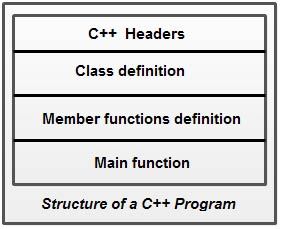
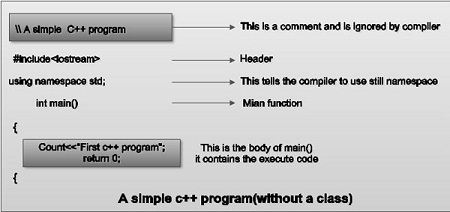
**Structure of a C+ + Program**

Programs are a sequence of instructions or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely, *headers, class definition, member functions definitions*and *main function.*

**

Note that C++ provides the flexibility of writing a program with or without a class and its member functions definitions. A simple C++ program (without a class) includes comments, headers, namespace, main() and input/output statements.

**Comments**are a vital element of a program that is used to increase the readability of a program and to describe its functioning. Comments are not executable statements and hence, do not increase the size of a file.



C++ supports two comment styles: single line comment and multiline comment. Single line comments are used to define line-by-line descriptions. Double slash *(//)*is used to represent single line comments. To understand the concept of single line comment, consider this statement.

**/ / An example to demonstrate single line comment It can also be written**

/**/ An example to demonstrate**

**/ / single line comment**

Multiline comments are used to define multiple lines descriptions and are represented as / \* \* /. For example, consider this statement.

**/\* An example to demonstrate multiline comment \*/**

Generally, multiline comments are not used in C++ as they require more space on the line. However, they are useful within the program statements where single line comments cannot be used. For example, consider this statement.

**for(int i = 0; i<10; //loop runs 10 times i++)**

Compiler ignores everything written after the single line comment and hence, an error occurs. Therefore, in this case multiline comments are used. For example, consider this statement.

**for(int i = 0; i<10; /\*loop runs 10 times \*/ i++)**

**Headers:** Generally, a program includes various programming elements like built-in functions, classes, keywords, constants, operators, etc., that are already defined in the standard C++ library. In order to use such pre-defined elements in a program, an appropriate header must be included in the program. The standard headers contain the information like prototype, definition and return type of library functions, [data type](https://ecomputernotes.com/java/data-type-variable-and-array/explain-data-types-in-java) of constants, etc. As a result, programmers do not need to explicitly declare (or define) the predefined programming elements.

Standard headers are specified in a program through the preprocessor directive” #include. In Figure, the iostream header is used. When the compiler processes the instruction #inc1ude<iostream>, it includes the contents of iostream in the program. This enables the programmer to use *standard input, output*and *error*facilities that are provided only through the standard streams defined in <iostream>. These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

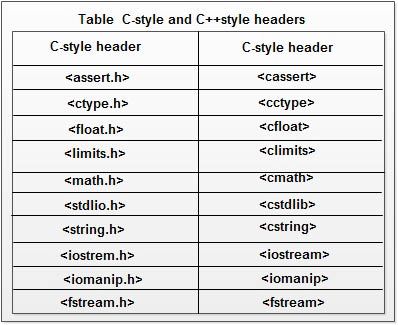
**• cin** (pronounced **“see in”**) : It is the standard input stream that is associated with the standard input device (keyboard) and is used to take the input from users.

**• cout** (pronounced **“see out”**) : It is the standard output stream that is associated with the standard [output device](https://ecomputernotes.com/fundamental/input-output-and-memory/list-various-input-and-output-devices) (monitor) and is used to display the output to users.

**• cerr** (pronounced **“see err”**) : It is the standard error stream that is associated with the standard error device (monitor) and is used to report errors to the users. The cerr object does not have a buffer (temporary storage area) and hence, immediately reports errors to users. ‘

**• clog** (pronounced **“see log”**): It is the buffered error stream that is associated with the standard error device ([computer](https://ecomputernotes.com/fundamental/introduction-to-computer/what-is-computer) screen) and is used to report errors to users. Unlike cerr, clog reports errors to users only when the buffer is full

For many years, C++ applied C-style headers, that is, .h extension in the headers. However, the standard C++ library introduced new-style headers that include only header name. Hence, the most modem compilers do not require any extension, though they support the older .h extension. Some of C-style headers and their equivalent C++ style headers are listed in Table.

**Namespace:** Since its creation, C++ has gone through many changes by the C++ Standards Committee. One of the new features added to this language is namespace. A namespace permits grouping of various entities like classes, objects, functions and various C++ tokens, etc., under a single name. Different users can create separate namespaces and thus can use similar names of the entities. This avoids compile-time error that may exist due to identical-name conflicts.

The C++ Standards Committee has rearranged the entities of the standard library under a namespace called std. In Figure, the statement using namespace std informs the compiler to include all the entities present in the namespace std. The entities of a namespace can be accessed in different ways which are listed here.

• By specifying the using directive

**using namespace std;**

**cout<<“Hello World”;**

• By specifying the full member name

**std: :cout<<“Hello World”;**

• By specifying the using declaration

**using std:: cout;**

**cout<<“Hello World”;**

As soon as the new-style header is included, its contents are included in the std namespace. Thus, all the modern C++ compilers support these statements.

**#include<iostream>**

**using namespace std;**

However, some old compilers may not support these statements. In that case, the statements are replaced by this single statement.

**#include<iostream.h>**

**Main Function:**The main () is a startup function that starts the execution of a c++ program. All C++ statements that need to be executed are written within main ( ). The compiler executes all the instructions written within the opening and closing curly braces’ {}’ that enclose the body of main ( ). Once all the instructions in main () are executed, the control passes out of main ( ), terminating the entire program and returning a value to the [operating system](https://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system).

By default, main () in C++ returns an int value to the [operating system](https://ecomputernotes.com/fundamental/disk-operating-system/what-is-operating-system). Therefore, main () should end with the return 0 statement. A return value zero indicates success and a non-zero value indicates failure or error.

# Tokens in C++ (Keywords, Identifiers, Constants, Strings, Operators, Special Symbols )

Tokens and keywords are undoubtedly notable features of C++.

Tokens act as building blocks of a program. Just like a living cell is the smallest possible unit of life, tokens in C++ are referred to as the smallest individual units in a program. Keywords in C++ help the user in framing statements and commands in a language. Each keyword conveys a unique connotation to the compiler to perform a specific task. Just like the combination of words helps us in framing sentences, the combination of keywords helps us in framing statements to perform logical operations in a programming language. Simply combining keywords wouldn’t help to serve the purpose.

As we need to use proper grammar to form a meaningful sentence, we need to be well-acquainted with the ***syntax of C++*** to instruct the compiler what to do. If these statements are not formed in a logical manner, they would sound gibberish and you would get a compilation error. Let’s discuss the concept of Tokens with Character set in C++ in detail.

## **1. C++ Character Set**

Before we begin with C++ tokens, let us understand what Character set has to offer.

C++ Character set is basically a set of valid characters that convey a specific connotation to the compiler. We use characters to represent letters, digits, special symbols, white spaces, and other characters.

The C++ character set consists of 3 main elements. They are:

1. **Letters:** These are alphabets ranging from A-Z and a-z (both uppercase and lowercase characters convey different meanings)
2. **Digits:** All the digits from 0 – 9 are valid in C++.
3. **Special symbols:** There are a variety of special symbols available in C++ like mathematical, logical and relational operators like **+,-, \*, /, \, ^, %, !, @, #, ^, &, (, ), [, ], ;** and many more.

## **2. Tokens in C++**

As discussed earlier, ***tokens in C++ are the smallest individual unit of a program***.

The following tokens are available in C++ which are similar to that seen in C with the addition of certain exclusive keywords, strings, and operators:

* Keywords
* Identifiers
* Constants
* Strings
* Special symbols
* Operators

## **3. C++ Keywords**

**Keywords in C++ refer to the pre-existing, reserved words, each holding its own position and power and has a specific function associated with it.**

It is important to note that we cannot use C++ keywords for assigning variable names as it would suggest a totally different meaning entirely and would be incorrect.

**Get the Samurai Technique to Learn**[**Arrays in C and C++**](https://data-flair.training/blogs/arrays-in-c-and-cpp/)

Here is a list of keywords available in C++ according to the latest standards:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| alignas | alignof | asm | auto | bool | break |
| case | catch | char | char16\_t | char32\_t | class |
| const | constexpr | const\_cast | continue | decltype | default |
| delete | double | do | dynamic\_cast | else | enum |
| explicit | export | extern | FALSE | float | for |
| friend | goto | if | inline | int | long |
| mutable | namespace | new | noexcept | nullptr | operator |
| private | protected | public | register | reinterpret\_cast | return |
| short | signed | sizeof | static | static\_assert | static\_cast |
| struct | switch | template | this | thread\_local | throw |
| TRUE | try | typedef | typeid | typename | union |
| unsigned | using | virtual | void | volatile | wchar\_t |
| while | – | – | – | – | – |

## **4. C++ Identifiers**

C++ allows the programmer to assign names of his own choice to variables, arrays, functions, structures, classes, and various other data structures called identifiers. The programmer may use the mixture of different types of character sets available in C++ to name an identifier.

