Lập trình

**C++**

# OVERVIEW

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

**Note** − A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time.

## **Object-Oriented Programming**

C++ fully supports object-oriented programming, including the four pillars of object-oriented development −

* Encapsulation
* Data hiding
* Inheritance
* Polymorphism

## **Standard Libraries**

Standard C++ consists of three important parts −

* The core language giving all the building blocks including variables, data types and literals, etc.
* The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
* The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

## **The ANSI Standard**

The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.

The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

## **Learning C++**

The most important thing while learning C++ is to focus on concepts.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

## **Use of C++**

C++ is used by hundreds of thousands of programmers in essentially every application domain.

C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under realtime constraints.

C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.

Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.

# ENVIRONMENT SETUP

## **Local Environment Setup**

If you are still willing to set up your environment for C++, you need to have the following two softwares on your computer.

### **Text Editor**

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.

The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.

A text editor should be in place to start your C++ programming.

### **C++ Compiler**

This is an actual C++ compiler, which will be used to compile your source code into final executable program.

Most C++ compilers don't care what extension you give to your source code, but if you don't specify otherwise, many will use .cpp by default.

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective Operating Systems.

## **Installing GNU C/C++ Compiler**

### **UNIX/Linux Installation**

If you are using **Linux or UNIX** then check whether GCC is installed on your system by entering the following command from the command line −

$ g++ -v

If you have installed GCC, then it should print a message such as the following −

Using built-in specs.

Target: i386-redhat-linux

Configured with: ../configure --prefix=/usr .......

Thread model: posix

gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)

If GCC is not installed, then you will have to install it yourself using the detailed instructions available at <https://gcc.gnu.org/install/>

### **Mac OS X Installation**

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's website and follow the simple installation instructions.

Xcode is currently available at [developer.apple.com/technologies/tools/](https://developer.apple.com/technologies/tools/).

### **Windows Installation**

To install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, [www.mingw.org](http://www.mingw.org/), and follow the link to the MinGW download page. Download the latest version of the MinGW installation program which should be named MinGW-<version>.exe.

While installing MinGW, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your **PATH** environment variable so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

# BASIC SYNTAX

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

* **Object** − Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
* **Class** − A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.
* **Methods** − A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
* **Instance Variables** − Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

## **C++ Program Structure**

Let us look at a simple code that would print the words *Hello World*.

[Live Demo](http://tpcg.io/n4BVuS)

#include <iostream>

using namespace std;

// main() is where program execution begins.

int main() {

cout << "Hello World"; // prints Hello World

return 0;

}

Let us look at the various parts of the above program −

* The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
* The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
* The next line '**// main() is where program execution begins.**' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
* The line **int main()** is the main function where program execution begins.
* The next line **cout << "Hello World";** causes the message "Hello World" to be displayed on the screen.
* The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

## **Compile and Execute C++ Program**

Let's look at how to save the file, compile and run the program. Please follow the steps given below −

* Open a text editor and add the code as above.
* Save the file as: hello.cpp
* Open a command prompt and go to the directory where you saved the file.
* Type 'g++ hello.cpp' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line and would generate a.out executable file.
* Now, type 'a.out' to run your program.
* You will be able to see ' Hello World ' printed on the window.

$ g++ hello.cpp

$ ./a.out

Hello World

Make sure that g++ is in your path and that you are running it in the directory containing file hello.cpp.

You can compile C/C++ programs using makefile. For more details, you can check our ['Makefile Tutorial'](https://www.tutorialspoint.com/makefile/index.htm).

## **Semicolons and Blocks in C++**

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are three different statements −

x = y;

y = y + 1;

add(x, y);

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example −

{

cout << "Hello World"; // prints Hello World

return 0;

}

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example −

x = y;

y = y + 1;

add(x, y);

is the same as

x = y; y = y + 1; add(x, y);

## **C++ Identifiers**

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers −

mohd zara abc move\_name a\_123

myname50 \_temp j a23b9 retVal

## **C++ Keywords**

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

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

## **Trigraphs**

A few characters have an alternative representation, called a trigraph sequence. A trigraph is a three-character sequence that represents a single character and the sequence always starts with two question marks.

Trigraphs are expanded anywhere they appear, including within string literals and character literals, in comments, and in preprocessor directives.

Following are most frequently used trigraph sequences −

|  |  |
| --- | --- |
| **Trigraph** | **Replacement** |
| ??= | # |
| ??/ | \ |
| ??' | ^ |
| ??( | [ |
| ??) | ] |
| ??! | | |
| ??< | { |
| ??> | } |
| ??- | ~ |

All the compilers do not support trigraphs and they are not advised to be used because of their confusing nature.

## **Whitespace in C++**

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it.

Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins.

### **Statement 1**

int age;

In the above statement there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

### **Statement 2**

fruit = apples + oranges; // Get the total fruit

In the above statement 2, no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

# COMMENTS

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /\* and end with \*/. For example −

/\* This is a comment \*/

/\* C++ comments can also

\* span multiple lines

\*/

A comment can also start with //, extending to the end of the line. For example −

[Live Demo](http://tpcg.io/Q4esaC)

#include <iostream>

using namespace std;

main() {

cout << "Hello World"; // prints Hello World

return 0;

}

When the above code is compiled, it will ignore **// prints Hello World** and final executable will produce the following result −

Hello World

Within a /\* and \*/ comment, // characters have no special meaning. Within a // comment, /\* and \*/ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example −

/\* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

\*/

# DATA TYPES

While writing program in any language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

## **Primitive Built-in Types**

C++ offers the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types −

|  |  |
| --- | --- |
| **Type** | **Keyword** |
| Boolean | bool |
| Character | char |
| Integer | int |
| Floating point | float |
| Double floating point | double |
| Valueless | void |
| Wide character | wchar\_t |

Several of the basic types can be modified using one or more of these type modifiers −

* signed
* unsigned
* short
* long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

|  |  |  |
| --- | --- | --- |
| **Type** | **Typical Bit Width** | **Typical Range** |
| char | 1byte | -127 to 127 or 0 to 255 |
| unsigned char | 1byte | 0 to 255 |
| signed char | 1byte | -127 to 127 |
| int | 4bytes | -2147483648 to 2147483647 |
| unsigned int | 4bytes | 0 to 4294967295 |
| signed int | 4bytes | -2147483648 to 2147483647 |
| short int | 2bytes | -32768 to 32767 |
| unsigned short int | 2bytes | 0 to 65,535 |
| signed short int | 2bytes | -32768 to 32767 |
| long int | 8bytes | -9223372036854775808 to 9223372036854775807 |
| signed long int | 8bytes | same as long int |
| unsigned long int | 8bytes | 0 to 18446744073709551615 |
| long long int | 8bytes | -(2^63) to (2^63)-1 |
| unsigned long long int | 8bytes | 0 to 18,446,744,073,709,551,615 |
| float | 4bytes |  |
| double | 8bytes |  |
| long double | 12bytes |  |
| wchar\_t | 2 or 4 bytes | 1 wide character |

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

Following is the example, which will produce correct size of various data types on your computer.

