**Benefits of C++ over C**

1. There is Stronger Type Checking in C++.
2. All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
3. C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
4. Exception Handling is there in C++.
5. The Concept of Virtual functions and also Constructors and Destructors for Objects.
6. Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.
7. Variables can be declared anywhere in the program in C++, but must be declared before they are used.

**C++ Character Set:**

Character set is asset of valid characters that a language can recognize. A character represents any letter, digit, or any other sign. The C++ has the following character set:

**Letters:** A-Z, a-z

**Digits:**  0-9

**Special Symbols:** Space + -\* / ^ \ () [] {} = !=

**White Spaces:** Blank space, Horizontal tab, Carriage return, Newline, Form feed

**Other Characters:**  C++ can process any of the 256 ASCII characters as data or as literals.

**TOKENS**

The smallest individual unit in a program is known as a token or a lexical unit.

C++ has the following tokens:

•Keywords

• Identifiers

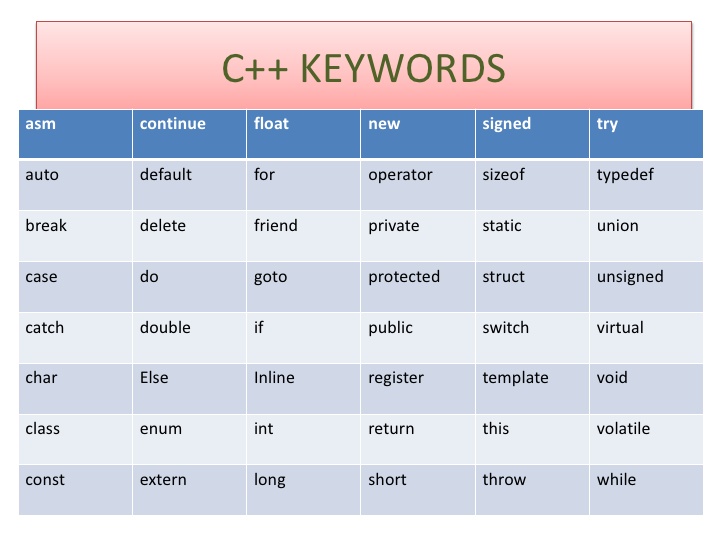
• Literals

• Punctuators

• Operators

1. **KEYWORDS**

Keywords are the words that convey a special meaning to the language compiler. These are reserved for special purpose and mus not be used as normal identifier names.



1. **Identifiers**

Identifiers are fundamental building blocks of a program and are used as the general terminology for the names given to different parts of a program viz. variables, objects, classes, functions, arrays, etc. Identifier forming rule of C++ states the following:

*An identifier is an arbitrarily long sequence of letters and digits. The first character must be a letter: the underscore (-) counts as a letter. Upper and lower-case letters are different. All characters are significant.*

Some valid identifiers:

Myfile DATE9\_7\_77 Z2T0Z9

MYFILE \_DS \_HJI3\_JK

\_CHK FILE13

Some invalid indentifiers:

DATA\_REC: contains special character- (other than A-Z, a-z and \_).

29CLCT: starting with a digit.

break: reserved keyword.

My.file: contains special character.

**3.Literals**

Literals( or constants) are data items that never change their value uring a program run. C++ allows several kinds of literals:

• Integer-constant

•Floating-constant

•Character-constant

•Boolean-constant

•string-constant

**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.

“abc”

“\ab”

“this is a pen”

**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

## Arithmetic Operators

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

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

|  |  |  |
| --- | --- | --- |
| **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:

|  |  |  |
| --- | --- | --- |
| **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 –

|  |  |  |
| --- | --- | --- |
| **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 performs 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 |