##### Rules for C++ Identifiers

There are certain rules to be followed by the user while naming identifiers, otherwise, you would get a compilation error. These rules are:

1. **First character:** The first character of the identifier in C++ should positively begin with either an alphabet or an underscore. It means that it strictly cannot begin with a number.
2. **No special characters:** C++ does not encourage the use of special characters while naming an identifier. It is evident that we cannot use special characters like the exclamatory mark or the “@” symbol.
3. **No keywords:** Using keywords as identifiers in C++ is strictly forbidden, as they are reserved words that hold a special meaning to the C++ compiler. If used purposely, you would get a compilation error.
4. **No white spaces:** Leaving a gap between identifiers is discouraged. White spaces incorporate blank spaces, newline, carriage return, and horizontal tab.
5. **Word limit:** The use of an arbitrarily long sequence of identifier names is restrained. The name of the identifier must not exceed 31 characters, otherwise, it would be insignificant.
6. **Case sensitive:** In C++, uppercase and lowercase characters connote different meanings.
7. Here is a table which illustrates the valid use of Identifiers:

|  |  |  |  |
| --- | --- | --- | --- |
| **Identifier Name** | **Valid or Invalid** | **Correction or alternative, if invalid** | **Elucidation if invalid** |
| 5th\_element | Invalid | element\_5 | It violates Rule 1 as it begins with a digit |
| \_delete | Valid | – | – |
| school.fee | Invalid | school\_fee | It violates Rule 2 as it contains a special character ‘.’ |
| register[5] | Invalid | Register[5] | It violates Rule 3 as it contains a keyword |
| Student[10] | Valid | – | – |
| employee name | Invalid | employee \_name | It violates Rule 4 as it contains a blank space |
| perimeter() | Valid | – | – |

## **5. C++ Constants**

Before we begin our discussion on constants in C++, it is important to note that we can use the terms “constants” and “literals” interchangeably.

As the name itself suggests, constants are referred to as fixed values that cannot change their value during the entire program run as soon as we define them.

**Syntax:**

const data\_type variable\_name = value;

#### **Types of Constants in C++**

The different types of constants are:

* **Integer constants –** These constants store values of the int data type.

For instance:

const int data = 5;

* **Floating constants –**These constants store values of the float data type.

For instance:

const float e = 2.71;

***Refer to***[***Constants and Literals in C++***](https://data-flair.training/blogs/constants-in-c-and-c-plus-plus/)***for a detailed description.***

* **Character constants –** These constants store values of the character data type.

For instance:

const char answer = ‘y’;

* **String constants –**These constants are also of the character data type but differ in the declaration part.

For instance:

const char title[] = ‘‘DataFlair’’;

* **Octal constants –** The number system which consists of only 8 digits, from 0 to 7 is called the octal number system. The constant octal values can be declared as:

const int oct = 034;

It is the octal equivalent of the digit 28 in the decimal number system.

* **Hexadecimal constants –**The number system which consists of 16 digits, from 0 to 9 and alphabets ‘a’ to ‘f’ is called hexadecimal number system. The constant hexadecimal values can be declared as:

const int hex = 0x40;

It is the hexadecimal equivalent of the digit 64 in the decimal number system.

## **6. C++ Strings**

Just like characters,[***strings in C++***](https://data-flair.training/blogs/strings-in-c-cpp/) are used to store letters and digits. Strings can be referred to as an array of characters as well as an individual data type.

It is enclosed within double quotes, unlike characters which are stored within single quotes. The termination of a string in C++ is represented by the null character, that is,***‘\0’.*** The size of a string is the number of individual characters it has.

In C++, a string can be declared in the following ways:

char name[30] = ‘’Hello!”; // The compiler reserves 30 bytes of memory for the string.

char name[] = “Hello!”; // The compiler reserves the required amount of memory for the string.

char name[30] = { ‘H’ , ’e’ , ’l’ , ’l’ , ’o’};; // This is how a string is represented as a set of characters.

string name = “Hello” // The compiler reserves 32 bytes of memory.

## **7. Special Symbols**

Apart from letters and digits, there are some special characters in [C++](https://en.wikipedia.org/wiki/C%2B%2B) which help you manipulate or perform data operations. Each special symbol has a specific meaning to the C++ compiler.

**Here is a table which illustrates some of the special characters in C:**

|  |  |  |
| --- | --- | --- |
| **Special Character** | **Trivial Name** | **Function** |
| [ ] | Square brackets | The opening and closing brackets of an array symbolize single and multidimensional subscripts. |
| () | Simple brackets | The opening and closing brackets represent function declaration and calls, used in print statements. |
| { } | Curly braces | The opening and closing curly brackets to denote the start and end of a particular fragment of code which may be functions, loops or conditional statements |
| , | Comma | We use commas to separate more than one statements, like in the declaration of different variable names |
| # | Hash / Pound / Preprocessor | The hash symbol represents a preprocessor directive used for denoting the use of a header file |
| \* | Asterisk | We use the asterisk symbol in various respects such as to declare pointers, used as an operand for multiplication |
| ~ | Tilde | We use the tilde symbol as a destructor to free memory |
| . | Period / dot | The use the dot operator to access a member of a structure |

## **8. C++ Operators**

Operators are tools or symbols which are used to perform a specific operation on data. Operations are performed on operands. Operators can be classified into three broad categories according to the number of operands used.

**Unary:** It involves the use of one a single operand. For instance, ’!’ is a unary operator which operates on a single variable, say ‘c’ as !c which denotes its negation or complement.

**Binary:** It involves the use of 2 operands. They are further classified as:

* Arithmetic
* Relational
* Logical
* Assignment
* Bitwise
* Conditional

**Ternary:** It involves the use of 3 operands. For instance, ?: is used to in place of if-else conditions.

# C++ Data Types

In C++, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

int age = 13;

Here, age is a variable of type int. Meaning, the variable can only store integers of either 2 or 4 bytes.

## C++ Fundamental Data Types

The table below shows the fundamental data types, their meaning, and their sizes (in bytes):

|  |  |  |
| --- | --- | --- |
| Data Type | Meaning | Size (in Bytes) |
| int | Integer | 2 or 4 |
| float | Floating-point | 4 |
| double | Double Floating-point | 8 |
| char | Character | 1 |
| wchar\_t | Wide Character | 2 |
| bool | Boolean | 1 |
| void | Empty | 0 |

Now, let us discuss these fundamental data types in more detail.

### 1. C++ int

* The int keyword is used to indicate integers.
* Its size is usually 4 bytes. Meaning, it can store values from **-2147483648 to 2147483647**.
* For example,

int salary = 85000;

### 2. C++ float and double

* float and double are used to store floating-point numbers (decimals and exponentials).
* The size of float is 4 bytes and the size of double is 8 bytes. Hence, double has two times the precision of float. To learn more, visit C++ float and double.
* For example,

float area = 64.74;

double volume = 134.64534;

As mentioned above, these two data types are also used for exponentials. For example,

double distance = 45E12 // 45E12 is equal to 45\*10^12

### 3. C++ char

* Keyword char is used for characters.
* Its size is 1 byte.
* Characters in C++ are enclosed inside single quotes ' '.
* For example,

char test = 'h';

**Note:** In C++, an integer value is stored in a char variable rather than the character itself. To learn more, visit [C++ characters](https://www.programiz.com/cpp-programming/char-type).

### 4. C++ wchar\_t

* Wide character wchar\_t is similar to the char data type, except its size is 2 bytes instead of 1.
* It is used to represent characters that require more memory to represent them than a single char.
* For example,

wchar\_t test = L'ם' // storing Hebrew character;

Notice the letter L before the quotation marks.

**Note:** There are also two other fixed-size character types char16\_t and char32\_t introduced in C++11.

### 5. C++ bool

* The bool data type has one of two possible values: true or false.
* Booleans are used in conditional statements and loops (which we will learn in later chapters).
* For example,

bool cond = false;

### 6. C++ void

* The void keyword indicates an absence of data. It means "nothing" or "no value".
* We will use void when we learn about functions and pointers.

**Note:** We cannot declare variables of the void type.

## C++ Type Modifiers

We can further modify some of the fundamental data types by using type modifiers. There are 4 type modifiers in C++. They are:

1. signed
2. unsigned
3. short
4. long

We can modify the following data types with the above modifiers:

* int
* double
* char

### C++ Modified Data Types List

|  |  |  |
| --- | --- | --- |
| Data Type | Size (in Bytes) | Meaning |
| signed int | 4 | used for integers (equivalent to int) |
| unsigned int | 4 | can only store positive integers |
| short | 2 | used for small integers (range **-32768 to 32767**) |
| unsigned short | 2 | used for small positive integers (range **0 to 65,535**) |
| long | at least 4 | used for large integers (equivalent to long int) |
| unsigned long | 4 | used for large positive integers or 0 (equivalent to unsigned long int) |
| long long | 8 | used for very large integers (equivalent to long long int). |
| unsigned long long | 8 | used for very large positive integers or 0 (equivalent to unsigned long long int) |
| long double | 12 | used for large floating-point numbers |
| signed char | 1 | used for characters (guaranteed range **-127 to 127**) |
| unsigned char | 1 | used for characters (range **0 to 255**) |

Let's see a few examples.

long b = 4523232;

long int c = 2345342;

long double d = 233434.56343;

short d = 3434233; // Error! out of range

unsigned int a = -5; // Error! can only store positive numbers or 0

## Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

We will learn about these derived data types in later tutorials.

# C++ Type Conversion

C++ allows us to convert data of one type to that of another. This is known as type conversion.

There are two types of type conversion in C++.

1. Implicit Conversion
2. Explicit Conversion (also known as Type Casting)

## Implicit Type Conversion

The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

Let us look at two examples of implicit type conversion.

### Example 1: Conversion From int to double

// Working of implicit type-conversion

#include <iostream>

using namespace std;

int main() {

// assigning an int value to num\_int

int num\_int = 9;

// declaring a double type variable

double num\_double;

// implicit conversion

// assigning int value to a double variable

num\_double = num\_int;

cout << "num\_int = " << num\_int << endl;

cout << "num\_double = " << num\_double << endl;

return 0;

}

**Output**

num\_int = 9

num\_double = 9

In the program, we have assigned an int data to a double variable.

num\_double = num\_int;

Here, the int value is automatically converted to double by the compiler before it is assigned to the num\_double variable. This is an example of implicit type conversion.