[Live Demo](http://tpcg.io/iKNn78)

#include <iostream>

using namespace std;

int main() {

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

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

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

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

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

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

cout << "Size of wchar\_t : " << sizeof(wchar\_t) << endl;

return 0;

}

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof()** operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine −

Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 4

Size of float : 4

Size of double : 8

Size of wchar\_t : 4

Following is another example:

#include <iostream>

#include <limits>

using namespace std;

int main() {

std::cout << "Int Min " << std::numeric\_limits<int>::min() << endl;

std::cout << "Int Max " << std::numeric\_limits<int>::max() << endl;

std::cout << "Unsigned Int Min " << std::numeric\_limits<unsigned int>::min() << endl;

std::cout << "Unsigned Int Max " << std::numeric\_limits<unsigned int>::max() << endl;

std::cout << "Long Int Min " << std::numeric\_limits<long int>::min() << endl;

std::cout << "Long Int Max " << std::numeric\_limits<long int>::max() << endl;

std::cout << "Unsigned Long Int Min " << std::numeric\_limits<unsigned long int>::min() <<endl;

std::cout << "Unsigned Long Int Max " << std::numeric\_limits<unsigned long int>::max() << endl;

}

## **typedef Declarations**

You can create a new name for an existing type using **typedef**. Following is the simple syntax to define a new type using typedef −

typedef type newname;

For example, the following tells the compiler that feet is another name for int −

typedef int feet;

Now, the following declaration is perfectly legal and creates an integer variable called distance −

feet distance;

## **Enumerated Types**

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration.

Creating an enumeration requires the use of the keyword **enum**. The general form of an enumeration type is −

enum enum-name { list of names } var-list;

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called colors and the variable c of type color. Finally, c is assigned the value "blue".

enum color { red, green, blue } c;

c = blue;

By default, the value of the first name is 0, the second name has the value 1, and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, **green** will have the value 5.

enum color { red, green = 5, blue };

Here, **blue** will have a value of 6 because each name will be one greater than the one that precedes it.

# KIỂU BIẾN

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive −

There are following basic types of variable in C++ as explained in last chapter −

|  |  |
| --- | --- |
| **Sr.No** | **Type & Description** |
| 1 | **bool**  Stores either value true or false. |
| 2 | **char**  Typically a single octet (one byte). This is an integer type. |
| 3 | **int**  The most natural size of integer for the machine. |
| 4 | **float**  A single-precision floating point value. |
| 5 | **double**  A double-precision floating point value. |
| 6 | **void**  Represents the absence of type. |
| 7 | **wchar\_t**  A wide character type. |

C++ also allows to define various other types of variables, which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures,** and **Classes**.

Following section will cover how to define, declare and use various types of variables.

## **Variable Definition in C++**

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type, and contains a list of one or more variables of that type as follows −

type variable\_list;

Here, **type** must be a valid C++ data type including char, w\_char, int, float, double, bool or any user-defined object, etc., and **variable\_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here −

int i, j, k;

char c, ch;

float f, salary;

double d;

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows −

type variable\_name = value;

Some examples are −

extern int d = 3, f = 5; // declaration of d and f.

int d = 3, f = 5; // definition and initializing d and f.

byte z = 22; // definition and initializes z.

char x = 'x'; // the variable x has the value 'x'.

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

## **Variable Declaration in C++**

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use **extern** keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

### **Example**

Try the following example where a variable has been declared at the top, but it has been defined inside the main function −

[Live Demo](http://tpcg.io/odXXRO)

#include <iostream>

using namespace std;

// Variable declaration:

extern int a, b;

extern int c;

extern float f;

int main () {

// Variable definition:

int a, b;

int c;

float f;

// actual initialization

a = 10;

b = 20;

c = a + b;

cout << c << endl ;

f = 70.0/3.0;

cout << f << endl ;

return 0;

}

When the above code is compiled and executed, it produces the following result −

30

23.3333

Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example −

// function declaration

int func();

int main() {

// function call

int i = func();

}

// function definition

int func() {

return 0;

}

## **Lvalues and Rvalues**

There are two kinds of expressions in C++ −

* **lvalue** − Expressions that refer to a memory location is called "lvalue" expression. An lvalue may appear as either the left-hand or right-hand side of an assignment.
* **rvalue** − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement −

int g = 20;

But the following is not a valid statement and would generate compile-time error −

10 = 20;

# PHẠM VI CỦA BIẾN

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

* Inside a function or a block which is called local variables,
* In the definition of function parameters which is called formal parameters.
* Outside of all functions which is called global variables.

We will learn what is a function and it's parameter in subsequent chapters. Here let us explain what are local and global variables.

