**C++ Variables, Literals and Constants**

In this tutorial, we will learn about variables, literals, and constants in C++ with the help of examples.

**C++ Variables**

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). For example,

int age = 14;

Here, age is a variable of the int data type, and we have assigned an integer value 14 to it.

**Note:** The int data type suggests that the variable can only hold integers. Similarly, we can use the double data type if we have to store decimals and exponentials.

We will learn about all the data types in detail in the next tutorial.

The value of a variable can be changed, hence the name **variable**.

int age = 14; // age is 14

age = 17; // age is 17

**Rules for naming a variable**

* A variable name can only have alphabets, numbers and the underscore \_.
* A variable name cannot begin with a number.
* Variable names cannot begin with an uppercase character.
* A variable name cannot be a [keyword](https://www.programiz.com/cpp-programming/keywords-identifiers). For example, int is a keyword that is used to denote integers.
* A variable name can start with an underscore. However, it's not considered a good practice.

**Note:** We should try to give meaningful names to variables. For example, first\_name is a better variable name than fn.

**C++ Literals**

Literals are data used for representing fixed values. They can be used directly in the code. For example: 1, 2.5, 'c' etc.

Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms.

Here's a list of different literals in C++ programming.

**1. Integers**

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

* decimal (base 10)
* octal (base 8)
* hexadecimal (base 16)

For example:

Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc

In C++ programming, octal starts with a 0, and hexadecimal starts with a 0x.

**2. Floating-point Literals**

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

**Note:** E-5 = 10-5

**3. Characters**

A character literal is created by enclosing a single character inside single quotation marks. For example: 'a', 'm', 'F', '2', '}' etc.

**4. Escape Sequences**

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C++ programming. For example, newline (enter), tab, question mark, etc.

In order to use these characters, escape sequences are used.

|  |  |
| --- | --- |
| Escape Sequences | Characters |
| \b | Backspace |
| \f | Form feed |
| \n | Newline |
| \r | Return |
| \t | Horizontal tab |
| \v | Vertical tab |
| \\ | Backslash |
| \' | Single quotation mark |
| \" | Double quotation mark |
| \? | Question mark |
| \0 | Null Character |

**5. String Literals**

A string literal is a sequence of characters enclosed in double-quote marks. For example:

|  |  |
| --- | --- |
| "good" | string constant |
| "" | null string constant |
| " " | string constant of six white space |
| "x" | string constant having a single character |
| "Earth is round\n" | prints string with a newline |

We will learn about strings in detail in the C++ string tutorial.

**C++ Constants**

In C++, we can create variables whose value cannot be changed. For that, we use the const keyword. Here's an example:

const int LIGHT\_SPEED = 299792458;

LIGHT\_SPEED = 2500 // Error! LIGHT\_SPEED is a constant.

Here, we have used the keyword const to declare a constant named LIGHT\_SPEED. If we try to change the value of LIGHT\_SPEED, we will get an error.

A constant can also be created using the #define preprocessor directive. We will learn about it in detail in the C++ Macros tutorial.

**C++ Data Types**

In this tutorial, we will learn about basic data types such as int, float, char, etc. in C++ programming with the help of examples.

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

int age = 13;

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

**C++ Fundamental Data Types**

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

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

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

**1. C++ int**

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

int salary = 85000;

**2. C++ float and double**

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

float area = 64.74;

double volume = 134.64534;

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

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

**3. C++ char**

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

char test = 'h';

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

**4. C++ wchar\_t**

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

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

Notice the letter L before the quotation marks.

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

**5. C++ bool**

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

bool cond = false;

**6. C++ void**

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

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

**C++ Type Modifiers**

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

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

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

* int
* double
* char

**C++ Modified Data Types List**

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

Let's see a few examples.

long b = 4523232;

long int c = 2345342;

long double d = 233434.56343;

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

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

**Derived Data Types**

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

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

**C++ Basic Input/Output**

In this tutorial, we will learn to use the cin object to take input from the user, and the cout object to display output to the user with the help of examples.