### Example 2: Automatic Conversion from double to int

//Working of Implicit type-conversion

#include <iostream>

using namespace std;

int main() {

int num\_int;

double num\_double = 9.99;

// implicit conversion

// assigning a double value to an int variable

num\_int = num\_double;

cout << "num\_int = " << num\_int << endl;

cout << "num\_double = " << num\_double << endl;

return 0;

}

**Output**

num\_int = 9

num\_double = 9.99

In the program, we have assigned a double data to an int variable.

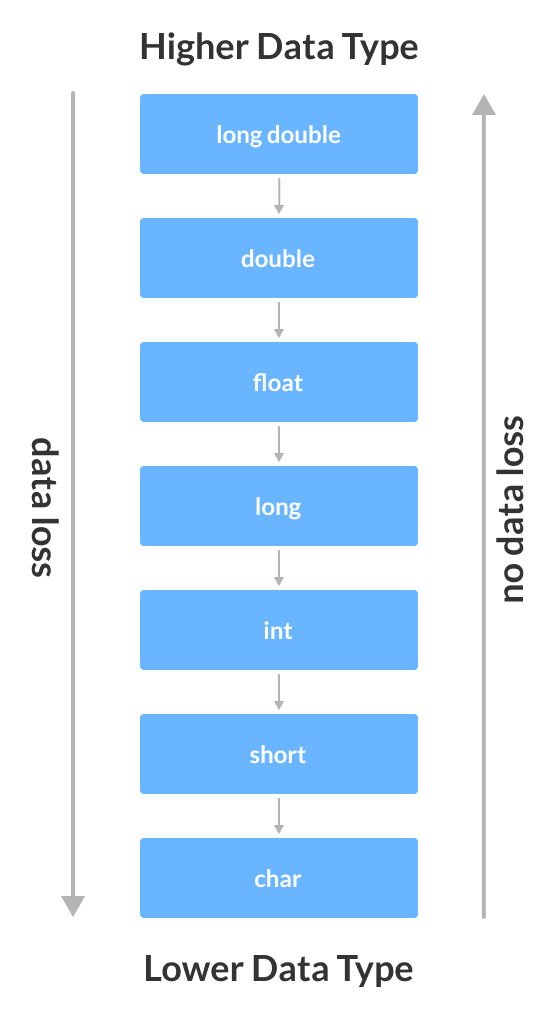
num\_int = num\_double;

Here, the double value is automatically converted to int by the compiler before it is assigned to the num\_int variable. This is also an example of implicit type conversion.

**Note:** Since int cannot have a decimal part, the digits after the decimal point are truncated in the above example.

### Data Loss During Conversion (Narrowing Conversion)

As we have seen from the above example, conversion from one data type to another is prone to data loss. This happens when data of a larger type is converted to data of a smaller type.

Possible Data Loss During Type Conversion

## C++ Explicit Conversion

When the user manually changes data from one type to another, this is known as **explicit conversion**. This type of conversion is also known as **type casting**.

There are three major ways in which we can use explicit conversion in C++. They are:

1. C-style type casting (also known as **cast notation**)
2. Function notation (also known as **old C++ style type casting**)
3. Type conversion operators

### C-style Type Casting

As the name suggests, this type of casting is favored by the **C programming language**. It is also known as **cast notation**.

The syntax for this style is:

(data\_type)expression;

For example,

// initializing int variable

int num\_int = 26;

// declaring double variable

double num\_double;

// converting from int to double

num\_double = (double)num\_int;

### Function-style Casting

We can also use the function like notation to cast data from one type to another.

The syntax for this style is:

data\_type(expression);

For example,

// initializing int variable

int num\_int = 26;

// declaring double variable

double num\_double;

// converting from int to double

num\_double = double(num\_int);

### Example 3: Type Casting

#include <iostream>

using namespace std;

int main() {

// initializing a double variable

double num\_double = 3.56;

cout << "num\_double = " << num\_double << endl;

// C-style conversion from double to int

int num\_int1 = (int)num\_double;

cout << "num\_int1 = " << num\_int1 << endl;

// function-style conversion from double to int

int num\_int2 = int(num\_double);

cout << "num\_int2 = " << num\_int2 << endl;

return 0;

}

**Output**

num\_double = 3.56

num\_int1 = 3

num\_int2 = 3

We used both the **C style type conversion** and the **function-style casting for type conversion** and displayed the results. Since they perform the same task, both give us the same output.

### Type Conversion Operators

Besides these two type castings, C++ also has four operators for type conversion. They are known as **type conversion operators**. They are:

* static\_cast
* dynamic\_cast
* const\_cast
* reinterpret\_cast

# C++ Preprocessor Directive

**WHAT ARE PREPROCESSOR DIRECTIVES?**

* The preprocessor directives are the lines that are included in a program which begins with the # and it is different from the typical source code.
* They are invoked by the compiler to process the programs before compilation. The preprocessor directive is placed at the top of the source code in a separate line beginning with the character # and followed by a directive name.
* The preprocessor directive statement cannot be with semi colon. Some of the examples of preprocessor directives are #define, #include, #indef, etc.

**Let us discuss preprocessor directives types:**

# ****MACROS****

The macros are the piece of code in a program which is given some name and whenever this name occurs then the compiler replaces the name with the  actual code. The #define directive is used to define a macro.

**Let us have a look at the example to understand this:**

#include <iostream>

  using namespace std;

// macro definition

#define range 10

int main()

{

int x;

{

    for (x = 0; x < range; x++)

{

        cout << x << "\n";

    }

    return 0;

}

}

# LET US HAVE A LOOK AT THE EXAMPLE:

#include <iostream>

  using namespace std;

// macro with parameter

#define AREA(l) (l \* l)

int main()

{

    int x = 10, area;

      area = AREA(x);

      cout << "Area of square is: " << area;

      return 0;

}

**FILE INCLUSION**

In this preprocessor directive it tells the compiler to include a file in the source code.

**THERE ARE TWO TYPES OF FILES WHICH CAN BE INCLUDED BY THE USER IN A PROGRAM:**

**1. HEADER FILE**

The header files contains the definition of predefined functions like cin, cout, etc. These files must be included for working with these functions; different functions are declared in different header files. For example functions which perform string operations are in ‘string’ file.

**SYNTAX:**

#include<filename>

In this syntax the filename is the name of the file that to be included. The ‘<’ and ‘>’ brackets tells the compiler to check for the file in the standard directory.

**2. USER DEFINED FILES**

When the program is very large so it is good to divide it into smaller files and include whenever needed. These types of files are user defined files.

**SYNTAX:**

#include “filename”

* **Conditional Compilation:**The conditional compilation directives are type of directives which helps to compile a specific code of the program or to skip compilation of some specific part of the program based on some conditions. We can perform this with the help of two preprocessing commands ifdef and endif.

#ifdef macro\_name

     Statement1;

     Statement2;

     .

     .

     .

     StatementN;

#endif

* **Other Directives:**There are some other directives apart from the directives discussed above. There are two directives which are not used:

1. **#undef Directive**:  It is used to undefine an existing macro.

**SYNTAX:**

#undef LIMIT

By using this statement it will undefine the existing macro LIMIT. After this statement every “ifdef LIMIT” statement will evaluate to false.

1. **#pragma Directive:** This directive is a special directive as it is used to turn on or off some of the features and it is a compiler specific which means it vary from compiler to compiler.

# Namespace in C++

Namespaces permit us to bunch named elements that in any case would have a worldwide degree into smaller extensions, giving them namespace scope. This permits putting together the components of projects into various consistent extensions alluded to by names.

* Namespace is a component included C++ and not present in C.
* A namespace is a revelatory district that gives a degree to the identifiers (names of the kinds, work, factors and so forth) inside it.
* Various namespace blocks with a similar name are permitted. All presentations inside those squares are proclaimed in the named scope.
* Namespaces in C++ are utilized to coordinate such a large number of classes with the goal that it tends to be not difficult to deal with the application.
* For getting to the class of a namespace, we need to utilize namespacename::classname. We can utilize catchphrases with the goal that we don’t need to utilize total names constantly.
* In C++, worldwide namespace is the root namespace. The global::std will consistently allude to the namespace “sexually transmitted disease” of C++ Framework.

#include <iostream>

**using** **namespace** std;

**namespace** First {

**void** sayHello() {

        cout<<"Hello First Namespace"<<endl;

    }

}

**namespace** Second  {

**void** sayHello() {

           cout<<"Hello Second Namespace"<<endl;

       }

}

**int** main()

{

 First::sayHello();

 Second::sayHello();

**return** 0;

}

## **C++ namespace example: by using keyword**

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

#include <iostream>

**using** **namespace** std;

**namespace** First{

**void** sayHello(){

      cout << "Hello First Namespace" << endl;

   }

}

**namespace** Second{

**void** sayHello(){

      cout << "Hello Second Namespace" << endl;

   }

}

**using** **namespace** First;

**int** main () {

   sayHello();

**return** 0;

}

# C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation.**

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation.**

## **I/O Library Header Files**

Let us see the common header files used in C++ programming are:

|  |  |
| --- | --- |
| **Header File** | **Function and Description** |
| <iostream> | It is used to define the **cout, cin and cerr** objects, which correspond to standard output stream, standard input stream and standard error stream, respectively. |
| <iomanip> | It is used to declare services useful for performing formatted I/O, such as **setprecision and setw.** |
| <fstream> | It is used to declare services for user-controlled file processing. |

## **Standard output stream (cout)**

The **cout** is a predefined object of **ostream** class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Let's see the simple example of standard output stream (cout):

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main( ) {
4. **char** ary[] = "Welcome to C++ tutorial";
5. cout << "Value of ary is: " << ary << endl;
6. }

Output:

Value of ary is: Welcome to C++ tutorial

## **Standard input stream (cin)**

The **cin** is a predefined object of **istream** class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

Let's see the simple example of standard input stream (cin):

#include <iostream>

**using** **namespace** std;

**int** main( ) {

**int** age;

   cout << "Enter your age: ";

   cin >> age;

   cout << "Your age is: " << age << endl;

}

Output:

Enter your age: 22

Your age is: 22

# Manipulators in C++

Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream. To access manipulators, the file iomanip.h should be included in the program. In this article, we have shared the list of Manipulators in C++ and functions of manipulators with examples.

