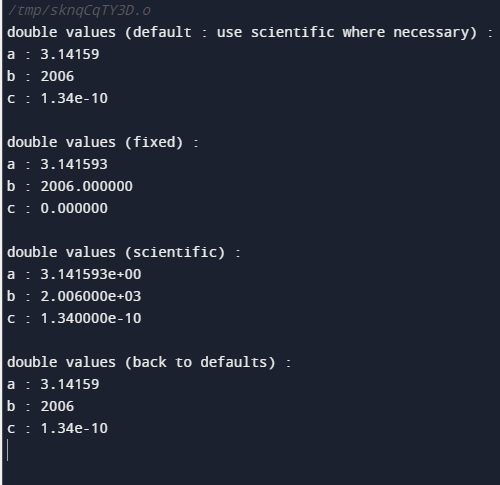
    std::cout.unsetf(std::ios::scientific | std::ios::fixed); // Hack

    std::cout << "a : " << a << std::endl;

    std::cout << "b : " << b << std::endl;

    std::cout << "c : " << c << std::endl;

    //===================================================================



Setprecision:

 //===================================================================

    std::cout << std::endl;

    //setprecision() : the number of digits printed out for a floating point. Default is 6

    a = 3.1415926535897932384626433832795;

    std::cout << std::endl;

    std::cout << "a (default precision(6)) : " << a <<  std::endl;

    std::cout << std::setprecision(10);

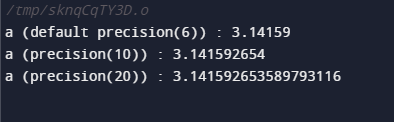
    std::cout << "a (precision(10)) : " << a << std::endl;

    std::cout << std::setprecision(20);

    std::cout << "a (precision(20)) : " << a << std::endl;

    //If the precision is bigger than supported by the type, you'll just print garbage.

    //===================================================================



Showpoint ,noshowpoint

//===================================================================

    std::cout << std::endl;

    //showpoint and noshowpoint : show trailing zeros if necessary

    //Force output of the decimal point

    double d {34.1};

    double e {101.99};

    double f {12.0};

    int    g {45};

    std::cout << std::endl;

    std::cout << "noshowpoint (default) : " << std::endl;

    std::cout << "d : " << d << std::endl;

    std::cout << "e : " << e << std::endl;

    std::cout << "f : " << f << std::endl; // 12

    std::cout << "g : " << g << std::endl;

    std::cout << std::endl;

    std::cout << "showpoint: " << std::endl;

    std::cout << std::showpoint;

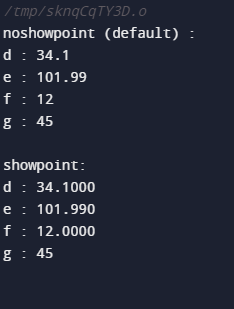
    std::cout << "d : " << d << std::endl;

    std::cout << "e : " << e << std::endl;

    std::cout << "f : " << f << std::endl; // 12.0

    std::cout << "g : " << g << std::endl;

    return 0;



Numeric limits:

#include <iostream>

#include <limits>

int main(){

    std::cout << "The range for short is from " << std::numeric\_limits<short>::min() << " to "

        << std::numeric\_limits<short>::max() << std::endl;

    std::cout << "The range for unsigned short is from " << std::numeric\_limits<unsigned short>::min() << " to "

        << std::numeric\_limits<unsigned short>::max() << std::endl;

    std::cout << "The range for int is from " << std::numeric\_limits<int>::min() << " to "

        << std::numeric\_limits<int>::max() << std::endl;

    std::cout << "The range for unsigned int is from " << std::numeric\_limits<unsigned int>::min() << " to "

        << std::numeric\_limits<unsigned int>::max() << std::endl;

    std::cout << "The range for long is from " << std::numeric\_limits<long>::min() << " to "

        << std::numeric\_limits<long>::max() << std::endl;

    std::cout << "The range for float is from " << std::numeric\_limits<float>::min() << " to "

        << std::numeric\_limits<float>::max() << std::endl;

    std::cout << "The range(with lowest) for float is from " << std::numeric\_limits<float>::lowest() << " to "

        << std::numeric\_limits<float>::max() << std::endl;

    std::cout << "The range(with lowest) for double is from " << std::numeric\_limits<double>::lowest() << " to "

        << std::numeric\_limits<double>::max() << std::endl;

    std::cout << "The range(with lowest) for long double is from " << std::numeric\_limits<long double>::lowest() << " to "

        << std::numeric\_limits<long double>::max() << std::endl;

    //Other facilities

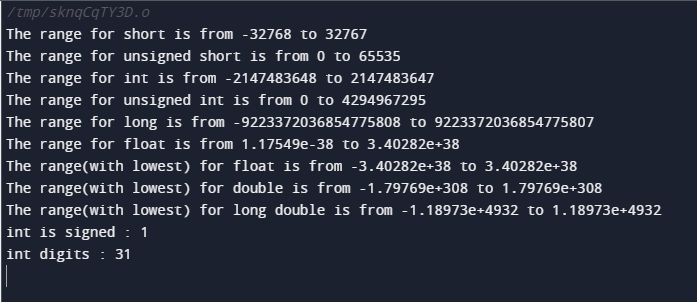
    //More info : https://en.cppreference.com/w/cpp/types/numeric\_limits

    std::cout << "int is signed : " << std::numeric\_limits<int>::is\_signed << std::endl;

    std::cout << "int digits : " << std::numeric\_limits<int>::digits << std::endl;

    return 0;

}



Math functions:

#include <iostream>

#include <cmath>

int main(){

    double weight { 7.7 };

    //floor

    std::cout << "Weight rounded to floor is : " << std::floor(weight) << std::endl;

    //ceil

    std::cout << "Weight rounded to ceil is : " << std::ceil(weight) << std::endl;

    //abs

    double savings {-5000 };

    std::cout << "Abs of weight is : " << std::abs(weight) << std::endl;

    std::cout << "Abs of savings is : " << std::abs(savings) << std::endl;

    //exp : f(x) = e ^ x , where e = 2.71828 . Test the result here against a calculator

    double exponential = std::exp(10);

    std::cout << "The exponential of 10 is : " << exponential << std::endl;

    //pow

    std::cout << "3 ^ 4 is : " << std::pow(3,4) << std::endl;

    std::cout << "9^3 is : " << std::pow(9,3) << std::endl;

    //log : reverse function of pow. if 2^3 = 8 , log 8 in base 2 = 3.  Log is like asking

    // to which exponent should we elevate 2 to get eight ? Log, by default computes the log

    // in base e. There also is another function which uses base 10 called log10

    // Try the reverse operation of  e^4 = 54.59 , it will be log 54.59 in base e = ?

    std::cout << "Log ; to get 54.59, you would elevate e to the power of : "

             << std::log(54.59) << std::endl;

    //log10 , 10 ^ 4 = 10000  , to get 10k , you'd need to elevate 10 to the power of ? , this is log in base 10

    std::cout << "To get 10000, you'd need to elevate 10 to the power of : "

                 << std::log10(10000) << std::endl; // 4

    //sqrt

    std::cout << "The square root of 81 is : " << std::sqrt(81) << std::endl;

    //round. Halfway points are rounded away from 0. 2,5 is rounded to 5 for example

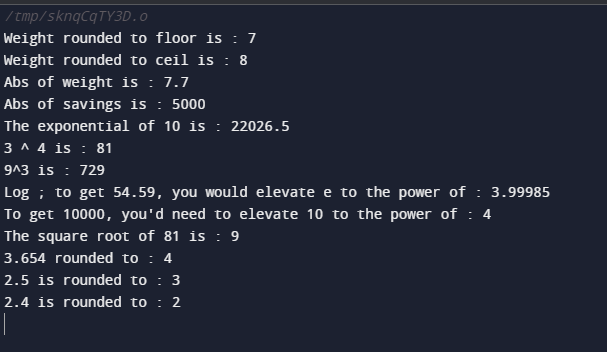
    std::cout << "3.654 rounded to : " << std::round(3.654) << std::endl;

    std::cout << "2.5 is rounded to : " << std::round(2.5) << std::endl;

    std::cout << "2.4 is rounded to : " << std::round(2.4) << std::endl;

    return 0;

}



Miscellaneous :

#include <iostream>

int main(){

    short int var1 {10}; // 2 bytes

    short int var2 {20};

    char var3 {40}; //1

    char var4 {50};

    std::cout << "size of var1 : " << sizeof(var1) << std::endl;

    std::cout << "size of var2 : " << sizeof(var2) << std::endl;

    std::cout << "size of var3 : " << sizeof(var3) << std::endl;

    std::cout << "size of var4 : " << sizeof(var4) << std::endl;

    auto result1 = var1 + var2 ;

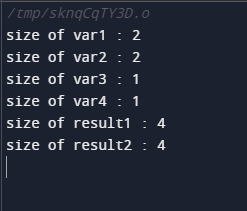
    auto result2 = var3 + var4;

    std::cout << "size of result1 : " << sizeof(result1) << std::endl; // 4

    std::cout << "size of result2 : " << sizeof(result2) << std::endl; // 4

    return 0;

}



Operator at once:

#include <iostream>

int main()

{

    int a = 23;

    int b = 234;

    double c = 3487.3;

    double d = -220;

    // // arithmetic operators in integers

    // std::cout << a + b << std::endl;

    // std::cout << a - b << std::endl;

    // std::cout << a \* b << std::endl;

    // std::cout << a / b << std::endl;

    // // std::cout << a % b << std::endl;

    // // arithmetic operators on double

    // std::cout << c + d << std::endl;

    // std::cout << c - d << std::endl;

    // std::cout << c \* d << std::endl;

    // std::cout << c / d << std::endl;

    // // std::cout << c % d << std::endl;//modulus operator does not work in the floating point numbers

    // \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    // precendence

    int sum = a + b \* c - d / a;

    std::cout << static\_cast<double>(sum) << std::endl;

    // \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    // assignment operators

    // std::cout << (a += 10) << std::endl; // 30

    // // working on the updated value of a thet is 30

    // std::cout << (a -= 10) << std::endl; // 30-10=20

    // std::cout << (a \*= 10) << std::endl; // 20\*10=200

    // std::cout << (a /= 10) << std::endl; // 200/10=20

    // std::cout << (a %= 10) << std::endl; // 20%10=0

    //\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    // relational operators

    // result is either true or false

    // std::cout << std::boolalpha;

    // std::cout << (a > b) << std::endl;  // false

    // std::cout << (a >= b) << std::endl; // false

    // std::cout << (a <= b) << std::endl; // true

    // std::cout << (a < b) << std::endl;  // true

    // std::cout << (a == b) << std::endl; // false

    // std::cout << (a != b) << std::endl; // true

    // \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    // \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    // logical operators

    // asssigning values specifically here only for the refernce

    // auto a = 0, b = 0;

    // true=anything>0 and anything<0

    // false=0

    // std::cout << (a && b) << std::endl;

    // std::cout << (a || b) << std::endl;

    // std::cout << (!b) << std::endl;

    // std::cout << (!a) << std::endl;

    // std::cout << ((!a) && b) << std::endl;

    // std::cout << ((!a) || b) << std::endl;

    // \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

    // mixed ;

    // std::cout << ((a > b) && (a < b)) << std::endl;

    // std::cout << ((a > b) || (a >= b)) << std::endl;

    // std::cout << ((a <= b) || (a > b)) << std::endl;

    // std::cout << ((!(a <= b)) || (a > b)) << std::endl;

    // std::cout << ((a > b) || (!(a < b))) << std::endl;

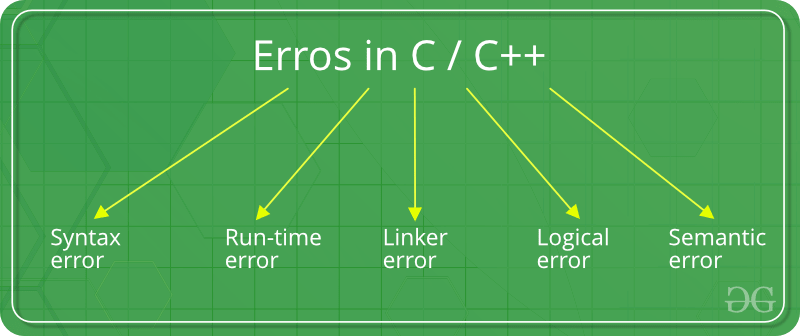
    // std::cout << (!(a >= b)) << std::endl;

}

# Errors in C/C++

Error is an illegal operation performed by the user which results in 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.   
The most common errors can be broadly classified as follows.

**Type of errors:**



1. **Syntax errors:** Errors that occur when you **violate the rules** of writing C/C++ syntax are known as syntax errors. This compiler error indicates something that must be fixed before the code can be compiled. All these errors are detected by compiler and thus are known as compile-time errors.   
   Most frequent syntax errors are:
   * Missing Parenthesis (**}**)
   * Printing the value of variable without declaring it
   * Missing semicolon like this:

|  |
| --- |
| // C++ program to illustrate  // syntax error    #include <iostream>  **using** **namespace** std;    **void** main()  {  **int** x = 10;  **int** y = 15;        cout << " "<< (x, y) // semicolon missed  }    // This code is contributed by shivanisinghss2110 |

**Error:**

error: expected ';' before '}' token

* Syntax of a basic construct is written wrong. For example : while loop

|  |
| --- |
| // C++ program to illustrate  // syntax error  #include <iostream>  **using** **namespace** std;    **int** main(**void**)  {      // while() cannot contain "." as an argument.  **while**(.)      {          cout <<"hello";      }  **return** 0;  }    // This code is contributed by shivanisinghss2110 |

**Error:**

error: expected expression before '.' token

while(.)

* In the given example, the syntax of while loop is incorrect. This causes a syntax error.

1. **Run-time Errors :** Errors which occur during program execution(run-time) after successful compilation are called run-time errors. One of the most common run-time error is division by zero also known as Division error. These types of error are hard to find as the compiler doesn’t point to the line at which the error occurs.   
   For more understanding run the example given below.

|  |
| --- |
| // C++ program to illustrate  // run-time error  #include <iostream>  #include <bits/stdc++.h>  **using** **namespace** std;    **void** main()  {  **int** n = 9, **div** = 0;        // wrong logic      // number is divided by 0,      // so this program abnormally terminates  **div** = n/0;       cout << "result = "<< **div**;  }    // This code is contributed by shivanisinghss2110 |

**Error:**

warning: division by zero [-Wdiv-by-zero]

div = n/0;

1. In the given example, there is Division by zero error. This is an example of run-time error i.e errors occurring while running the program.

* **Linker Errors:**These error occurs when after compilation we link the different object files with main’s object using *Ctrl+F9* key(RUN). These are errors generated when the executable of the program cannot be generated. This may be due to wrong function prototyping, incorrect header files. One of the most common linker error is writing **Main()** instead of **main()**.

|  |
| --- |
| // C++ program to illustrate  // linker error  #include <bits/stdc++.h>  **using** **namespace** std;    **void** Main() // Here Main() should be main()  {  **int** a = 10;      cout << " "<< a;  }    // This code is contributed by shivanisinghss2110 |

**Error:**

(.text+0x20): undefined reference to `main'

**Logical Errors :**On compilation and execution of a program, desired output is not obtained when certain input values are given. These types of errors which provide incorrect output but appears to be error free are called logical errors. These are one of the most common errors done by beginners of programming.   
These errors solely depend on the logical thinking of the programmer and are easy to detect if we follow the line of execution and determine why the program takes that path of execution. 

|  |
| --- |
| // C++ program to illustrate  // logical error  **int** main()  {  **int** i = 0;        // logical error : a semicolon after loop  **for**(i = 0; i < 3; i++);      {         cout << "loop ";  **continue**;      }  **return** 0;  }      // This code is contributed by shivanisinghss2110. |

1. No output

**Semantic errors :** This error occurs when the statements written in the program are not meaningful to the compiler.

|  |
| --- |
| // C++ program to illustrate  // semantic error  **int** main()  {  **int** a, b, c;      a + b = c; //semantic error  }    // This code is contributed by sarajadhav12052009 |

**Error:**

error: value required as left operand of assignment

a + b = c; //semantic error

bitwise operator:

#include <iostream>

#include <bitset>

int main(){

    unsigned short int data {15};

    std::cout << "data (dec) : " <<std::showbase <<  std::dec << data << std::endl;

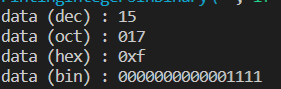
    std::cout << "data (oct) : " <<std::showbase <<  std::oct << data << std::endl;

    std::cout << "data (hex) : " <<std::showbase <<  std::hex << data << std::endl;

    std::cout << "data (bin) : " << std::bitset<16>(data) << std::endl;

    return 0;

}



Shifting of operator:

#include <iostream>

#include <bitset>

int main(){

    unsigned short int value {0xff0u};

    std::cout << "Size of short int " << sizeof(short int) <<  std::endl;//  16 bits

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift left by one bit

    value = static\_cast<unsigned short int>(value << 1);

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift left by one bit

    value = static\_cast<unsigned short int>(value << 1);

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift left by one bit

    value = static\_cast<unsigned short int>(value << 1);

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift left by one bit

    value = static\_cast<unsigned short int>(value << 1);

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift left by one bit

    value = static\_cast<unsigned short int>(value << 1);

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift right by one bit

    value = static\_cast<unsigned short int>(value >> 1);

    std::cout << "value : " << std::bitset<16>(value)

    << ", dec : " << value << std::endl;

    //Shift by multiple bits in one go

    //Shift right by four bits

    value = static\_cast<unsigned short int>(value >> 4);

    std::cout << "value : " << std::bitset<16>(value)

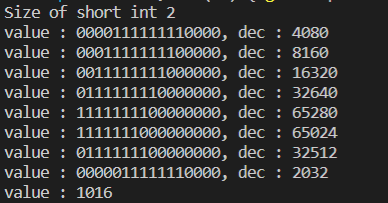
    << ", dec : " << value << std::endl;

    //

    std::cout << "value : " << (value >> 1) << std::endl;

    return 0;

}



Bitwise logical operator:

#include <iostream>

#include <iomanip>

#include <bitset>

int main(){

    int COLUMN\_WIDTH {20};

    unsigned char value1 {0x3}; // 0000 0011

    unsigned char value2 {0x5}; // 0000 0101

    std::cout << std::setw(COLUMN\_WIDTH) << "value1 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(value1) << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "value2 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(value2) << std::endl;

    //AND

    std::cout << std::endl;

    std::cout << "Bitwise AND :  " << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "value1 & value2 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(value1 & value2) << std::endl;

    std::cout << std::endl;

    //OR

    std::cout << std::endl;

    std::cout << "Bitwise OR :  " << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "value1 | value2 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(value1 | value2) << std::endl;

    std::cout << std::endl;

    //NOT

    std::cout << std::endl;

    std::cout << "Bitwise NOT " << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "~value1 : "

        << std::setw(COLUMN\_WIDTH) <<  std::bitset<8>(~value1) << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "~value2 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(~value2) << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "~01011001 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(~0b01011001) << std::endl;//Using bin literal

    std::cout << std::setw(COLUMN\_WIDTH) << "~01011001 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(~0x59) << std::endl;//Using hex literal

    std::cout << std::endl;

    //XOR

    std::cout << std::endl;

    std::cout << "Bitwise XOR :  " << std::endl;

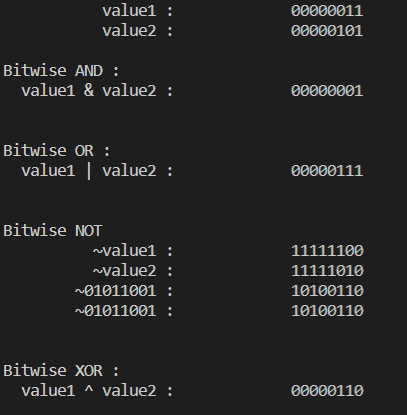
    std::cout << std::setw(COLUMN\_WIDTH) << "value1 ^ value2 : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(value1 ^ value2) << std::endl;

    std::cout << std::endl;

    return 0;

}



Compound operators:

#include <iostream>

#include <iomanip>

#include <bitset>

int main(){

    const int COLUMN\_WIDTH {20};

    std::cout << std::endl;

    std::cout << "Compound bitwise assignment operators" << std::endl;

    unsigned char sandbox\_var{0b00110100}; // 8 bits : positive numbers only

    //Print out initial value

    std::cout << std::endl;

    std::cout << "Initial value :  " << std::endl;

    std::cout << std::setw(COLUMN\_WIDTH) << "sandbox\_var : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(sandbox\_var) << std::endl;

    std::cout << std::endl;

    //Compound left shift

    std::cout << std::endl;

    std::cout << "Shift left 2 bit positions in place :  " << std::endl;

    sandbox\_var <<= 2;

    std::cout << std::setw(COLUMN\_WIDTH) << "sandbox\_var : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(sandbox\_var) << std::endl;

    std::cout << std::endl;

    //Compound right shift

    std::cout << std::endl;

    std::cout << "Shift right 4 bit positions in place :  " << std::endl;

    sandbox\_var >>= 4;

    std::cout << std::setw(COLUMN\_WIDTH) << "sandbox\_var : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(sandbox\_var) << std::endl;

    std::cout << std::endl;

    //Compound OR with 0000 0010 to have all lower 4 bits turned on

    std::cout << std::endl;

    std::cout << "Compound OR with 0000 0010 :  " << std::endl;

    sandbox\_var |= 0b00001111;

    std::cout << std::setw(COLUMN\_WIDTH) << "sandbox\_var : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(sandbox\_var) << std::endl;

    std::cout << std::endl;

    //Compound AND with 0000 1100 to turn off the 2 lowest bits

    std::cout << std::endl;

    std::cout << "Compound AND with 0000 1100 :  " << std::endl;

    sandbox\_var &= 0b000000000;

    std::cout << std::setw(COLUMN\_WIDTH) << "sandbox\_var : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(sandbox\_var) << std::endl;

    std::cout << std::endl;

    //XOR with 00000011 to turn on the 4 lowest bits again

    std::cout << std::endl;

    std::cout << "Compound XOR with 0000 0011 :  " << std::endl;

    sandbox\_var ^= 0b00000011;

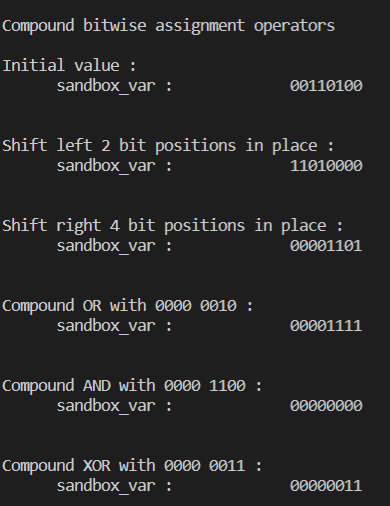
    std::cout << std::setw(COLUMN\_WIDTH) << "sandbox\_var : "

        << std::setw(COLUMN\_WIDTH) << std::bitset<8>(sandbox\_var) << std::endl;

    std::cout << std::endl;

    return 0;

}



Masking:

#include <iostream>

#include <iomanip>

#include <bitset>

int main()

{

    const int COLUMN\_WIDTH{20};

    // Highlight position for bit of interest with a 1

    // Mask other positions with 0

    const unsigned char mask\_bit\_0{0b00000001}; // Bit0

    const unsigned char mask\_bit\_1{0b00000010}; // Bit1

    const unsigned char mask\_bit\_2{0b00000100}; // Bit2

    const unsigned char mask\_bit\_3{0b00001000}; // Bit3

    const unsigned char mask\_bit\_4{0b00010000}; // Bit4

    const unsigned char mask\_bit\_5{0b00100000}; // Bit5

    const unsigned char mask\_bit\_6{0b01000000}; // Bit6

    const unsigned char mask\_bit\_7{0b10000000}; // Bit7

    // Sandbox variable

    unsigned char var{0b00000000}; // Starts off all bits off

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Set a few bits : make them 1's regardless of what's in there

    // SETTING BITS

    // Setting : |= with mask of the bit

    // Set bit 1

    std::cout << "Setting bit in position 1" << std::endl;

    var |= mask\_bit\_1;

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Set bit 5

    std::cout << "Setting bit in position 5" << std::endl;

    var |= mask\_bit\_5;

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // RESETTING BITS : set to 0

    // Resetting : &= (~mask)

    // Reset bit 1

    std::cout << "Resetting bit in position 1" << std::endl;

    var &= (~mask\_bit\_1);

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Reset bit 5

    std::cout << "Resetting bit in position 1" << std::endl;

    var &= (~mask\_bit\_5);

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Set all bits

    std::cout << "Setting all bits" << std::endl;

    var |= (mask\_bit\_0 | mask\_bit\_1 | mask\_bit\_2 | mask\_bit\_3 |

            mask\_bit\_4 | mask\_bit\_5 | mask\_bit\_6 | mask\_bit\_7);

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Reset bits at pos 0,2,4,6

    std::cout << "Reset bits at pos 0,2,4,6" << std::endl;

    var &= ~(mask\_bit\_0 | mask\_bit\_2 | mask\_bit\_4 | mask\_bit\_6);

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Check state of a bit

    std::cout << std::endl;

    std::cout << "Checking the state of each bit position (on/off)" << std::endl;

    std::cout << "bit0 is " << ((var & mask\_bit\_0) >> 0) << std::endl;

    std::cout << "bit1 is " << ((var & mask\_bit\_1) >> 1) << std::endl;

    std::cout << "bit2 is " << ((var & mask\_bit\_2) >> 2) << std::endl;

    std::cout << "bit3 is " << ((var & mask\_bit\_3) >> 3) << std::endl;

    std::cout << "bit4 is " << ((var & mask\_bit\_4) >> 4) << std::endl;

    std::cout << "bit5 is " << ((var & mask\_bit\_5) >> 5) << std::endl;

    std::cout << std::boolalpha;

    std::cout << "bit6 is " << ((var & mask\_bit\_6) >> 6) << std::endl;

    std::cout << "bit6 is " << static\_cast<bool>(var & mask\_bit\_6) << std::endl;

    std::cout << "bit7 is " << ((var & mask\_bit\_7) >> 7) << std::endl;

    std::cout << "bit7 is " << static\_cast<bool>(var & mask\_bit\_7) << std::endl;

    // Toggle bits

    // Toggle : var ^ mask

    // Toggle bit 0

    std::cout << std::endl;

    std::cout << "Toggle bit 0" << std::endl;

    var ^= mask\_bit\_0;

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Toggle bit7

    std::cout << "Toggle bit 7" << std::endl;

    var ^= mask\_bit\_7;