**C++ Output**

In C++, cout sends formatted output to standard output devices, such as the screen. We use the cout object along with the << operator for displaying output.

**Example 1: String Output**

#include <iostream>

using namespace std;

int main() {

// prints the string enclosed in double quotes

cout << "This is C++ Programming";

return 0;

}

**Output**

This is C++ Programming

**How does this program work?**

* We first include the iostream header file that allows us to display output.
* The cout object is defined inside the std namespace. To use the std namespace, we used the using namespace std; statement.
* Every C++ program starts with the main() function. The code execution begins from the start of the main() function.
* cout is an object that prints the string inside quotation marks " ". It is followed by the << operator.
* return 0; is the "exit status" of the main() function. The program ends with this statement, however, this statement is not mandatory.

**Note:** If we don't include the using namespace std; statement, we need to use std::cout instead of cout.

#include <iostream>

int main() {

// prints the string enclosed in double quotes

std::cout << "This is C++ Programming";

return 0;

}

**Example 2: Numbers and Characters Output**

To print the numbers and character variables, we use the same cout object but without using quotation marks.

#include <iostream>

using namespace std;

int main() {

int num1 = 70;

double num2 = 256.783;

char ch = 'A';

cout << num1 << endl; // print integer

cout << num2 << endl; // print double

cout << "character: " << ch << endl; // print char

return 0;

}

**Output**

70

256.783

character: A

**Notes:**

* The endl manipulator is used to insert a new line. That's why each output is displayed in a new line.
* The << operator can be used more than once if we want to print different variables, strings and so on in a single statement. For example:

cout << "character: " << ch << endl;

**C++ Input**

In C++, cin takes formatted input from standard input devices such as the keyboard. We use the cin object along with the >> operator for taking input.

**Example 3: Integer Input/Output**

#include <iostream>

using namespace std;

int main() {

int num;

cout << "Enter an integer: ";

cin >> num; // Taking input

cout << "The number is: " << num;

return 0;

}

**Output**

Enter an integer: 70

The number is: 70

In the program, we used

cin >> num;

to take input from the user. The input is stored in the variable num. We use the >> operator with cin to take input.

**Note:** If we don't include the using namespace std; statement, we need to use std::cin instead of cin.

**C++ Taking Multiple Inputs**

#include <iostream>

using namespace std;

int main() {

char a;

int num;

cout << "Enter a character and an integer: ";

cin >> a >> num;

cout << "Character: " << a << endl;

cout << "Number: " << num;

return 0;

}

**Output**

Enter a character and an integer: F

23

Character: F

Number: 23

**C++ Type Conversion**

In this tutorial, we will learn about the basics of C++ type conversion with the help of examples.

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

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

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

**Implicit Type Conversion**

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

Let us look at two examples of implicit type conversion.

**Example 1: Conversion From int to double**

// Working of implicit type-conversion

#include <iostream>

using namespace std;

int main() {

// assigning an int value to num\_int

int num\_int = 9;

// declaring a double type variable

double num\_double;

// implicit conversion

// assigning int value to a double variable

num\_double = num\_int;

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

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

return 0;

}

**Output**

num\_int = 9

num\_double = 9

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

num\_double = num\_int;

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

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

//Working of Implicit type-conversion

#include <iostream>

using namespace std;

int main() {

int num\_int;

double num\_double = 9.99;

// implicit conversion

// assigning a double value to an int variable

num\_int = num\_double;

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

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

return 0;

