**Q.1. what is different b/w C and C++?**

**Ans :-** **C Programming is an ANSI/ISO standard and powerful programming language for developing real time applications. C programming language was invented by Dennis Ritchie at the Bell Laboratories in 1972. It was invented for implementing UNIX operating system. C is most widely used programming language even today. All other programming languages were derived directly or indirectly from C programming concepts. This tutorial explains all basic concepts in C like history of C language, data types, keywords, constants, variables, operators, expressions, control statements, array, pointer, string, library functions, structures and unions etc.**

**C++ is a programming language which is a superset of the C language. The C language is known as C language because it is a successor of B language and was developed by Dennis Ritchie. C++ is a middle level programming language and it is developed by Bjarne Stroustrup in 1979 at Bell Labs as the extension to C language and was originally named as C with Classes but later it was renamed as C++ in 1983. The C++ language can run on various platforms like Windows, Mac OS, and even on various versions of UNIX. The C++ is an object oriented language. This C++ tutorial will help new learners to understand all C++ concepts in easy way with examples.**

**Q.2. what is basic history of C++?**

**Ans:-** **C++ is a programming language which is a superset of the C language. The C language is known as C language because it is a successor of B language and was developed by Dennis Ritchie. C++ is a middle level programming language and it is developed by Bjarne Stroustrup in 1979 at Bell Labs as the extension to C language and was originally named as C with Classes but later it was renamed as C++ in 1983. The C++ language can run on various platforms like Windows, Mac OS, and even on various versions of UNIX. The C++ is an object oriented language. This C++ tutorial will help new learners to understand all C++ concepts in easy way with examples.**

**C++ is a programming language which includes the main four pillars of object oriented development. Those concepts are:**

* **Data Hiding**
* **Inheritance**
* **Encapsulation**
* **Polymorphism**
* **What are the features of C++?**
* **C++ is an object oriented programming language. Some of the features are:**
* **It is a mid-level programming language.**
* **It supports the dynamic memory allocation.**
* **C++ is a machine independent but not platform independent.**
* **It provides lots of inbuilt functions.**
* **Setting C++ Local Environment**
* **Now to learn C++ programming language you need to set up the C++ programming environment, so that you can compile and execute the programs. So, now you will learn how to set up the local environment on the system.**
* **Text Editor – The text editor is used to type your program and they are Notepad, Epsilon, vim or vi, etc. The text editor can have different name and version that can vary on different operating systems. For example, the Notepad editor will be used on Windows and vim or vi can be used on Linux or UNIX as well as Windows. And the files which you create on editor are known as source files and saved with the .cpp , .cp or .c extension.**
* **Compiler – The compiler is that which will be used to compile the source code into executable program. By default, the compiler use .cpp extension.**
* **Now, let us have a look at how to install GNU C/C++ compiler:**
* **Installing on UNIX/LINUX**
* **So, here you will learn how to install gcc compiler on Linux or UNIX and then check whether gcc is installed on your system or not by entering the following command on the command line:**

**Q.3.** **what are literals?**

**Ans:-**

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 −

#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 −

#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 −

#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.

**Q.4.** **what is the difference between keyword and identifier?**

**Ans:-**

**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** |  |

**Q.5.** **Describe different types of operators in C++.**

**Ans:-**

* **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 −**

|  |  |  |
| --- | --- | --- |
| * **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 −**

|  |  |  |
| --- | --- | --- |
| * **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.**

|  |  |  |
| --- | --- | --- |
| * **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** |