## **What are Manipulators in C++?**

manipulators are simply an instruction to the output stream that modify the output in various ways. In other words, we can say that Manipulators are operators that are used to format the data display.

### Advantages and Purpose of Manipulators

* It is mainly used to make up the program structure.
* Manipulators functions are special stream function that changes certain format and characteristics of the input and output.
* To carry out the operations of the manipulators <iomanip.h> must be included.
* Manipulators functions are specially designed to be used in conjunction with insertion (<<) and extraction (>>) operator on stream objects.
* Manipulators are used to changing the format of parameters on streams and to insert or extract certain special characters.

# endl and setw manipulators in C++

* **Manipulators** are functions that are used to format or modify the output stream in various ways.
* They have a special characteristic that they are used along with insertion (<<) operator to change the format of the data.
* There are many manipulators in[C++](https://www.chewcode.com/topic/c/). But as of now, we will be dealing only with the **endl and setw** manipulators.
* To be able to use manipulators in our program, we must include **<iomanip> header file** in our source program.
* But here is an exception – the endl manipulator can be used without including the <iomanip> file.

Now let us see how these two manipulators work.

## **1. endl**

The endl manipulator works the same way as the ‘\n’ character in C++. That is the endl manipulator outputs the subsequent data or text in the next line.

But the difference is that endl also flushes the output buffer when it is used in a program.

Here is a sample program that shows how endl works.

#include<iostream>

#include<conio.h>

using namespace std;

int main()

{

int num;

cout<<"Enter your roll number: ";

cin>>num;

cout<<"Hello roll number "<<num<<endl;

cout<<"Welcome to your new class!!";

getch();

return 0;

}

In the above program, endl operator is used in the fourth statement inside main() function.  
After this statement, the string “Welcome to your new class!!” is printed in the next line on the output screen.

## **2. setw manipulator**

This manipulator is used to set the width of the output in a program. It takes up an argument ‘n’ which is the width of the field in which the output is to be displayed.

By default, the output in the field is right-aligned.

Here is a sample program to show how it works.

#include<iostream>

#include<conio.h>

#include<iomanip>

using namespace std;

int main()

{

int num1,num2,num3;

cout<<"Enter three numbers:\n";

cin>>num1>>num2>>num3;

cout<<"\nDisplaying the three numbers\n"

<<"Num1:"<<setw(8)<<num1<<endl

<<"Num2:"<<setw(8)<<num2<<endl

<<"Num3:"<<setw(8)<<num3<<endl;

getch();

return 0;

}

In the program above, we ask the user to input three integers and then we display them using setw() manipulator.

Notice how we have cascaded the ‘<<‘ operator to display the three numbers. We could have used individual cout statements for displaying each number.

Here is a live preview of this program along with its output. We can see how each of the three numbers is right aligned in a field width of size 8.

### **setfill()**is a function in Manipulators in C++**:**

It replaces setw(whitespaces )’s with a different character. It’s similar to setw() in that it manipulates output, but the only parameter required is a single character. It’s worth noting that a character is contained in single quotes.

setfill(char ch)

**For example**,

cout<< setfill('\*') << endl;

cout << setw(5) << number5 << setw(5) << number6 << endl;

The output of the above will be ‘\*’ character between variable *number5* and variable *number6*.

**Example Program:**

We will use the above setw() example with a little modifications.

#include <cstdlib>

#include <iostream>

#include <iomanip>

using namespace std;

int main()

{

    //variable declaration

int number1, number2, total;

  //variable initialization

number1 = 100;

number2 = 345;

// expression

total = number1 + number2;

//printing output with setw

 cout << endl;

 cout << endl;

cout << setfill('\*') << endl;

cout << setw(5) << number1 << " + " << setw(5) << number2 << " = "  << setw(6) << total << endl;

system("PAUSE");

return EXIT\_SUCCESS;

}

**Output**:

\*\*100 + \*\*345 = \*\*\*445

# C++ Memory Management: new and delete

C++ allows us to allocate the memory of a variable or an array in run time. This is known as dynamic memory allocation.

In other programming languages such as Java and Python, the compiler automatically manages the memories allocated to variables. But this is not the case in C++.

In C++, we need to deallocate the dynamically allocated memory manually after we have no use for the variable.

We can allocate and then deallocate memory dynamically using the new and delete operators respectively.

## **C++ new Operator**

The new operator allocates memory to a variable. For example,

// declare an int pointer

int\* pointVar;

// dynamically allocate memory

// using the new keyword

pointVar = new int;

// assign value to allocated memory

\*pointVar = 45;

Here, we have dynamically allocated memory for an int variable using the new operator.

Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location.

In the case of an array, the new operator returns the address of the first element of the array.

From the example above, we can see that the syntax for using the new operator is

pointerVariable = new dataType;

## **delete Operator**

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable.

For this, the delete operator is used. It returns the memory to the operating system. This is known as **memory deallocation**.

The syntax for this operator is

delete pointerVariable;

Consider the code:

// declare an int pointer

int\* pointVar;

// dynamically allocate memory

// for an int variable

pointVar = new int;

// assign value to the variable memory

\*pointVar = 45;

// print the value stored in memory

cout << \*pointVar; // Output: 45

// deallocate the memory

delete pointVar;

Here, we have dynamically allocated memory for an int variable using the pointer pointVar.

After printing the contents of pointVar, we deallocated the memory using delete.

**Note**: If the program uses a large amount of unwanted memory using new, the system may crash because there will be no memory available for the operating system. In this case, the delete operator can help the system from crash.

## Example 1: C++ Dynamic Memory Allocation

#include <iostream>

using namespace std;

int main() {

// declare an int pointer

int\* pointInt;

// declare a float pointer

float\* pointFloat;

// dynamically allocate memory

pointInt = new int;

pointFloat = new float;

// assigning value to the memory

\*pointInt = 45;

\*pointFloat = 45.45f;

cout << \*pointInt << endl;

cout << \*pointFloat << endl;

// deallocate the memory

delete pointInt;

  delete pointFloat;

return 0;

}

**Output**

45

45.45

In this program, we dynamically allocated memory to two variables of int and float types. After assigning values to them and printing them, we finally deallocate the memories using the code

delete pointInt;

delete pointFloat;

**Note:** Dynamic memory allocation can make memory management more efficient.

Especially for arrays, where a lot of the times we don't know the size of the array until the run time.

## Example 2: C++ new and delete Operator for Arrays

// C++ Program to store GPA of n number of students and display it

// where n is the number of students entered by the user

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter total number of students: ";

cin >> num;

float\* ptr;

// memory allocation of num number of floats

ptr = new float[num];

cout << "Enter GPA of students." << endl;

for (int i = 0; i < num; ++i) {

cout << "Student" << i + 1 << ": ";

cin >> \*(ptr + i);

}

cout << "\nDisplaying GPA of students." << endl;

for (int i = 0; i < num; ++i) {

cout << "Student" << i + 1 << " :" << \*(ptr + i) << endl;

}

// ptr memory is released

delete[] ptr;

return 0;

}

**Output**

Enter total number of students: 4

Enter GPA of students.

Student1: 3.6

Student2: 3.1

Student3: 3.9

Student4: 2.9

Displaying GPA of students.

Student1 :3.6

Student2 :3.1

Student3 :3.9

Student4 :2.9

In this program, we have asked the user to enter the number of students and store it in the num variable.

Then, we have allocated the memory dynamically for the float array using new.

We enter data into the array (and later print them) using pointer notation.

After we no longer need the array, we deallocate the array memory using the code delete[] ptr;.

Notice the use of [] after delete. We use the square brackets [] in order to denote that the memory deallocation is that of an array.

# Statements and flow control

A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (;), and are executed in the same order in which they appear in a program.  
  
But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.

Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple C++ statement, -such as a single instruction, terminated with a semicolon (;) - or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {}:  
  
{ statement1; statement2; statement3; }  
  
The entire block is considered a single statement (composed itself of multiple substatements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

### **Selection statements: if and else**

The if keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:  
  
if (condition) statement  
  
Here, condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is not executed (it is simply ignored), and the program continues right after the entire selection statement.  
For example, the following code fragment prints the message (x is 100), only if the value stored in the x variable is indeed 100:

|  |  |  |
| --- | --- | --- |
| 1 2 | if (x == 100)  cout << "x is 100"; |  |

If x is not exactly 100, this statement is ignored, and nothing is printed.  
If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces ({}), forming a block:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | if (x == 100)  {  cout << "x is ";  cout << x;  } |  |

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

|  |  |  |
| --- | --- | --- |
|  | if (x == 100) { cout << "x is "; cout << x; } |  |

Selection statements with if can also specify what happens when the condition is not fulfilled, by using the else keyword to introduce an alternative statement. Its syntax is:  
  
if (condition) statement1 else statement2   
where statement1 is executed in case condition is true, and in case it is not, statement2 is executed.  
  
For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | if (x == 100)  cout << "x is 100";  else  cout << "x is not 100"; |  |

This prints x is 100, if indeed x has a value of 100, but if it does not, and only if it does not, it prints x is not 100 instead.  
Several if + else structures can be concatenated with the intention of checking a range of values. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 | if (x > 0)  cout << "x is positive";  else if (x < 0)  cout << "x is negative";  else  cout << "x is 0"; |  |

This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: {}.

### Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords while, do, and for.

#### The while loop

The simplest kind of loop is the while-loop. Its syntax is:  
  
while (expression) statement  
  
The while-loop simply repeats statement while expression is true. If, after any execution of statement, expression is no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // custom countdown using while  #include <iostream>  using namespace std;  int main ()  {  int n = 10;  while (n>0) {  cout << n << ", ";  --n;  }  cout << "liftoff!\n";  } | 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff! | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

The first statement in main sets n to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition n>0 (that n is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (n>0) remains being true.  
  