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    // Toggle multiple bits in one go : the 4 higher bits

    std::cout << "Toggle multiple bits in one go : the 4 higher bits" << std::endl;

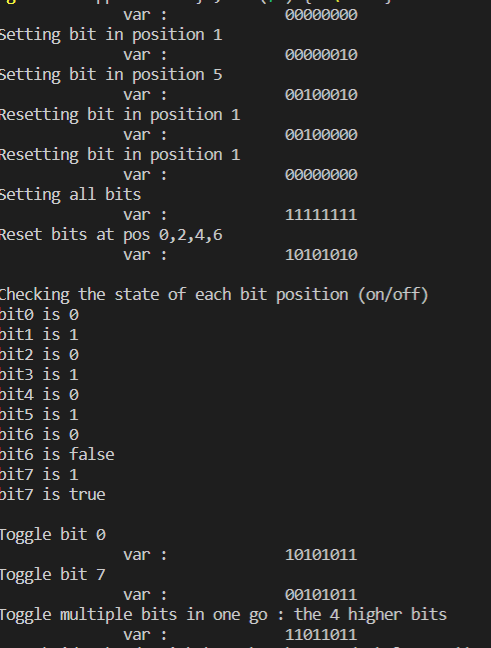
    var ^= (mask\_bit\_7 | mask\_bit\_6 | mask\_bit\_5 | mask\_bit\_4);

    std::cout << std::setw(COLUMN\_WIDTH) << "var : "

              << std::setw(COLUMN\_WIDTH) << std::bitset<8>(var) << std::endl;

    return 0;

}



#include <iostream>

const unsigned char mask\_bit\_0 {0b00000001} ;//Bit0

const unsigned char mask\_bit\_1 {0b00000010} ;//Bit1

const unsigned char mask\_bit\_2 {0b00000100} ;//Bit2

const unsigned char mask\_bit\_3 {0b00001000} ;//Bit3

const unsigned char mask\_bit\_4 {0b00010000} ;//Bit4

const unsigned char mask\_bit\_5 {0b00100000} ;//Bit5

const unsigned char mask\_bit\_6 {0b01000000} ;//Bit6

const unsigned char mask\_bit\_7 {0b10000000} ;//Bit7

void use\_options\_v0 (bool flag0, bool flag1, bool flag2, bool flag3,

      bool flag4, bool flag5, bool flag6, bool flag7){

    std::cout << "Flag0 is : " << flag0 << ", do something with it." <<  std::endl;

    std::cout << "Flag1 is : " << flag1 << ", do something with it."<<  std::endl;

    std::cout << "Flag2 is : " << flag2 << ", do something with it." <<  std::endl;

    std::cout << "Flag3 is : " << flag3 << ", do something with it."<<  std::endl;

    std::cout << "Flag4 is : " << flag4 << ", do something with it."<<  std::endl;

    std::cout << "Flag5 is : " << flag5 << ", do something with it."<<  std::endl;

    std::cout << "Flag6 is : " << flag6 << ", do something with it."<<  std::endl;

    std::cout << "Flag7 is : " << flag7 << ", do something with it."<<  std::endl;

}

void use\_options\_v1(unsigned char flags){

    std::cout << "bit0 is " << ((flags & mask\_bit\_0) >> 0 ) << ", do something with it!"<< std::endl;

    std::cout << "bit1 is " << ((flags & mask\_bit\_1) >> 1 ) <<", do something with it!"<< std::endl;

    std::cout << "bit2 is " << ((flags & mask\_bit\_2) >> 2 ) <<", do something with it!"<< std::endl;

    std::cout << "bit3 is " << ((flags & mask\_bit\_3) >> 3 ) <<", do something with it!"<< std::endl;

    std::cout << "bit4 is " << ((flags & mask\_bit\_4) >> 4 ) <<", do something with it!"<< std::endl;

    std::cout << "bit5 is " << ((flags & mask\_bit\_5) >> 5 ) <<", do something with it!"<< std::endl;

    std::cout << "bit6 is " << ((flags & mask\_bit\_6) >> 6 ) <<", do something with it!"<< std::endl;

    std::cout << "bit7 is " << ((flags & mask\_bit\_7) >> 7 ) <<", do something with it!"<< std::endl;

}

int main(){

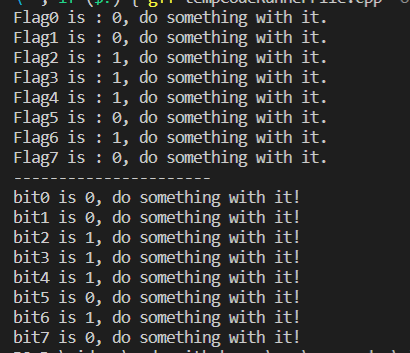
    use\_options\_v0(0,0,1,1,1,0,1,0);

    std::cout << "----------------------" << std::endl;

    use\_options\_v1(mask\_bit\_2 | mask\_bit\_3 | mask\_bit\_4 | mask\_bit\_6);

    return 0;

}



Packing color information:

#include <iostream>

int main(){

    const unsigned int red\_mask {0xFF000000};

    const unsigned int green\_mask {0x00FF0000};

    const unsigned int blue\_mask {0x0000FF00};

    const unsigned int alpha\_mask {0x000000FF};// Transparency information

    unsigned int my\_color {0xAABCDE00};

    //We shift to make sure the color byte of interest is in the

    // lower index byte position so that we can interpret that as an integer,

    // which will be between 0 and 255.

    //Set some format options

    std::cout << std::dec << std::showbase << std::endl;

    std::cout << "Red is : " <<  ((my\_color & red\_mask) >> 24) << std::endl;

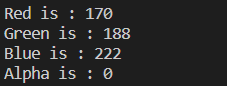
    std::cout << "Green is : " <<  ((my\_color & green\_mask) >> 16) << std::endl;

    std::cout << "Blue is : " <<  ((my\_color & blue\_mask) >> 8) << std::endl;

    std::cout << "Alpha is : " <<  ((my\_color & alpha\_mask) >> 0) << std::endl;

    return 0;

}

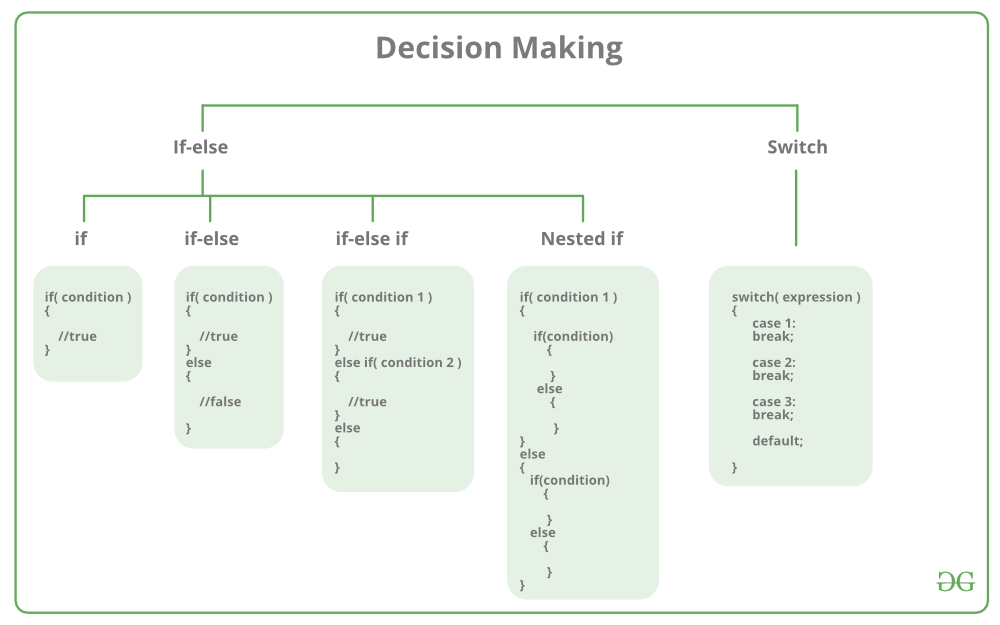


--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Refer here for the archive of [Basics in C++](https://www.geeksforgeeks.org/tag/cpp-basics/).

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

# 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.



**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**: 

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(condition)

statement1;

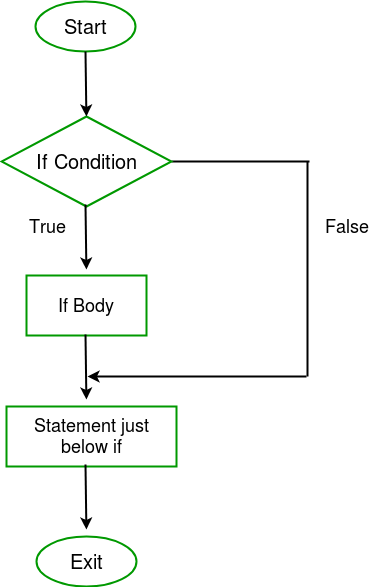
statement2;

// Here if the condition is true, if block

// will consider only statement1 to be inside

// its block.

**Flowchart** 



* C
* C++

|  |
| --- |
| // C++ program to illustrate If statement  #include<iostream>  **using** **namespace** std;    **int** main()      {  **int** i = 10;    **if** (i > 15)          {             cout<<"10 is less 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.

**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 *if*statement to execute a block of code when the condition is false.   
**Syntax**: 

if (condition)

{

// Executes this block if

// condition is true

}

else

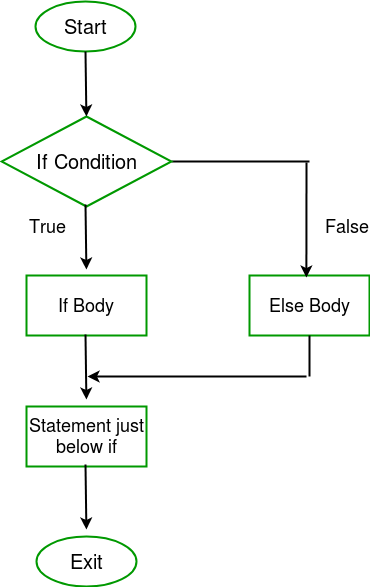
{

// Executes this block if

// condition is false

}

**Flowchart**: 



**Example:**

* C
* 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.

**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)

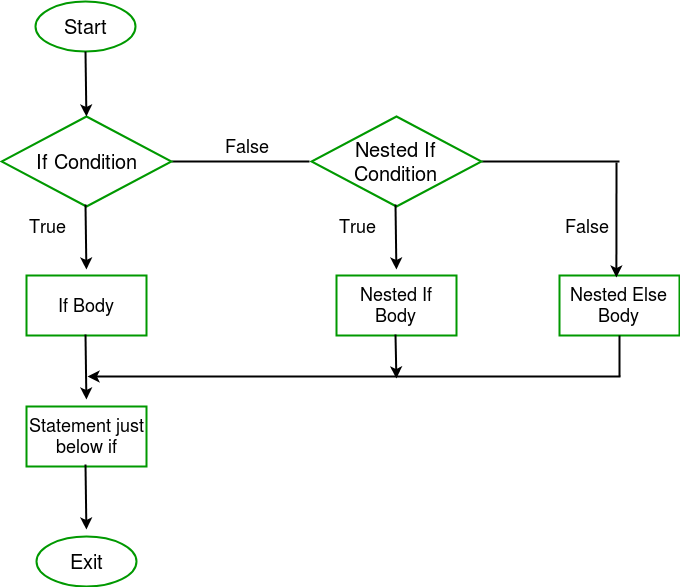
{

// Executes when condition2 is true

}

}

**Flowchart** 



|  |
| --- |
| **Example:**  // 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

**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 are true, then the final else statement will be executed.   
**Syntax:** 

if (condition)

statement;

else if (condition)

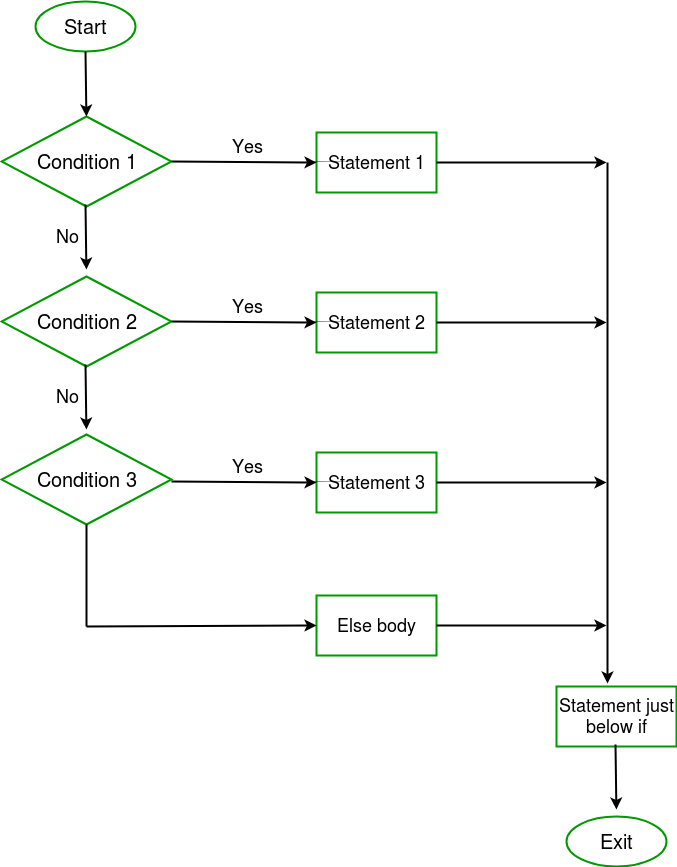
statement;

.

.

else

statement;



**Example:** 

* C
* 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

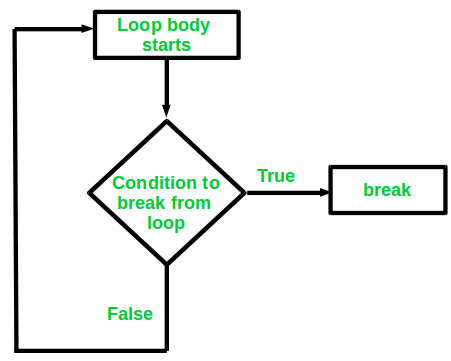
**Jump Statements in C/C++**

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

1. [**C break:**](https://www.geeksforgeeks.org/break-statement-cc/) 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;

1. 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.



1. **Example:**

|  |
| --- |
| // CPP program to illustrate  // Linear Search  #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;  } |

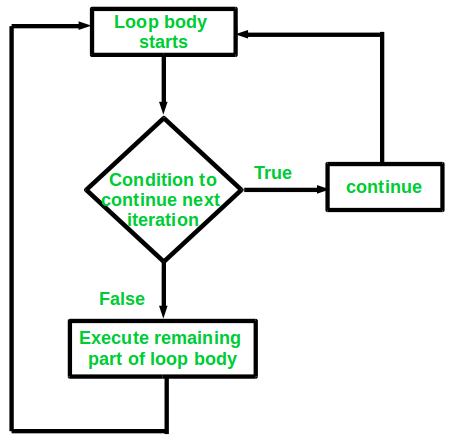


**Output:**

Element found at position: 3

1. [**C**](https://www.geeksforgeeks.org/continue-statement-cpp/)continues[**:**](https://www.geeksforgeeks.org/continue-statement-cpp/) This loop control statement is just like the [break statement](https://www.geeksforgeeks.org/break-statement-cc/). 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;



1. **Example:**

* C
* 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`.

|  |
| --- |
| #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  \*/ |

1. [**C goto:**](https://www.geeksforgeeks.org/goto-statement-in-c-cpp/) 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:

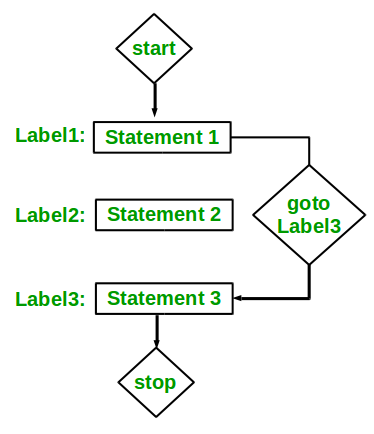
. | .

. | .

. | .

label: | goto label;

1. In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here 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.



1. Below are some examples of how to use goto statement:  
   **Examples:**

* C
* 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

2. [**C return:**](https://www.geeksforgeeks.org/return-statement-in-c-cpp-with-examples/) 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 return 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 is must be returned.   
   **Syntax:**

return[expression];

* **Example:**

|  |
| --- |
| // 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

**My codes:**

**If condition:**

#include <iostream>

int main(){

    int number1 {75};

    int number2 {60};

    bool result = (number1 < number2);//Expression yielding the condition

    /\*

    std::cout << std::boolalpha << "result : " << result << std::endl;

    std::cout << std::endl;

    std::cout << "free standing if statement" << std::endl;

    //if(result){

    if(result == true){

        std::cout << number1 << " is less than " << number2 << std::endl;

    }

    //if(!result){

    if(!(result == true)){

        std::cout << number1 << " is NOT less than " << number2 << std::endl;

    }

    \*/

    //Using else

    /\*

    std::cout << std::endl;

    std::cout << "using the else clause : " << std::endl;

    if(result == true){

        std::cout << number1 << " is less than " << number2 << std::endl;

    }else{

        std::cout << number1 << " is NOT less than " << number2 << std::endl;

    }

    \*/

    //Use expression as condition directly

    /\*

    std::cout << std::endl;

    std::cout << "Using expression as condition : " << std::endl;

    if(number1 < number2){

        std::cout << number1 << " is less than " << number2 << std::endl;

    }else{

        std::cout << number1 << " is NOT less than " << number2 << std::endl;

    }

    \*/

    //Nesting if statements

    std::cout << std::endl;

    std::cout << "Nesting if statements" << std::endl;

    bool red = false;

    bool green {true};

    bool yellow {false};

    bool police\_stop{true};

    /\*

     \*      If green : go

     \*      If red, yellow : stop

     \*      If green and police\_stop : stop

     \* \*/

     /\*

     if(red){

         std::cout << "Stop" << std::endl;

     }

     if(yellow){

         std::cout << "Slow down" << std::endl;

     }

     if(green){

         std::cout << "Go" << std::endl;

     }

     \*/

    /\*

    std::cout << std::endl;

     std::cout << "Police officer stops(verbose)" << std::endl;

     if(green){

         if(police\_stop){

             std::cout << "Stop" << std::endl;

         }

         else{

             std::cout << "Go" << std::endl;

         }

     }

     \*/

     std::cout << std::endl;

     std::cout << "Police officer stops(less verbose)" << std::endl;

     if(green && !police\_stop){

         std::cout << "Go" << std::endl;

     }else{

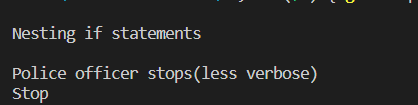
         std::cout << "Stop" << std::endl;

     }

    return 0;

}

Output:



Ifelse:

#include <iostream>

// Tools

const int Pen{ 10 };

const int Marker{ 20 };

const int Eraser{ 30 };

const int Rectangle{ 40 };

const int Circle{ 50 };

const int Ellipse{ 60 };

int main(){

    int tool {Eraser};

    if (tool == Pen) {

        std::cout << "Active tool is pen" << std::endl;

        //Do the actual painting

    }

    else if (tool == Marker) {

        std::cout << "Active tool is Marker" << std::endl;

    }

    else if (tool == Eraser) {

        std::cout << "Active tool is Eraser" << std::endl;

    }

    else if (tool == Rectangle) {

        std::cout << "Active tool is Rectangle" << std::endl;

    }

    else if (tool == Circle) {

        std::cout << "Active tool is Circle" << std::endl;

    }

    else if (tool == Ellipse) {

        std::cout << "Active tool is Ellipse" << std::endl;

    }

    std::cout << "Moving on" << std::endl;

    return 0;

}



Switch case:

#include <iostream>

#include <string>

// Tools

const int Pen{10};

const int Marker{20};

const int Eraser{30};

const int Rectangle{40};

const int Circle{50};

const int Ellipse{60};

int main()

{

     int tool{Eraser};

     switch (tool)

     {

     case Pen:

     {

          std::cout << "Active tool is Pen" << std::endl;

     }

     break;

     case Marker:

     {

          std::cout << "Active tool is Marker" << std::endl;

     }

     break;

     case Eraser:

     case Rectangle:

     case Circle:

     {

          std::cout << "Drawing Shapes" << std::endl;

     }

     break;

     case Ellipse:

     {

          std::cout << "Active tool is Ellipse" << std::endl;

     }

     break;

     default:

     {

          std::cout << "No match found" << std::endl;

     }

     break;

     }

     std::cout << "Moving on" << std::endl;

     /\*

     // Condition can only be integer of enum (We'll learn about enums later in the course)

     std::string name {"John"};

     switch (name) // Compiler error!

     {

     }

     \*/

     return 0;

}



When function is used as to run if else blocks:

#include <iostream>

bool car()

{

    std::cout << "car function running" << std::endl;

    return false;

}

bool house()

{

    std::cout << "house function running" << std::endl;

    return true;

}

bool job()

{

    std::cout << "job function running" << std::endl;

    return false;

}

bool spouse()

{

    std::cout << "spouse function running" << std::endl;

    return false;

}

int main()

{

    /\*

    bool a{ true };

    bool b{ true };

    bool c{ true };

    bool d{ false };

    bool p{ false };

    bool q{ false };

    bool r{ false };

    bool m{ true };

    //AND : If one of the operands is 0, the result is 0

    std::cout << std::endl;

    std::cout << "AND short circuit" << std::endl;

    bool result = a && b && c && d;

    std::cout << "result : " << std::boolalpha << result << std::endl;

    //OR : If one of the operands is 1, the result is 1.

    std::cout << std::endl;

    std::cout << "OR short circuit" << std::endl;

    result = p || q || r || m;

    std::cout << "result : " << std::boolalpha << result << std::endl;

    \*/

    if (car() && house() && job() && spouse())

    {

        std::cout << "I am happy" << std::endl;

    }

    else

    {

        std::cout << "I am sad" << std::endl;

    }

    if (car() || house() || job() || spouse())

    {

        std::cout << "I am happy" << std::endl;

    }

    else

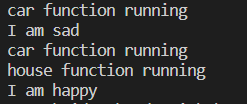
    {

        std::cout << "I am sad" << std::endl;

    }

    return 0;

}



Using true or false to run if else blocks:

#include <iostream>

int main()

{

    int condition{9};

    bool bool\_condition = condition;

    std::cout << std::boolalpha;

    if (bool\_condition)

    {

        std::cout << "We have a " << bool\_condition << " in our variable " << std::endl; // different from 0

    }

    else

    {

        std::cout << "We have " << bool\_condition << " in our variable" << std::endl; // zero

    }

    return 0;

}



Ternary operator:

#include <iostream>

int main(){

    int max{};

    int a{35};

    int b{200};

    std::cout << std::endl;

    std::cout << "using regular if " << std::endl;

    /\*

    if(a >  b){

        max = a;

    }else{

        max = b;

    }

    \*/

    max = (a > b)? a : b; // Ternary operator

    std::cout << "max : " << max << std::endl;

    return 0;

}



Using constexpr to run the ifelse block:

#include <iostream>

int main(){

   constexpr bool condition {false};

   if constexpr (condition){

       std::cout << "Condition is true" << std::endl;

   }else{

       std::cout << "Condition is false" << std::endl;

   }

    return 0;

}



Initializer inside the if else block:

#include <iostream>

int main()

{

    bool go{true};

    if (int speed{10} /\*initialsation\*/; go /\*actual condition which will run the if condition\*/)

    {

        std::cout << "speed : " << speed << std::endl;

        if (speed > 5)

        {

            std::cout << "Slow down!" << std::endl;

        }

        else

        {

            std::cout << "All good!" << std::endl;

        }

    }

    else

    {

        std::cout << "speed : " << speed << std::endl;

        std::cout << "Stop" << std::endl;

    }

    // std::cout << "Out of the if block , speed : " << speed << std::endl;not allowed ,not dfined will give error.

    return 0;

}



Initialization inside the switch case:

#include <iostream>

#include <string>

// Tools

const int Pen{10};

const int Marker{20};

const int Eraser{30};

const int Rectangle{40};

const int Circle{50};

const int Ellipse{60};

int main()

{

     int tool{Eraser};

     switch (double strength{3.56}; tool)

     {

     case Pen:

     {

          std::cout << "Active tool is Pen. strength : " << strength << std::endl;

     }

     break;

     case Marker:

     {

          std::cout << "Active tool is Marker. strength : " << strength << std::endl;

     }

     break;

     case Eraser:

     case Rectangle:

     case Circle:

     {

          std::cout << "Drawing Shapes. strength : " << strength << std::endl;

     }

     break;

     case Ellipse:

     {

          std::cout << "Active tool is Ellipse. strength : " << strength << std::endl;

     }

     break;

     default:

     {

          std::cout << "No match found. strength : " << strength << std::endl;

     }

     break;

     }

     std::cout << "Moving on" << std::endl;

     // strength++;

     return 0;

}



**Loops :**

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(general) Method (Iterative Method)

Manually we have to write the print() for C and 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++ program to illustrate 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";      cout << "Hello World\n";      cout << "Hello World\n";      cout << "Hello World\n";      cout << "Hello World\n";      cout << "Hello World\n";  **return** 0;  } |

**Output:**

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

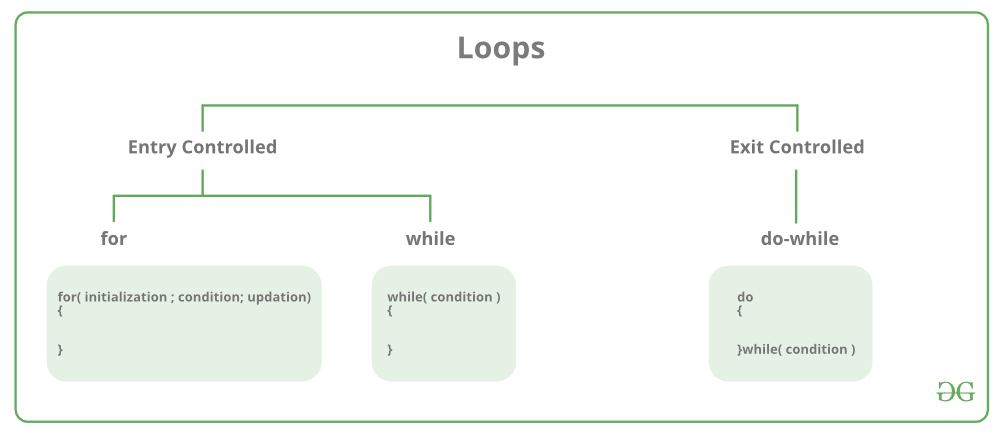
### Using Loops

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.