**Q.6. what do you understand by the term pre processor directive?**

**Ans:-**

* **The preprocessors are the directives, which give instructions to the compiler to preprocess the information before actual compilation starts.**
* **All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;).**
* **You already have seen a #include directive in all the examples. This macro is used to include a header file into the source file.**
* **There are number of preprocessor directives supported by C++ like #include, #define, #if, #else, #line, etc. Let us see important directives −**
* **The #define Preprocessor**
* **The #define preprocessor directive creates symbolic constants. The symbolic constant is called a macro and the general form of the directive is −**
* **#define macro-name replacement-text**
* **When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example −**
* **#include <iostream>**
* **using namespace std;**
* **#define PI 3.14159**
* **int main () {**
* **cout << "Value of PI :" << PI << endl;**
* **return 0;**
* **}**
* **Now, let us do the preprocessing of this code to see the result assuming we have the source code file. So let us compile it with -E option and redirect the result to test.p. Now, if you check test.p, it will have lots of information and at the bottom, you will find the value replaced as follows −**
* **$gcc -E test.cpp > test.p**
* **...**
* **int main () {**
* **cout << "Value of PI :" << 3.14159 << endl;**
* **return 0;**
* **}**
* **Function-Like Macros**
* **You can use #define to define a macro which will take argument as follows −**
* **#include <iostream>**
* **using namespace std;**
* **#define MIN(a,b) (((a)<(b)) ? a : b)**
* **int main () {**
* **int i, j;**
* **i = 100;**
* **j = 30;**
* **cout <<"The minimum is " << MIN(i, j) << endl;**
* **return 0;**
* **}**
* **If we compile and run above code, this would produce the following result −**
* **The minimum is 30**
* **Conditional Compilation**
* **There are several directives, which can be used to compile selective portions of your program's source code. This process is called conditional compilation.**
* **The conditional preprocessor construct is much like the ‘if’ selection structure. Consider the following preprocessor code −**
* **#ifndef NULL**
* **#define NULL 0**
* **#endif**
* **You can compile a program for debugging purpose. You can also turn on or off the debugging using a single macro as follows −**
* **#ifdef DEBUG**
* **cerr <<"Variable x = " << x << endl;**
* **#endif**
* **This causes the cerr statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statment to comment out a portion of the program as follows −**
* **#if 0**
* **code prevented from compiling**
* **#endif**
* **Let us try the following example −**
* **#include <iostream>**
* **using namespace std;**
* **#define DEBUG**
* **#define MIN(a,b) (((a)<(b)) ? a : b)**
* **int main () {**
* **int i, j;**
* **i = 100;**
* **j = 30;**
* **#ifdef DEBUG**
* **cerr <<"Trace: Inside main function" << endl;**
* **#endif**
* **#if 0**
* **/\* This is commented part \*/**
* **cout << MKSTR(HELLO C++) << endl;**
* **#endif**
* **cout <<"The minimum is " << MIN(i, j) << endl;**
* **#ifdef DEBUG**
* **cerr <<"Trace: Coming out of main function" << endl;**
* **#endif**
* **return 0;**
* **}**
* **If we compile and run above code, this would produce the following result −**
* **The minimum is 30**
* **Trace: Inside main function**
* **Trace: Coming out of main function**
* **The # and ## Operators**
* **The # and ## preprocessor operators are available in C++ and ANSI/ISO C. The # operator causes a replacement-text token to be converted to a string surrounded by quotes.**
* **Consider the following macro definition −**
* **Live Dem**
* **#include <iostream>**
* **using namespace std;**
* **#define MKSTR( x ) #x**
* **int main () {**
* **cout << MKSTR(HELLO C++) << endl;**
* **return 0;**
* **}**
* **If we compile and run above code, this would produce the following result −**
* **HELLO C++**
* **Let us see how it worked. It is simple to understand that the C++ preprocessor turns the line −**
* **cout << MKSTR(HELLO C++) << endl;**
* **Above line will be turned into the following line −**
* **cout << "HELLO C++" << endl;**
* **The ## operator is used to concatenate two tokens. Here is an example −**
* **#define CONCAT( x, y ) x ## y**
* **When CONCAT appears in the program, its arguments are concatenated and used to replace the macro. For example, CONCAT(HELLO, C++) is replaced by "HELLO C++" in the program as follows.**
* **#include <iostream>**
* **using namespace std;**
* **#define concat(a, b) a ## b**
* **int main() {**
* **int xy = 100;**
* **cout << concat(x, y);**
* **return 0;**
* **}**
* **If we compile and run above code, this would produce the following result −**
* **100**
* **Let us see how it worked. It is simple to understand that the C++ preprocessor transforms −**
* **cout << concat(x, y);**
* **Above line will be transformed into the following line −**
* **cout << xy;**
* **Predefined C++ Macros**
* **C++ provides a number of predefined macros mentioned below −**

|  |  |
| --- | --- |
| * **Sr.No** | * **Macro & Description** |
| * **1** | * **\_\_LINE\_\_** * **This contains the current line number of the program when it is being compiled.** |
| * **2** | * **\_\_FILE\_\_** * **This contains the current file name of the program when it is being compiled.** |
| * **3** | * **\_\_DATE\_\_** * **This contains a string of the form month/day/year that is the date of the translation of the source file into object code.** |
| * **4** | * **\_\_TIME\_\_** * **This contains a string of the form hour:minute:second that is the time at which the program was compiled.** |

* **Let us see an example for all the above macros −**
* **#include <iostream>**
* **using namespace std;**
* **int main () {**
* **cout << "Value of \_\_LINE\_\_ : " << \_\_LINE\_\_ << endl;**
* **cout << "Value of \_\_FILE\_\_ : " << \_\_FILE\_\_ << endl;**
* **cout << "Value of \_\_DATE\_\_ : " << \_\_DATE\_\_ << endl;**
* **cout << "Value of \_\_TIME\_\_ : " << \_\_TIME\_\_ << endl;**
* **return 0;**
* **}**
* **If we compile and run above code, this would produce the following result −**
* **Value of \_\_LINE\_\_ : 6**
* **Value of \_\_FILE\_\_ : test.cpp**
* **Value of \_\_DATE\_\_ : Feb 28 2011**
* **Value of \_\_TIME\_\_ : 18:52:48**

**Q.7. what are the different data types in C++?**

**Ans**

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 | Range | 0 to 65,535 |
| signed short int | Range | -32768 to 32767 |
| long int | 4bytes | -2,147,483,648 to 2,147,483,647 |
| signed long int | 4bytes | same as long int |
| unsigned long int | 4bytes | 0 to 4,294,967,295 |
| float | 4bytes | +/- 3.4e +/- 38 (~7 digits) |
| double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| long double | 8bytes | +/- 1.7e +/- 308 (~15 digits) |
| 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

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.

**Q.8. what is object? Explain classes in object oriented programming with suitable example.**

**Ans:-**

## Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box −

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

## Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear −

[Live Demo](http://tpcg.io/JeEWd6)

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

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

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members cannot be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

## Classes and Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below −

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [**Class Member Functions**](https://www.tutorialspoint.com/cplusplus/cpp_class_member_functions.htm)  A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. |
| 2 | [**Class Access Modifiers**](https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm)  A class member can be defined as public, private or protected. By default members would be assumed as private. |
| 3 | [**Constructor & Destructor**](https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm)  A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted. |
| 4 | [**Copy Constructor**](https://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm)  The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| 5 | [**Friend Functions**](https://www.tutorialspoint.com/cplusplus/cpp_friend_functions.htm)  A **friend** function is permitted full access to private and protected members of a class. |
| 6 | [**Inline Functions**](https://www.tutorialspoint.com/cplusplus/cpp_inline_functions.htm)  With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. |
| 7 | [**this Pointer**](https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm)  Every object has a special pointer **this** which points to the object itself. |
| 8 | [**Pointer to C++ Classes**](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_class.htm)  A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it. |
| 9 | [**Static Members of a Class**](https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm)  Both data members and function members of a class can be declared as static. |