## **Local Variables**

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables −

[Live Demo](http://tpcg.io/QIjnPh)

#include <iostream>

using namespace std;

int main () {

// Local variable declaration:

int a, b;

int c;

// actual initialization

a = 10;

b = 20;

c = a + b;

cout << c;

return 0;

}

## **Global Variables**

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables −

[Live Demo](http://tpcg.io/dRHHpD)

#include <iostream>

using namespace std;

// Global variable declaration:

int g;

int main () {

// Local variable declaration:

int a, b;

// actual initialization

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;

}

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example −

[Live Demo](http://tpcg.io/dt7MP9)

#include <iostream>

using namespace std;

// Global variable declaration:

int g = 20;

int main () {

// Local variable declaration:

int g = 10;

cout << g;

return 0;

}

When the above code is compiled and executed, it produces the following result −

10

## **Initializing Local and Global Variables**

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

|  |  |
| --- | --- |
| **Data Type** | **Initializer** |
| int | 0 |
| char | '\0' |
| float | 0 |
| double | 0 |
| pointer | NULL |

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

# CONTRAINS / LITERALS

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

## **Integer Literals**

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

212 // Legal

215u // Legal

0xFeeL // Legal

078 // Illegal: 8 is not an octal digit

032UU // Illegal: cannot repeat a suffix

Following are other examples of various types of Integer literals −

85 // decimal

0213 // octal

0x4b // hexadecimal

30 // int

30u // unsigned int

30l // long

30ul // unsigned long

## **Floating-point Literals**

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals −

3.14159 // Legal

314159E-5L // Legal

510E // Illegal: incomplete exponent

210f // Illegal: no decimal or exponent

.e55 // Illegal: missing integer or fraction

## **Boolean Literals**

There are two Boolean literals and they are part of standard C++ keywords −

* A value of **true** representing true.
* A value of **false** representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

## **Character Literals**

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in **wchar\_t** type of variable . Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes −

|  |  |
| --- | --- |
| **Escape sequence** | **Meaning** |
| \\ | \ character |
| \' | ' character |
| \" | " character |
| \? | ? character |
| \a | Alert or bell |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Carriage return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \ooo | Octal number of one to three digits |
| \xhh . . . | Hexadecimal number of one or more digits |

Following is the example to show a few escape sequence characters −

[Live Demo](http://tpcg.io/8RqQ5S)

#include <iostream>

using namespace std;

int main() {

cout << "Hello\tWorld\n\n";

return 0;

}

When the above code is compiled and executed, it produces the following result −

Hello World

## **String Literals**

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"

## **Defining Constants**

There are two simple ways in C++ to define constants −

* Using **#define** preprocessor.
* Using **const** keyword.

## **The #define Preprocessor**

Following is the form to use #define preprocessor to define a constant −

#define identifier value

Following example explains it in detail −

[Live Demo](http://tpcg.io/N6xxDP)

#include <iostream>

using namespace std;

#define LENGTH 10

#define WIDTH 5

#define NEWLINE '\n'

int main() {

int area;

area = LENGTH \* WIDTH;

cout << area;

cout << NEWLINE;

return 0;

}

When the above code is compiled and executed, it produces the following result −

50

## **The const Keyword**

You can use **const** prefix to declare constants with a specific type as follows −

const type variable = value;

Following example explains it in detail −

[Live Demo](http://tpcg.io/IA1cea)

#include <iostream>

using namespace std;

int main() {

const int LENGTH = 10;

const int WIDTH = 5;

const char NEWLINE = '\n';

int area;

area = LENGTH \* WIDTH;

cout << area;

cout << NEWLINE;

return 0;

}

When the above code is compiled and executed, it produces the following result −

50

Note that it is a good programming practice to define constants in CAPITALS.

# MODIFIER TYPES

C++ allows the **char, int,**and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here −

* signed
* unsigned
* long
* short

The modifiers **signed, unsigned, long,** and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to char, and **long** can be applied to double.

The modifiers **signed** and **unsigned** can also be used as prefix to **long** or **short** modifiers. For example, **unsigned long int**.

C++ allows a shorthand notation for declaring **unsigned, short,** or **long** integers. You can simply use the word **unsigned, short,** or **long,** without **int**. It automatically implies **int**. For example, the following two statements both declare unsigned integer variables.

unsigned x;

unsigned int y;

To understand the difference between the way signed and unsigned integer modifiers are interpreted by C++, you should run the following short program −

[Live Demo](http://tpcg.io/nWAgjz)

#include <iostream>

using namespace std;

/\* This program shows the difference between

\* signed and unsigned integers.

\*/

int main() {

short int i; // a signed short integer

short unsigned int j; // an unsigned short integer

j = 50000;

i = j;

cout << i << " " << j;

return 0;

}

When this program is run, following is the output −

-15536 50000

The above result is because the bit pattern that represents 50,000 as a short unsigned integer is interpreted as -15,536 by a short.

## **Type Qualifiers in C++**

The type qualifiers provide additional information about the variables they precede.

|  |  |
| --- | --- |
| **Sr.No** | **Qualifier & Meaning** |
| 1 | **const**  Objects of type **const** cannot be changed by your program during execution. |
| 2 | **volatile**  The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program. |
| 3 | **restrict**  A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict. |

# STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

* auto
* register
* static
* extern
* mutable

## **The auto Storage Class**

The **auto** storage class is the default storage class for all local variables.

{

int mount;

auto int month;

}

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

## **The register Storage Class**

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

{

register int miles;

}

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

## **The static Storage Class**

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

[Live Demo](http://tpcg.io/20HjiV)

#include <iostream>

// Function declaration

void func(void);

static int count = 10; /\* Global variable \*/

main() {

while(count--) {

func();

}

return 0;

}

// Function definition

void func( void ) {

static int i = 5; // local static variable

i++;

std::cout << "i is " << i ;

std::cout << " and count is " << count << std::endl;

}

When the above code is compiled and executed, it produces the following result −

i is 6 and count is 9

i is 7 and count is 8

i is 8 and count is 7

i is 9 and count is 6

i is 10 and count is 5

i is 11 and count is 4

i is 12 and count is 3

i is 13 and count is 2

i is 14 and count is 1

i is 15 and count is 0

## **The extern Storage Class**

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern*is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

### **First File: main.cpp**

#include <iostream>

int count ;

extern void write\_extern();

main() {

count = 5;

write\_extern();

}

### **Second File: support.cpp**

#include <iostream>

extern int count;

void write\_extern(void) {

std::cout << "Count is " << count << std::endl;

}

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows −

$g++ main.cpp support.cpp -o write

This will produce **write** executable program, try to execute **write** and check the result as follows −

$./write

5

## **The mutable Storage Class**

The **mutable** specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function.

# OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operators
* Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

## **Arithmetic Operators**

There are following arithmetic operators supported by C++ language −

Assume variable A holds 10 and variable B holds 20, then −

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_arithmatic_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| \* | Multiplies both operands | A \* B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | [Increment operator](https://www.tutorialspoint.com/cplusplus/cpp_increment_decrement_operators.htm), increases integer value by one | A++ will give 11 |
| -- | [Decrement operator](https://www.tutorialspoint.com/cplusplus/cpp_increment_decrement_operators.htm), decreases integer value by one | A-- will give 9 |

## **Relational Operators**

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_relational_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## **Logical Operators**

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then −

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_logical_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Called Logical AND operator. If both the operands are non-zero, then condition becomes true. | (A && B) is false. |
| || | Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true. | (A || B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false. | !(A && B) is true. |

## **Bitwise Operators**

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; now in binary format they will be as follows −

A = 0011 1100

B = 0000 1101

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

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then −

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_bitwise_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

## **Assignment Operators**

There are following assignment operators supported by C++ language −

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

## **Misc Operators**

The following table lists some other operators that C++ supports.

|  |  |
| --- | --- |
| **Sr.No** | **Operator & Description** |
| 1 | **sizeof**  [sizeof operator](https://www.tutorialspoint.com/cplusplus/cpp_sizeof_operator.htm) returns the size of a variable. For example, sizeof(a), where ‘a’ is integer, and will return 4. |
| 2 | **Condition ? X : Y**  [Conditional operator (?)](https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.htm). If Condition is true then it returns value of X otherwise returns value of Y. |
| 3 | **,**  [Comma operator](https://www.tutorialspoint.com/cplusplus/cpp_comma_operator.htm) causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list. |
| 4 | **. (dot) and -> (arrow)**  [Member operators](https://www.tutorialspoint.com/cplusplus/cpp_member_operators.htm) are used to reference individual members of classes, structures, and unions. |
| 5 | **Cast**  [Casting operators](https://www.tutorialspoint.com/cplusplus/cpp_casting_operators.htm) convert one data type to another. For example, int(2.2000) would return 2. |
| 6 | **&**  [Pointer operator &](https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm) returns the address of a variable. For example &a; will give actual address of the variable. |
| 7 | **\***  [Pointer operator \*](https://www.tutorialspoint.com/cplusplus/cpp_pointer_operators.htm) is pointer to a variable. For example \*var; will pointer to a variable var. |

## **Operators Precedence in C++**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example x = 7 + 3 \* 2; here, x is assigned 13, not 20 because operator \* has higher precedence than +, so it first gets multiplied with 3\*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_operators_precedence.htm)

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

# LOOP TYPES

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

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

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages −



C++ programming language provides the following type of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No** | **Loop Type & Description** |
| 1 | [while loop](https://www.tutorialspoint.com/cplusplus/cpp_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 2 | [for loop](https://www.tutorialspoint.com/cplusplus/cpp_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 3 | [do...while loop](https://www.tutorialspoint.com/cplusplus/cpp_do_while_loop.htm)  Like a ‘while’ statement, except that it tests the condition at the end of the loop body. |
| 4 | [nested loops](https://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm)  You can use one or more loop inside any another ‘while’, ‘for’ or ‘do..while’ loop. |

## **Loop Control Statements**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements.

|  |  |
| --- | --- |
| **Sr.No** | **Control Statement & Description** |
| 1 | [break statement](https://www.tutorialspoint.com/cplusplus/cpp_break_statement.htm)  Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch. |
| 2 | [continue statement](https://www.tutorialspoint.com/cplusplus/cpp_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [goto statement](https://www.tutorialspoint.com/cplusplus/cpp_goto_statement.htm)  Transfers control to the labeled statement. Though it is not advised to use goto statement in your program. |

## **The Infinite Loop**

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.

#include <iostream>

using namespace std;

int main () {

for( ; ; ) {

printf("This loop will run forever.\n");

}

return 0;

}

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the ‘for (;;)’ construct to signify an infinite loop.

**NOTE** − You can terminate an infinite loop by pressing Ctrl + C keys.

# DECISION MAKING

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



C++ programming language provides following types of decision making statements.

|  |  |
| --- | --- |
| **Sr.No** | **Statement & Description** |
| 1 | [if statement](https://www.tutorialspoint.com/cplusplus/cpp_if_statement.htm)  An ‘if’ statement consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statement](https://www.tutorialspoint.com/cplusplus/cpp_if_else_statement.htm)  An ‘if’ statement can be followed by an optional ‘else’ statement, which executes when the boolean expression is false. |
| 3 | [switch statement](https://www.tutorialspoint.com/cplusplus/cpp_switch_statement.htm)  A ‘switch’ statement allows a variable to be tested for equality against a list of values. |
| 4 | [nested if statements](https://www.tutorialspoint.com/cplusplus/cpp_nested_if.htm)  You can use one ‘if’ or ‘else if’ statement inside another ‘if’ or ‘else if’ statement(s). |
| 5 | [nested switch statements](https://www.tutorialspoint.com/cplusplus/cpp_nested_switch.htm)  You can use one ‘switch’ statement inside another ‘switch’ statement(s). |

## **The ? : Operator**

We have covered [conditional operator “? :”](https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.htm) in previous chapter which can be used to replace **if...else** statements. It has the following general form −

Exp1 ? Exp2 : Exp3;

Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ‘?’ expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ‘?’ expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

# FUNCTIONS

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

## **Defining a Function**

The general form of a C++ function definition is as follows −

return\_type function\_name( parameter list ) {

body of the function

}

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

* **Return Type** − A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return\_type is the keyword **void**.
* **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
* **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
* **Function Body** − The function body contains a collection of statements that define what the function does.

## **Example**

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both −

// function returning the max between two numbers

int max(int num1, int num2) {

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

## **Function Declarations**

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return\_type function\_name( parameter list );

For the above defined function max(), following is the function declaration −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

## **Calling a Function**

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it’s return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example −

[Live Demo](http://tpcg.io/XuXfLz)

#include <iostream>

using namespace std;

// function declaration

int max(int num1, int num2);

int main () {

// local variable declaration:

int a = 100;

int b = 200;

int ret;

// calling a function to get max value.

ret = max(a, b);

cout << "Max value is : " << ret << endl;

return 0;

}

// function returning the max between two numbers

int max(int num1, int num2) {

// local variable declaration

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

Max value is : 200

## **Function Arguments**

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function −

|  |  |
| --- | --- |
| **Sr.No** | **Call Type & Description** |
| 1 | [Call by Value](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_value.htm)  This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument. |
| 2 | [Call by Pointer](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_pointer.htm)  This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |
| 3 | [Call by Reference](https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm)  This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument. |

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

## **Default Values for Parameters**

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example −

[Live Demo](http://tpcg.io/KoJLUN)

#include <iostream>

using namespace std;

int sum(int a, int b = 20) {

int result;

result = a + b;

return (result);

}

int main () {

// local variable declaration:

int a = 100;

int b = 200;

int result;

// calling a function to add the values.

result = sum(a, b);

cout << "Total value is :" << result << endl;

// calling a function again as follows.

result = sum(a);

cout << "Total value is :" << result << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Total value is :300

Total value is :120

# NUMBERS

Normally, when we work with Numbers, we use primitive data types such as int, short, long, float and double, etc. The number data types, their possible values and number ranges have been explained while discussing C++ Data Types.