* An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.
* **Counter not Reached:**If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeats it.
* **Counter reached:** If the condition has been reached, the next instruction “falls through” to the next sequential instruction or branches outside the loop.

**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** – 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

}

***Example:***

*for(int i = 0; i < n; i++){*

*}*

In for loop, a loop variable is used to control the loop. First, initialize this loop variable to some value, then check whether this variable is less than or greater than the counter value. If the statement is true, then the loop body is executed and the loop variable gets updated. Steps are repeated till the exit condition comes.

* **Initialization Expression**: In this expression, we have to initialize the loop counter to some value. for example: int i=1;
* **Test Expression**: In this expression, we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression otherwise we will exit from the for a loop. For example: i <= 10;
* **Update Expression**: After executing the loop body this expression increments/decrements the loop variable by some value. for example: i++;

**Equivalent Flow Diagram for loop:**

Equivalent flow diagram for loops in C and C++

**Example:**

* C
* C++

|  |
| --- |
| // C++ program to illustrate for loop  #include <iostream>  **using** **namespace** std;    **int** main()  {  **for** (**int** i = 1; i <= 10; i++)      {          cout << "Hello World\n";      }    **return** 0;  } |

**Output:**

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

Hello World

### 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.  
**Syntax**: 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.

**initialization expression;**

while (**test\_expression**)

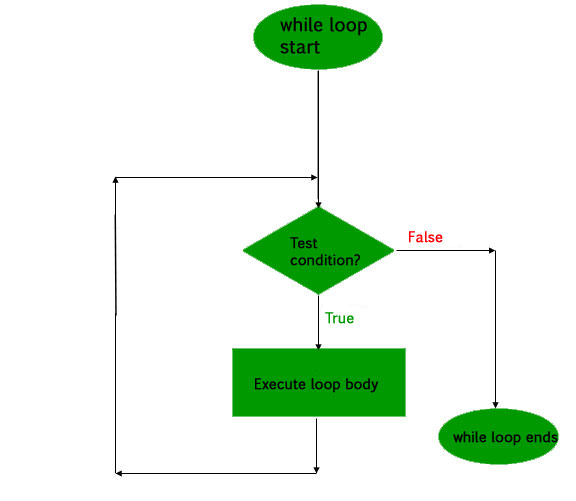
{

// statements

**update\_expression;**

}

**Flow Diagram**: 



**Example:**

* C
* C++

|  |
| --- |
| // C++ program to illustrate 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

### 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

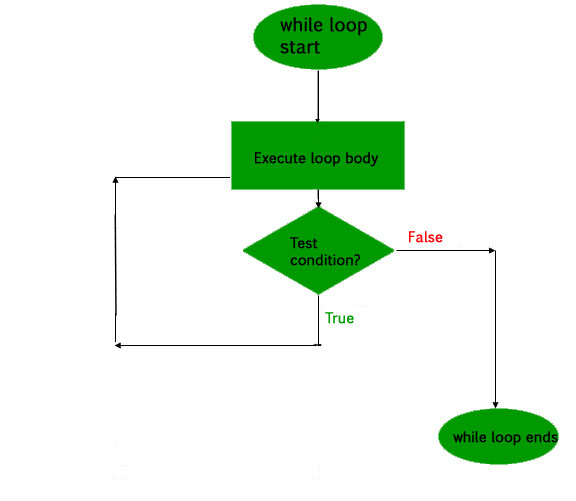
{

// statements

**update\_expression;**

} while (**test\_expression**);

**Note**: Notice the semi – colon(“;”) in the end of loop.  
**Flow Diagram**: 



**Example:**

* C
* C++

|  |
| --- |
| // C++ program to illustrate 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

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++

|  |
| --- |
| // C++ program to demonstrate infinite loops  // using for and while  // 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:**

This loop will run forever.

This loop will run forever.

...................

**Using While loop:**

* C
* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {    **while** (1)          cout << "This loop will run forever.\n";  **return** 0;  } |

**Output:**

This loop will run forever.

This loop will run forever.

...................

**Using Do-While loop:**

* C
* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main() {    **do**{          cout << "This loop will run forever.\n";      } **while**(1);    **return** 0;  } |

**Output:**

This loop will run forever.

This loop will run forever.

...................

**More Advanced Looping Techniques**

* [Range-based for loop in C++](https://www.geeksforgeeks.org/range-based-loop-c/)

# Range-based for loop in C++:

# Range-based for loop in C++ is added since C++ 11. It executes a for loop over a range. Used as a more readable equivalent to the traditional for loop operating over a range of values, such as all elements in a container.

**Syntax :**

**for ( range\_declaration : range\_expression )**

**loop\_statement**

**Parameters :**

**range\_declaration :**

a declaration of a named variable, whose type is the

type of the element of the sequence represented by

range\_expression, or a reference to that type.

Often uses the auto specifier for automatic type

deduction.

**range\_expression :**

any expression that represents a suitable sequence

or a braced-init-list.

**loop\_statement :**

any statement, typically a compound statement, which

is the body of the loop.

C++ implementation :

|  |
| --- |
| // Illustration of range-for loop  // using CPP code  #include <iostream>  #include <vector>  #include <map>    //Driver  **int** main()  {      // Iterating over whole array      std::vector<**int**> v = {0, 1, 2, 3, 4, 5};  **for** (**auto** i : v)          std::cout << i << ' ';        std::cout << '\n';        // the initializer may be a braced-init-list  **for** (**int** n : {0, 1, 2, 3, 4, 5})          std::cout << n << ' ';        std::cout << '\n';        // Iterating over array  **int** a[] = {0, 1, 2, 3, 4, 5};  **for** (**int** n : a)          std::cout << n << ' ';        std::cout << '\n';        // Just running a loop for every array      // element  **for** (**int** n : a)          std::cout << "In loop" << ' ';        std::cout << '\n';        // Printing string characters      std::string str = "Geeks";  **for** (**char** c : str)          std::cout << c << ' ';        std::cout << '\n';        // Printing keys and values of a map      std::map <**int**, **int**> MAP({{1, 1}, {2, 2}, {3, 3}});  **for** (**auto** i : MAP)          std::cout << '{' << i.first << ", "                    << i.second << "}\n";  } |

Output:

0 1 2 3 4 5

0 1 2 3 4 5

0 1 2 3 4 5

In loop In loop In loop In loop In loop In loop

G e e k s

{1, 1}

{2, 2}

{3, 3}

* [for\_each loop in C++](https://www.geeksforgeeks.org/for_each-loop-c/)

# for\_each loop in C++:

Apart from the [generic looping techniques](https://www.geeksforgeeks.org/decision-making-c-c-else-nested-else/), such as “for, while and do-while”, C++ in its language also allows us to use another functionality which solves the same purpose termed “for-each” loops. This loop accepts a function which executes over each of the container elements. This loop is defined in the header file “algorithm”: ***#include<algorithm>***, and hence has to be included for successful operation of this loop.

* It is versatile, i.e.  Can work with any container.
* It reduces chances of errors one can commit using generic for loop
* It makes code more readable
* for\_each loops improve overall performance of code

**Syntax:**

**for\_each (InputIterator** start\_iter**, InputIterator** last\_iter**, Function fnc)**

**start\_iter :** The **beginning** position

from where function operations has to be executed.

**last\_iter** : The **ending** position

till where function has to be executed.

**fnc/obj\_fnc :** The 3rd argument is a function or

an object function which operation would be applied to each element.

|  |
| --- |
| // C++ code to demonstrate the  // working of for\_each loop    #include<iostream>  #include<vector>  #include<algorithm>  **using** **namespace** std;    // helper function 1  **void** printx2(**int** a)  {      cout << a \* 2 << " ";  }    // helper function 2  // object type function  **struct** Class2  {  **void** operator() (**int** a)      {          cout << a \* 3 << " ";      }  } ob1;      **int** main()  {        // initializing array  **int** arr[5] = { 1, 5, 2, 4, 3 };        cout << "Using Arrays:" << endl;        // printing array using for\_each      // using function      cout << "Multiple of 2 of elements are : ";      for\_each(arr, arr + 5, printx2);        cout << endl;        // printing array using for\_each      // using object function      cout << "Multiple of 3 of elements are : ";      for\_each(arr, arr + 5, ob1);        cout << endl;        // initializing vector      vector<**int**> arr1 = { 4, 5, 8, 3, 1 };        cout << "Using Vectors:" << endl;          // printing array using for\_each      // using function      cout << "Multiple of 2 of elements are : ";      for\_each(arr1.begin(), arr1.end(), printx2);        cout << endl;        // printing array using for\_each      // using object function      cout << "Multiple of 3 of elements are : ";      for\_each(arr1.begin(), arr1.end(), ob1);        cout << endl;    } |

**Output**

Using Arrays:

Multiple of 2 of elements are : 2 10 4 8 6

Multiple of 3 of elements are : 3 15 6 12 9

Using Vectors:

Multiple of 2 of elements are : 8 10 16 6 2

Multiple of 3 of elements are : 12 15 24 9 3

**Exceptions and for\_each:**

In the cases of exceptions, if the function throws an exception or if any of the operations on iterators throws an exception, **for\_each** loop will also throw an exception and **break/terminate** the loop.

**Note:**

* Invalid arguments may leads to [Undefined behavior](https://www.geeksforgeeks.org/undefined-behavior-c-cpp/).
* For\_each can not work with pointers of an array (An array pointer do not know its size, for\_each loops will not work with arrays without knowing the size of an array)
* CPP

|  |
| --- |
| // C++ code to demonstrate the working  // of for\_each with Exception    #include<iostream>  #include<vector>  #include<algorithm>  **using** **namespace** std;    // Helper function 1  **void** printx2(**int** a)  {      cout << a \* 2 << " ";  **if** ( a % 2 == 0)      {  **throw** a;      }    }    // Helper function 2  // object type function  **struct** Class2  {  **void** operator() (**int** a)      {          cout << a \* 3 << " ";  **if** ( a % 2 == 0)          {  **throw** a;            }      }  } ob1;      **int** main()  {        // Initializing array  **int** arr[5] = { 1, 5, 2, 4, 3 };        cout << "Using Array" << endl;        // Printing Exception using for\_each      // using function  **try**      {          for\_each(arr, arr + 5, printx2);      }  **catch**(**int** i)      {          cout << "\nThe Exception element is : " << i ;      }      cout << endl;        // Printing Exception using for\_each      // using object function  **try**      {          for\_each(arr, arr + 5, ob1);      }  **catch**(**int** i)      {          cout << "\nThe Exception element is : " << i ;      }        // Initializing vector      vector<**int**> arr1 = { 1, 3, 6, 5, 1 };        cout << "\nUsing Vector" << endl;        // Printing Exception using for\_each      // using function  **try**      {          for\_each(arr1.begin(), arr1.end(), printx2);      }  **catch**(**int** i)      {          cout << "\nThe Exception element is : " << i ;      }      cout << endl;        // printing Exception using for\_each      // using object function  **try**      {          for\_each(arr1.begin(), arr1.end(), ob1);      }  **catch**(**int** i)      {          cout << "\nThe Exception element is : " << i ;      }  } |

**Output**

Using Array

2 10 4

The Exception element is : 2

3 15 6

The Exception element is : 2

Using Vector

2 6 12

The Exception element is : 6

3 9 18

The Exception element is : 6

**Using Lambdas:**

With the introduction of lambda functions, this can be easily used to make the whole thing inline which is very compact and useful for people looking for using functional programming.

* C++

|  |
| --- |
| #include <bits/stdc++.h>  #include <iostream>  **using** **namespace** std;    **int** main()  {        vector<**int**> vec{ 1, 2, 3, 4, 5 };        // this increases all the values in the vector by 1;      for\_each(vec.begin(), vec.end(), [](**int**& a) { a++; });        // this prints all the values in the vector;      for\_each(vec.begin(), vec.end(),               [](**int** a) { cout << a << " " << endl; });    **return** 0;  } |

**Output**

2

3

4

5

6

**Important Points:**

* Use for 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

**My codes on loops:**

**For loop:**

#include <iostream>

int main()

{

    // Print I love C++ 10 times : The bad way

    /\*

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    \*/

    // for loop : the good way

    // for (unsigned int i{0}; i < 10000; ++i)

    // {

    //     // Whatever we want the loop to run

    //     std::cout << i << " : I love C++" << std::endl;

    // }

    // std::cout << "Loop done!" << std::endl;

    std::cout << "===================================================" << std::endl;

    // Use size\_t : a representation of some unsigned int for positive numbers [sizes]

    for (size\_t i{0}; i < 10; ++i)

    {

        std::cout << i << " : I love C++" << std::endl;

    }

    std::cout << "Loop done!" << std::endl;

    std::cout << "===================================================" << std::endl;

    // sizeof(size\_t)

    std::cout << "sizeof(size\_t) : " << sizeof(size\_t) << std::endl;

    std::cout << "===================================================" << std::endl;

    // Scope of the iterator

    for (size\_t i{0}; i < 10; ++i)

    {

        std::cout << i << " : I love C++" << std::endl;

    }

    std::cout << "Loop done!" << std::endl;

    // std::cout << "i : " << i << std::endl;Compiler error : i is not in scope

    std::cout << "===================================================" << std::endl;

    // Iterator declared outside the loop

    size\_t i{0}; // Iterator defined outside

    for (i; i < 10; ++i)

    {

        std::cout << i << " : I love C++" << std::endl;

    }

    std::cout << "Loop done!" << std::endl;

    std::cout << "i : " << i << std::endl;

    std::cout << "===================================================" << std::endl;

    // Leave out the iterator declaration part

    // size\_t i{0}; // Iterator defined outside

    // for (; i < 10; ++i)

    // {

    //     std::cout << i << " : I love C++" << std::endl;

    // }

    // std::cout << "Loop done!" << std::endl;

    // std::cout << "i : " << i << std::endl;

    std::cout << "===================================================" << std::endl;

    // Don't hard code values : BAD!

    // const size\_t COUNT{100};

    // for(size\_t i{0} ; i < COUNT ; ++i){

    //     std::cout << i << " : I love C++" << std::endl;

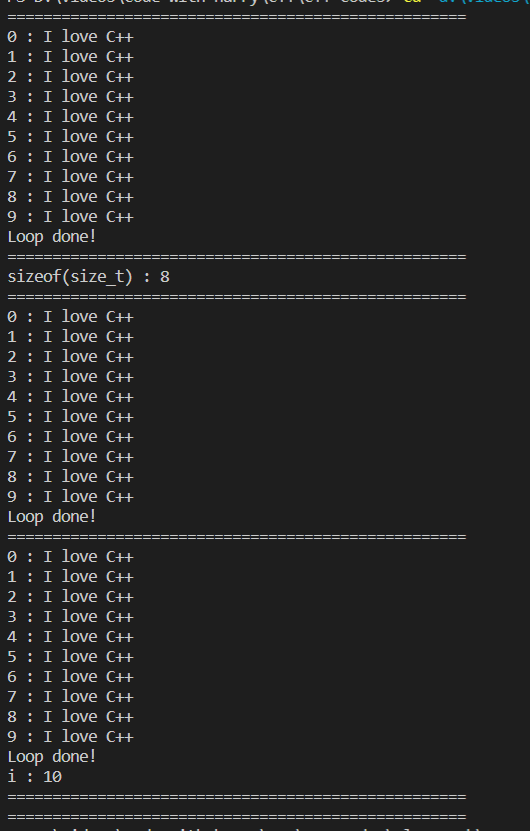
    // }

    // std::cout << "Loop done!" << std::endl;

    return 0;

}

**Output:**

****

For multiple parametered loop:

#include <iostream>

int main(){

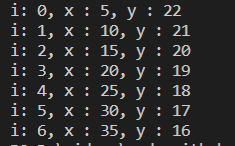
    for (size\_t i{0} , x {5}, y{22} ; y > 15 ; ++i , x+=5 , y-=1){

        std::cout << "i: " << i << ", x : " << x << ", y : " << y << std::endl;

    }

    return 0;

}



Comma operator :

#include <iostream>

int main(){

    //The comma operator combines

    //two or more  expressions into a single expression,

    // where the value of the operation is the value of its right operand

    int increment {5};

    int number1 {10};

    int number2 {20};

    int number3 {25};

    int result = (number1 \*= ++increment, number2 - (++increment), number3 += ++increment);

    std::cout << "number1 : " << number1 << std::endl; // 60

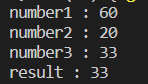
    std::cout << "number2 : " << number2 << std::endl; // 20

    std::cout << "number3 : " << number3 << std::endl; // 33

    std::cout << "result : " <<  result << std::endl; // 33

    return 0;

}



Ranged for loop:

#include <iostream>

int main()

{

    int bag\_of\_values[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // This is a collection of integers

    // The variable value will be assigned a value from the values array on each iteration

    // for (size\_t i{0}; i < 10; ++i)

    // {

    //     std::cout << "value : " << bag\_of\_values[i] << std::endl;

    // }

    // ==============================================================

    // range based:

    for (int value : bag\_of\_values)

    {

        // value holds a copy of the current iteration in the whole bag

        std::cout << " value : " << value << std::endl;

    }

    // Specify the collection in place

    for (int value : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

    {

        // value holds a copy of the current iteration in the whole bag

        std::cout << " value : " << value << std::endl;

    }

    // Auto type deduction

    for (auto value : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

    {

        // value holds a copy of the current iteration in the whole bag

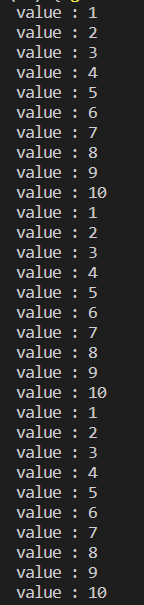
        std::cout << " value : " << value << std::endl;

    }

    return 0;

}

Output:



While loop:

#include <iostream>

int main(){

    //Print I love C++ 10 times

    /\*

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    \*/

    const size\_t COUNT{100};

    size\_t i{0}; // Iterator declaration

    while(i < COUNT ){ // Test

       std::cout << i << " : I love C++" << std::endl;

       ++i; // Incrementation

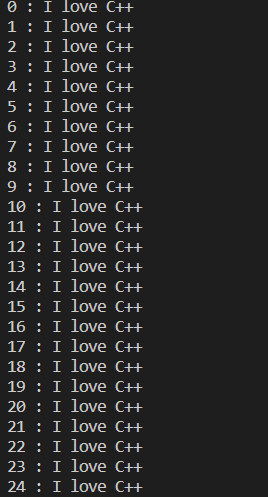
    }

    std::cout << "Loop done!" << std::endl;

    return 0;

}

Output:



Huge loops:

#include <iostream>

int main()

{

    /\*

    for(size\_t i{0} ; i < 100000 ; ++i){

        //std::cout << "i : " << i << std::endl;

    }

    std::cout << "Done!" << std::endl;

    \*/

    size\_t i{0}; // Iterator

    while (i < 100000)

    {

        // std::cout << "i : " << i << std::endl;

        ++i;

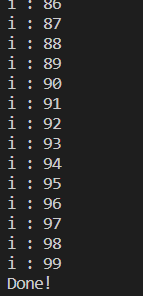
    }

    std::cout << "Done!" << std::endl;

    return 0;

}

Output:



Do while loop:

#include <iostream>

int main(){

    //Print I love C++ 10 times

    /\*

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    std::cout << "I love C++" << std::endl;

    \*/

    const int COUNT{0};

    size\_t i{0}; // Iterator declaration

    do{

        std::cout << i << " : I love C++" << std::endl;

        ++i; // Incrementation

    }while( i < COUNT);

    std::cout << "Loop done!" << std::endl;

    return 0;

}



Infinite loop:

#include <iostream>

int main(){

   //Infinite loop : for loop

   /\*

   for(size\_t i{};true ; ++i){

       std::cout << i <<  " : I love C++" << std::endl;

   }

   \*/

    //Infinite loop : while loop

    /\*

    size\_t i{0};

    while(true){

        std::cout << i  << " : I love C++" << std::endl;

        ++i;

    }

    \*/

    //Infinite loop : do while loop

    size\_t i{0};

    do{

        std::cout << i  << " : I love C++" << std::endl;

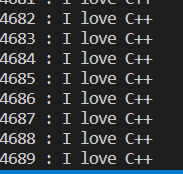
        ++i;

    }while(true);

    return 0;

}

output=:



Decrementing loops:

#include <iostream>

int main(){

    const size\_t COUNT {5};

    /\*

    // For loops

    std::cout << "Incrementing for loop : " << std::endl;

    for (size\_t i{} ; i < COUNT ; ++i){

        std::cout << "i  : " << i << std::endl;

    }

    std::cout << std::endl;

    std::cout << "Decrementing for loop : " << std::endl;

    for(size\_t i{COUNT} ; i > 0 ; --i){

        std::cout << "i : " << i << std::endl;

    }

    \*/

    //Range based for loops can't be made to run in reverse order

    //While loop

    /\*

    //Incrementing while

    std::cout << std::endl;

    std::cout << "Incrementing while" << std::endl;

    size\_t i{0};

    while( i < COUNT){

        std::cout << "i : " << i << std::endl;

        ++i;

    }

    //Decrementing while

    std::cout << std::endl;

    std::cout << "Decrementing while : " << std::endl;

    i = COUNT;

    while( i > 0){

        std::cout << "i : " << i << std::endl;

        --i;

    }

    \*/

   //Do while loop

    /\*

    //Incrementing do while

    std::cout << std::endl;

    std::cout << "Incrementing do while" << std::endl;

    size\_t i = 0;

    do {

        std::cout << "i : " << i << std::endl;

        ++i;

    }while ( i < COUNT);

    //Decrementing do while

    std::cout << std::endl;

    std::cout << "Decrementing do while" << std::endl;

    i = COUNT;

    do {

        std::cout << "i : " << i << std::endl;

        --i;

    }while ( i > 0);

    \*/

   //Infinite loop

    //Decrementing do while

    std::cout << std::endl;

    std::cout << "Decrementing do while" << std::endl;

    unsigned int i = COUNT;

    do {

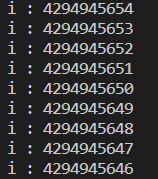
        std::cout << "i : " << i << std::endl;

        --i;

    }while ( i >= 0);

    return 0;

}



Nested loops:

#include <iostream>

#include <iomanip>

int main(){

    //Setw on numbers before you print them

    const size\_t ROWS {12};

    const size\_t COLS {3};

    /\*

    std::cout << "Tabular data visualization with nested for loops : " << std::endl;

    for (size\_t row{0} ; row < ROWS ; ++ row){

        for (size\_t col{0} ; col < COLS ; ++col){

            std::cout  << "( row "  <<  std::setw(2) << row << ",col " << std::setw(2) <<  col << ")  ";

        }

        std::cout << std::endl;

    }

    \*/

   //While loop

   /\*

    std::cout << std::endl;

    std::cout << "Tabular data visualization with nested while loops : " << std::endl;

    //Remember to reset col to 0 after the inner loop is done for the next row

    // to use the right columns.

    size\_t row {0};

    size\_t col {0};

    while(row < ROWS){

        while(col < COLS){

            std::cout  << "( row " << std::setw(2) <<  row << ",col "<< std::setw(2) <<  col << ") ";

            ++col;

        }

        std::cout << std::endl;

        col = 0 ;   // Reset col to 0 to allow printing from col 0 . col is in main

                    // function local scope now.

        ++row;

    }

    \*/

    std::cout << std::endl;

    std::cout << "Tabular data visualization with nested do while loops : " << std::endl;

    size\_t row = 0;

    size\_t col = 0;

    do { // row

        do {

            std::cout  << "( row " << std::setw(2) <<  row << ",col "<< std::setw(2) <<  col << ")   ";

            ++col;

        }while(col < COLS);

        std::cout << std::endl;

        col = 0 ;   // Reset col to 0 to allow printing from col 0 . col is in main

                    // function local scope now.

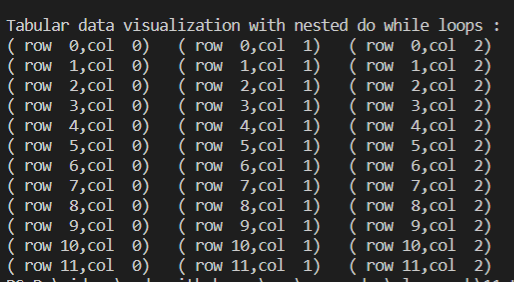
        ++row;

    }while(row < ROWS);

    return 0;

}

Output:



Break and continue in loops:

#include <iostream>

int main(){

    /\*

    // break and continue : for loops

    const size\_t COUNT{20};

    for(size\_t i{0} ; i < COUNT ; ++i ){

        if(i==5)

            continue;

        if(i == 11)

            break; // Breaks out of the loop

        std::cout << "i : " << i << std::endl;

    }

    std::cout << "Loop done!" << std::endl;

    \*/

   // break and continue : while loop

    /\*

   size\_t i{0};

   while (i < 20){

       if(i==5){

            ++i;

            continue;

       }

     if(i==11)

        break;

        std::cout << "i : " << i << std::endl;

        ++i;

   }

   std::cout << "Loop done!" << std::endl;

   \*/

  //break and continue : do while loop

  size\_t i{0};

  do{

      if(i==5){

            ++i;

            continue;

       }

     if(i==11)

        break;

        std::cout << "i : " << i << std::endl;

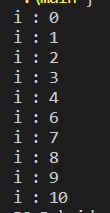
        ++i;

  }while(i <20);

    return 0;

}

Output:



For loop with initialization:

#include <iostream>

int main()

{

    for (double multiplier{4}; auto i : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10})

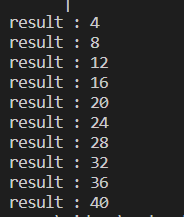
    {

        std::cout << "result : " << (i \* multiplier) << std::endl;

    }

    return 0;

}

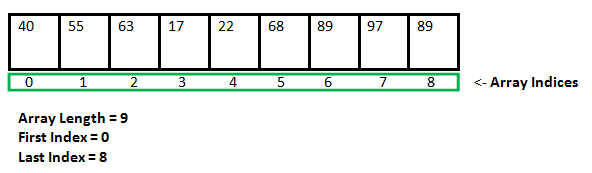


# Arrays in C/C++

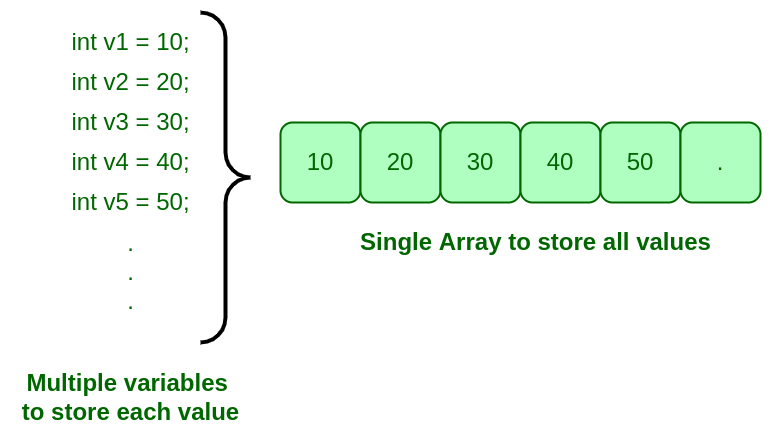
1.It is a group of variables of similar data types referred to by a single element.