The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. n is assigned a value
2. The while condition is checked (n>0). At this point there are two possibilities:
   * condition is true: the statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:  
   cout << n << ", ";  
   --n;  
   (prints the value of n and decreases n by 1)
4. End of block. Return automatically to step 2.
5. Continue the program right after the block:  
   print liftoff! and end the program.

A thing to consider with while-loops is that the loop should end at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become false at some point. Otherwise, the loop will continue looping forever. In this case, the loop includes --n, that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) false after a certain number of loop iterations. To be more specific, after 10 iterations, n becomes 0, making the condition no longer true, and ending the while-loop.  
  
Note that the complexity of this loop is trivial for a computer, and so the whole countdown is performed instantly, without any practical delay between elements of the count (if interested, see [sleep\_for](https://www.cplusplus.com/sleep_for) for a countdown example with delays).

#### The do-while loop

A very similar loop is the do-while loop, whose syntax is:  
  
do statement while (condition);  
  
It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | // echo machine  #include <iostream>  #include <string>  using namespace std;  int main ()  {  string str;  do {  cout << "Enter text: ";  getline (cin,str);  cout << "You entered: " << str << '\n';  } while (str != "goodbye");  } | Enter text: hello  You entered: hello  Enter text: who's there?  You entered: who's there?  Enter text: goodbye  You entered: goodbye | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

The do-while loop is usually preferred over a while-loop when the statement needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering goodbye, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

#### The for loop

The for loop is designed to iterate a number of times. Its syntax is:  
  
for (initialization; condition; increase) statement;  
  
Like the while-loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization and an increase expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as condition.  
  
It works in the following way:

1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. increase is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 | // countdown using a for loop  #include <iostream>  using namespace std;  int main ()  {  for (int n=10; n>0; n--) {  cout << n << ", ";  }  cout << "liftoff!\n";  } | 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff! | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, for (;n<10;) is a loop without *initialization* or *increase* (equivalent to a while-loop); and for (;n<10;++n) is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with true as condition (i.e., an infinite loop).  
  
Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | for ( n=0, i=100 ; n!=i ; ++n, --i )  {  // whatever here...  } |  |

This loop will execute 50 times if neither n or i are modified within the loop:  
  
  
  
n starts with a value of 0, and i with 100, the condition is n!=i (i.e., that n is not equal to i). Because n is increased by one, and i decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both n and i are equal to 50.

#### Range-based for loop

The for-loop has another syntax, which is used exclusively with ranges:  
  
for ( declaration : range ) statement;  
  
This kind of for loop iterates over all the elements in range, where declaration declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end; Most of these types have not yet been introduced in this tutorial, but we are already acquainted with at least one kind of range: strings, which are sequences of characters.  
  
An example of range-based for loop using strings:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | // range-based for loop  #include <iostream>  #include <string>  using namespace std;  int main ()  {  string str {"Hello!"};  for (char c : str)  {  cout << "[" << c << "]";  }  cout << '\n';  } | [H][e][l][l][o][!] | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

Note how what precedes the colon (:) in the for loop is the declaration of a char variable (the elements in a string are of type char). We then use this variable, c, in the statement block to represent the value of each of the elements in the range.  
  
This loop is automatic and does not require the explicit declaration of any counter variable.  
  
Range based loops usually also make use of type deduction for the type of the elements with auto. Typically, the range-based loop above can also be written as:

|  |  |  |
| --- | --- | --- |
| 1 2 | for (auto c : str)  cout << "[" << c << "]"; |  |

Here, the type of c is automatically deduced as the type of the elements in str.

### Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations.

#### The break statement

break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // break loop example  #include <iostream>  using namespace std;  int main ()  {  for (int n=10; n>0; n--)  {  cout << n << ", ";  if (n==3)  {  cout << "countdown aborted!";  break;  }  }  } | 10, 9, 8, 7, 6, 5, 4, 3, countdown aborted! | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

#### The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 | // continue loop example  #include <iostream>  using namespace std;  int main ()  {  for (int n=10; n>0; n--) {  if (n==5) continue;  cout << n << ", ";  }  cout << "liftoff!\n";  } | 10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff! | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

#### The goto statement

goto allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.  
  
The destination point is identified by a *label*, which is then used as an argument for the goto statement. A *label* is made of a valid identifier followed by a colon (:).  
  
goto is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++. But, just as an example, here is a version of our countdown loop using goto:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | // goto loop example  #include <iostream>  using namespace std;  int main ()  {  int n=10;  mylabel:  cout << n << ", ";  n--;  if (n>0) goto mylabel;  cout << "liftoff!\n";  } | 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff! | [Edit & Run](https://www.cplusplus.com/doc/tutorial/control/) |

### Another selection statement: switch.

The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating if-else statements, but limited to constant expressions. Its most typical syntax is:

switch (expression)

{

case constant1:

group-of-statements-1;

break;

case constant2:

group-of-statements-2;

break;

.

.

.

default:

default-group-of-statements

}

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1; if it is, it executes group-of-statements-1 until it finds the break statement. When it finds this break statement, the program jumps to the end of the entire switch statement (the closing brace).  
  
If expression was not equal to constant1, it is then checked against constant2. If it is equal to this, it executes group-of-statements-2 until a break is found, when it jumps to the end of the switch.  
  
Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the default: label, if it exists (since it is optional).  
  
Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement:

|  |  |
| --- | --- |
| **switch example** | **if-else equivalent** |
| switch (x) {  case 1:  cout << "x is 1";  break;  case 2:  cout << "x is 2";  break;  default:  cout << "value of x unknown";  } | if (x == 1) {  cout << "x is 1";  }  else if (x == 2) {  cout << "x is 2";  }  else {  cout << "value of x unknown";  } |

The switch statement has a somewhat peculiar syntax inherited from the early times of the first C compilers, because it uses labels instead of blocks. In the most typical use (shown above), this means that break statements are needed after each group of statements for a particular label. If break is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement (such as break) is reached.  
  
If the example above lacked the break statement after the first group for case one, the program would not jump automatically to the end of the switch block after printing x is 1, and would instead continue executing the statements in case two (thus printing also x is 2). It would then continue doing so until a break statement is encountered, or the end of the switch block. This makes unnecessary to enclose the statements for each case in braces {}, and can also be useful to execute the same group of statements for different possible values. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 | switch (x) {  case 1:  case 2:  case 3:  cout << "x is 1, 2 or 3";  break;  default:  cout << "x is not 1, 2 nor 3";  } |  |

Notice that switch is limited to compare its evaluated expression against labels that are constant expressions. It is not possible to use variables as labels or ranges, because they are not valid C++ constant expressions.  
  
To check for ranges or values that are not constant, it is better to use concatenations of if and else if statements.

**C++ Function Overloading**

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

// same name different arguments

int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }

Here, all 4 functions are overloaded functions.

Notice that the return types of all these 4 functions are not the same. Overloaded functions may or may not have different return types but they must have different arguments. For example,

// Error code

int test(int a) { }

double test(int b){ }

Here, both functions have the same name, the same type, and the same number of arguments. Hence, the compiler will throw an error.

**Example 1: Overloading Using Different Types of Parameter**

// Program to compute absolute value

// Works for both int and float

#include <iostream>

using namespace std;

// function with float type parameter

float absolute(float var){

if (var < 0.0)

var = -var;

return var;

}

// function with int type parameter

int absolute(int var) {

if (var < 0)

var = -var;

return var;

}

int main() {

// call function with int type parameter

cout << "Absolute value of -5 = " << absolute(-5) << endl;

// call function with float type parameter

cout << "Absolute value of 5.5 = " << absolute(5.5f) << endl;

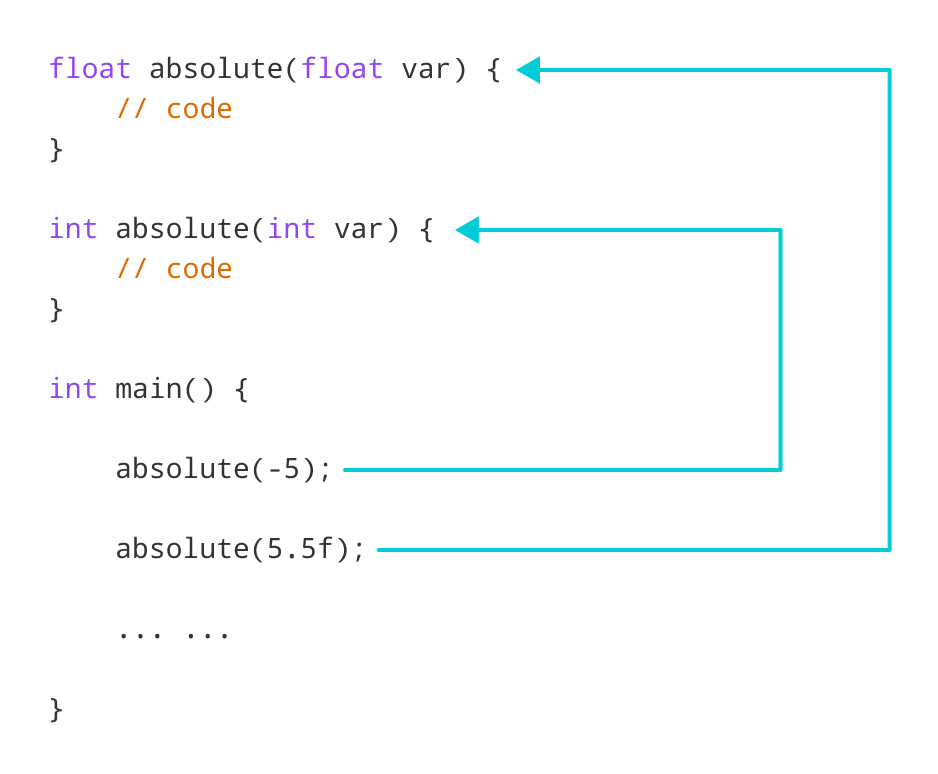
return 0;

}

**Output**

Absolute value of -5 = 5

Absolute value of 5.5 = 5.5



Working of overloading for the absolute() function

In this program, we overload the absolute() function. Based on the type of parameter passed during the function call, the corresponding function is called.