## **Defining Numbers in C++**

You have already defined numbers in various examples given in previous chapters. Here is another consolidated example to define various types of numbers in C++ −

[Live Demo](http://tpcg.io/Q21eQX)

#include <iostream>

using namespace std;

int main () {

// number definition:

short s;

int i;

long l;

float f;

double d;

// number assignments;

s = 10;

i = 1000;

l = 1000000;

f = 230.47;

d = 30949.374;

// number printing;

cout << "short s :" << s << endl;

cout << "int i :" << i << endl;

cout << "long l :" << l << endl;

cout << "float f :" << f << endl;

cout << "double d :" << d << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

short s :10

int i :1000

long l :1000000

float f :230.47

double d :30949.4

## **Math Operations in C++**

In addition to the various functions you can create, C++ also includes some useful functions you can use. These functions are available in standard C and C++ libraries and called **built-in** functions. These are functions that can be included in your program and then use.

C++ has a rich set of mathematical operations, which can be performed on various numbers. Following table lists down some useful built-in mathematical functions available in C++.

To utilize these functions you need to include the math header file **<cmath>**.

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **double cos(double);**  This function takes an angle (as a double) and returns the cosine. |
| 2 | **double sin(double);**  This function takes an angle (as a double) and returns the sine. |
| 3 | **double tan(double);**  This function takes an angle (as a double) and returns the tangent. |
| 4 | **double log(double);**  This function takes a number and returns the natural log of that number. |
| 5 | **double pow(double, double);**  The first is a number you wish to raise and the second is the power you wish to raise it t |
| 6 | **double hypot(double, double);**  If you pass this function the length of two sides of a right triangle, it will return you the length of the hypotenuse. |
| 7 | **double sqrt(double);**  You pass this function a number and it gives you the square root. |
| 8 | **int abs(int);**  This function returns the absolute value of an integer that is passed to it. |
| 9 | **double fabs(double);**  This function returns the absolute value of any decimal number passed to it. |
| 10 | **double floor(double);**  Finds the integer which is less than or equal to the argument passed to it. |

Following is a simple example to show few of the mathematical operations −

[Live Demo](http://tpcg.io/qU7JSv)

#include <iostream>

#include <cmath>

using namespace std;

int main () {

// number definition:

short s = 10;

int i = -1000;

long l = 100000;

float f = 230.47;

double d = 200.374;

// mathematical operations;

cout << "sin(d) :" << sin(d) << endl;

cout << "abs(i) :" << abs(i) << endl;

cout << "floor(d) :" << floor(d) << endl;

cout << "sqrt(f) :" << sqrt(f) << endl;

cout << "pow( d, 2) :" << pow(d, 2) << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

sign(d) :-0.634939

abs(i) :1000

floor(d) :200

sqrt(f) :15.1812

pow( d, 2 ) :40149.7

## **Random Numbers in C++**

There are many cases where you will wish to generate a random number. There are actually two functions you will need to know about random number generation. The first is **rand()**, this function will only return a pseudo random number. The way to fix this is to first call the **srand()** function.

Following is a simple example to generate few random numbers. This example makes use of **time()** function to get the number of seconds on your system time, to randomly seed the rand() function −

[Live Demo](http://tpcg.io/Z1ZN0e)

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

int main () {

int i,j;

// set the seed

srand( (unsigned)time( NULL ) );

/\* generate 10 random numbers. \*/

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

// generate actual random number

j = rand();

cout <<" Random Number : " << j << endl;

}

return 0;

}

When the above code is compiled and executed, it produces the following result –

Random Number : 1748144778

Random Number : 630873888

Random Number : 2134540646

Random Number : 219404170

Random Number : 902129458

Random Number : 920445370

Random Number : 1319072661

Random Number : 257938873

Random Number : 1256201101

Random Number : 580322989

# ARRAYS

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

## **Declaring Arrays**

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows −

type arrayName [ arraySize ];

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement −

double balance[10];

## **Initializing Arrays**

You can initialize C++ array elements either one by one or using a single statement as follows −

double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array −

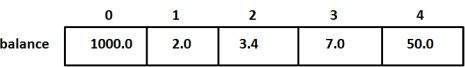
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};

You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above −



## **Accessing Array Elements**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays −

[Live Demo](http://tpcg.io/QRe4fY)

#include <iostream>

using namespace std;

#include <iomanip>

using std::setw;

int main () {

int n[ 10 ]; // n is an array of 10 integers

// initialize elements of array n to 0

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

n[ i ] = i + 100; // set element at location i to i + 100

}

cout << "Element" << setw( 13 ) << "Value" << endl;

// output each array element's value

for ( int j = 0; j < 10; j++ ) {

cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;

}

return 0;

}

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result −

Element Value

0 100

1 101

2 102

3 103

4 104

5 105

6 106

7 107

8 108

9 109

## **Arrays in C++**

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Multi-dimensional arrays](https://www.tutorialspoint.com/cplusplus/cpp_multi_dimensional_arrays.htm)  C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array. |
| 2 | [Pointer to an array](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_an_array.htm)  You can generate a pointer to the first element of an array by simply specifying the array name, without any index. |
| 3 | [Passing arrays to functions](https://www.tutorialspoint.com/cplusplus/cpp_passing_arrays_to_functions.htm)  You can pass to the function a pointer to an array by specifying the array's name without an index. |
| 4 | [Return array from functions](https://www.tutorialspoint.com/cplusplus/cpp_return_arrays_from_functions.htm)  C++ allows a function to return an array. |

# STRINGS

C++ provides following two types of string representations −

* The C-style character string.
* The string class type introduced with Standard C++.