1. Its elements are stored in a contiguous memory location.
2. The size of the array should be mentioned while declaring it.
3. Array elements are always counted from zero (0) onward.
4. Array elements can be accessed using the position of the element in the array.
5. The array can have one or more dimensions.

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



**Why do we need arrays?**   
We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.



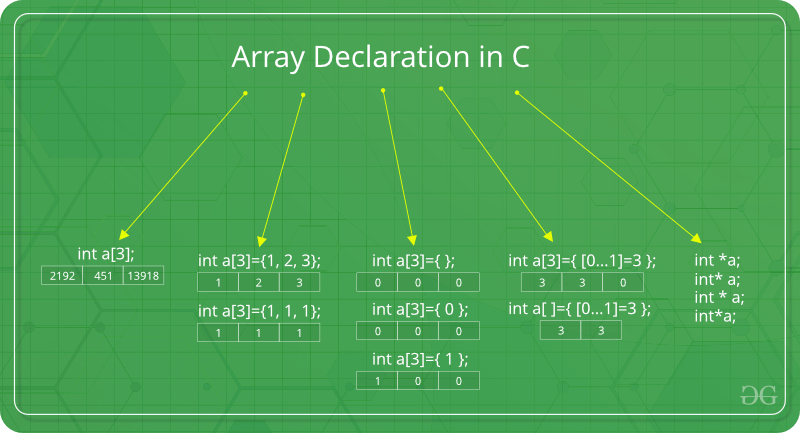
#### **Advantages:-**

* Code Optimization:  we can retrieve or sort the data efficiently.
* Random access: We can get any data located at an index position.

#### **Disadvantages:-**

* Size Limit: We can store only the fixed size of elements in the array. It doesn’t grow its size at runtime.

**Array declaration in C/C++:** 



**Note:**In the above image int a[3]={[0…1]=3}; this kind of declaration has been obsolete since GCC 2.5

There are various ways in which we can declare an array. It can be done by specifying its type and size, initializing it or both.

**Array declaration by specifying the size**

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {      // array declaration by specifying size  **int** arr1[10];        // With recent C/C++ versions, we can also      // declare an array of user specified size  **int** n = 10;  **int** arr2[n];    **return** 0;  }  // This code is contributed by sarajadhav12052009 |

**Array declaration by initializing element.**

|  |
| --- |
| // Array declaration by initializing elements  #include <iostream>  **using** **namespace** std;  **int** main()  {  **int** arr[] = { 10, 20, 30, 40};  **return** 0;  // Compiler creates an array of size 4.  // above is same as  "int arr[4] = {10, 20, 30, 40}"  } |

**Array declaration by specifying the size and initializing elements**

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {      // Array declaration by specifying size and initializing      // elements  **int** arr[6] = { 10, 20, 30, 40 };        // Compiler creates an array of size 6, initializes first      // 4 elements as specified by user and rest two elements as      // 0. above is same as  "int arr[] = {10, 20, 30, 40, 0, 0}"    **return** 0;  }    // This code is contributed by sarajadhav12052009 |

**Advantages of an Array in C/C++:**

1. Random access of elements using the array index.
2. Use of fewer lines of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing fewer lines of code.

**Disadvantages of an Array in C/C++:**

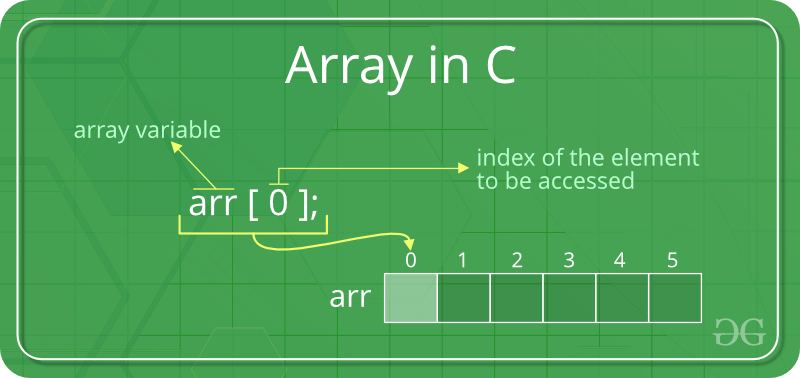
1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

**Facts about Array in C/C++:**

* **Accessing Array Elements:**

Array elements are accessed by using an integer index. Array index starts with 0 and goes till the size of the array minus 1.

* The name of the array is also a pointer to the first element of the array.



**Example:**

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **int** arr[5];      arr[0] = 5;      arr[2] = -10;        // this is same as arr[1] = 2      arr[3 / 2] = 2;      arr[3] = arr[0];        cout << arr[0] << " " << arr[1] << " " << arr[2] << " "           << arr[3];    **return** 0;  } |

**Output**

5 2 -10 5

**No Index Out of bound Checking:**   
There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

|  |
| --- |
| // This C++ program compiles fine  // as index out of bound  // is not checked in C.    #include <iostream>  **using** **namespace** std;    **int** main()  {  **int** arr[2];        cout << arr[3] << " ";      cout << arr[-2] << " ";    **return** 0;  } |

**Output**

211343841 4195777

In C, it is not a compiler error to initialise an array with more elements than the specified size. For example, the below program compiles fine and shows just a Warning.

* C

|  |
| --- |
| #include <stdio.h>  **int** main()  {        // Array declaration by initializing it      // with more elements than specified size.  **int** arr[2] = { 10, 20, 30, 40, 50 };    **return** 0;  } |

**Warnings:**

prog.c: In function 'main':

prog.c:7:25: warning: excess elements in array initializer

int arr[2] = { 10, 20, 30, 40, 50 };

^

prog.c:7:25: note: (near initialization for 'arr')

prog.c:7:29: warning: excess elements in array initializer

int arr[2] = { 10, 20, 30, 40, 50 };

^

prog.c:7:29: note: (near initialization for 'arr')

prog.c:7:33: warning: excess elements in array initializer

int arr[2] = { 10, 20, 30, 40, 50 };

^

prog.c:7:33: note: (near initialization for 'arr')

* **Note:** The program won’t compile in C++. If we save the above program as a .cpp, the program generates compiler error *“error: too many initializers for ‘int [2]'”*.

**The elements are stored at contiguous memory locations**

**Example:**

|  |
| --- |
| // C++ program to demonstrate that array elements  // are stored contiguous locations    #include <iostream>  **using** **namespace** std;    **int** main()  {      // an array of 10 integers.      // If arr[0] is stored at      // address x, then arr[1] is      // stored at x + sizeof(int)      // arr[2] is stored at x +      // sizeof(int) + sizeof(int)      // and so on.  **int** arr[5], i;        cout << "Size of integer in this compiler is "           << **sizeof**(**int**) << "\n";  **for** (i = 0; i < 5; i++)          // The use of '&' before a variable name, yields          // address of variable.          cout << "Address arr[" << i << "] is " << &arr[i]               << "\n";  **return** 0;  } |

**Output**

Size of integer in this compiler is 4

Address arr[0] is 0x7fff7a02db20

Address arr[1] is 0x7fff7a02db24

Address arr[2] is 0x7fff7a02db28

Address arr[3] is 0x7fff7a02db2c

Address arr[4] is 0x7fff7a02db30

Another way to traverse the array

|  |
| --- |
| #include<bits/stdc++.h>  **using** **namespace** std;    **int** main()  {  **int** arr[6]={11,12,13,14,15,16};      // Way 1  **for**(**int** i=0;i<6;i++)          cout<<arr[i]<<" ";      cout<<endl;        // Way 2      cout<<"By Other Method:"<<endl;  **for**(**int** i=0;i<6;i++)          cout<<i[arr]<<" ";        cout<<endl;  **return** 0;  }    // Contributed by Akshay Pawar ( Username - akshaypawar4) |

**Output**

11 12 13 14 15 16

By Other Method:

11 12 13 14 15 16

**Array vs Pointers**

Arrays and pointers are two different things (we can check by applying sizeof). The confusion happens because the array name indicates the address of the first element and arrays are always passed as pointers (even if we use a square bracket). Please see the [Difference between pointer and array in C?](https://www.geeksforgeeks.org/difference-pointer-array-c/) for more details.

**What is a vector in C++?**

A vector in C++ is a class in STL that represents an array. The advantages of vectors over normal arrays are,

* We do not need pass size as an extra parameter when we declare a vector i.e, Vectors support dynamic sizes (we do not have to specify size of a vector initially). We can also resize a vector.
* Vectors have many in-built functions like removing an element, etc.

To know more about functionalities provided by vectors, please refer to [vector in C++](https://www.geeksforgeeks.org/vector-in-cpp-stl/) for more details.

#include <iostream>

int main()

{

    // Declare an array of ints

    int scores[10]; // Junk data

    // Read data

    std::cout << " scores [0] : " << scores[0] << std::endl;

    std::cout << " scores [1] : " << scores[1] << std::endl;

    // Read with a loop

    for (size\_t i{0}; i < 10; ++i)

    {

        std::cout << "scores [" << i << "] : " << scores[i] << std::endl;

    }

    scores[0] = 20;

    scores[1] = 21;

    scores[2] = 22;

    // Print the data out

    for (size\_t i{0}; i < 10; ++i)

    {

        std::cout << "scores [" << i << "] : " << scores[i] << std::endl;

    }

    // Write data in a loop

    for (size\_t i{0}; i < 10; ++i)

    {

        scores[i] = i \* 10;

    }

    // Print the data out

    for (size\_t i{0}; i < 10; ++i)

    {

        std::cout << "scores [" << i << "] : " << scores[i] << std::endl;

    }

    // Declare and initialize at the same time

    double salaries[5]{12.7, 7.5, 13.2, 8.1, 9.3};

    for (size\_t i{0}; i < 5; ++i)

    {

        std::cout << "salary[" << i << "] : " << salaries[i] << std::endl;

    }

    // If you don't initialize all the elements, those you leave out

    // are initialized to 0

    int families[5]{12, 7, 5};

    for (size\_t i{0}; i < 5; ++i)

    {

        std::cout << "families[" << i << "] : " << families[i] << std::endl;

    }

    // Omit the size of the array at declaration

    int class\_sizes[]{10, 12, 15, 11, 18, 17, 23, 56};

    // Will print this with a range based for loop

    for (auto value : class\_sizes)

    {

        std::cout << "value : " << value << std::endl;

    }

    // Read only arrays

    const int birds[]{10, 12, 15, 11, 18, 17, 23, 56};

    //    birds[2] = 8;

    // Sum up scores array, store result in sum

    int score[]{2, 5, 8, 2, 5, 6, 9};

    int sum{0};

    for (int element : score)

    {

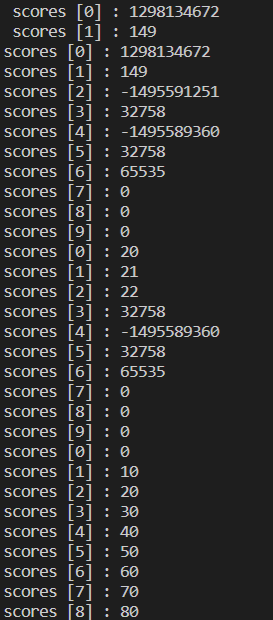
        sum += element;

    }

    std::cout << "Score sum : " << sum << std::endl;

    return 0;

}



Size of the array:

#include <iostream>

int main()

{

    int i;

    int scores[]{1, 2, 5};

    // int count{std::size(scores)}; // std::size( C++17)

    std::cout << "sizeof(scores) : " << sizeof(scores) << std::endl;

    std::cout << "sizeof(scores[0]) : " << sizeof(scores[0]) << std::endl;

    // std::cout << "count : " << count << std::endl;

    int count{sizeof(scores) / sizeof(scores[0])};

    for (size\_t i{0}; i < count; ++i)

    {

        std::cout << "scores [" << i << "] : " << scores[i] << std::endl;

    }

    // Range based for loop

    for (auto i : scores)

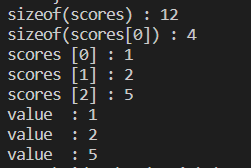
    {

        std::cout << "value  : " << i << std::endl;

    }

    return 0;

}



Char array:

#include <iostream>

int main()

{

    // Declare array

    char message[] = "Hello there modified by Rohit";

    std::cout << message << std::endl;

    // int data[5] {1,2,3,3,3};

    // std::cout << "data : " << data << std::endl;

    // Print out the array through looping

    std::cout << "message : ";

    for (auto c : message)

    {

        std::cout << c;

    }

    std::cout << std::endl;

    // Change characters in our array

    message[1] = 'a';

    // Print out the array through looping

    std::cout << "message : ";

    for (auto c : message)

    {

        std::cout << c;

    }

    std::cout << std::endl;

    // Will probably print garbage after your char array

    std::cout << "message : " << message << std::endl;

    // If a character array is null terminated, it's called as C-String

    char message1[]{'H', 'e', 'l', 'l', 'o', '\0'};

    std::cout << "message1 : " << message1 << std::endl;

    std::cout << "sizeof(message1) : " << sizeof(message1) << std::endl;

    char message2[6]{'H', 'e', 'l', 'l', 'o'};

    std::cout << "\nmessage2 : " << message2 << std::endl;

    std::cout << "sizeof(message2) : " << sizeof(message2) << std::endl;

    char message3[]{'H', 'e', 'l', 'l', 'o'}; // This is not a c string ,

                                              // as there is not null character

    std::cout << "\nmessage3 : " << message3 << std::endl;

    std::cout << "sizeof(message3) : " << sizeof(message3) << std::endl;

    // String literal

    /\*

    char message4 [] {"Hello"};

    std::cout << "message4 : " << message4 << std::endl;

    std::cout << "sizeof(message4) : " << sizeof(message4) << std::endl;

    \*/

    // Can't safely print out arrays other than those of characters

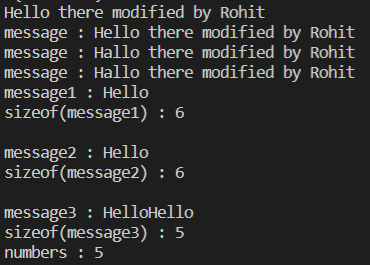
    int numbers[]{1, 2, 3, 4, 5};

    std::cout << "numbers : " << numbers[4] << std::endl;

    return 0;

}

Output:



Array bounds:

#include <iostream>

int main(){

    int numbers[] {1,2,3,4,5,6,7,8,9,0};

    //Read beyond bounds : Will read garbage or crash your program

    //std::cout << "numbers[12] : " << numbers[12] << std::endl;

    //Write beyond bounds. The compiler allows it. But you don't own

    //the memory at index 12, so other programs may modify it and your

    //program may read bogus data at a later time. Or you can even

    //corrupt data used by other parts of your ptogram

    numbers[129] = 1000;

    std::cout << "numbers[129] : " << numbers[129] << std::endl;

    std::cout << "Program ending...." << std::endl;

    return 0;

}



# Multidimensional Arrays in C / C++

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays are stored in row-major order.

The***general form of declaring N-dimensional arrays*** is:

data\_type array\_name[size1][size2]....[sizeN];

* **data\_type**: Type of data to be stored in the array.
* **array\_name**: Name of the array
* **size1, size2,… ,sizeN**: Sizes of the dimension

**Examples**:

**Two dimensional array:** int two\_d[10][20];

**Three dimensional array:** int three\_d[10][20][30];

**Size of Multidimensional Arrays:**

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.   
**For example:**

* The array **int x[10][20]** can store total (10\*20) = 200 elements.
* Similarly array **int x[5][10][20]** can store total (5\*10\*20) = 1000 elements.

### Two-Dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one-dimensional array for easier understanding.

The basic form of declaring a two-dimensional array of size x, y:   
**Syntax:**

**data\_type array\_name[x][y];**

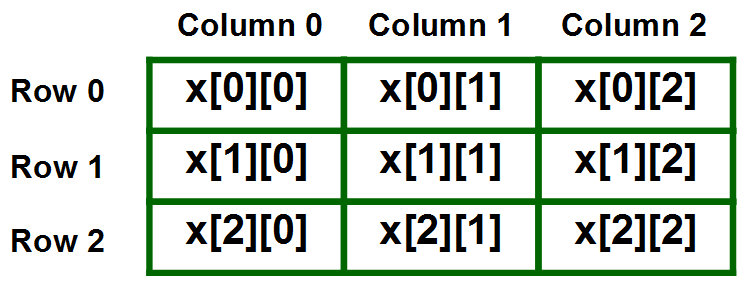
Here,**data\_type**is the type of data to be stored.

We can declare a two-dimensional integer array say ‘x’ of size 10,20 as:

int x[10][20];

Elements in two-dimensional arrays are commonly referred to by x[i][j] where i is the row number and ‘j’ is the column number.

A two – dimensional array can be seen as a table with ‘x’ rows and ‘y’ columns where the row number ranges from 0 to (x-1) and the column number ranges from 0 to (y-1). A two – dimensional array ‘x’ with 3 rows and 3 columns is shown below:



**Initializing Two – Dimensional Arrays**: There are various ways in which a Two-Dimensional array can be initialized.

**First Method**:

int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in order, the first 4 elements from the left in the first row, the next 4 elements in the second row, and so on.

**Second Method**:

int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};

**Third Method:**

int x[3][4];

for(int i = 0; i < 3; i++){

for(int j = 0; j < 4; j++){

cin >> x[i][j];

}

}

**Fourth Method(Dynamic Allocation):**

int\*\* x = new int\*[3];

for(int i = 0; i < 3; i++){

x[i] = new int[4];

for(int j = 0; j < 4; j++){

cin >> x[i][j];

}

}

This type of initialization makes use of nested braces. Each set of inner braces represents one row. In the above example, there is a total of three rows so there are three sets of inner braces.

**Accessing Elements of Two-Dimensional Arrays:** Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

**Example:**

int x[2][1];

The above example represents the element present in the third row and second column.

**Note**: In arrays, if the size of an array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is 2+1 = 3. To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two ‘*for*‘ loops. One to traverse the rows and another to traverse columns.   
 

**Example:**

|  |
| --- |
| // C++ Program to print the elements of a  // Two-Dimensional array  #include<iostream>  **using** **namespace** std;    **int** main()  {      // an array with 3 rows and 2 columns.  **int** x[3][2] = {{0,1}, {2,3}, {4,5}};        // output each array element's value  **for** (**int** i = 0; i < 3; i++)      {  **for** (**int** j = 0; j < 2; j++)          {              cout << "Element at x[" << i                   << "][" << j << "]: ";              cout << x[i][j]<<endl;          }      }    **return** 0;  } |

**Output:**

Element at x[0][0]: 0

Element at x[0][1]: 1

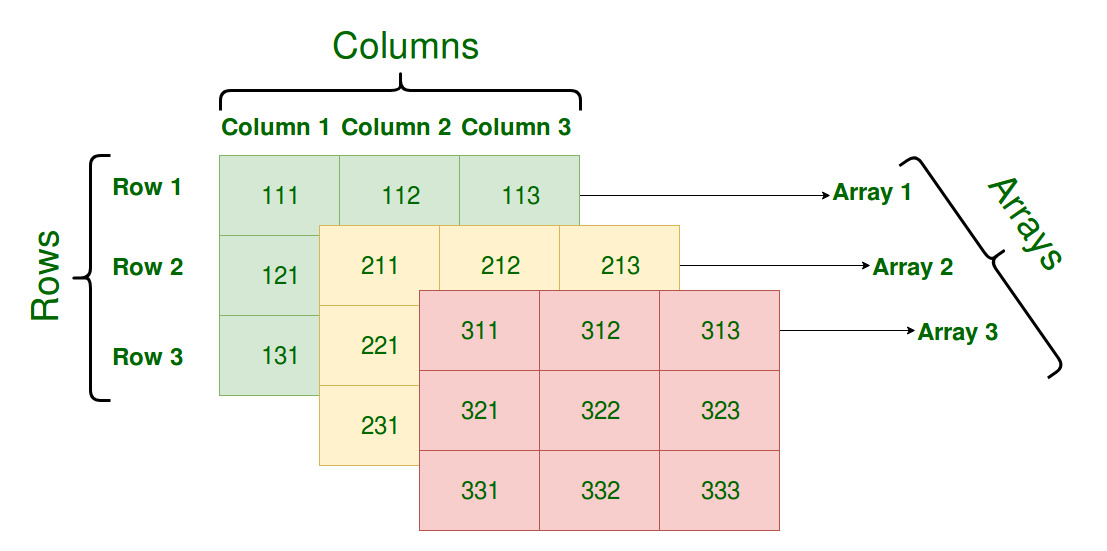
Element at x[1][0]: 2

Element at x[1][1]: 3

Element at x[2][0]: 4

Element at x[2][1]: 5

### Three-Dimensional Array



**Initializing Three-Dimensional Array**: Initialization in a Three-Dimensional array is the same as that of Two-dimensional arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

**Method 1**:

int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

11, 12, 13, 14, 15, 16, 17, 18, 19,

20, 21, 22, 23};

**Method 2(Better)**:

int x[2][3][4] =

{

{ {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },

{ {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }

};

**Accessing elements in Three-Dimensional Arrays**: Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

|  |
| --- |
| // C++ program to print elements of Three-Dimensional  // Array  #include <iostream>  **using** **namespace** std;    **int** main()  {      // initializing the 3-dimensional array  **int** x[2][3][2] = { { { 0, 1 }, { 2, 3 }, { 4, 5 } },                         { { 6, 7 }, { 8, 9 }, { 10, 11 } } };      // output each element's value  **for** (**int** i = 0; i < 2; ++i) {  **for** (**int** j = 0; j < 3; ++j) {  **for** (**int** k = 0; k < 2; ++k) {                  cout << "Element at x[" << i << "][" << j                       << "][" << k << "] = " << x[i][j][k]                       << endl;              }          }      }  **return** 0;  } |

**Output:**

Element at x[0][0][0] = 0

Element at x[0][0][1] = 1

Element at x[0][1][0] = 2

Element at x[0][1][1] = 3

Element at x[0][2][0] = 4

Element at x[0][2][1] = 5

Element at x[1][0][0] = 6

Element at x[1][0][1] = 7

Element at x[1][1][0] = 8

Element at x[1][1][1] = 9

Element at x[1][2][0] = 10

Element at x[1][2][1] = 11

In similar ways, we can create arrays with any number of dimensions. However, the complexity also increases as the number of dimensions increases. The most used multidimensional array is the Two-Dimensional Array.

#include <iostream>

int main()

{

    // 2D array

    // int packages [3][4] ; // 12 where we can store ints

    int packages[][4]{

        {1, 2, 3, 4},

        {5, 6, 7, 8},

        {9, 10, 11, 12},

        {3, 4, 5, 6}};

    // Read data from a 2D array

    /\*

    for(size\_t i{0} ; i < 3; ++ i){

        for(size\_t j{0}; j < 4 ; ++j){

            std::cout << packages[i][j] << "   ";

        }

        std::cout << std::endl;

    }

    \*/

    // Use std::size to query the size of array dimensions

    /\*

    for(size\_t i{0} ; i < std::size(packages); ++ i){

        for(size\_t j{0}; j < std::size(packages[i])  ; ++j){

            std::cout << packages[i][j] << "   ";

        }

        std::cout << std::endl;

    }

    \*/

    // 3D arrays are defined in the same way. We just use three sets of indexes

    //  3 lights per room, 5 rooms per house 7 houses per block

    int house\_block[7][5][3]{

        {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}, {10, 11, 12}, {13, 14, 15}},

        {{16, 17, 18}, {19, 20, 21}, {22, 23, 24}, {25, 26, 27}, {28, 29, 30}},

        {{31, 32, 33}, {34, 35, 36}, {37, 38, 39}, {40, 41, 42}, {43, 44, 45}},

        {{46, 47, 48}, {49, 50, 51}, {52, 53, 54}, {55, 56, 57}, {58, 59, 60}},

        {{61, 62, 63}, {64, 65, 66}, {67, 68, 69}, {70, 71, 72}, {73, 74, 75}},

        {{76, 77, 78}, {79, 80, 81}, {82, 83, 84}, {85, 86, 87}, {88, 89, 90}},

        {{91, 92, 93}, {94, 95, 96}, {97, 98, 99}, {100, 101, 102}, {103, 104, 105}}};

    for (size\_t i{0}; i < std::size(house\_block); ++i)

    {

        for (size\_t j{0}; j < std::size(house\_block[i]); ++j)

        {

            for (size\_t k{0}; k < std::size(house\_block[i][j]); ++k)

            {

                std::cout << house\_block[i][j][k] << "     ";

            }

        }

    }

    /\*

    const size\_t num\_cols{3};

    int packages2 [] [5] {

        {1,2,3},

        {4,5,6},

        {7,8,9},

        {10,11,12},

        {100,110,120} // Can add as many triplets in packages2 as we want

    };

    std::cout << std::endl;

    std::cout << "Ommiting leftmost dimension for 2d array : " << std::endl;

    for (size\_t i{0} ; i < std::size(packages2) ; ++i){

        for( size\_t j{0} ; j < std::size(packages2[i]) ; ++j){

            std::cout << "Item (" << i << "," << j << ") : " << packages2[i][j] << std::endl;

        }

    }

    \*/

    // For 3d and really any multi dimensional array , you have to specify

    // the number of elements in []'s , only the left most is not mandatory.

    // Below is the example for 3D reproduced.Omitting the 5 or the 3 or both

    // will cause a compile error.