**Example 2: Overloading Using Different Number of Parameters**

#include <iostream>

using namespace std;

// function with 2 parameters

void display(int var1, double var2) {

cout << "Integer number: " << var1;

cout << " and double number: " << var2 << endl;

}

// function with double type single parameter

void display(double var) {

cout << "Double number: " << var << endl;

}

// function with int type single parameter

void display(int var) {

cout << "Integer number: " << var << endl;

}

int main() {

int a = 5;

double b = 5.5;

// call function with int type parameter

display(a);

// call function with double type parameter

display(b);

// call function with 2 parameters

display(a, b);

return 0;

}

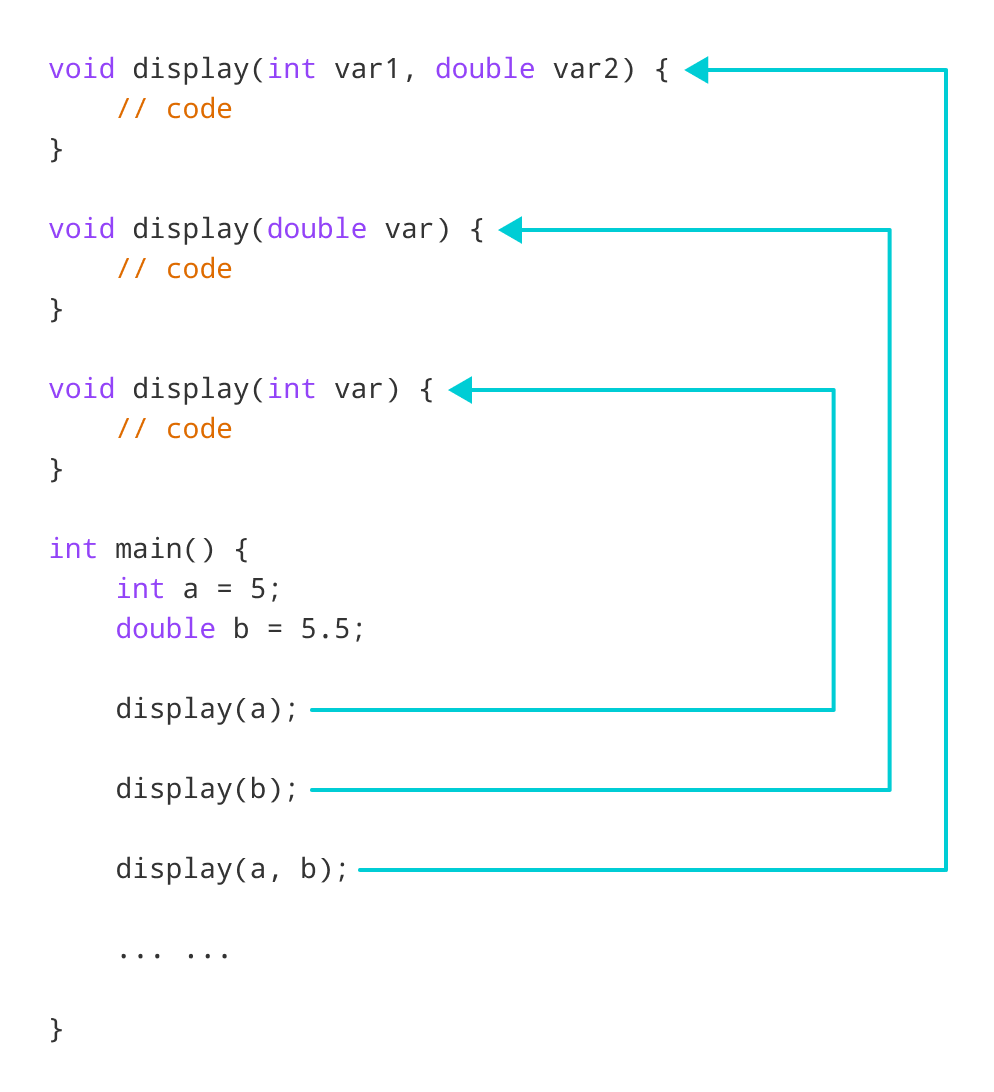
**Output**

Integer number: 5

Float number: 5.5

Integer number: 5 and double number: 5.5

Here, the display() function is called three times with different arguments. Depending on the number and type of arguments passed, the corresponding display() function is called.



Working of overloading for the display() function

The return type of all these functions is the same but that need not be the case for function overloading.

# C++ Inline Functions

In this tutorial, we will learn about inline functions in C++ and how to use them with the help of examples.

In C++, we can declare a function as inline. This copies the function to the location of the function call in compile-time and may make the program execution faster.

## Inline Functions

To create an inline function, we use the inline keyword. For example,

inline returnType functionName(parameters) {

// code

}

Notice the use of keyword inline before the function definition.

## C++ Inline Function

#include <iostream>

using namespace std;

inline void displayNum(int num) {

cout << num << endl;

}

int main() {

// first function call

displayNum(5);

// second function call

displayNum(8);

// third function call

displayNum(666);

return 0;

}

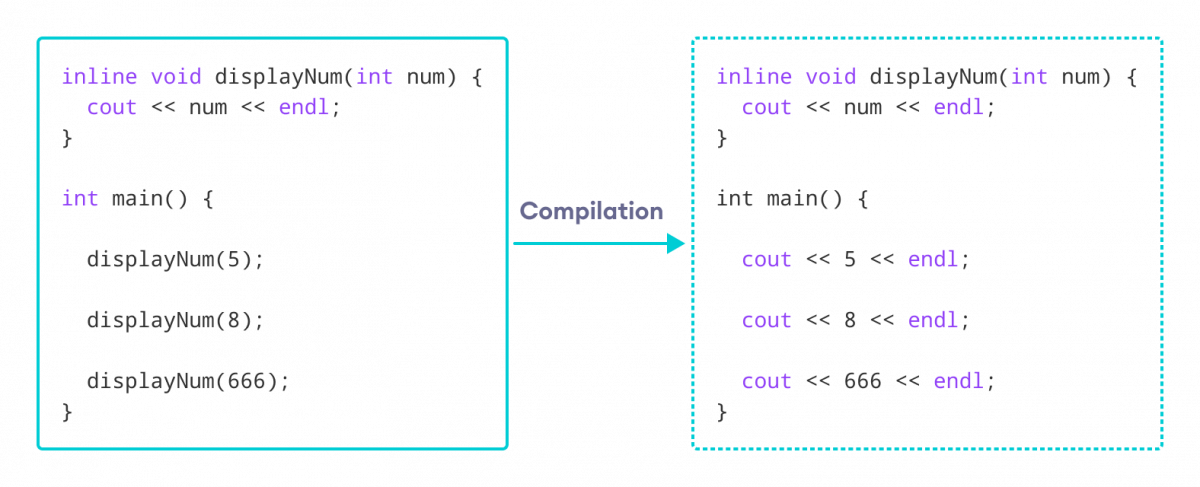
**Output**

5

8

666

Here is how this program works:

Working of inline functions in C++

Here, we created an inline function named displayNum() that takes a single integer as a parameter.

We then called the function 3 times in the main() function with different arguments. Each time displayNum() is called, the compiler copies the code of the function to that call location.

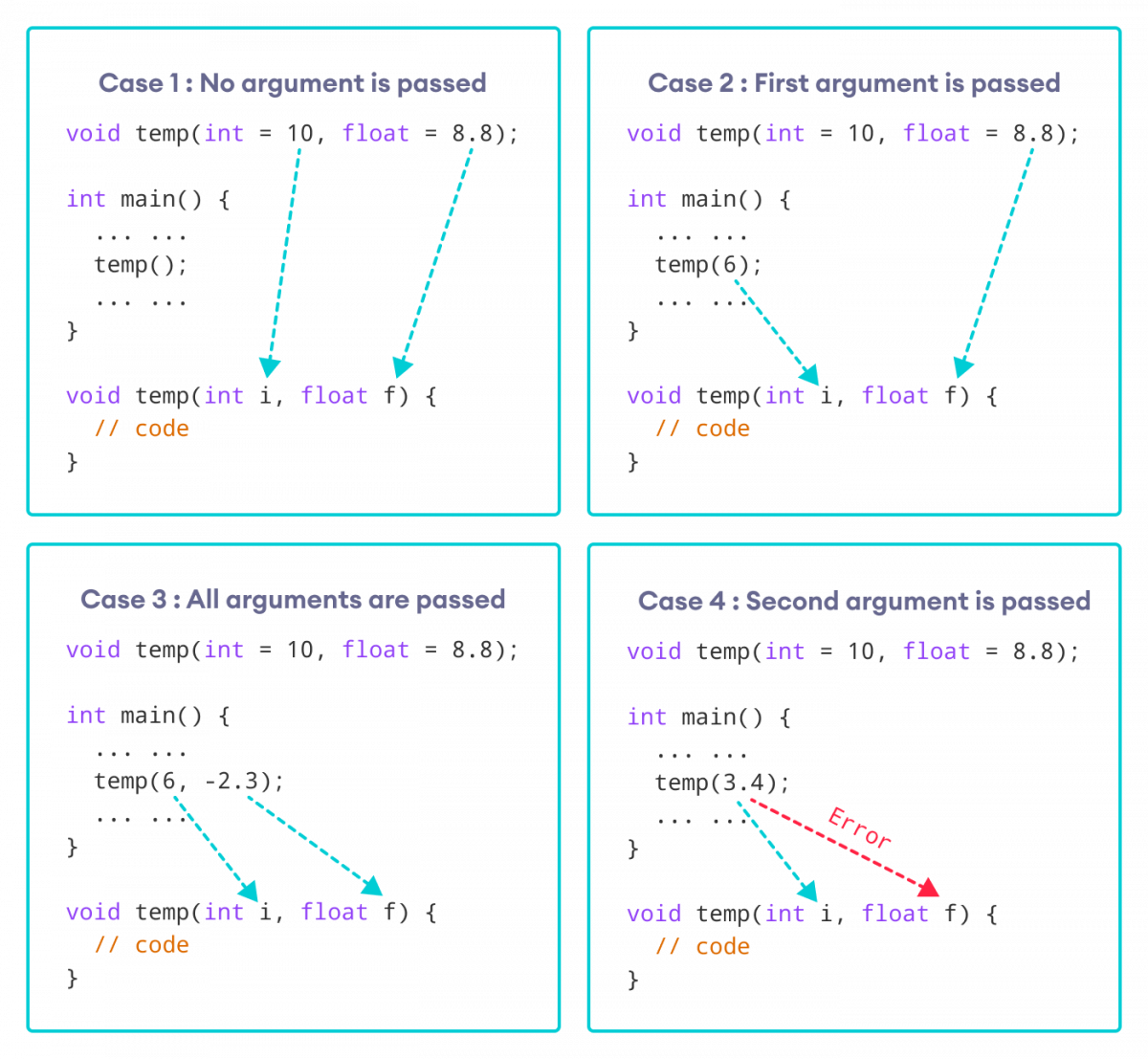
# C++ Programming Default Arguments (Parameters)