## **The C-Style Character String**

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

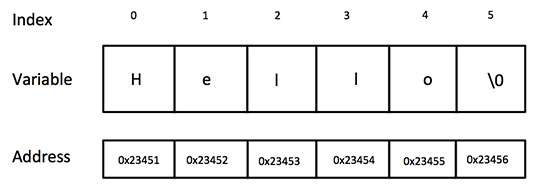
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization, then you can write the above statement as follows −

char greeting[] = "Hello";

Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string −

[Live Demo](http://tpcg.io/OzjL2L)

#include <iostream>

using namespace std;

int main () {

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

cout << "Greeting message: ";

cout << greeting << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Greeting message: Hello

C++ supports a wide range of functions that manipulate null-terminated strings −

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **strcpy(s1, s2);**  Copies string s2 into string s1. |
| 2 | **strcat(s1, s2);**  Concatenates string s2 onto the end of string s1. |
| 3 | **strlen(s1);**  Returns the length of string s1. |
| 4 | **strcmp(s1, s2);**  Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2. |
| 5 | **strchr(s1, ch);**  Returns a pointer to the first occurrence of character ch in string s1. |
| 6 | **strstr(s1, s2);**  Returns a pointer to the first occurrence of string s2 in string s1. |

Following example makes use of few of the above-mentioned functions −

[Live Demo](http://tpcg.io/oA6mP3)

#include <iostream>

#include <cstring>

using namespace std;

int main () {

char str1[10] = "Hello";

char str2[10] = "World";

char str3[10];

int len ;

// copy str1 into str3

strcpy( str3, str1);

cout << "strcpy( str3, str1) : " << str3 << endl;

// concatenates str1 and str2

strcat( str1, str2);

cout << "strcat( str1, str2): " << str1 << endl;

// total lenghth of str1 after concatenation

len = strlen(str1);

cout << "strlen(str1) : " << len << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

strcpy( str3, str1) : Hello

strcat( str1, str2): HelloWorld

strlen(str1) : 10

## **The String Class in C++**

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example −

[Live Demo](http://tpcg.io/upuWFC)

#include <iostream>

#include <string>

using namespace std;

int main () {

string str1 = "Hello";

string str2 = "World";

string str3;

int len ;

// copy str1 into str3

str3 = str1;

cout << "str3 : " << str3 << endl;

// concatenates str1 and str2

str3 = str1 + str2;

cout << "str1 + str2 : " << str3 << endl;

// total length of str3 after concatenation

len = str3.size();

cout << "str3.size() : " << len << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

str3 : Hello

str1 + str2 : HelloWorld

str3.size() : 10

# POINTERS

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined −

[Live Demo](http://tpcg.io/fQ1InE)

#include <iostream>

using namespace std;

int main () {

int var1;

char var2[10];

cout << "Address of var1 variable: ";

cout << &var1 << endl;

cout << "Address of var2 variable: ";

cout << &var2 << endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var1 variable: 0xbfebd5c0

Address of var2 variable: 0xbfebd5b6

## **What are Pointers?**

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

type \*var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

int \*ip; // pointer to an integer

double \*dp; // pointer to a double

float \*fp; // pointer to a float

char \*ch // pointer to character

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## **Using Pointers in C++**

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations −

[Live Demo](http://tpcg.io/gTdFv7)

#include <iostream>

using namespace std;

int main () {

int var = 20; // actual variable declaration.

int \*ip; // pointer variable

ip = &var; // store address of var in pointer variable

cout << "Value of var variable: ";

cout << var << endl;

// print the address stored in ip pointer variable

cout << "Address stored in ip variable: ";

cout << ip << endl;

// access the value at the address available in pointer

cout << "Value of \*ip variable: ";

cout << \*ip << endl;

return 0;

}

When the above code is compiled and executed, it produces result something as follows −

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of \*ip variable: 20

## **Pointers in C++**

Pointers have many but easy concepts and they are very important to C++ programming. There are following few important pointer concepts which should be clear to a C++ programmer −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Null Pointers](https://www.tutorialspoint.com/cplusplus/cpp_null_pointers.htm)  C++ supports null pointer, which is a constant with a value of zero defined in several standard libraries. |
| 2 | [Pointer Arithmetic](https://www.tutorialspoint.com/cplusplus/cpp_pointer_arithmatic.htm)  There are four arithmetic operators that can be used on pointers: ++, --, +, - |
| 3 | [Pointers vs Arrays](https://www.tutorialspoint.com/cplusplus/cpp_pointers_vs_arrays.htm)  There is a close relationship between pointers and arrays. |
| 4 | [Array of Pointers](https://www.tutorialspoint.com/cplusplus/cpp_array_of_pointers.htm)  You can define arrays to hold a number of pointers. |
| 5 | [Pointer to Pointer](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_pointer.htm)  C++ allows you to have pointer on a pointer and so on. |
| 6 | [Passing Pointers to Functions](https://www.tutorialspoint.com/cplusplus/cpp_passing_pointers_to_functions.htm)  Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function. |
| 7 | [Return Pointer from Functions](https://www.tutorialspoint.com/cplusplus/cpp_return_pointer_from_functions.htm)  C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well. |

# REFERENCES

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

## **References vs Pointers**

References are often confused with pointers but three major differences between references and pointers are −

* You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
* Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
* A reference must be initialized when it is created. Pointers can be initialized at any time.

## **Creating References in C++**

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. For example, suppose we have the following example −

int i = 17;

We can declare reference variables for i as follows.

int& r = i;

Read the & in these declarations as **reference**. Thus, read the first declaration as "r is an integer reference initialized to i" and read the second declaration as "s is a double reference initialized to d.". Following example makes use of references on int and double −

[Live Demo](http://tpcg.io/SS8zU0)

#include <iostream>

using namespace std;

int main () {

// declare simple variables

int i;

double d;

// declare reference variables

int& r = i;

double& s = d;

i = 5;

cout << "Value of i : " << i << endl;

cout << "Value of i reference : " << r << endl;

d = 11.7;

cout << "Value of d : " << d << endl;

cout << "Value of d reference : " << s << endl;

return 0;

}

When the above code is compiled together and executed, it produces the following result −

Value of i : 5

Value of i reference : 5

Value of d : 11.7

Value of d reference : 11.7

References are usually used for function argument lists and function return values. So following are two important subjects related to C++ references which should be clear to a C++ programmer −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [References as Parameters](https://www.tutorialspoint.com/cplusplus/passing_parameters_by_references.htm)  C++ supports passing references as function parameter more safely than parameters. |
| 2 | [Reference as Return Value](https://www.tutorialspoint.com/cplusplus/returning_values_by_reference.htm)  You can return reference from a C++ function like any other data type. |

# DATE & TIME

The C++ standard library does not provide a proper date type. C++ inherits the structs and functions for date and time manipulation from C. To access date and time related functions and structures, you would need to include <ctime> header file in your C++ program.