    /\*

    int house\_block1 [] [5] [3] {

        {

            {1,2,3},{4,5,6},{7,8,9},{10,11,12},{13,14,15}

        },

        {

            {16,17,18},{19,20,21},{22,23,24},{25,26,27},{28,29,30}

        },

        {

            { 31,32,33},{34,35,36},{37,38,39},{40,41,42},{43,44,45}

        },

        {

             {46,47,48},{49,50,51},{52,53,54},{55,56,57},{58,59,60}

        }

    };

    for (size\_t i {0} ; i < std::size(house\_block1 ); ++i){

        for( size\_t j{0}; j < std::size(house\_block1[i]) ; ++j){

            std::cout << "[";

            for( size\_t k{0}; k < std::size(house\_block1[i][j]) ; ++k){

                std::cout << house\_block1[i][j][k] << " ";

            }

            std::cout <<"] " ;//Separate elements for good visualization

        }

        std::cout << std::endl;//Separate elements for good visualization

    }

    \*/

    // If you put less elements than the amount specified in the declaration, the compiler

    // will automaticaly fill out those left out with 0 or '\0' is the array is of char.

    //  Change the array to int house\_block [] [5] [4], and because all inner arrays have 3,

    //  elements, the fourth will be autofilled with 0.

    //  Also remove some other elements, I chose the {4,5,6} array but you can add others

    //  when recording to make this a bit fun. The left out indexes will be auto filled with

    //  zeros

    /\*

     int house\_block2 [] [5] [4] {  // Try to make the last dimension 5 and show

                                     //that the compiler appends 2 zeros

         {

             {1,2,3},{4},{7,8,9},{10,11,12},{13,14,15}  //The one element array will be autofilled

                                                     // with zeros to complete 4 elements

         },

         {

             {16,17,18},{19,20,21},{22,23,24},{25,26,27},{28,29,30}

         },

         {

             { 31,32,33},{34,35,36},{37,38,39},{40,41,42},{43,44,45}

         },

         {

              {46,47,48},{49,50,51},{52,53,54},{55,56,57}//,{58,59,60}  // The spots for {58,59,60

                                                                        // Will be autofilled with 0

         }

     };

     //Modify data

     house\_block2[0][2][1] = 1021;

     std::cout << std::endl;

     std::cout << "Printing out 3d house\_block array with defaulted elements : " << std::endl;

     for (size\_t i {0} ; i < std::size(house\_block2 ); ++i){

         for( size\_t j{0}; j < std::size(house\_block2[i]) ; ++j){

             std::cout << "[";

             for( size\_t k{0}; k < std::size(house\_block2[i][j]) ; ++k){

                 std::cout << house\_block2[i][j][k] << " ";

             }

             std::cout <<"] " ;//Separate elements for good visualization

         }

         std::cout << std::endl;//Separate elements for good visualization

     }

 \*/

    return 0;

}

**Multidimensional array of characters:**

#include <iostream>

int main(){

    const size\_t name\_length{15};

    char members [][name\_length] {

        {'J','o','h','n'},

        {'S','a','m','u','e','l',},

        {'R','a','s','h','i','d'},

        {'R','o','d','r','i','g','e','z'}

    };

    //Declaring a 2D array

    //Printing out like this is unsafe : may go over and print

    //outside your valid memory block

    //until a terminating null character is encountered.

    /\*

    std::cout << "Unsafe printing of members : " << std::endl;

    for (size\_t i {0}; i < std::size(members) ; ++i){

        std::cout << members[i] << std::endl;

    }

    \*/

    //Can loop around manually printing out each character

    /\*

    std::cout << std::endl;

    std::cout << "Printing out character by character manually : " << std::endl;

    for (size\_t i{0} ; i < std::size(members) ; ++i){

        for (size\_t j{0} ; j < std::size(members[i]) ; ++j){

            std::cout << members[i][j] ;

        }

        std::cout << std::endl;

    }

    \*/

    //Better : Using C-string litterals

    //Compared to initialization with charactes with in '', this

    // is even easier to type. The entire string is a single entity

    //you can manage easily.

    /\*

    char members1 [][name\_length] {

        "John",

        "Samuel",

        "Rashid",

        "Rodriguez"

    };

    //Printing out members1

    std::cout << "Printing out members1 (C-string literals) : " << std::endl;

    for (size\_t i {0}; i < std::size(members1) ; ++i){

        std::cout << members1[i] << std::endl;

    }

    \*/

    //Updating our prediction declaration in the fortune teller game

    char predictions [] [90] {

        "a lot of kinds running in the backyard!",

        "a lot of empty beer bootles on your work table.",

        "you Partying too much with kids wearing weird clothes.",

        "you running away from something really scary",

        "clouds gathering in the sky and an army standing ready for war",

        "dogs running around in a deserted empty city",

        "a lot of cars stuck in a terrible traffic jam",

        "you sitting in the dark typing lots of lines of code on your dirty computer",

        "you yelling at your boss. And oh no! You get fired!",

        "you laughing your lungs out. I've never seen this before." ,

        "Uhm , I don't see anything!"

    };

    std::cout << std::endl;

    std::cout << "Prediction : " << predictions[5] << std::endl;

    return 0;

}

# How to print size of array parameter in C++?

How to compute the size of an array parameter in a function?  
Consider below C++ program:

* CPP

|  |
| --- |
| // A C++ program to show that it is wrong to  // compute size of an array parameter in a function  #include <iostream>  **using** **namespace** std;    **void** findSize(**int** arr[])  {      cout << **sizeof**(arr) << endl;  }    **int** main()  {  **int** a[10];      cout << **sizeof**(a) << " ";      findSize(a);  **return** 0;  } |

Output:

40 8

The above output is for a machine where the size of an integer is 4 bytes and the size of a pointer is 8 bytes.  
The **cout** statement inside main prints 40, and **cout** in findSize prints 8. The reason is, arrays are always passed pointers in functions, i.e., findSize(int arr[]) and findSize(int \*arr) mean exactly same thing. Therefore the cout statement inside findSize() prints the size of a pointer. See [this](https://www.geeksforgeeks.org/using-sizof-operator-with-array-paratmeters/) and [this](https://www.geeksforgeeks.org/why-c-treats-array-parameters-as-pointers/) for details.  
**How to find the size of an array in function?**   
We can pass a ‘reference to the array’. 

* CPP

|  |
| --- |
| // A C++ program to show that we can use reference to  // find size of array  #include <iostream>  **using** **namespace** std;    **void** findSize(**int** (&arr)[10])  {      cout << **sizeof**(arr) << endl;  }    **int** main()  {  **int** a[10];      cout << **sizeof**(a) << " ";      findSize(a);  **return** 0;  } |

Output:

40 40

The above program doesn’t look good as we have a hardcoded size of the array parameter. We can do it better using [templates in C++](http://geeksquiz.com/templates-cpp/).

|  |
| --- |
| // A C++ program to show that we use template and  // reference to find size of integer array parameter  #include <iostream>  **using** **namespace** std;    **template** <**size\_t** n>  **void** findSize(**int** (&arr)[n])  {      cout << **sizeof**(**int**) \* n << endl;  }    **int** main()  {  **int** a[10];      cout << **sizeof**(a) << " ";      findSize(a);  **return** 0;  } |

Output:

40 40

We can make a generic function as well:

|  |
| --- |
| // A C++ program to show that we use template and  // reference to find size of any type array parameter  #include <iostream>  **using** **namespace** std;    **template** <**typename** T, **size\_t** n>  **void** findSize(T (&arr)[n])  {      cout << **sizeof**(T) \* n << endl;  }    **int** main()  {  **int** a[10];      cout << **sizeof**(a) << " ";      findSize(a);    **float** f[20];      cout << **sizeof**(f) << " ";      findSize(f);  **return** 0;  } |

Output:

40 40

80 80

Now the next step is to print the size of a dynamically allocated array. It’s your task man! I’m giving you a hint.

* CPP

|  |
| --- |
| #include <iostream>  #include <cstdlib>  **using** **namespace** std;    **int** main()  {  **int** \*arr = (**int**\*)**malloc**(**sizeof**(**int**) \* 20);  **return** 0;  } |

# std::string class in C++

C++ has in its definition a way to represent a **sequence of characters as an object of the class**. This class is called std:: string. String class stores the characters as a sequence of bytes with the functionality of allowing **access to the single-byte character**.

### String vs Character Array

| String | Char Array |
| --- | --- |
| A string is a **class that defines objects** that be represented as a stream of characters. | A character array is simply an **array of characters** that can be terminated by a null character. |
| In the case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated,**no memory is wasted**. | The size of the character array has to be **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is also wasted** |
| As strings are represented as objects, **no array decay** occurs. | There is a**threat of**[**array decay**](https://www.geeksforgeeks.org/what-is-array-decay-in-c-how-can-it-be-prevented/) in the case of the character array. |
| **Strings are slower** when compared to implementation than character array. | Implementation of**character array is faster** than std:: string. |
| String class defines**a number of functionalities** that allow manifold operations on strings. | Character arrays **do not offer** many **inbuilt functions**to manipulate strings. |

## **Operations on Strings**

### 1) Input Functions

| Function | Definition |
| --- | --- |
| [getline()](https://www.geeksforgeeks.org/getline-string-c/) | This function is used to store a stream of characters as entered by the user in the object memory. |
| [push\_back()](https://www.geeksforgeeks.org/stdstringpush_back-in-cpp/) | This function is used to input a character at the end of the string. |
| pop\_back() | Introduced from C++11(for strings), this function is used to delete the last character from the string. |

**Example:**

* CPP

|  |
| --- |
| // C++ Program to demonstrate the working of  // getline(), push\_back() and pop\_back()  #include <iostream>  #include <string> // for string class  **using** **namespace** std;    // Driver Code  **int** main()  {      // Declaring string      string str;        // Taking string input using getline()      getline(cin, str);        // Displaying string      cout << "The initial string is : ";      cout << str << endl;        // Inserting a character      str.push\_back('s');        // Displaying string      cout << "The string after push\_back operation is : ";      cout << str << endl;        // Deleting a character      str.pop\_back();        // Displaying string      cout << "The string after pop\_back operation is : ";      cout << str << endl;    **return** 0;  } |

**Output:**

The initial string is : geeksforgeek

The string after push\_back operation is : geeksforgeeks

The string after pop\_back operation is : geeksforgeek

### **2) Capacity Functions**

| Function | Definition |
| --- | --- |
| capacity() | This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently. |
| [resize()](https://www.geeksforgeeks.org/stdstringresize-in-c/) | This function changes the size of the string, the size can be increased or decreased. |
| length() | This function finds the length of the string. |
| shrink\_to\_fit() | This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters has to be made. |

**Example:**

* CPP

|  |
| --- |
| // C++ Program to demonstrate the working of  // capacity(), resize() and shrink\_to\_fit()  #include <iostream>  #include <string> // for string class  **using** **namespace** std;    // Driver Code  **int** main()  {      // Initializing string      string str = "geeksforgeeks is for geeks";        // Displaying string      cout << "The initial string is : ";      cout << str << endl;        // Resizing string using resize()      str.resize(13);        // Displaying string      cout << "The string after resize operation is : ";      cout << str << endl;        // Displaying capacity of string      cout << "The capacity of string is : ";      cout << str.capacity() << endl;        // Displaying length of the string      cout << "The length of the string is :" << str.length()           << endl;        // Decreasing the capacity of string      // using shrink\_to\_fit()      str.shrink\_to\_fit();        // Displaying string      cout << "The new capacity after shrinking is : ";      cout << str.capacity() << endl;    **return** 0;  } |

**Output**

The initial string is : geeksforgeeks is for geeks

The string after resize operation is : geeksforgeeks

The capacity of string is : 26

The length of the string is :13

The new capacity after shrinking is : 15

### 3) Iterator Functions

| Function | Definition |
| --- | --- |
| begin() | This function returns an iterator to the beginning of the string. |
| end() | This function returns an iterator to the end of the string. |
| rbegin() | This function returns a reverse iterator pointing at the end of the string. |
| rend() | This function returns a reverse iterator pointing at beginning of the string. |

**Example:**

* CPP

|  |
| --- |
| // C++ Program to demonstrate the working of  // begin(), end(), rbegin(), rend()  #include <iostream>  #include <string> // for string class  **using** **namespace** std;    // Driver Code  **int** main()  {      // Initializing string`      string str = "geeksforgeeks";        // Declaring iterator      std::string::iterator it;        // Declaring reverse iterator      std::string::reverse\_iterator it1;        // Displaying string      cout << "The string using forward iterators is : ";  **for** (it = str.begin(); it != str.end(); it++)          cout << \*it;      cout << endl;        // Displaying reverse string      cout << "The reverse string using reverse iterators is "              ": ";  **for** (it1 = str.rbegin(); it1 != str.rend(); it1++)          cout << \*it1;      cout << endl;    **return** 0;  } |

**Output**

The string using forward iterators is : geeksforgeeks

The reverse string using reverse iterators is : skeegrofskeeg

**4) Manipulating Functions:**

| Function | Definition |
| --- | --- |
| copy(“char array”, len, pos) | This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied, and starting position in the string to start copying. |
| swap() | This function swaps one string with other. |

**Example:**

* CPP

|  |
| --- |
| // C++ Program to demonstrate the working of  // copy() and swap()  #include <iostream>  #include <string> // for string class  **using** **namespace** std;    // Driver Code  **int** main()  {      // Initializing 1st string      string str1 = "geeksforgeeks is for geeks";        // Declaring 2nd string      string str2 = "geeksforgeeks rocks";        // Declaring character array  **char** ch[80];        // using copy() to copy elements into char array      // copies "geeksforgeeks"      str1.copy(ch, 13, 0);        // Displaying char array      cout << "The new copied character array is : ";      cout << ch << endl;        // Displaying strings before swapping      cout << "The 1st string before swapping is : ";      cout << str1 << endl;      cout << "The 2nd string before swapping is : ";      cout << str2 << endl;        // using swap() to swap string content      str1.swap(str2);        // Displaying strings after swapping      cout << "The 1st string after swapping is : ";      cout << str1 << endl;      cout << "The 2nd string after swapping is : ";      cout << str2 << endl;    **return** 0;  } |

**Output**

The new copied character array is : geeksforgeeks

The 1st string before swapping is : geeksforgeeks is for geeks

The 2nd string before swapping is : geeksforgeeks rocks

The 1st string after swapping is : geeksforgeeks rocks

The 2nd string after swapping is : geeksforgeeks is for geeks

**Must Read:**[C++ String Class and its Applications](https://www.geeksforgeeks.org/c-string-class-and-its-applications/)

This article is contributed by [**Manjeet Singh**](https://auth.geeksforgeeks.org/profile.php?user=manjeet_04&list=practice). If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](https://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.

# Raw String Literal in C++

A Literal is a constant variable whose value does not change during the lifetime of the program. Whereas, a raw string literal is a string in which the escape characters like ‘ **\n**, **\t,** or **\”** ‘ of C++ are not processed. Hence, a raw string literal that starts with**R”( and ends in )”.**

**The syntax for Raw string Literal:**

R "delimiter( raw\_characters )delimiter" // delimiter is the end of logical entity

Here, delimiter is optional and it can be a character except the backslash{ / }, whitespaces{  }, and parentheses { () }.

These raw string literals allow a series of characters by writing precisely its contents like raw character sequence**.**

**Example:**

**Ordinary String Literal**

"\\\\n"

**Raw String Literal**

\/-- Delimiter

R"(\\n)"

/\-- Delimiter

**Difference between an Ordinary String Literal and a Raw String Literal:**

| **Ordinary String Literal** | **Raw String Literal** |
| --- | --- |
| It does not need anything to be defined. | It needs a defined line{ parentheses ()} to start with the prefix **R.** |
| It does not allow/include nested characters. | It allows/includes nested character implementation. |
| It does not ignore any special meaning of character and implements their special characteristic. | It ignores all the special characters like **\n**and**\t** and treats them like normal text. |

**Example of Raw String Literal:**

* CPP

|  |
| --- |
| // C++ program to demonstrate working of raw string literal  #include <iostream>  **using** **namespace** std;    // Driver Code  **int** main()  {      // A Normal string      string string1 = "Geeks.\nFor.\nGeeks.\n";        // A Raw string      string string2 = R"(Geeks.\nFor.\nGeeks.\n)";        cout << string1 << endl;        cout << string2 << endl;    **return** 0;  } |

**Output**

Geeks.

For.

Geeks.

Geeks.\nFor.\nGeeks.\n

# Array of Strings in C++ – 5 Different Ways to Create

In C++, a string is usually just an array of (or a reference/points to) characters that ends with the NULL character ‘**\0**‘. A string is a 1-dimensional array of characters and an array of strings is a 2-dimensional array of characters.

***Below are the 5 different ways to create an Array of Strings in C++:***

* Using Pointers
* Using 2-D Array
* Using the String Class
* Using the Vector Class
* Using the Array Class

## **1. Using Pointers**

[Pointers](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) are the symbolic representation of an address. In simple words, a pointer is something that stores the address of a variable in it. In this method, an array of string literals is created by an array of pointers in which the character or string pointer points to the very first value of the created array and will always point to it until it is traversed.

**Example:**

* CPP

|  |
| --- |
| // C++ program to demonstrate array of strings using  // pointers character array  #include <iostream>  #include <stdio.h>    **int** main()  {      // Initialize array of pointer  **const** **char**\* colour[4]          = { "Blue", "Red", "Orange", "Yellow" };        // Printing Strings stored in 2D array  **for** (**int** i = 0; i < 4; i++)          std::cout << colour[i] << "\n";    **return** 0;  } |

**Output:**

Blue

Red

Orange

Yellow

* The number of strings is fixed, but needn’t be. The 4 may be omitted, and the compiler will compute the correct size.
* These strings are constants and their contents cannot be changed. Because string literals (literally, the quoted strings) exist in a read-only area of memory, we must specify “const” here to prevent unwanted accesses that may crash the program.

## **2. Using a 2D array**

A [2-D array](https://www.geeksforgeeks.org/multidimensional-arrays-c-cpp/) is the simplest form of a multidimensional array in which it stores the data in a tabular form. This method is useful when the length of all strings is known and a particular memory footprint is desired. Space for strings will be allocated in a single block

**Example**:

* CPP

|  |
| --- |
| // C++ program to demonstrate array of strings using  // 2D character array  #include <iostream>    **int** main()  {      // Initialize 2D array  **char** colour[4][10]          = { "Blue", "Red", "Orange", "Yellow" };        // Printing Strings stored in 2D array  **for** (**int** i = 0; i < 4; i++)          std::cout << colour[i] << "\n";    **return** 0;  } |

**Output:**

Blue

Red

Orange

Yellow

* Both the number of strings and the size of strings are fixed. The 4, again, may be left out, and the appropriate size will be computed by the compiler. The second dimension, however, must be given (in this case, 10), so that the compiler can choose an appropriate memory layout.
* Each string can be modified but will take up the full space given by the second dimension. Each will be laid out next to each other in memory, and can’t change size.
* Sometimes, control over the memory footprint is desirable, and this will allocate a region of memory with a fixed, regular layout.

## **3. Using the String class**

The STL [string](https://www.geeksforgeeks.org/c-string-class-and-its-applications/) or [string class](https://www.geeksforgeeks.org/stdstring-class-in-c/) may be used to create an array of mutable strings. In this method, the size of the string is not fixed, and the strings can be changed which somehow makes it dynamic in nature nevertheless **std::string**can be used to create a string array using in-built functions.

**Example:**

* CPP

|  |
| --- |
| // C++ program to demonstrate array of strings using  // array of strings.  #include <iostream>  #include <string>    **int** main()  {      // Initialize String Array      std::string colour[4]          = { "Blue", "Red", "Orange", "Yellow" };        // Print Strings  **for** (**int** i = 0; i < 4; i++)          std::cout << colour[i] << "\n";  } |

**Output:**

Blue

Red

Orange

Yellow

The array is of fixed size, but needn’t be. Again, the 4 here may be omitted, and the compiler will determine the appropriate size of the array. The strings are also mutable, allowing them to be changed.

## **4. Using the vector class**

A [**vector**](https://www.geeksforgeeks.org/vector-in-cpp-stl/) is a dynamic array that doubles its size whenever a new character is added that exceeds its limit. The STL container vector can be used to dynamically allocate an array that can vary in size.

This is only usable in C++, as C does not have classes. Note that the initializer-list syntax here requires a compiler that supports the 2011 C++ standard, and though it is quite likely your compiler does, it is something to be aware of.

* CPP

|  |
| --- |
| // C++ program to demonstrate array of strings using vector  #include <iostream>  #include <string>  #include <vector>    **int** main()  {      // Declaring Vector of String type      // Values can be added here using initializer-list      // syntax      std::vector<std::string> colour{ "Blue", "Red",                                       "Orange" };        // Strings can be added at any time with push\_back      colour.push\_back("Yellow");        // Print Strings stored in Vector  **for** (**int** i = 0; i < colour.size(); i++)          std::cout << colour[i] << "\n";  } |

**Output:**

Blue

Red

Orange

Yellow

* Vectors are dynamic arrays, and allow you to add and remove items at any time.
* Any type or class may be used in vectors, but a given vector can only hold one type.

## **5. Using the Array Class**

An array is a homogeneous mixture of data that is stored continuously in the memory space. The STL [container array](https://www.geeksforgeeks.org/array-class-c/) can be used to allocate a fixed-size array. It may be used very similarly to a vector, but the size is always fixed.

**Example:**

* C++

|  |
| --- |
| // C++ program to demonstrate array of string  // using STL array  #include <array>  #include <iostream>  #include <string>    **int** main()  {      // Initialize array      std::array<std::string, 4> colour{ "Blue", "Red",                                         "Orange", "Yellow" };        // Printing Strings stored in array  **for** (**int** i = 0; i < 4; i++)          std::cout << colour[i] << "\n";    **return** 0;  } |

**Output**

Blue

Red

Orange

Yellow

These are by no means the only ways to make a collection of strings. C++ offers several [container](https://www.geeksforgeeks.org/containers-cpp-stl/) classes, each of which has various tradeoffs and features, and all of them exist to fill requirements that you will have in your projects. Explore and have fun!

**Conclusion:** Out of all the methods, Vector seems to be the best way for creating an array of Strings in C++.

# Tokenizing a string in C++

Tokenizing a string denotes splitting a string with respect to some delimiter(s). There are many ways to tokenize a string. In this article four of them are explained:

### Using stringstream

A **stringstream** associates a string object with a stream allowing you to read from the string as if it were a stream.

Below is the C++ implementation :

* C++

|  |
| --- |
| // Tokenizing a string using stringstream  #include <bits/stdc++.h>    **using** **namespace** std;    **int** main()  {        string line = "GeeksForGeeks is a must try";        // Vector of string to save tokens      vector <string> tokens;        // stringstream class check1      stringstream check1(line);        string intermediate;        // Tokenizing w.r.t. space ' '  **while**(getline(check1, intermediate, ' '))      {          tokens.push\_back(intermediate);      }        // Printing the token vector  **for**(**int** i = 0; i < tokens.size(); i++)          cout << tokens[i] << '\n';  } |

**Output**

GeeksForGeeks

is

a

must

try

### Using **strtok()**

// Splits str[] according to given delimiters.

// and returns next token. It needs to be called

// in a loop to get all tokens. It returns NULL

// when there are no more tokens.

**char \* strtok(char str[], const char \*delims);**

Below is the C++ implementation :

* C++

|  |
| --- |
| // C/C++ program for splitting a string  // using strtok()  #include <stdio.h>  #include <string.h>    **int** main()  {  **char** str[] = "Geeks-for-Geeks";        // Returns first token  **char** \*token = **strtok**(str, "-");        // Keep printing tokens while one of the      // delimiters present in str[].  **while** (token != NULL)      {  **printf**("%s\n", token);          token = **strtok**(NULL, "-");      }    **return** 0;  } |

**Output**

Geeks

for

Geeks

#### **Another Example of strtok() :**

* C

|  |
| --- |
| // C code to demonstrate working of  // strtok  #include <string.h>  #include <stdio.h>    // Driver function  **int** main()  {   // Declaration of string  **char** gfg[100] = " Geeks - for - geeks - Contribute";        // Declaration of delimiter  **const** **char** s[4] = "-";  **char**\* tok;        // Use of strtok      // get first token      tok = **strtok**(gfg, s);        // Checks for delimiter  **while** (tok != 0) {  **printf**(" %s\n", tok);            // Use of strtok          // go through other tokens          tok = **strtok**(0, s);      }    **return** (0);  } |

**Output**

Geeks

for

geeks

Contribute

### Using **strtok\_r()**

Just like strtok() function in C, **strtok\_r()** does the same task of parsing a string into a sequence of tokens. strtok\_r() is a reentrant version of strtok().

There are two ways we can call strtok\_r()

// The third argument saveptr is a pointer to a char \*

// variable that is used internally by strtok\_r() in

// order to maintain context between successive calls

// that parse the same string.

**char \*strtok\_r(char \*str, const char \*delim, char \*\*saveptr);**

Below is a simple C++ program to show the use of strtok\_r() :

* C++

|  |
| --- |
| // C/C++ program to demonstrate working of strtok\_r()  // by splitting string based on space character.  #include<stdio.h>  #include<string.h>    **int** main()  {  **char** str[] = "Geeks for Geeks";  **char** \*token;  **char** \*rest = str;    **while** ((token = strtok\_r(rest, " ", &rest)))  **printf**("%s\n", token);    **return**(0);  } |

**Output**

Geeks

for

Geeks

### Using std::sregex\_token\_iterator

In this method the tokenization is done on the basis of regex matches. Better for use cases when multiple delimiters are needed.