In C++ programming, we can provide default values for [function](https://www.programiz.com/cpp-programming/function) parameters.

If a function with default arguments is called without passing arguments, then the default parameters are used.

However, if arguments are passed while calling the function, the default arguments are ignored.

## Working of default arguments

How default arguments work in C++

We can understand the working of default arguments from the image above:

1. When temp() is called, both the default parameters are used by the function.
2. When temp(6) is called, the first argument becomes 6 while the default value is used for the second parameter.
3. When temp(6, -2.3) is called, both the default parameters are overridden, resulting in i = 6 and f = -2.3.
4. When temp(3.4) is passed, the function behaves in an undesired way because the second argument cannot be passed without passing the first argument.  
     
   Therefore, 3.4 is passed as the first argument. Since the first argument has been defined as int, the value that is actually passed is 3.

## Example: Default Argument

#include <iostream>

using namespace std;

// defining the default arguments

void display(char = '\*', int = 3);

int main() {

int count = 5;

cout << "No argument passed: ";

// \*, 3 will be parameters

display();

cout << "First argument passed: ";

// #, 3 will be parameters

display('#');

cout << "Both arguments passed: ";

// $, 5 will be parameters

display('$', count);

return 0;

}

void display(char c, int count) {

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

{

cout << c;

}

cout << endl;

}

**Output**

No argument passed: \*\*\*

First argument passed: ###

Both arguments passed: $$$$$

Here is how this program works:

1. display() is called without passing any arguments. In this case, display() uses both the default parameters c = '\*' and n = 1.
2. display('#') is called with only one argument. In this case, the first becomes '#'. The second default parameter n = 1 is retained.
3. display('#', count) is called with both arguments. In this case, default arguments are not used.

We can also define the default parameters in the function definition itself. The program below is equivalent to the one above.

#include <iostream>

using namespace std;

// defining the default arguments

void display(char c = '\*', int count = 3) {

for(int i = 1; i <= count; ++i) {

cout << c;

}

cout << endl;

}

int main() {

int count = 5;

cout << "No argument passed: ";

// \*, 3 will be parameters

display();

cout << "First argument passed: ";

// #, 3 will be parameters

display('#');

cout << "Both argument passed: ";

// $, 5 will be parameters

display('$', count);

return 0;

}

## Things to Remember

1. Once we provide a default value for a parameter, all subsequent parameters must also have default values. For example,
2. // Invalid
3. void add(int a, int b = 3, int c, int d);
4. // Invalid
5. void add(int a, int b = 3, int c, int d = 4);
6. // Valid

void add(int a, int c, int b = 3, int d = 4);

1. If we are defining the default arguments in the function definition instead of the function prototype, then the function must be defined before the function call.
2. // Invalid code
3. int main() {
4. // function call
5. display();
6. }
7. void display(char c = '\*', int count = 5) {

// code

}

# C++ Call by Reference: Using pointers

This method used is called passing by value because the actual value is passed.

However, there is another way of passing arguments to a function where the actual values of arguments are not passed. Instead, the reference to values is passed.

For example,

// function that takes value as parameter

void func1(int numVal) {

// code

}

// function that takes reference as parameter

// notice the & before the parameter

void func2(int &numRef) {

// code

}

int main() {

int num = 5;

// pass by value

func1(num);

// pass by reference

func2(num);

return 0;

}

Notice the & in void func2(int &numRef). This denotes that we are using the address of the variable as our parameter.

So, when we call the func2() function in main() by passing the variable num as an argument, we are actually passing the address of num variable instead of the value **5**.



C++ Pass by Value vs. Pass by Reference

## Example 1: Passing by reference without pointers

#include <iostream>

using namespace std;

// function definition to swap values

void swap(int &n1, int &n2) {

int temp;

temp = n1;

n1 = n2;

n2 = temp;

}

int main()

{

// initialize variables

int a = 1, b = 2;

cout << "Before swapping" << endl;

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

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

// call function to swap numbers

swap(a, b);

cout << "\nAfter swapping" << endl;

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

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

return 0;

}

**Output**

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

In this program, we passed the variables a and b to the swap() function. Notice the function definition,

void swap(int &n1, int &n2)

Here, we are using & to denote that the function will accept addresses as its parameters.

Hence, the compiler can identify that instead of actual values, the reference of the variables is passed to function parameters.

In the swap() function, the function parameters n1 and n2 are pointing to the same value as the variables a and b respectively. Hence the swapping takes place on actual value.

The same task can be done using the pointers.

## Example 2: Passing by reference using pointers

#include <iostream>

using namespace std;

// function prototype with pointer as parameters

void swap(int\*, int\*);

int main()

{

// initialize variables

int a = 1, b = 2;

cout << "Before swapping" << endl;

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

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

// call function by passing variable addresses

swap(&a, &b);

cout << "\nAfter swapping" << endl;

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

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

return 0;

}

// function definition to swap numbers

void swap(int\* n1, int\* n2) {

int temp;

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

}

**Output**

Before swapping

a = 1

b = 2

After swapping

a = 2

b = 1

Here, we can see the output is the same as the previous example. Notice the line,

// &a is address of a

// &b is address of b

swap(&a, &b);

Here, the address of the variable is passed during the function call rather than the variable.

Since the address is passed instead of value, a dereference operator \* must be used to access the value stored in that address.

temp = \*n1;

\*n1 = \*n2;

\*n2 = temp;

\*n1 and \*n2 gives the value stored at address n1 and n2 respectively.

Since n1 and n2 contain the addresses of a and b, anything is done to \*n1 and \*n2 will change the actual values of a and b.

Hence, when we print the values of a and b in the main() function, the values are changed.

# C++ Return by Reference

In C++ Programming, not only can you pass values by reference to a [function](https://www.programiz.com/cpp-programming/function) but you can also return a value by reference.

To understand this feature, you should have the knowledge of:

* [Global variables](https://www.programiz.com/cpp-programming/storage-class#global_variable)

## Example: Return by Reference

#include <iostream>

using namespace std;

// global variable

int num;

// function declaration

int& test();

int main() {

  // assign 5 to num variable

  // equivalent to num = 5;

test() = 5;

cout << num;

return 0;

}

// function definition

// returns the address of num variable

int& test() {

return num;

}

**Output**

5

In program above, the return type of function test() is int&. Hence, this function returns a reference of the variable num.

The return statement is return num;. Unlike return by value, this statement doesn't return value of num, instead it returns the variable itself (address).

So, when the **variable** is returned, it can be assigned a value as done in test() = 5;

This stores **5** to the variable num, which is displayed onto the screen.

### Important Things to Remember When Returning by Reference.

* Ordinary function returns value but this function doesn't. Hence, you cannot return a constant from the function.
* int& test() {
* return 2;

}

* You cannot return a local variable from this function.
* int& test() {
* int n = 2;
* return n;

}

# Storage Classes in C++

## Scope and Visibility

### Scope

The scope of an identifier is a part of the program in which the identifier can be used to access its object. There are different categories of scope: block (or local), function, function prototype, and file. These categories depend on how and where identifiers are declared.

* **Block**: The scope of an identifier with block (or local) scope starts at the declaration point and ends at the end of the block containing the declaration (such block is known as the enclosing block). Parameter declarations with a function definition also have block scope, limited to the scope of the function body.
* **File**: File scope identifiers, also known as globals, are declared outside of all blocks; their scope is from the point of declaration to the end of the source file.
* **Function**: The only identifiers having function scope are statement labels. Label names can be used with goto statements anywhere in the function in which the label is declared. Labels are declared implicitly by writing label\_name: followed by a statement. Label names must be unique within a function.
* **Function prototype**: Identifiers declared within the list of parameter declarations in a function prototype (not as a part of a function definition) have a function prototype scope. This scope ends at the end of the function prototype.

### Visibility

The visibility of an identifier is a region of the program source code from which an identifier’s associated object can be legally accessed.

Scope and visibility usually coincide, though there are circumstances under which an object becomes temporarily hidden by the appearance of a duplicate identifier: the object still exists but the original identifier cannot be used to access it until the scope of the duplicate identifier ends.