There are four time-related types: **clock\_t, time\_t, size\_t**, and **tm**. The types - clock\_t, size\_t and time\_t are capable of representing the system time and date as some sort of integer.

The structure type **tm** holds the date and time in the form of a C structure having the following elements −

struct tm {

int tm\_sec; // seconds of minutes from 0 to 61

int tm\_min; // minutes of hour from 0 to 59

int tm\_hour; // hours of day from 0 to 24

int tm\_mday; // day of month from 1 to 31

int tm\_mon; // month of year from 0 to 11

int tm\_year; // year since 1900

int tm\_wday; // days since sunday

int tm\_yday; // days since January 1st

int tm\_isdst; // hours of daylight savings time

}

Following are the important functions, which we use while working with date and time in C or C++. All these functions are part of standard C and C++ library and you can check their detail using reference to C++ standard library given below.

|  |  |
| --- | --- |
| **Sr.No** | **Function & Purpose** |
| 1 | **time\_t time(time\_t \*time);**  This returns the current calendar time of the system in number of seconds elapsed since January 1, 1970. If the system has no time, .1 is returned. |
| 2 | **char \*ctime(const time\_t \*time);**  This returns a pointer to a string of the form *day month year hours:minutes:seconds year\n\0*. |
| 3 | **struct tm \*localtime(const time\_t \*time);**  This returns a pointer to the **tm** structure representing local time. |
| 4 | **clock\_t clock(void);**  This returns a value that approximates the amount of time the calling program has been running. A value of .1 is returned if the time is not available. |
| 5 | **char \* asctime ( const struct tm \* time );**  This returns a pointer to a string that contains the information stored in the structure pointed to by time converted into the form: day month date hours:minutes:seconds year\n\0 |
| 6 | **struct tm \*gmtime(const time\_t \*time);**  This returns a pointer to the time in the form of a tm structure. The time is represented in Coordinated Universal Time (UTC), which is essentially Greenwich Mean Time (GMT). |
| 7 | **time\_t mktime(struct tm \*time);**  This returns the calendar-time equivalent of the time found in the structure pointed to by time. |
| 8 | **double difftime ( time\_t time2, time\_t time1 );**  This function calculates the difference in seconds between time1 and time2. |
| 9 | **size\_t strftime();**  This function can be used to format date and time in a specific format. |

## **Current Date and Time**

Suppose you want to retrieve the current system date and time, either as a local time or as a Coordinated Universal Time (UTC). Following is the example to achieve the same −

[Live Demo](http://tpcg.io/DHKMA9)

#include <iostream>

#include <ctime>

using namespace std;

int main() {

// current date/time based on current system

time\_t now = time(0);

// convert now to string form

char\* dt = ctime(&now);

cout << "The local date and time is: " << dt << endl;

// convert now to tm struct for UTC

tm \*gmtm = gmtime(&now);

dt = asctime(gmtm);

cout << "The UTC date and time is:"<< dt << endl;

}

When the above code is compiled and executed, it produces the following result −

The local date and time is: Sat Jan 8 20:07:41 2011

The UTC date and time is:Sun Jan 9 03:07:41 2011

## **Format Time using struct tm**

The **tm** structure is very important while working with date and time in either C or C++. This structure holds the date and time in the form of a C structure as mentioned above. Most of the time related functions makes use of tm structure. Following is an example which makes use of various date and time related functions and tm structure −

While using structure in this chapter, I'm making an assumption that you have basic understanding on C structure and how to access structure members using arrow -> operator.

[Live Demo](http://tpcg.io/SMnO0N)

#include <iostream>

#include <ctime>

using namespace std;

int main() {

// current date/time based on current system

time\_t now = time(0);

cout << "Number of sec since January 1,1970 is:: " << now << endl;

tm \*ltm = localtime(&now);

// print various components of tm structure.

cout << "Year:" << 1900 + ltm->tm\_year<<endl;

cout << "Month: "<< 1 + ltm->tm\_mon<< endl;

cout << "Day: "<< ltm->tm\_mday << endl;

cout << "Time: "<< 5+ltm->tm\_hour << ":";

cout << 30+ltm->tm\_min << ":";

cout << ltm->tm\_sec << endl;

}

When the above code is compiled and executed, it produces the following result −

Number of sec since January 1,1970 is:: 1588485717

Year:2020

Month: 5

Day: 3

Time: 11:31:57

# BASIC INPUT / INPUT

The C++ standard libraries provide an extensive set of input/output capabilities which we will see in subsequent chapters. This chapter will discuss very basic and most common I/O operations required for C++ programming.

C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

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

There are following header files important to C++ programs −

|  |  |
| --- | --- |
| **Sr.No** | **Header File & Function and Description** |
| 1 | **<iostream>**  This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively. |
| 2 | **<iomanip>**  This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and **setprecision**. |
| 3 | **<fstream>**  This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter. |

## **The Standard Output Stream (cout)**

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

[Live Demo](http://tpcg.io/qFaiKd)

#include <iostream>

using namespace std;

int main() {

char str[] = "Hello C++";

cout << "Value of str is : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Value of str is : Hello C++

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

## **The Standard Input Stream (cin)**

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

[Live Demo](http://tpcg.io/NDGHmy)

#include <iostream>

using namespace std;

int main() {

char name[50];

cout << "Please enter your name: ";

cin >> name;

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

}

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

Please enter your name: cplusplus

Your name is: cplusplus

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following −

cin >> name >> age;

This will be equivalent to the following two statements −

cin >> name;

cin >> age;

## **The Standard Error Stream (cerr)**

The predefined object **cerr** is an instance of **ostream** class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object **cerr** is un-buffered and each stream insertion to cerr causes its output to appear immediately.