Below is a simple C++ program to show the use of std::sregex\_token\_iterator:

* C++

|  |
| --- |
| // CPP program for above approach  #include <iostream>  #include <regex>  #include <string>  #include <vector>    /\*\*   \* @brief Tokenize the given vector     according to the regex   \* and remove the empty tokens.   \*   \* @param str   \* @param re   \* @return std::vector<std::string>   \*/  std::vector<std::string> tokenize(  **const** std::string str,  **const** std::regex re)  {      std::sregex\_token\_iterator it{ str.begin(),                               str.end(), re, -1 };      std::vector<std::string> tokenized{ it, {} };        // Additional check to remove empty strings      tokenized.erase(          std::remove\_if(tokenized.begin(),                              tokenized.end(),                         [](std::string **const**& s) {  **return** s.size() == 0;                         }),          tokenized.end());    **return** tokenized;  }    // Driver Code  **int** main()  {  **const** std::string str = "Break string                     a,spaces,and,commas";  **const** std::regex re(R"([\s|,]+)");        // Function Call  **const** std::vector<std::string> tokenized =                             tokenize(str, re);    **for** (std::string token : tokenized)          std::cout << token << std::endl;  **return** 0;  } |

**Output**

Break

string

a

spaces

and

commas

**Time Complexity:** **O(n \* d)** where n is the length of string and d is the number of delimiters.  
**Auxiliary Space:** **O(n)**

# strrchr() function in C/C++

**strrchr() function**   
In C++, strrchr() is a predefined function used for string handling. cstring is the header file required for string functions.  
This function returns a pointer to the last occurrence of a character in a string.   
The character whose last occurrence we want to find is passed as the second argument to the function and the string in which we have to find the character is passed as the first argument to the function.   
**Syntax** 

char \*strrchr(const char \*str, int c)

Here, str is the string and c is the character to be located. It is passed as its int promotion, but it is internally converted back to char.   
**Application**   
Given a string in C++, we need to find the last occurrence of a character, let’s say ‘a’.  
Examples: 

Input : string = 'This is a string'

Output :9

Input :string = 'My name is Ayush'

Output :12

**Algorithm**   
1. Pass the given string in the strrchr() function and mention the character you need to point to.   
2. The function returns a value, print the value.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

* CPP

|  |
| --- |
| // C++ program to demonstrate working strchr()  #include <iostream>  #include <cstring>  **using** **namespace** std;    **int** main()  {  **char** str[] = "This is a string";  **char** \* ch = **strrchr**(str,'a');    cout << ch - str + 1;  **return** 0;  } |

Output: 

9

**C Examples :** 

* C

|  |
| --- |
| // C code to demonstrate the working of  // strrchr()    #include <stdio.h>  #include <string.h>    // Driver function  **int** main()  {        // initializing variables  **char** st[] = "GeeksforGeeks";  **char** ch = 'e';  **char**\* val;        // Use of strrchr()      // returns "ks"      val = **strrchr**(st, ch);    **printf**("String after last %c is :  %s \n", ch, val);    **char** ch2 = 'm';        // Use of strrchr()      // returns null      // test for null      val = **strrchr**(st, ch2);    **printf**("String after last %c is :  %s ", ch2, val);    **return** (0);  } |

Output: 

String after last e is : eks

String after last m is : (null)

**Practical Application:**Since it returns the entire string after the last occurrence of a particular character, it can be used to **extract the suffix of a string**. For e.g to know the entire leading zeroes in a denomination when we know the first number. This example is demonstrated below.

|  |
| --- |
| // C code to demonstrate the application of  // strrchr()    #include <stdio.h>  #include <string.h>    // Driver function  **int** main()  {        // initializing the denomination  **char** denom[] = "Rs 10000000";        // Printing original string  **printf**("The original string is : %s", denom);        // initializing the initial number  **char** first = '1';  **char**\* entire;        // Use of strrchr()      // returns entire number      entire = **strrchr**(denom, first);    **printf**("\nThe denomination value is : %s ", entire);    **return** (0);  } |

Output: 

The original string is : Rs 10000000

The denomination value is : 10000000

# stringstream in C++ and its Applications

A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin). To use stringstream, we need to include ***sstream*** header file. The stringstream class is extremely useful in parsing input.

**Basic methods are:**

1. **clear()-**To clear the stream.
2. **str()-**To get and set string object whose content is present in the stream.
3. **operator <<-**Add a string to the stringstream object.
4. **operator >>-**Read something from the stringstream object.

**Examples:**

**1. Count the number of words in a string**

**Examples:**

***Input:****Asipu Pawan Kumar****Output:****3*

***Input:****Geeks For Geeks Ide****Output:****4*

Below is the C++ program to implement the above approach-

* C++

|  |
| --- |
| // C++ program to count words in  // a string using stringstream.  #include <iostream>  #include <sstream>  #include<string>  **using** **namespace** std;    **int** countWords(string str)  {      // Breaking input into word      // using string stream        // Used for breaking words      stringstream s(str);        // To store individual words      string word;    **int** count = 0;  **while** (s >> word)          count++;  **return** count;  }    // Driver code  **int** main()  {      string s = "geeks for geeks geeks "                 "contribution placements";      cout << " Number of words are: " << countWords(s);  **return** 0;  } |

**Output**

Number of words are: 6

**2. Print frequencies of individual words in a string**

**Examples:**

***Input:****Geeks For Geeks Quiz Geeks Quiz Practice Practice****Output:****For -> 1  
             Geeks -> 3  
             Practice -> 2  
             Quiz -> 2*

***Input:****Word String Frequency String****Output:****Frequency -> 1  
              String -> 2  
              Word -> 1*

Below is the C++ program to implement the above approach-

* C++

|  |
| --- |
| // C++ program to demonstrate use  // of stringstream to count  // frequencies of words.  #include <bits/stdc++.h>  #include <iostream>  #include <sstream>  #include<string>  **using** **namespace** std;    **void** printFrequency(string st)  {      // Each word it mapped to      // it's frequency      map<string, **int**>FW;        // Used for breaking words      stringstream ss(st);        // To store individual words      string Word;    **while** (ss >> Word)          FW[Word]++;        map<string, **int**>::iterator m;  **for** (m = FW.begin(); m != FW.end(); m++)          cout << m->first << "-> "               << m->second << "\n";  }    // Driver code  **int** main()  {      string s = "Geeks For Geeks Ide";      printFrequency(s);  **return** 0;  } |

**Output**

For-> 1

Geeks-> 2

Ide-> 1

[Removing spaces from a string using Stringstream](https://www.geeksforgeeks.org/removing-spaces-string-using-stringstream/)  
[Converting Strings to Numbers in C/C++](https://www.geeksforgeeks.org/converting-strings-numbers-cc/)

# Functions in C/C++:

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

The general form of a function is:

# Syntax:

# return\_type function\_name([ arg1\_type arg1\_name, ... ]) { code }

**Example:**  
Below is a simple C/C++ program to demonstrate functions.

* C
* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** max(**int** x, **int** y)  {  **if** (x > y)  **return** x;  **else**  **return** y;  }    **int** main() {  **int** a = 10, b = 20;        // Calling above function to find max of 'a' and 'b'  **int** m = max(a, b);        cout << "m is " << m;  **return** 0;  } |

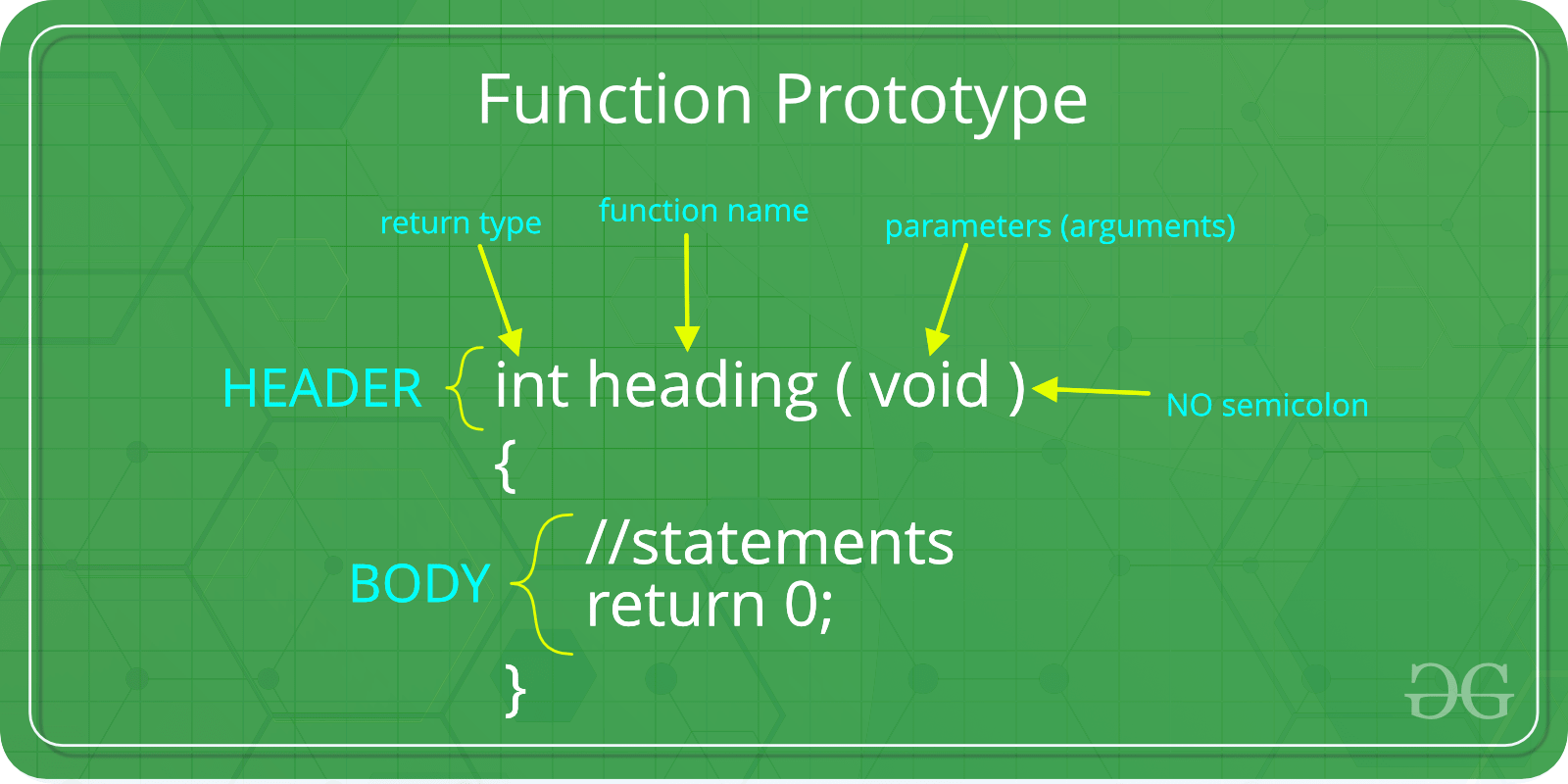
**Output:**

m is 20

**Why do we need functions?**

* Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
* Functions make code modular. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.
* Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

**Function Declaration**  
A function declaration tells the compiler about the number of parameters function takes, data-types of parameters, and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in below declarations)



|  |
| --- |
| // A function that takes two integers as parameters  // and returns an integer  **int** max(**int**, **int**);    // A function that takes an int pointer and an int variable as parameters  // and returns a pointer of type int  **int** \*swap(**int**\*,**int**);    // A function that takes a charas parameters  // and returns an reference variable  **char** \*call(**char** b);    // A function that takes a char and an int as parameters  // and returns an integer  **int** fun(**char**, **int**); |

It is always recommended to declare a function before it is used (See [this](https://www.geeksforgeeks.org/what-is-the-purpose-of-a-function-prototype/), [this](https://www.geeksforgeeks.org/g-fact-95/)and [this](https://www.geeksforgeeks.org/importance-of-function-prototype-in-c/)for details)

In C, we can do both declaration and definition at the same place, like done in the above example program.

C also allows to declare and define functions separately, this is especially needed in the case of library functions. The library functions are declared in header files and defined in library files. Below is an example declaration.

**Parameter Passing to functions**  
The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.  
The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.  
There are two most popular ways to pass parameters.

***Pass by Value:*** In this parameter passing method, values of actual parameters are copied to function’s formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

***Pass by Reference*** Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

* C
* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **void** fun(**int** x) {      x = 30;  }    **int** main() {  **int** x = 20;      fun(x);      cout << "x = " << x;  **return** 0;  } |

**Output:**

x = 20

However, in C, we can use pointers to get the effect of pass-by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator \* is used to access the value at an address. In the statement ‘\*ptr = 30’, value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement ‘fun(&x)’, the address of x is passed so that x can be modified using its address.

* C
* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **void** fun(**int** \*ptr)  {      \*ptr = 30;  }    **int** main() {  **int** x = 20;      fun(&x);      cout << "x = " << x;    **return** 0;  } |

Output:

x = 30

**Following are some important points about functions in C.**  
**1)**Every C program has a function called main() that is called by operating system when a user runs the program.

**2)** Every function has a return type. If a function doesn’t return any value, then void is used as a return type. Moreover, if the return type of the function is void, we still can use return statement in the body of function definition by not specifying any constant, variable, etc. with it, by only mentioning the ‘return;’ statement which would symbolize the termination of the function as shown below:

|  |
| --- |
| **void** function name(**int** a)  {  .......  //Function Body  **return**;  //Function execution would get terminated  } |

**3)** In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.  
**4)** Empty parameter list in C means that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use “void fun(void)”.  
As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.  
**5)**If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:  
int function name();  
And in that case, if the return type of that function is different than INT, compiler would show an error.

**Main Function:**  
The main function is a special function. Every C++ program must contain a function named main. It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.

**Types of main Function:**

1) The first type is – main function without parameters :

|  |
| --- |
| // Without Parameters  **int** main()  {     ...  **return** 0;  } |

2) Second type is main function with parameters :

|  |
| --- |
| // With Parameters  **int** main(**int** argc, **char** \* **const** argv[])  {     ...  **return** 0;  } |

The reason for having the parameter option for the main function is to allow input from the command line.

When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named argv.

Since the main function has the return type of int, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program’s execution was. Returning 0 signals that there were no problems.

**More on Functions in C/C++:**

* [Quiz on function in C](https://www.geeksforgeeks.org/c-language-2-gq/functions-gq/)
* [Importance of function prototype in C](https://www.geeksforgeeks.org/importance-of-function-prototype-in-c/)
* [Functions that are executed before and after main() in C](https://www.geeksforgeeks.org/functions-that-are-executed-before-and-after-main-in-c/)
* [return statement vs exit() in main()](https://www.geeksforgeeks.org/return-statement-vs-exit-in-main/)
* [How to Count Variable Numbers of Arguments in C?,](https://www.geeksforgeeks.org/how-to-count-variable-numbers-of-arguments-in-c/)
* [What is evaluation order of function parameters in C?](https://www.geeksforgeeks.org/g-fact-20/)
* [Does C support function overloading?](https://www.geeksforgeeks.org/does-c-support-function-overloading/)
* [How can we return multiple values from a function?](https://www.geeksforgeeks.org/how-can-i-return-multiple-values-from-a-function/)
* [What is the purpose of a function prototype?](https://www.geeksforgeeks.org/what-is-the-purpose-of-a-function-prototype/)
* [Static functions in C](https://www.geeksforgeeks.org/what-are-static-functions-in-c/)
* [exit(), abort() and assert()](https://www.geeksforgeeks.org/understanding-exit-abort-and-assert/)
* [Implicit return type int in C](https://www.geeksforgeeks.org/implicit-return-type-int-c-language/)
* [What happens when a function is called before its declaration in C?](https://www.geeksforgeeks.org/g-fact-95/)

# Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn’t provide a value for the argument. In case any value is passed, the default value is overridden.

**1)**The following is a simple C++ example to demonstrate the use of default arguments. Here, we don’t have to write 3 sum functions; only one function works by using the default values for 3rd and 4th arguments.

* CPP

|  |
| --- |
| // CPP Program to demonstrate Default Arguments  #include <iostream>  **using** **namespace** std;    // A function with default arguments,  // it can be called with  // 2 arguments or 3 arguments or 4 arguments.  **int** sum(**int** x, **int** y, **int** z = 0, **int** w = 0) //assigning default values to z,w as 0  {  **return** (x + y + z + w);  }    // Driver Code  **int** main()  {      // Statement 1      cout << sum(10, 15) << endl;        // Statement 2      cout << sum(10, 15, 25) << endl;        // Statement 3      cout << sum(10, 15, 25, 30) << endl;  **return** 0;  } |

**Output**

25

50

80

**Explanation:** In statement 1, only two values are passed, hence the variables z and w take the default values of 0. In statement 2, three values are passed, so the value of z is overridden with 25. In statement 3, four values are passed, so the values of z and w are overridden with 25 and 30 respectively.

**2)**If function overloading is done containing the default arguments, then we need to make sure it is not ambiguous to the compiler, otherwise it will throw an error. The following is the modified version of the above program:

* CPP

|  |
| --- |
| // CPP Program to demonstrate Function overloading in  // Default Arguments  #include <iostream>  **using** **namespace** std;    // A function with default arguments, it can be called with  // 2 arguments or 3 arguments or 4 arguments.  **int** sum(**int** x, **int** y, **int** z = 0, **int** w = 0)  {  **return** (x + y + z + w);  }  **int** sum(**int** x, **int** y, **float** z = 0, **float** w = 0)  {  **return** (x + y + z + w);  }  // Driver Code  **int** main()  {      cout << sum(10, 15) << endl;      cout << sum(10, 15, 25) << endl;      cout << sum(10, 15, 25, 30) << endl;  **return** 0;  } |

**Error:**

prog.cpp: In function 'int main()':

prog.cpp:17:20: error: call of overloaded

'sum(int, int)' is ambiguous

cout << sum(10, 15) << endl;

^

prog.cpp:6:5: note: candidate:

int sum(int, int, int, int)

int sum(int x, int y, int z=0, int w=0)

^

prog.cpp:10:5: note: candidate:

int sum(int, int, float, float)

int sum(int x, int y, float z=0, float w=0)

^

**3)**A constructor can contain default parameters as well. A default constructor can either have no parameters or parameters with default arguments.

* C++

|  |
| --- |
| // CPP code to demonstrate use of default arguments in  // Constructors    #include <iostream>  **using** **namespace** std;  **class** A {  **public**:  **int** sum = 0;      A(); // default constructor with no argument      A(**int** x = 0); // default constructor with one                          // arguments    }; |

**Explanation**: Here, we see a default constructor with no arguments and a default constructor with one default argument. The default constructor with argument has a default parameter x, which has been assigned a value of 0.

**Key Points:**

* Default arguments are different from constant arguments as constant arguments can’t be changed whereas default arguments can be overwritten if required.
* Default arguments are overwritten when the calling function provides values for them. For example, calling the function sum(10, 15, 25, 30) overwrites the values of z and w to 25 and 30 respectively.
* When a function is called, the arguments are copied from the calling function to the called function in the order left to right. Therefore, sum(10, 15, 25) will assign 10, 15, and 25 to x, y, and z respectively, which means that only the default value of w is used.
* Once a default value is used for an argument in the function definition, all subsequent arguments to it must have a default value as well. It can also be stated that the default arguments are assigned from right to left. For example, the following function definition is invalid as the subsequent argument of the default variable z is not default.

// Invalid because z has default value, but w after it doesn't have a default value

int sum(int x, int y, int z = 0, int w).

**Advantages of Default Arguments:**

* Default arguments are useful when we want to increase the capabilities of an existing function as we can do it just by adding another default argument to the function.
* It helps in reducing the size of a program.
* It provides a simple and effective programming approach.
* Default arguments improve the consistency of a program.

**Disadvantages of Default Arguments:**

* It increases the execution time as the compiler needs to replace the omitted arguments by their default values in the function call.

# Inline Functions in C++

Inline function is one of the important feature of C++. So, let’s first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because execution time of small function is less than the switching time

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

inline return-type function-name(parameters)

{

// function code

}

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:  
1) If a function contains a loop. (for, while, do-while)  
2) If a function contains static variables.  
3) If a function is recursive.  
4) If a function return type is other than void, and the return statement doesn’t exist in function body.  
5) If a function contains switch or goto statement.

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

**Inline function disadvantages:**  
1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  **inline** **int** cube(**int** s)  {  **return** s\*s\*s;  }  **int** main()  {      cout << "The cube of 3 is: " << cube(3) << "\n";  **return** 0;  } //Output: The cube of 3 is: 27 |

**Inline function and classes:**  
It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.  
For example:

|  |
| --- |
| **class** S  {  **public**:  **inline** **int** square(**int** s) // redundant use of inline      {          // this function is automatically inline          // function body      }  }; |

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.  
For example:

|  |
| --- |
| **class** S  {  **public**:  **int** square(**int** s); // declare the function  };    **inline** **int** S::square(**int** s) // use inline prefix  {    } |

The following program demonstrates this concept:

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  **class** operation  {  **int** a,b,add,sub,mul;  **float** **div**;  **public**:  **void** get();  **void** sum();  **void** difference();  **void** product();  **void** division();  };  **inline** **void** operation :: get()  {      cout << "Enter first value:";      cin >> a;      cout << "Enter second value:";      cin >> b;  }    **inline** **void** operation :: sum()  {      add = a+b;      cout << "Addition of two numbers: " << a+b << "\n";  }    **inline** **void** operation :: difference()  {      sub = a-b;      cout << "Difference of two numbers: " << a-b << "\n";  }    **inline** **void** operation :: product()  {      mul = a\*b;      cout << "Product of two numbers: " << a\*b << "\n";  }    **inline** **void** operation ::division()  {  **div**=a/b;      cout<<"Division of two numbers: "<<a/b<<"\n" ;  }    **int** main()  {      cout << "Program using inline function\n";      operation s;      s.get();      s.sum();      s.difference();      s.product();      s.division();  **return** 0;  } |

Output:

Enter first value: 45

Enter second value: 15

Addition of two numbers: 60

Difference of two numbers: 30

Product of two numbers: 675

Division of two numbers: 3

**What is wrong with macro?**

Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.  
Example:

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  **class** S  {  **int** m;  **public**:  #define MAC(S::m)    // error  }; |

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this. One other thing is that the macros are managed by preprocessor and inline functions are managed by C++ compiler.

Remember: It is true that all the functions defined inside the class are implicitly inline and C++ compiler will perform inline call of these functions, but C++ compiler cannot perform inlining if the function is virtual. The reason is call to a virtual function is resolved at runtime instead of compile time. Virtual means wait until runtime and inline means during compilation, if the compiler doesn’t know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time. An example where inline function has no effect at all:

|  |
| --- |
| **inline** **void** show()  {      cout << "value of S = " << S << endl;  } |

The above function relatively takes a long time to execute. In general function which performs input output (I/O) operation shouldn’t be defined as inline because it spends a considerable amount of time. Technically inlining of show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline. Programming languages like Java & C# doesn’t support inline functions.  
But in Java, the compiler can perform inlining when the small final method is called, because final methods can’t be overridden by sub classes and call to a final method is resolved at compile time. In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Last thing to keep in mind that inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can’t provide better result. In other words don’t expect better performance of program. Don’t make every function inline. It is better to keep inline functions as small as possible.

# Return From Void Functions in C++

Void functions are known as**Non-Value Returning functions**. They are “void” due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. **Void functions do not have a return type, but they can do return values.** Some of the cases are listed below:  
   
**1) A Void Function Can Return:**We can simply write a return statement in a void fun(). In fact, it is considered a good practice (for readability of code) to write a return; statement to indicate the end of the function.

* CPP

|  |
| --- |
| // CPP Program to demonstrate void functions  #include <iostream>  **using** **namespace** std;    **void** fun()  {      cout << "Hello";        // We can write return in void  **return**;  }    // Driver Code  **int** main()  {      fun();  **return** 0;  } |

**Output**

Hello

**2) A void fun() can return another void function:** A void function can also call another void function while it is terminating. For example,

* CPP

|  |
| --- |
| // C++ code to demonstrate void()  // returning void()  #include <iostream>  **using** **namespace** std;    // A sample void function  **void** work()  {      cout << "The void function has returned "              " a void() !!! \n";  }    // Driver void() returning void work()  **void** test()  {      // Returning void function  **return** work();  }    // Driver Code  **int** main()  {      // Calling void function      test();  **return** 0;  } |

**Output**

The void function has returned a void() !!!

The above code explains how void() can actually be useful to return void functions without giving errors.  
   
**3) A void() can return a void value:**A void() cannot return a value that can be used. But it can return a value that is void without giving an error. For example,

* CPP

|  |
| --- |
| // C++ code to demonstrate void()  // returning a void value  #include <iostream>  **using** **namespace** std;    // Driver void() returning a void value  **void** test()  {      cout << "Hello";        // Returning a void value  **return** (**void**)"Doesn't Print";  }    // Driver Code  **int** main()  {      test();  **return** 0;  } |

**Output**

Hello

# Functors in C++

Please note that the title is **Functors** (Not Functions)!!

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

**Functors** are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

Below program uses [transform() in STL](https://www.geeksforgeeks.org/transform-c-stl-perform-operation-elements/) to add 1 to all elements of arr[].

|  |
| --- |
| // A C++ program uses transform() in STL to add  // 1 to all elements of arr[]  #include <bits/stdc++.h>  **using** **namespace** std;    **int** increment(**int** x) {  **return** (x+1); }    **int** main()  {  **int** arr[] = {1, 2, 3, 4, 5};  **int** n = **sizeof**(arr)/**sizeof**(arr[0]);        // Apply increment to all elements of      // arr[] and store the modified elements      // back in arr[]      transform(arr, arr+n, arr, increment);    **for** (**int** i=0; i<n; i++)          cout << arr[i] <<" ";    **return** 0;  } |

Output:

2 3 4 5 6

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what’s happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the operator().

**The line,**

MyFunctor(10);

**Is same as**

MyFunctor.operator()(10);

Let’s delve deeper and understand how this can actually be used in conjunction with STLs.

|  |
| --- |
| // C++ program to demonstrate working of  // functors.  #include <bits/stdc++.h>  **using** **namespace** std;    // A Functor  **class** increment  {  **private**:  **int** num;  **public**:      increment(**int** n) : num(n) {  }        // This operator overloading enables calling      // operator function () on objects of increment  **int** operator () (**int** arr\_num) **const** {  **return** num + arr\_num;      }  };    // Driver code  **int** main()  {  **int** arr[] = {1, 2, 3, 4, 5};  **int** n = **sizeof**(arr)/**sizeof**(arr[0]);  **int** to\_add = 5;        transform(arr, arr+n, arr, increment(to\_add));    **for** (**int** i=0; i<n; i++)          cout << arr[i] << " ";  } |

Output:

6 7 8 9 10

Thus, here, Increment is a functor, a c++ class that acts as a function.

**The line,**

transform(arr, arr+n, arr, increment(to\_add));

**is the same as writing below two lines,**

// Creating object of increment

increment obj(to\_add);

// Calling () on object

transform(arr, arr+n, arr, obj);

Thus, an object a is created that overloads the operator(). Hence, functors can be used effectively in conjunction with C++ STLs.