Technically, visibility cannot exceed a scope, but a scope can exceed visibility. See the following example:

**void** f (**int** i) {

**int** j; *// auto by default*

j = 3; *// int i and j are in scope and visible*

{ *// nested block*

**double** j; *// j is local name in the nested block*

j = 0.1; *// i and double j are visible;*

*// int j = 3 in scope but hidden*

}

*// double j out of scope*

j += 1; *// int j visible and = 4*

}

*// i and j are both out of scope*

# C++ Storage Class

Every variable in C++ has two features: type and storage class. Type specifies the type of data that can be stored in a variable. For example: int, float, char etc.

And, storage class controls two different properties of a variable: lifetime (determines how long a variable can exist) and scope (determines which part of the program can access it).

Depending upon the storage class of a variable, it can be divided into 4 major types:

* [Local variable](https://www.programiz.com/cpp-programming/storage-class#local_variable)
* [Global variable](https://www.programiz.com/cpp-programming/storage-class#global_variable)
* [Static local variable](https://www.programiz.com/cpp-programming/storage-class#static_variable)
* [Register Variable](https://www.programiz.com/cpp-programming/storage-class#register%20variable)
* [Thread Local Storage](https://www.programiz.com/cpp-programming/storage-class#thread_local_storage)

## Local Variable

A variable defined inside a function (defined inside [function](https://www.programiz.com/cpp-programming/function) body between braces) is called a local variable or automatic variable.

Its scope is only limited to the function where it is defined. In simple terms, local variable exists and can be accessed only inside a function.

The life of a local variable ends (It is destroyed) when the function exits.

### Example 1: Local variable

#include <iostream>

using namespace std;

void test();

int main()

{

// local variable to main()

int var = 5;

test();

// illegal: var1 not declared inside main()

var1 = 9;

}

void test()

{

// local variable to test()

int var1;

var1 = 6;

// illegal: var not declared inside test()

cout << var;

}

The variable var cannot be used inside test() and var1 cannot be used inside main() function.

Keyword auto was also used for defining local variables before as: auto int var;

But, after [C++11 auto](http://en.cppreference.com/w/cpp/language/auto) has a different meaning and should not be used for defining local variables.

## Global Variable

If a variable is defined outside all functions, then it is called a global variable.

The scope of a global variable is the whole program. This means, It can be used and changed at any part of the program after its declaration.

Likewise, its life ends only when the program ends.

### Example 2: Global variable

#include <iostream>

using namespace std;

// Global variable declaration

int c = 12;

void test();

int main()

{

++c;

// Outputs 13

cout << c <<endl;

test();

return 0;

}

void test()

{

++c;

// Outputs 14

cout << c;

}

**Output**

13

14

In the above program, c is a global variable.

This variable is visible to both functions main() and test() in the above program.

### Static Local variable

Keyword static is used for specifying a static variable. For example:

... .. ...

int main()

{

static float a;

... .. ...

}

A static local variable exists only inside a function where it is declared (similar to a local variable) but its lifetime starts when the function is called and ends only when the program ends.

The main difference between local variable and static variable is that, the value of static variable persists the end of the program.

### Example 3: Static local variable

#include <iostream>

using namespace std;

void test()

{

// var is a static variable

static int var = 0;

++var;

cout << var << endl;

}

int main()

{

test();

test();

return 0;

}

**Output**

1

2

In the above program, test() function is invoked 2 times.

During the first call, variable var is declared as static variable and initialized to 0. Then 1 is added to var which is displayed in the screen.

When the function test() returns, variable var still exists because it is a static variable.

During second function call, no new variable var is created. The same var is increased by 1 and then displayed to the screen.

**Output of above program if var was not specified as static variable**

1

1

## Register Variable (Deprecated in C++11)

Keyword register is used for specifying register variables.

Register variables are similar to automatic variables and exists inside a particular function only. It is supposed to be faster than the local variables.

If a program encounters a register variable, it stores the variable in processor's register rather than memory if available. This makes it faster than the local variables.

However, this keyword was deprecated in C++11 and should not be used.

# C++ Pointers

In this tutorial, we will learn about pointers in C++ and their working with the help of examples.

In C++, pointers are variables that store the memory addresses of other variables.

## Address in C++

If we have a variable var in our program, &var will give us its address in the memory. For example,

### Example 1: Printing Variable Addresses in C++

#include <iostream>

using namespace std;

int main()

{

// declare variables

int var1 = 3;

int var2 = 24;

int var3 = 17;

// print address of var1

cout << "Address of var1: "<< &var1 << endl;

// print address of var2

cout << "Address of var2: " << &var2 << endl;

// print address of var3

cout << "Address of var3: " << &var3 << endl;

}

**Output**

Address of var1: 0x7fff5fbff8ac

Address of var2: 0x7fff5fbff8a8

Address of var3: 0x7fff5fbff8a4

Here, 0x at the beginning represents the address is in the hexadecimal form.

Notice that the first address differs from the second by 4 bytes and the second address differs from the third by 4 bytes.

This is because the size of an int variable is 4 bytes in a 64-bit system.

**Note:**You may not get the same results when you run the program.

## C++ Pointers

As mentioned above, pointers are used to store addresses rather than values.

Here is how we can declare pointers.

int \*pointVar;

Here, we have declared a pointer pointVar of the int type.

We can also declare pointers in the following way.

int\* pointVar; // preferred syntax

Let's take another example of declaring pointers.

int\* pointVar, p;

Here, we have declared a pointer pointVar and a normal variable p.  
  
**Note:** The \* operator is used after the data type to declare pointers.

### Assigning Addresses to Pointers

Here is how we can assign addresses to pointers:

int\* pointVar, var;

var = 5;

// assign address of var to pointVar pointer

pointVar = &var;

Here, 5 is assigned to the variable var. And, the address of var is assigned to the pointVar pointer with the code pointVar = &var.

### Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the \* operator. For example:

int\* pointVar, var;

var = 5;

// assign address of var to pointVar

pointVar = &var;

// access value pointed by pointVar

cout << \*pointVar << endl; // Output: 5

In the above code, the address of var is assigned to pointVar. We have used the \*pointVar to get the value stored in that address.

When \* is used with pointers, it's called the **dereference operator**. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, \*pointVar = var.

**Note: In C++,** pointVar and \*pointVar is completely different. We cannot do something like \*pointVar = &var;

### Example 2: Working of C++ Pointers

#include <iostream>

using namespace std;

int main() {

int var = 5;

// declare pointer variable

int\* pointVar;

// store address of var

pointVar = &var;

// print value of var

cout << "var = " << var << endl;

// print address of var

cout << "Address of var (&var) = " << &var << endl

<< endl;

// print pointer pointVar

cout << "pointVar = " << pointVar << endl;

// print the content of the address pointVar points to

cout << "Content of the address pointed to by pointVar (\*pointVar) = " << \*pointVar << endl;

return 0;

}

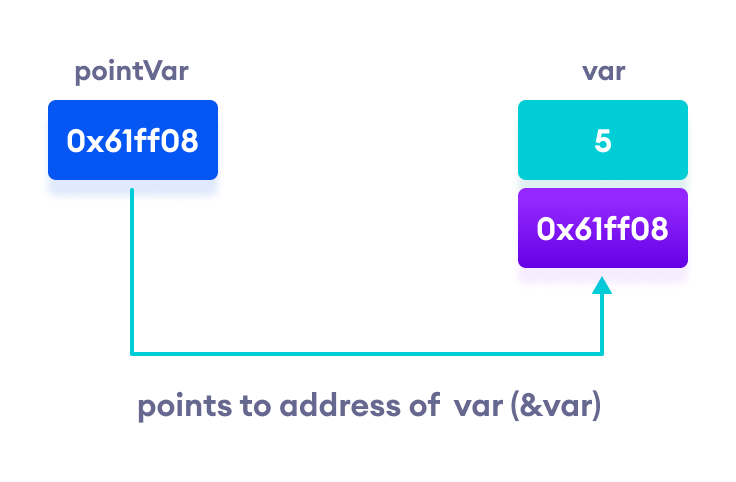
**Output**

var = 5

Address of var (&var) = 0x61ff08

pointVar = 0x61ff08

Content of the address pointed to by pointVar (\*pointVar) = 5

Working of C++ pointers

### Changing Value Pointed by Pointers

If pointVar points to the address of var, we can change the value of var by using \*pointVar.

For example,

int var = 5;

int\* pointVar;

// assign address of var

pointVar = &var;

// change value at address pointVar

\*pointVar = 1;

cout << var << endl; // Output: 1

Here, pointVar and &var have the same address, the value of var will also be changed when \*pointVar is changed.

### Example 3: Changing Value Pointed by Pointers

#include <iostream>

using namespace std;

int main() {

int var = 5;

int\* pointVar;

// store address of var

pointVar = &var;

// print var

cout << "var = " << var << endl;

// print \*pointVar

cout << "\*pointVar = " << \*pointVar << endl

<< endl;

cout << "Changing value of var to 7:" << endl;

// change value of var to 7

var = 7;

// print var

cout << "var = " << var << endl;

// print \*pointVar

cout << "\*pointVar = " << \*pointVar << endl

<< endl;

cout << "Changing value of \*pointVar to 16:" << endl;

// change value of var to 16

\*pointVar = 16;

// print var

cout << "var = " << var << endl;

// print \*pointVar

cout << "\*pointVar = " << \*pointVar << endl;

return 0;

}

**Output**

var = 5

\*pointVar = 5

Changing value of var to 7:

var = 7

\*pointVar = 7

Changing value of \*pointVar to 16:

var = 16

\*pointVar = 16

## Common mistakes when working with pointers

Suppose, we want a pointer varPoint to point to the address of var. Then,

int var, \*varPoint;

// Wrong!

// varPoint is an address but var is not

varPoint = var;

// Wrong!

// &var is an address

// \*varPoint is the value stored in &var

\*varPoint = &var;

// Correct!

// varPoint is an address and so is &var

varPoint = &var;

// Correct!

// both \*varPoint and var are values

\*varPoint = var;