The **cerr** is also used in conjunction with the stream insertion operator as shown in the following example.

[Live Demo](http://tpcg.io/3xp5lM)

#include <iostream>

using namespace std;

int main() {

char str[] = "Unable to read....";

cerr << "Error message : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

## **The Standard Log Stream (clog)**

The predefined object **clog** is an instance of **ostream** class. The clog object is said to be attached to the standard error device, which is also a display screen but the object **clog** is buffered. This means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

The **clog** is also used in conjunction with the stream insertion operator as shown in the following example.

[Live Demo](http://tpcg.io/Dewejb)

#include <iostream>

using namespace std;

int main() {

char str[] = "Unable to read....";

clog << "Error message : " << str << endl;

}

When the above code is compiled and executed, it produces the following result −

Error message : Unable to read....

You would not be able to see any difference in cout, cerr and clog with these small examples, but while writing and executing big programs the difference becomes obvious. So it is good practice to display error messages using cerr stream and while displaying other log messages then clog should be used.

# DATA STRUCTES

C/C++ arrays allow you to define variables that combine several data items of the same kind, but **structure** is another user defined data type which allows you to combine data items of different kinds.

Structures are used to represent a record, suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

## **Defining a Structure**

To define a structure, you must use the struct statement. The struct statement defines a new data type, with more than one member, for your program. The format of the struct statement is this −

struct [structure tag] {

member definition;

member definition;

...

member definition;

} [one or more structure variables];

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure −

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

} book;

## **Accessing Structure Members**

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use **struct** keyword to define variables of structure type. Following is the example to explain usage of structure −

[Live Demo](http://tpcg.io/zvIEi0)

#include <iostream>

#include <cstring>

using namespace std;

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main() {

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// book 1 specification

strcpy( Book1.title, "Learn C++ Programming");

strcpy( Book1.author, "Chand Miyan");

strcpy( Book1.subject, "C++ Programming");

Book1.book\_id = 6495407;

// book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Yakit Singha");

strcpy( Book2.subject, "Telecom");

Book2.book\_id = 6495700;

// Print Book1 info

cout << "Book 1 title : " << Book1.title <<endl;

cout << "Book 1 author : " << Book1.author <<endl;

cout << "Book 1 subject : " << Book1.subject <<endl;

cout << "Book 1 id : " << Book1.book\_id <<endl;

// Print Book2 info

cout << "Book 2 title : " << Book2.title <<endl;

cout << "Book 2 author : " << Book2.author <<endl;

cout << "Book 2 subject : " << Book2.subject <<endl;

cout << "Book 2 id : " << Book2.book\_id <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result −

Book 1 title : Learn C++ Programming

Book 1 author : Chand Miyan

Book 1 subject : C++ Programming

Book 1 id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Yakit Singha

Book 2 subject : Telecom

Book 2 id : 6495700

## **Structures as Function Arguments**

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example −

[Live Demo](http://tpcg.io/xvb7BQ)

#include <iostream>

#include <cstring>

using namespace std;

void printBook( struct Books book );

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main() {

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// book 1 specification

strcpy( Book1.title, "Learn C++ Programming");

strcpy( Book1.author, "Chand Miyan");

strcpy( Book1.subject, "C++ Programming");

Book1.book\_id = 6495407;

// book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Yakit Singha");

strcpy( Book2.subject, "Telecom");

Book2.book\_id = 6495700;

// Print Book1 info

printBook( Book1 );

// Print Book2 info

printBook( Book2 );

return 0;

}

void printBook( struct Books book ) {

cout << "Book title : " << book.title <<endl;

cout << "Book author : " << book.author <<endl;

cout << "Book subject : " << book.subject <<endl;

cout << "Book id : " << book.book\_id <<endl;

}

When the above code is compiled and executed, it produces the following result −

Book title : Learn C++ Programming

Book author : Chand Miyan

Book subject : C++ Programming

Book id : 6495407

Book title : Telecom Billing

Book author : Yakit Singha

Book subject : Telecom

Book id : 6495700

## **Pointers to Structures**

You can define pointers to structures in very similar way as you define pointer to any other variable as follows −

struct Books \*struct\_pointer;

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows −

struct\_pointer = &Book1;

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows −

struct\_pointer->title;

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept −

[Live Demo](http://tpcg.io/P8IeUE)

#include <iostream>

#include <cstring>

using namespace std;

void printBook( struct Books \*book );

struct Books {

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main() {

struct Books Book1; // Declare Book1 of type Book

struct Books Book2; // Declare Book2 of type Book

// Book 1 specification

strcpy( Book1.title, "Learn C++ Programming");

strcpy( Book1.author, "Chand Miyan");

strcpy( Book1.subject, "C++ Programming");

Book1.book\_id = 6495407;

// Book 2 specification

strcpy( Book2.title, "Telecom Billing");

strcpy( Book2.author, "Yakit Singha");

strcpy( Book2.subject, "Telecom");

Book2.book\_id = 6495700;

// Print Book1 info, passing address of structure

printBook( &Book1 );

// Print Book1 info, passing address of structure

printBook( &Book2 );

return 0;

}

// This function accept pointer to structure as parameter.

void printBook( struct Books \*book ) {

cout << "Book title : " << book->title <<endl;

cout << "Book author : " << book->author <<endl;

cout << "Book subject : " << book->subject <<endl;

cout << "Book id : " << book->book\_id <<endl;

}

When the above code is compiled and executed, it produces the following result −

Book title : Learn C++ Programming

Book author : Chand Miyan

Book subject : C++ Programming

Book id : 6495407

Book title : Telecom Billing

Book author : Yakit Singha

Book subject : Telecom

Book id : 6495700

## **The typedef Keyword**

There is an easier way to define structs or you could "alias" types you create. For example −

typedef struct {

char title[50];

char author[50];

char subject[100];

int book\_id;

} Books;

Now, you can use *Books* directly to define variables of *Books* type without using struct keyword. Following is the example −

Books Book1, Book2;

You can use **typedef** keyword for non-structs as well as follows −

typedef long int \*pint32;

pint32 x, y, z;

x, y and z are all pointers to long ints.