# Pointers in C and C++ | Set 1 (Introduction, Arithmetic and Array)

// General syntax

**datatype \*var\_name;**

// An example pointer "ptr" that holds

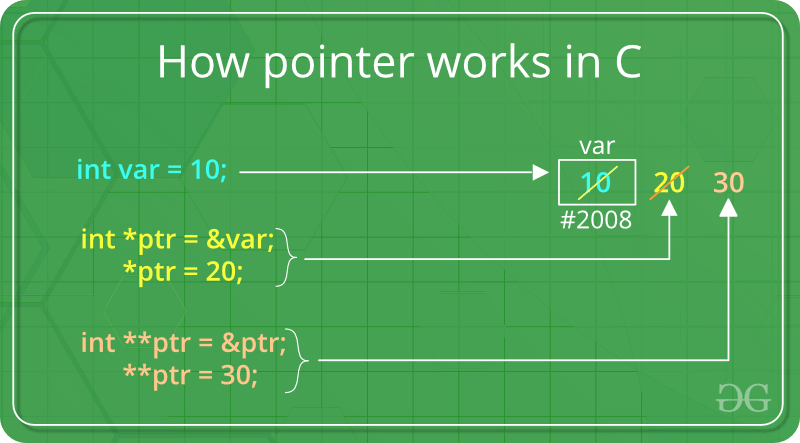
// address of an integer variable or holds

// address of a memory whose value(s) can

// be accessed as integer values through "ptr"

**int \*ptr;**

**Using a Pointer:**



To use pointers in C, we must understand below two operators. 

* To access address of a variable to a pointer, we use the unary operator **&**(ampersand) that returns the address of that variable. For example &x gives us address of variable x.
* C

|  |
| --- |
| // The output of this program can be different  // in different runs. Note that the program  // prints address of a variable and a variable  // can be assigned different address in different  // runs.  #include <stdio.h>    **int** main()  {  **int** x;        // Prints address of x  **printf**("%p", &x);    **return** 0;  } |

* One more operator is **unary \*** (Asterisk) which is used for two things :
  + To declare a pointer variable: When a pointer variable is declared in C/C++, there must be a \* before its name.
* C

|  |
| --- |
| // C program to demonstrate declaration of  // pointer variables.  #include <stdio.h>  **int** main()  {  **int** x = 10;        // 1) Since there is \* in declaration, ptr      // becomes a pointer variable (a variable      // that stores address of another variable)      // 2) Since there is int before \*, ptr is      // pointer to an integer type variable  **int** \*ptr;        // & operator before x is used to get address      // of x. The address of x is assigned to ptr.      ptr = &x;    **return** 0;  } |

* To access the value stored in the address we use the unary operator (\*) that returns the value of the variable located at the address specified by its operand. This is also called **Dereferencing**.
* C++
* C

|  |
| --- |
| // C++ program to demonstrate use of \* for pointers in C++  #include <iostream>  **using** **namespace** std;    **int** main()  {      // A normal integer variable  **int** Var = 10;        // A pointer variable that holds address of var.  **int** \*ptr = &Var;        // This line prints value at address stored in ptr.      // Value stored is value of variable "var"      cout << "Value of Var = "<< \*ptr << endl;        // The output of this line may be different in different      // runs even on same machine.      cout << "Address of Var = " <<  ptr << endl;        // We can also use ptr as lvalue (Left hand      // side of assignment)      \*ptr = 20; // Value at address is now 20        // This prints 20      cout << "After doing \*ptr = 20, \*ptr is "<< \*ptr << endl;    **return** 0;  }    // This code is contributed by  // shubhamsingh10 |

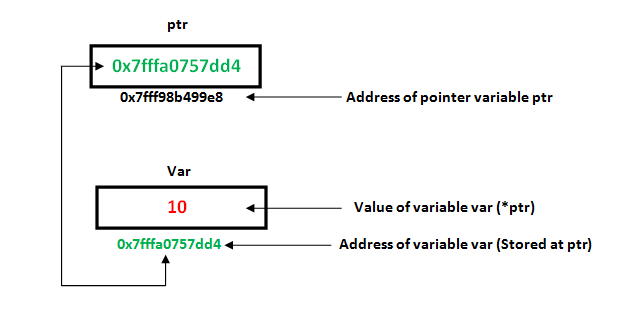
* **Output :**

Value of Var = 10

Address of Var = 0x7fffa057dd4

After doing \*ptr = 20, \*ptr is 20

* Below is pictorial representation of above program:



**Pointer Expressions and Pointer Arithmetic**   
A limited set of arithmetic operations can be performed on pointers. A pointer may be: 

* incremented ( ++ )
* decremented ( — )
* an integer may be added to a pointer ( + or += )
* an integer may be subtracted from a pointer ( – or -= )

Pointer arithmetic is meaningless unless performed on an array.   
Note : Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses.

|  |
| --- |
| // C++ program to illustrate Pointer Arithmetic  // in C/C++  #include <bits/stdc++.h>    // Driver program  **int** main()  {      // Declare an array  **int** v[3] = {10, 100, 200};        // Declare pointer variable  **int** \*ptr;        // Assign the address of v[0] to ptr      ptr = v;    **for** (**int** i = 0; i < 3; i++)      {  **printf**("Value of \*ptr = %d\n", \*ptr);  **printf**("Value of ptr = %p\n\n", ptr);            // Increment pointer ptr by 1          ptr++;      }  } |

Output:Value of \*ptr = 10

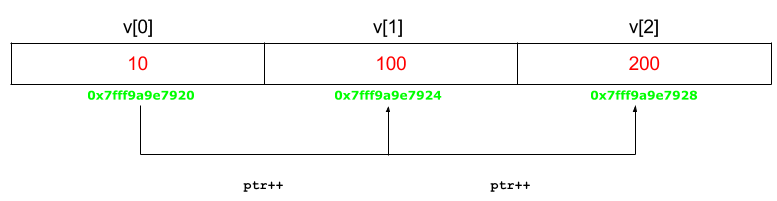
Value of ptr = 0x7ffcae30c710

Value of \*ptr = 100

Value of ptr = 0x7ffcae30c714

Value of \*ptr = 200

Value of ptr = 0x7ffcae30c718

[](https://media.geeksforgeeks.org/wp-content/uploads/Untitled-presentation-31.png)

**Array Name as Pointers**  
An array name acts like a pointer constant. The value of this pointer constant is the address of the first element.   
For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably. 

|  |
| --- |
| // C++ program to illustrate Array Name as Pointers in C++  #include <bits/stdc++.h>  **using** **namespace** std;    **void** geeks()  {      // Declare an array  **int** val[3] = { 5, 10, 15};        // Declare pointer variable  **int** \*ptr;        // Assign address of val[0] to ptr.      // We can use ptr=&val[0];(both are same)      ptr = val ;      cout << "Elements of the array are: ";      cout << ptr[0] << " " << ptr[1] << " " << ptr[2];    **return**;  }    // Driver program  **int** main()  {      geeks();  **return** 0;  } |

Output:

Elements of the array are: 5 10 15

Untitled presentation (2)

Now if this ptr is sent to a function as an argument then the array val can be accessed in a similar fashion.   
   
**Pointers and Multidimensional Arrays**   
Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration 

int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };

**In general, nums[i][j] is equivalent to \*(\*(nums+i)+j)**

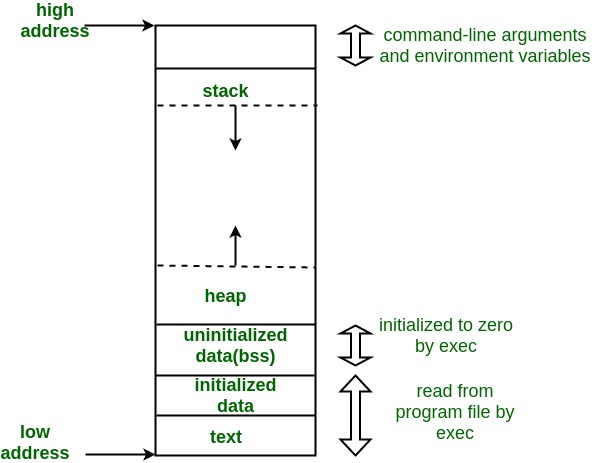
| Pointer Notation | Array Notation | Value |
| --- | --- | --- |
| \*(\*nums) | nums[0][0] | 16 |
| \*(\*nums + 1) | nums[0][1] | 18 |
| \*(\*nums + 2) | nums[0][2] | 20 |
| \*(\*(nums + 1)) | nums[1][0] | 25 |
| \*(\*(nums + 1) + 1) | nums[1][1] | 26 |
| \*(\*(nums + 1) + 2) | nums[1][2] | 27 |

**Related Articles**:

[**Applications of pointers in C/C++.**](https://www.geeksforgeeks.org/applications-of-pointers-in-c-cpp/)

# Memory Layout of C Programs

A typical memory representation of a C program consists of the following sections.  
1. Text segment  (i.e. instructions)  
2. Initialized data segment   
3. Uninitialized data segment  (bss)  
4. Heap   
5. Stack



A typical memory layout of a running process  
**1. Text Segment:**   
A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.  
As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it. 

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

**2. Initialized Data Segment:**   
Initialized data segment, usually called simply the Data Segment. A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.  
Note that, the data segment is not read-only, since the values of the variables can be altered at run time.  
This segment can be further classified into the initialized read-only area and the initialized read-write area.  
For instance, the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in the initialized read-write area. And a global C statement like const char\* string = “hello world” makes the string literal “hello world” to be stored in the initialized read-only area and the character pointer variable string in the initialized read-write area.  
Ex: static int i = 10 will be stored in the data segment and global int i = 10 will also be stored in data segment

**3. Uninitialized Data Segment:**   
Uninitialized data segment often called the “**bss**” segment, named after an ancient assembler operator that stood for “**block started by symbol**.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing  
uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.  
For instance, a variable declared static int i; would be contained in the BSS segment.   
For instance, a global variable declared int j; would be contained in the BSS segment.

**4. Stack:**   
The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow in opposite directions.)  
The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture, it grows toward address zero; on some other architectures, it grows in the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.  
Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

**5. Heap:**   
Heap is the segment where dynamic memory allocation usually takes place.  
The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

Examples.  
The size(1) command reports the sizes (in bytes) of the text, data, and bss segments. ( for more details please refer man page of size(1) )

1. Check the following simple C program

* C

|  |
| --- |
| #include <stdio.h>    **int** main(**void**)  {  **return** 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 8 1216 4c0 memory-layout

# Applications of Pointers in C/C++

2. Let us add one global variable in the program, now check the size of bss (highlighted in red color).

* C

|  |
| --- |
| #include <stdio.h>    **int** global; /\* Uninitialized variable stored in bss\*/    **int** main(**void**)  {  **return** 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **12** 1220 4c4 memory-layout

3. Let us add one static variable which is also stored in bss.

* C

|  |
| --- |
| #include <stdio.h>    **int** global; /\* Uninitialized variable stored in bss\*/    **int** main(**void**)  {  **static** **int** i; /\* Uninitialized static variable stored in bss \*/  **return** 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 248 **16** 1224 4c8 memory-layout

4. Let us initialize the static variable which will then be stored in the Data Segment (DS)

* C

|  |
| --- |
| #include <stdio.h>    **int** global; /\* Uninitialized variable stored in bss\*/    **int** main(**void**)  {  **static** **int** i = 100; /\* Initialized static variable stored in DS\*/  **return** 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **252 12** 1224 4c8 memory-layout

5. Let us initialize the global variable which will then be stored in the Data Segment (DS)

* C

|  |
| --- |
| #include <stdio.h>    **int** global = 10; /\* initialized global variable stored in DS\*/    **int** main(**void**)  {  **static** **int** i = 100; /\* Initialized static variable stored in DS\*/  **return** 0;  } |

[narendra@CentOS]$ gcc memory-layout.c -o memory-layout

[narendra@CentOS]$ size memory-layout

text data bss dec hex filename

960 **256 8** 1224 4c8 memory-layout

* **To pass arguments by reference**. Passing by reference serves two purposes

(i) **To modify variable of function in other.** Example to swap two variables;

* C
* C++

|  |
| --- |
| // C program to demonstrate that we can change  // local values of one function in another using pointers.    #include <stdio.h>    **void** swap(**int**\* x, **int**\* y)  {  **int** temp = \*x;      \*x = \*y;      \*y = temp;  }    **int** main()  {  **int** x = 10, y = 20;      swap(&x, &y);  **printf**("%d %d\n", x, y);  **return** 0;  } |

Output :

20 10

(ii) **For efficiency purpose**. Example passing large structure without reference would create a copy of the structure (hence wastage of space).   
Note : The above two can also be achieved through [References in C++](https://www.geeksforgeeks.org/references-in-c/). 

* **For accessing array elements.** Compiler internally uses pointers to access array elements.
* C
* C++

|  |
| --- |
| // C program to demonstrate that compiler  // internally uses pointer arithmetic to access  // array elements.    #include <stdio.h>    **int** main()  {  **int** arr[] = { 100, 200, 300, 400 };        // Compiler converts below to \*(arr + 2).  **printf**("%d ", arr[2]);        // So below also works.  **printf**("%d\n", \*(arr + 2));    **return** 0;  } |

Output :

300 300

* **To return multiple values.** Example returning square and square root of numbers.
* C
* C++

|  |
| --- |
| // C program to demonstrate that using a pointer  // we can return multiple values.    #include <math.h>  #include <stdio.h>    **void** fun(**int** n, **int**\* square, **double**\* sq\_root)  {      \*square = n \* n;      \*sq\_root = **sqrt**(n);  }    **int** main()  {    **int** n = 100;  **int** sq;  **double** sq\_root;      fun(n, &sq, &sq\_root);    **printf**("%d %f\n", sq, sq\_root);  **return** 0;  } |

Output :

10000 10

* [**Dynamic memory allocation**](https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/): We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, it is not deleted until we explicitly delete it.
* C
* C++

|  |
| --- |
| // C program to dynamically allocate an  // array of given size.    #include <stdio.h>  #include <stdlib.h>  **int**\* createArr(**int** n)  {  **int**\* arr = (**int**\*)(**malloc**(n \* **sizeof**(**int**)));  **return** arr;  }  **int** main()  {  **int**\* pt = createArr(10);  **return** 0;  } |

**Some Questions Regarding Pointers:**

1. *What are the uses of a pointer?*  
   Ans. Pointer is used in the following cases  
                i) It is used to access array elements  
               ii) It is used for dynamic memory allocation.  
              iii) It is used in Call by reference  
              iv) It is used in data structures like trees, graph, linked list etc.
2. *Are pointers integer?*  
   Ans. No, pointers are not integers. A pointer is an address and a positive number.
3. *What does the error ‘Null Pointer Assignment’ means and what causes this error?*  
   Ans. As null pointer points to nothing so accessing a uninitialized pointer or invalid location may cause an error.
4. *How pointer variables are initialized?*  
   Ans. Pointer variables are initialized by one of the following ways.  
                I. Static memory allocation  
               II. Dynamic memory allocation
5. *What is pointer to a pointer?*  
   Ans. If a pointer variable points another pointer value. Such a situation is known as a pointer to a  
   pointer.  
   Example:  
      int \*p1,\*\*p2,v=10;  
      P1=&v; p2=&p1;  
      Here p2 is a pointer to a pointer
6. *What is an array of pointers?*  
   Ans: If the elements of an array are addresses, such an array is called an array of pointers.

* **To implement data structures.**   
  Example [linked list](https://www.geeksforgeeks.org/data-structures/linked-list/), [tree](https://www.geeksforgeeks.org/binary-tree-data-structure/), etc. We cannot use [C++ references](https://www.geeksforgeeks.org/references-in-c/) to implement these data structures because references are fixed to a location (For example, we can not traverse a linked list using references)
* **To do system level programming where memory addresses are useful**. For example shared memory used by multiple threads. For more examples, see [IPC through shared memory](https://www.geeksforgeeks.org/ipc-shared-memory/), [Socket Programming in C/C++](https://www.geeksforgeeks.org/socket-programming-in-cc-handling-multiple-clients-on-server-without-multi-threading/), etc

# Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

* If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
* Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.  
Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.  
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Let’s look at each of them in greater detail.

### C malloc() method

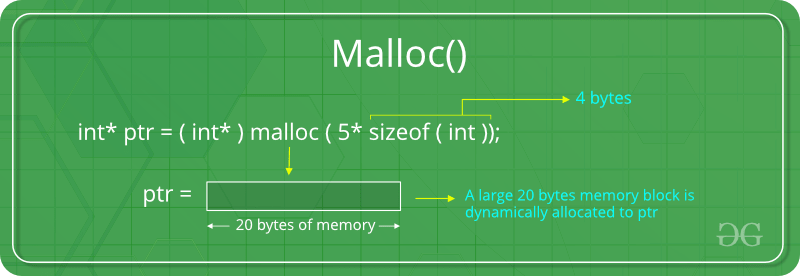
The **“malloc”** or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

**Syntax:**

ptr = (cast-type\*) malloc(byte-size)

**For Example:**

***ptr = (int\*) malloc(100 \* sizeof(int));*** *Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.*



If space is insufficient, allocation fails and returns a NULL pointer.

**Example:**

* C

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    **int** main()  {        // This pointer will hold the      // base address of the block created  **int**\* ptr;  **int** n, i;        // Get the number of elements for the array  **printf**("Enter number of elements:");  **scanf**("%d",&n);  **printf**("Entered number of elements: %d\n", n);        // Dynamically allocate memory using malloc()      ptr = (**int**\*)**malloc**(n \* **sizeof**(**int**));        // Check if the memory has been successfully      // allocated by malloc or not  **if** (ptr == NULL) {  **printf**("Memory not allocated.\n");  **exit**(0);      }  **else** {            // Memory has been successfully allocated  **printf**("Memory successfully allocated using malloc.\n");            // Get the elements of the array  **for** (i = 0; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array  **printf**("The elements of the array are: ");  **for** (i = 0; i < n; ++i) {  **printf**("%d, ", ptr[i]);          }      }    **return** 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,

### C calloc() method

1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

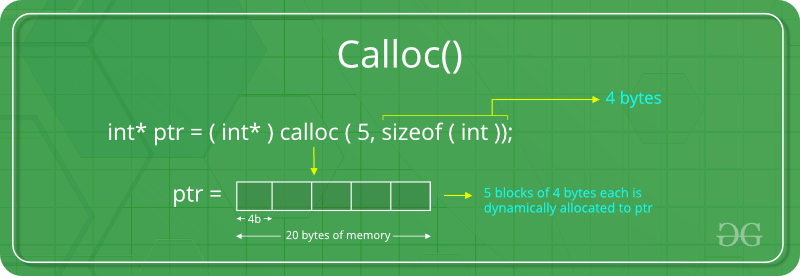
**Syntax:**

ptr = (cast-type\*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.

**For Example:**

***ptr = (float\*) calloc(25, sizeof(float));*** *This statement allocates contiguous space in memory for 25 elements each with the size of the float.*



If space is insufficient, allocation fails and returns a NULL pointer.

**Example:**

* C

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    **int** main()  {        // This pointer will hold the      // base address of the block created  **int**\* ptr;  **int** n, i;        // Get the number of elements for the array      n = 5;  **printf**("Enter number of elements: %d\n", n);        // Dynamically allocate memory using calloc()      ptr = (**int**\*)**calloc**(n, **sizeof**(**int**));        // Check if the memory has been successfully      // allocated by calloc or not  **if** (ptr == NULL) {  **printf**("Memory not allocated.\n");  **exit**(0);      }  **else** {            // Memory has been successfully allocated  **printf**("Memory successfully allocated using calloc.\n");            // Get the elements of the array  **for** (i = 0; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array  **printf**("The elements of the array are: ");  **for** (i = 0; i < n; ++i) {  **printf**("%d, ", ptr[i]);          }      }    **return** 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using calloc.

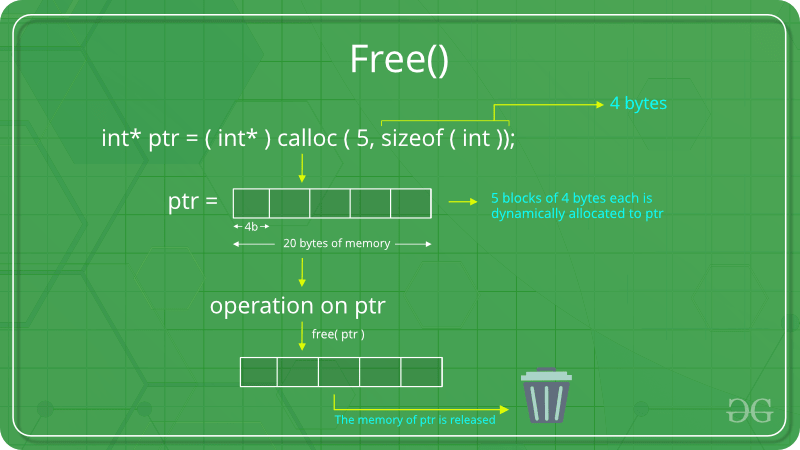
The elements of the array are: 1, 2, 3, 4, 5

### C free() method

**“free”** method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

**Syntax:**

free(ptr);



**Example:**

* C

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    **int** main()  {        // This pointer will hold the      // base address of the block created  **int** \*ptr, \*ptr1;  **int** n, i;        // Get the number of elements for the array      n = 5;  **printf**("Enter number of elements: %d\n", n);        // Dynamically allocate memory using malloc()      ptr = (**int**\*)**malloc**(n \* **sizeof**(**int**));        // Dynamically allocate memory using calloc()      ptr1 = (**int**\*)**calloc**(n, **sizeof**(**int**));        // Check if the memory has been successfully      // allocated by malloc or not  **if** (ptr == NULL || ptr1 == NULL) {  **printf**("Memory not allocated.\n");  **exit**(0);      }  **else** {            // Memory has been successfully allocated  **printf**("Memory successfully allocated using malloc.\n");            // Free the memory  **free**(ptr);  **printf**("Malloc Memory successfully freed.\n");            // Memory has been successfully allocated  **printf**("\nMemory successfully allocated using calloc.\n");            // Free the memory  **free**(ptr1);  **printf**("Calloc Memory successfully freed.\n");      }    **return** 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.

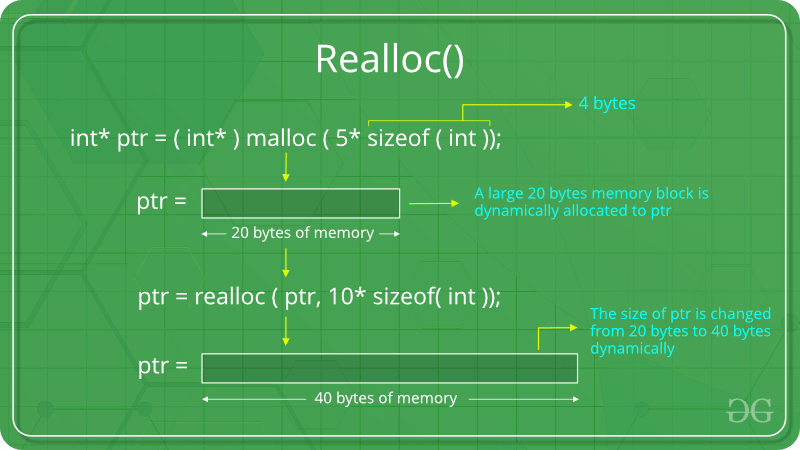
### C realloc() method

**“realloc”** or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

**Syntax:**

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.



If space is insufficient, allocation fails and returns a NULL pointer.

**Example:**

* C

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>    **int** main()  {        // This pointer will hold the      // base address of the block created  **int**\* ptr;  **int** n, i;        // Get the number of elements for the array      n = 5;  **printf**("Enter number of elements: %d\n", n);        // Dynamically allocate memory using calloc()      ptr = (**int**\*)**calloc**(n, **sizeof**(**int**));        // Check if the memory has been successfully      // allocated by malloc or not  **if** (ptr == NULL) {  **printf**("Memory not allocated.\n");  **exit**(0);      }  **else** {            // Memory has been successfully allocated  **printf**("Memory successfully allocated using calloc.\n");            // Get the elements of the array  **for** (i = 0; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array  **printf**("The elements of the array are: ");  **for** (i = 0; i < n; ++i) {  **printf**("%d, ", ptr[i]);          }            // Get the new size for the array          n = 10;  **printf**("\n\nEnter the new size of the array: %d\n", n);            // Dynamically re-allocate memory using realloc()          ptr = **realloc**(ptr, n \* **sizeof**(**int**));            // Memory has been successfully allocated  **printf**("Memory successfully re-allocated using realloc.\n");            // Get the new elements of the array  **for** (i = 5; i < n; ++i) {              ptr[i] = i + 1;          }            // Print the elements of the array  **printf**("The elements of the array are: ");  **for** (i = 0; i < n; ++i) {  **printf**("%d, ", ptr[i]);          }    **free**(ptr);      }    **return** 0;  } |

**Output:**

Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

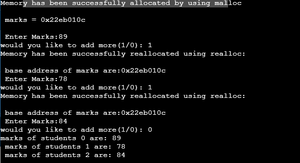
The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

One another example for realloc() method is:

* C

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  **int** main()  {  **int** index = 0, i = 0, n,          \*marks; // this marks pointer hold the base address                  // of  the block created  **int** ans;      marks = (**int**\*)**malloc**(**sizeof**(  **int**)); // dynamically allocate memory using malloc      // check if the memory is successfully allocated by      // malloc or not?  **if** (marks == NULL) {  **printf**("memory cannot be allocated");      }  **else** {          // memory has successfully allocated  **printf**("Memory has been successfully allocated by "                 "using malloc\n");  **printf**("\n marks = %pc\n",                 marks); // print the base or beginning                         // address of allocated memory  **do** {  **printf**("\n Enter Marks\n");  **scanf**("%d", &marks[index]); // Get the marks  **printf**("would you like to add more(1/0): ");  **scanf**("%d", &ans);    **if** (ans == 1) {                  index++;                  marks = (**int**\*)**realloc**(                      marks,                      (index + 1)                          \* **sizeof**(  **int**)); // Dynamically reallocate                                     // memory by using realloc                  // check if the memory is successfully                  // allocated by realloc or not?  **if** (marks == NULL) {  **printf**("memory cannot be allocated");                  }  **else** {  **printf**("Memory has been successfully "                             "reallocated using realloc:\n");  **printf**(                          "\n base address of marks are:%pc",                          marks); ////print the base or                                  ///beginning address of                                  ///allocated memory                  }              }          } **while** (ans == 1);          // print the marks of the students  **for** (i = 0; i <= index; i++) {  **printf**("marks of students %d are: %d\n ", i,                     marks[i]);          }  **free**(marks);      }  **return** 0;  } |

**Output:**



# Opaque Pointer

**What is an opaque pointer?**  
Opaque as the name suggests is something we can’t see through. e.g. wood is opaque. Opaque pointer is a pointer which points to a data structure whose contents are not exposed at the time of its definition.

Following pointer is opaque. One can’t know the data contained in STest structure by looking at the definition.

struct STest\* pSTest;

It is safe to assign NULL to an opaque pointer.

pSTest = NULL;

**Why Opaque pointer?**

There are places where we just want to hint the compiler that “Hey! This is some data structure which will be used by our clients. Don’t worry, clients will provide its implementation while preparing compilation unit”. Such type of design is robust when we deal with shared code. Please see below example:

Let’s say we are working on an app to deal with images. Since we are living in a world where everything is moving to cloud and devices are very affordable to buy, we want to develop apps for windows, android and apple platforms. So, it would be nice to have a good design which is robust, scalable and flexible as per our requirements. We can have shared code which would be used by all platforms and then different end-point can have platform specific code.  
To deal with images, we have a CImage class exposing APIs to deal with various image operations (scale, rotate, move, save etc).  
Since all the platforms will be providing same operations, we would define this class in a header file. But the way an image is handled might differ across platforms. Like Apple can have different mechanism to access pixels of an image than Windows does. This means that APIs might demand different set of info to perform operations. So to work on shared code, this is what we would like to do:

**Image.h :** A header file to store class declaration.

|  |
| --- |
| // This class provides API to deal with various  // image operations. Different platforms can  // implement these operations in different ways.  **class** CImage  {  **public**:      CImage();      ~CImage();  **struct** SImageInfo\* pImageInfo;  **void** Rotate(**double** angle);  **void** Scale(**double** scaleFactorX,  **double** scaleFactorY);  **void** Move(**int** toX, **int** toY);  **private**:  **void** InitImageInfo();  }; |

**Image.cpp :** Code that will be shared across different end-points

|  |
| --- |
| // Constructor and destructor for CImage  CImage::CImage()  {      InitImageInfo();  }    CImage::~CImage()  {      // Destroy stuffs here  } |

**Image\_windows.cpp :** Code specific to Windows will reside here

|  |
| --- |
| **struct** SImageInfo  {     // Windows specific DataSet  };    **void** CImage::InitImageInfo()  {      pImageInfo = **new** SImageInfo;      // Initialize windows specific info here  }    **void** CImage::Rotate()  {      // Make use of windows specific SImageInfo  } |

**Image\_apple.cpp :** Code specific to Apple will reside here

|  |
| --- |
| **struct** SImageInfo  {      // Apple specific DataSet  };  **void** CImage::InitImageInfo()  {      pImageInfo = **new** SImageInfo;        // Initialize apple specific info here  }  **void** CImage::Rotate()  {      // Make use of apple specific SImageInfo  } |

As it can be seen from the above example, **while defining blueprint of the CImage class we are only mentioning that there is a SImageInfo data structure.  
The content of SImageInfo is unknown. Now it is the responsibility of clients(windows, apple, android) to define that data structure and use it as per their requirement**. If in future we want to develop app for a new end-point ‘X’, the design is already there. We only need to define SImageInfo for end-point ‘X’ and use it accordingly.

Please note that the above explained example is one way of doing this. Design is all about discussion and requirement. A good design is decided taking many factors into account. We can also have platform specific classes like CImageWindows, CImageApple and put all platform specific code there.

# References in C++

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

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **int** x = 10;        // ref is a reference to x.  **int**& ref = x;        // Value of x is now changed to 20      ref = 20;      cout << "x = " << x << '\n';        // Value of x is now changed to 30      x = 30;      cout << "ref = " << ref << '\n';    **return** 0;  } |

**Output:**

**X=20;**

ref = 30

**Applications :**

**1. Modify the passed parameters in a function**: If a function receives a reference to a variable, it can modify the value of the variable. For example, the following program variables are swapped using references.

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **void** swap(**int**& first, **int**& second)  {  **int** temp = first;      first = second;      second = temp;  }    **int** main()  {  **int** a = 2, b = 3;      swap(a, b);      cout << a << " " << b;  **return** 0;  } |

**Output:**

3 2

**2. Avoiding a** **copy of large structures**: Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes wastage of CPU time and memory. We can use references to avoid this.

* CPP

|  |
| --- |
| **struct** Student {      string name;      string address;  **int** rollNo;  }    // If we remove & in below function, a new  // copy of the student object is created.  // We use const to avoid accidental updates  // in the function as the purpose of the function  // is to print s only.  **void** print(**const** Student &s)  {      cout << s.name << "  " << s.address << "  " << s.rollNo           << '\n';  } |

**3. In For Each Loop to modify all objects**: We can use references in for each loop to modify all elements.

* CPP

|  |
| --- |
| #include <bits/stdc++.h>  **using** **namespace** std;    **int** main()  {      vector<**int**> vect{ 10, 20, 30, 40 };        // We can modify elements if we      // use reference  **for** (**int**& x : vect) {          x = x + 5;      }        // Printing elements  **for** (**int** x : vect) {          cout << x << " ";      }      cout << '\n';    **return** 0;  } |

**4. For Each Loop to avoid the** **copy of objects**: We can use references in each loop to avoid a copy of individual objects when objects are large.

* CPP

|  |
| --- |
| #include <bits/stdc++.h>  **using** **namespace** std;    **int** main()  {      vector<string> vect{ "geeksforgeeks practice",                           "geeksforgeeks write",                           "geeksforgeeks ide" };        // We avoid copy of the whole string      // object by using reference.  **for** (**const** **auto**& x : vect) {          cout << x << '\n';      }    **return** 0;  } |

**References vs Pointers:**

Both references and pointers can be used to change the local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain. Despite the above similarities, there are the following differences between references and pointers.

1. A pointer can be declared as void but a reference can never be void For example

int a = 10;

void\* aa = &a; // it is valid

void& ar = a; // it is not valid

2. The pointer variable has n-levels/multiple levels of indirection i.e. single-pointer, double-pointer, triple-pointer. Whereas, the reference variable has only one/single level of indirection. The following code reveals the mentioned points:

3. Reference variable cannot be updated.

4. Reference variable is an internal pointer.

5. Declaration of a Reference variable is preceded with the ‘&’ symbol ( but do not read it as “address of”).

* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **int** i = 10; // simple or ordinary variable.  **int**\* p = &i; // single pointer  **int**\*\* pt = &p; // double pointer  **int**\*\*\* ptr = &pt; // triple pointer      // All the above pointers differ in the value they store      // or point to.      cout << "i = " << i << "\t"           << "p = " << p << "\t"           << "pt = " << pt << "\t"           << "ptr = " << ptr << '\n';  **int** a = 5; // simple or ordinary variable  **int**& S = a;  **int**& S0 = S;  **int**& S1 = S0;      cout << "a = " << a << "\t"           << "S = " << S << "\t"           << "S0 = " << S0 << "\t"           << "S1 = " << S1 << '\n';      // All the above references do not differ in their      // values as they all refer to the same variable.  } |

* **References are less powerful than pointers**  
  **1)** Once a reference is created, it cannot be later made to reference another object; it cannot be reset. This is often done with pointers.   
  **2)** References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.   
  **3)** A reference must be initialized when declared. There is no such restriction with pointers.  
  Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don’t have the above restrictions and can be used to implement all data structures. References being more powerful in Java is the main reason Java doesn’t need pointers.
* **References are safer and easier to use:**  
  ***1) Safer****:* Since references must be initialized, wild references like [wild pointers](https://www.geeksforgeeks.org/what-are-wild-pointers-how-can-we-avoid/) are unlikely to exist. It is still possible to have references that don’t refer to a valid location (See questions 5 and 6 in the below exercise)   
  ***2) Easier to use:*** References don’t need a dereferencing operator to access the value. They can be used like normal variables. ‘&’ operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->) is needed to access members.
* Together with the above reasons, there are few places like the copy constructor argument where pointer cannot be used. Reference must be used to pass the argument in the copy constructor. Similarly, references must be used for overloading some operators like ++.

**Exercise:**   
Predict the output of the following programs. If there are compilation errors, then fix them.  
**Question 1**

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int**& fun()  {  **static** **int** x = 10;  **return** x;  }    **int** main()  {      fun() = 30;      cout << fun();  **return** 0;  } |

**Question 2**

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** fun(**int**& x) { **return** x; }    **int** main()  {      cout << fun(10);  **return** 0;  } |

**Question 3**

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **void** swap(**char**\*& str1, **char**\*& str2)  {  **char**\* temp = str1;      str1 = str2;      str2 = temp;  }    **int** main()  {  **char**\* str1 = "GEEKS";  **char**\* str2 = "FOR GEEKS";      swap(str1, str2);      cout << "str1 is " << str1 << '\n';      cout << "str2 is " << str2 << '\n';  **return** 0;  } |

**Question 4**

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **int** x = 10;  **int**\* ptr = &x;  **int**&\* ptr1 = ptr;  } |

**Question 5**

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int** main()  {  **int**\* ptr = NULL;  **int**& ref = \*ptr;      cout << ref << '\n';  } |

**Question 6**

* CPP

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **int**& fun()  {  **int** x = 10;  **return** x;  }  **int** main()  {      fun() = 30;      cout << fun();  **return** 0;  } |

**Related Articles :**

* [Pointers vs References in C++](https://www.geeksforgeeks.org/pointers-vs-references-cpp/)
* [When do we pass arguments by reference or pointer?](https://www.geeksforgeeks.org/when-do-we-pass-arguments-by-reference-or-pointer/)
* [Can references refer to invalid location in C++?](https://www.geeksforgeeks.org/g-fact-25/)
* [Passing by pointer Vs Passing by Reference in C++](https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-c/)

# ‘this’ pointer in C++:

To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.  
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?  
The compiler supplies an implicit pointer along with the names of the functions as ‘this’.  
The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).  
For a class X, the type of this pointer is ‘X\* ‘. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X \*’ (see [this GFact](https://www.geeksforgeeks.org/g-fact-77/))

In the early version of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.  
C++ lets object destroy themselves by calling the following code :

|  |
| --- |
| **delete** **this**; |

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

Following are the situations where ‘this’ pointer is used:

**1) When local variable’s name is same as member’s name**

|  |
| --- |
| #include<iostream>  **using** **namespace** std;    /\* local variable is same as a member's name \*/  **class** Test  {  **private**:  **int** x;  **public**:  **void** setX (**int** x)     {         // The 'this' pointer is used to retrieve the object's x         // hidden by the local variable 'x'  **this**->x = x;     }  **void** print() { cout << "x = " << x << endl; }  };    **int** main()  {     Test obj;  **int** x = 20;     obj.setX(x);     obj.print();  **return** 0;  } |

Output:

x = 20

For constructors, [initializer list](https://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/) can also be used when parameter name is same as member’s name.

**2) To return reference to the calling object**

|  |
| --- |
| /\* Reference to the calling object can be returned \*/  Test& Test::func ()  {     // Some processing  **return** \***this**;  } |

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

|  |
| --- |
| #include<iostream>  **using** **namespace** std;    **class** Test  {  **private**:  **int** x;  **int** y;  **public**:    Test(**int** x = 0, **int** y = 0) { **this**->x = x; **this**->y = y; }    Test &setX(**int** a) { x = a; **return** \***this**; }    Test &setY(**int** b) { y = b; **return** \***this**; }  **void** print() { cout << "x = " << x << " y = " << y << endl; }  };    **int** main()  {    Test obj1(5, 5);      // Chained function calls.  All calls modify the same object    // as the same object is returned by reference    obj1.setX(10).setY(20);      obj1.print();  **return** 0;  } |

Output:

x = 10 y = 20

**Exercise:**  
Predict the output of following programs. If there are compilation errors, then fix them.

**Question 1**

|  |
| --- |
| #include<iostream>  **using** **namespace** std;    **class** Test  {  **private**:  **int** x;  **public**:    Test(**int** x = 0) { **this**->x = x; }  **void** change(Test \*t) { **this** = t; }  **void** print() { cout << "x = " << x << endl; }  };    **int** main()  {    Test obj(5);    Test \*ptr = **new** Test (10);    obj.change(ptr);    obj.print();  **return** 0;  } |

**Question 2**

|  |
| --- |
| #include<iostream>  **using** **namespace** std;    **class** Test  {  **private**:  **int** x;  **int** y;  **public**:    Test(**int** x = 0, **int** y = 0) { **this**->x = x; **this**->y = y; }  **static** **void** fun1() { cout << "Inside fun1()"; }  **static** **void** fun2() { cout << "Inside fun2()"; **this**->fun1(); }  };    **int** main()  {    Test obj;    obj.fun2();  **return** 0;  } |

**Question 3**

|  |
| --- |
| #include<iostream>  **using** **namespace** std;    **class** Test  {  **private**:  **int** x;  **int** y;  **public**:    Test (**int** x = 0, **int** y = 0) { **this**->x = x; **this**->y = y; }    Test setX(**int** a) { x = a; **return** \***this**; }    Test setY(**int** b) { y = b; **return** \***this**; }  **void** print() { cout << "x = " << x << " y = " << y << endl; }  };  **int** main()  {    Test obj1;    obj1.setX(10).setY(20);    obj1.print();  **return** 0;  } |

**Question 4**

|  |
| --- |
| #include<iostream>  **using** **namespace** std;    **class** Test  {  **private**:  **int** x;  **int** y;  **public**:    Test(**int** x = 0, **int** y = 0) { **this**->x = x; **this**->y = y; }  **void** setX(**int** a) { x = a; }  **void** setY(**int** b) { y = b; }  **void** destroy()  { **delete** **this**; }  **void** print() { cout << "x = " << x << " y = " << y << endl; }  };    **int** main()  {    Test obj;    obj.destroy();    obj.print();  **return** 0;  } |

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# Smart Pointers in C++ and How to Use Them:

In this article, we will be discussing smart pointers in C++. What are Smart Pointers, why, and how to use them properly?

Pointers are used for accessing the resources which are external to the program – like heap memory. So, for accessing the heap memory (if anything is created inside heap memory), pointers are used. When accessing any external resource we just use a copy of the resource. If we make any change to it, we just change it in the copied version. But, if we use a pointer to the resource, we’ll be able to change the original resource.

## Problems with Normal Pointers

Take a look at the code below.

* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **class** Rectangle {  **private**:  **int** length;  **int** breadth;  };    **void** fun()  {      // By taking a pointer p and      // dynamically creating object      // of class rectangle      Rectangle\* p = **new** Rectangle();  }    **int** main()  {      // Infinite Loop  **while** (1) {          fun();      }  } |

In function *fun*, it creates a pointer that is pointing to the *Rectangle* object. The object *Rectangle* contains two integers, *length* and *breadth*. When the function *fun* ends, p will be destroyed as it is a local variable. But, the memory it consumed won’t be deallocated because we forgot to use *delete p;* at the end of the function. That means the memory won’t be free to be used by other resources. But, we don’t need the variable anymore, but we need the memory.

In function *main*, *fun* is called in an infinite loop. That means it’ll keep creating *p*. It’ll allocate more and more memory but won’t free them as we didn’t deallocate it. The memory that’s wasted can’t be used again. Which is a memory leak. The entire *heap* memory may become useless for this reason. C++11 comes up with a solution to this problem, Smart Pointer.

## Introduction of Smart Pointers

As we’ve known unconsciously not deallocating a pointer causes a memory leak that may lead to crash of the program. Languages Java, C# has *Garbage Collection Mechanisms* to smartly deallocate unused memory to be used again. The programmer doesn’t have to worry about any memory leak. C++11 comes up with its own mechanism that’s *Smart Pointer*. When the object is destroyed it frees the memory as well. So, we don’t need to delete it as Smart Pointer does will handle it.

A *Smart Pointer* is a wrapper class over a pointer with an operator like \* and -> overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers* it can deallocate and free destroyed object memory.

The idea is to take a class with a pointer, [destructor](https://www.geeksforgeeks.org/destructors-c/)and [overloaded operators](https://www.geeksforgeeks.org/operator-overloading-c/) like \* and ->. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or reference count can be decremented). Consider the following simple *SmartPtr* class.

* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    **class** SmartPtr {  **int**\* ptr; // Actual pointer  **public**:      // Constructor: Refer [https://](NULL) www.geeksforgeeks.org/g-fact-93/      // for use of explicit keyword  **explicit** SmartPtr(**int**\* p = NULL) { ptr = p; }        // Destructor      ~SmartPtr() { **delete** (ptr); }        // Overloading dereferencing operator  **int**& operator\*() { **return** \*ptr; }  };    **int** main()  {      SmartPtr ptr(**new** **int**());      \*ptr = 20;      cout << \*ptr;        // We don't need to call delete ptr: when the object      // ptr goes out of scope, the destructor for it is automatically      // called and destructor does delete ptr.    **return** 0;  } |

**Output:**

20

This only works for *int*. So, we’ll have to create Smart Pointer for every object? No, there’s a solution, *Template*. In the code below as you can see *T* can be of any type. Read more about [Template](https://www.geeksforgeeks.org/templates-cpp/) here.

* C++

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    // A generic smart pointer class  **template** <**class** T>  **class** SmartPtr {      T\* ptr; // Actual pointer  **public**:      // Constructor  **explicit** SmartPtr(T\* p = NULL) { ptr = p; }        // Destructor      ~SmartPtr() { **delete** (ptr); }        // Overloading dereferencing operator      T& operator\*() { **return** \*ptr; }        // Overloading arrow operator so that      // members of T can be accessed      // like a pointer (useful if T represents      // a class or struct or union type)      T\* operator->() { **return** ptr; }  };    **int** main()  {      SmartPtr<**int**> ptr(**new** **int**());      \*ptr = 20;      cout << \*ptr;  **return** 0;  } |

**Output:**

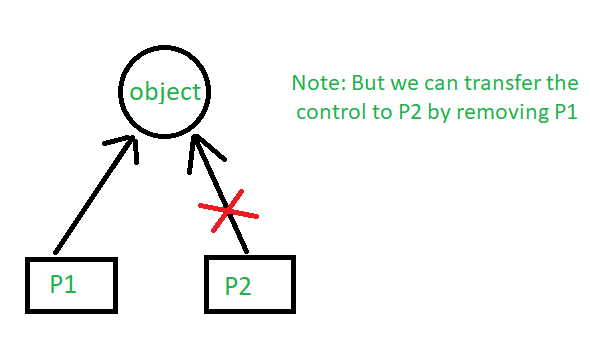
20

**Note:** Smart pointers are also useful in the management of resources, such as file handles or network sockets.

## Types of Smart Pointers

### **1. unique\_ptr**

*unique\_ptr* stores one pointer only. We can assign a different object by removing the current object from the pointer. Notice the code below. First, the *unique\_pointer* is pointing to *P1*. But, then we remove *P1* and assign *P2* so the pointer now points to *P2*.



* C++14

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  #include <memory>    **class** Rectangle {  **int** length;  **int** breadth;    **public**:      Rectangle(**int** l, **int** b){          length = l;          breadth = b;      }    **int** area(){  **return** length \* breadth;      }  };    **int** main(){        unique\_ptr<Rectangle> P1(**new** Rectangle(10, 5));      cout << P1->area() << endl; // This'll print 50        // unique\_ptr<Rectangle> P2(P1);      unique\_ptr<Rectangle> P2;      P2 = move(P1);        // This'll print 50      cout << P2->area() << endl;        // cout<<P1->area()<<endl;  **return** 0;  } |

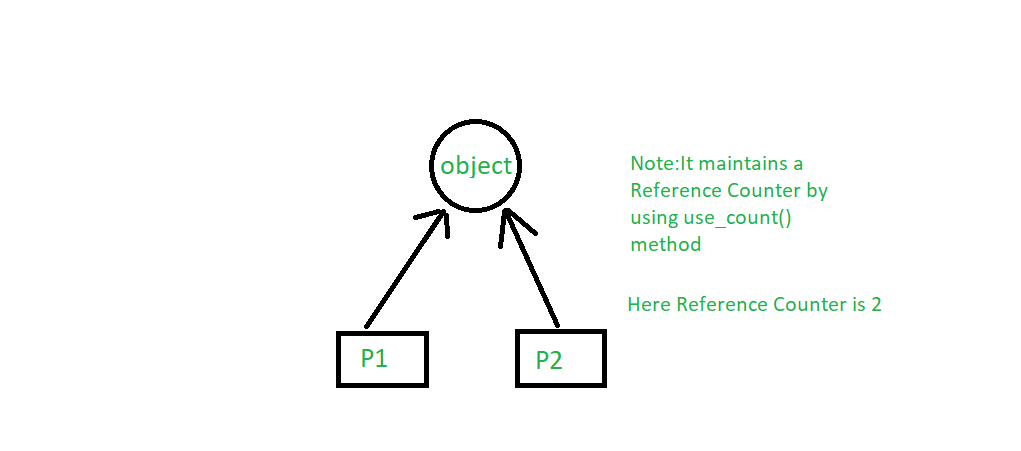
**Output:**

50

50

### 2. shared\_ptr

By using *shared\_ptr* more than one pointer can point to this one object at a time and it’ll maintain a **Reference Counter** using ***use\_count()* method.**



* C++14

|  |
| --- |
| #include <iostream>  **using** **namespace** std;  #include <memory>    **class** Rectangle {  **int** length;  **int** breadth;    **public**:      Rectangle(**int** l, **int** b)      {          length = l;          breadth = b;      }    **int** area()      {  **return** length \* breadth;      }  };    **int** main()  {        shared\_ptr<Rectangle> P1(**new** Rectangle(10, 5));      // This'll print 50      cout << P1->area() << endl;        shared\_ptr<Rectangle> P2;      P2 = P1;        // This'll print 50      cout << P2->area() << endl;        // This'll now not give an error,      cout << P1->area() << endl;        // This'll also print 50 now      // This'll print 2 as Reference Counter is 2      cout << P1.use\_count() << endl;  **return** 0;  } |

**Output:**

50

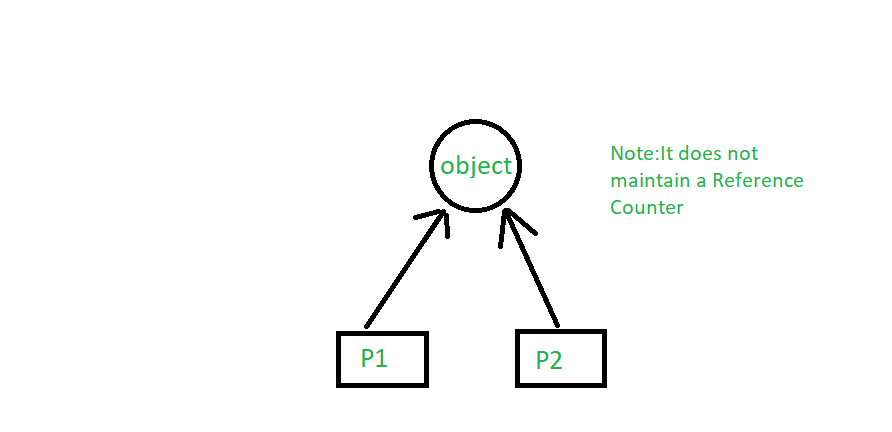
50

50

2

### 3. weak\_ptr

It’s much more similar to shared\_ptr except it’ll not maintain a **Reference Counter**. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock.**



C++ libraries provide implementations of smart pointers in the form of [auto\_ptr](http://en.wikipedia.org/wiki/Auto_ptr" \t "_blank), [unique\_ptr](http://en.wikipedia.org/wiki/Smart_pointer" \l "unique_ptr" \t "_blank), [shared\_ptr and weak\_ptr](http://en.wikipedia.org/wiki/Smart_pointer" \l "shared_ptr_and_weak_ptr" \t "_blank)

# Pointers vs References in C++

C and C++ support pointers, which is different from most other programming languages such as Java, Python, Ruby, Perl and PHP as they only support references. But interestingly, C++, along with pointers, also supports references.

On the surface, both references and pointers are very similar as both are used to have one variable provide access to another. With both providing lots of the same capabilities, it’s often unclear what is different between these mechanisms. In this article, I will try to illustrate the differences between pointers and references.

[Pointers](https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/): A pointer is a variable that holds the memory address of another variable. A pointer needs to be dereferenced with the **\*** operator to access the memory location it points to.

[References](https://www.geeksforgeeks.org/references-in-c/): A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.   
A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e., the compiler will apply the **\***operator for you.

int i = 3;

// A pointer to variable i or "stores the address of i"

int \*ptr = &i;

// A reference (or alias) for i.

int &ref = i;

**Differences**:

1.**Initialization:** A pointer can be initialized in this way:

int a = 10;

int \*p = &a;

// OR

int \*p;

p = &a;

We can declare and initialize pointer at same step or in multiple line.

2. While in references,

int a = 10;

int &p = a; // It is correct

// but

int &p;

p = a; // It is incorrect as we should declare and initialize references at single step

**NOTE:**This difference may vary from compiler to compiler. The above difference is with respect to Turbo IDE.

3. **Reassignment:** A pointer can be re-assigned. This property is useful for the implementation of data structures like a linked list, a tree, etc. See the following example:

int a = 5;

int b = 6;

int \*p;

p = &a;

p = &b;

4. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

int a = 5;

int b = 6;

int &p = a;

int &p = b; // This will throw an error of "multiple declaration is not allowed"

// However it is valid statement,

int &q = p;

5. **Memory Address:** A pointer has its own memory address and size on the stack, whereas a reference shares the same memory address with the original variable but also takes up some space on the stack.

int &p = a;

cout << &p << endl << &a;

6. **NULL value:**A pointer can be assigned NULL directly, whereas a reference cannot be. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into an exception situation.

7. **Indirection:** You can have a pointer to pointer (known as a double pointer) offering extra levels of indirection, whereas references only offer one level of indirection. For example,

In Pointers,

int a = 10;

int \*p;

int \*\*q; // It is valid.

p = &a;

q = &p;

// Whereas in references,

int &p = a;

int &&q = p; // It is reference to reference, so it is an error

8. **Arithmetic operations:** Various arithmetic operations can be performed on pointers, whereas there is no such thing called Reference Arithmetic (however, you can perform pointer arithmetic on the address of an object pointed to by a reference, as in &obj + 5).

**When to use What**

The performances are exactly the same as references are implemented internally as pointers. But still, you can keep some points in your mind to decide when to use what:

* Use references:
  + In function parameters and return types.
* Use pointers:
  + If pointer arithmetic or passing a NULL pointer is needed. For example, for arrays (Note that accessing an array is implemented using pointer arithmetic).
  + To implement data structures like a linked list, a tree, etc. and their algorithms. This is so because, in order to point to different cells, we have to use the concept of pointers.

[Quoted in C++ FAQ Lite](https://isocpp.org/wiki/faq/references#refs-vs-ptrs): Use references when you can, and pointers when you have to. References are usually preferred over pointers whenever you don’t need “reseating”. This usually means that references are most useful in a class’s public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function’s parameter or return value needs a “sentinel” reference — a reference that does not refer to an object. This is usually best done by returning/taking a pointer, and giving the “nullptr” value this special significance (references must always alias objects, not a dereferenced null pointer).