}

**Output**

num\_int = 9

num\_double = 9.99

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

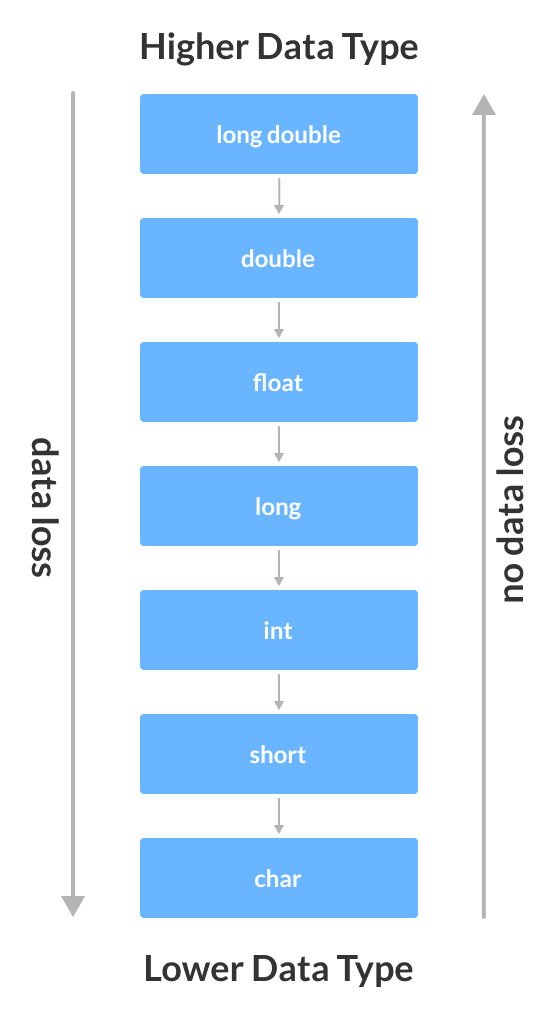
num\_double = num\_int;

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

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

**Data Loss During Conversion (Narrowing Conversion)**

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

Possible Data Loss During Type Conversion

**C++ Explicit Conversion**

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

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

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

**C-style Type Casting**

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

The syntax for this style is:

(data\_type)expression;

For example,

// initializing int variable

int num\_int = 26;

// declaring double variable

double num\_double;

// converting from int to double

num\_double = (double)num\_int;

**Function-style Casting**

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

The syntax for this style is:

data\_type(expression);

For example,

// initializing int variable

int num\_int = 26;

// declaring double variable

double num\_double;

// converting from int to double

num\_double = double(num\_int);

**Example 3: Type Casting**

#include <iostream>

using namespace std;

int main() {

// initializing a double variable

double num\_double = 3.56;

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

// C-style conversion from double to int

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

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

// function-style conversion from double to int

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

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

return 0;

}

**Output**

num\_double = 3.56

num\_int1 = 3

num\_int2 = 3

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

**Type Conversion Operators**

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

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

We will learn about these casts in later tutorials.

**Recommended Tutorials:**

* [*C++ string to int and Vice-versa*](https://www.programiz.com/cpp-programming/string-int-conversion)
* [*C++ string to float, double and Vice-versa*](https://www.programiz.com/cpp-programming/string-float-conversion)

# C++ Operators

#### In this tutorial, we will learn about the different types of operators in C++ with the help of examples. In programming, an operator is a symbol that operates on a value or a variable.

Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while - is an operator used for subtraction.

Operators in C++ can be classified into 6 types:

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

## 1. C++ Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

a + b;

Here, the + operator is used to add two variables a and b. Similarly there are various other arithmetic operators in C++.

|  |  |
| --- | --- |
| Operator | Operation |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulo Operation (Remainder after division) |

### Example 1: Arithmetic Operators

#include <iostream>

using namespace std;

int main() {

int a, b;

a = 7;

b = 2;

// printing the sum of a and b

cout << "a + b = " << (a + b) << endl;

// printing the difference of a and b

cout << "a - b = " << (a - b) << endl;

// printing the product of a and b

cout << "a \* b = " << (a \* b) << endl;

// printing the division of a by b

cout << "a / b = " << (a / b) << endl;

// printing the modulo of a by b

cout << "a % b = " << (a % b) << endl;

return 0;

}

**Output**

a + b = 9

a - b = 5

a \* b = 14

a / b = 3

a % b = 1

Here, the operators +, - and \* compute addition, subtraction, and multiplication respectively as we might have expected.

**/ Division Operator**

Note the operation (a / b) in our program. The / operator is the division operator.

As we can see from the above example, if an integer is divided by another integer, we will get the quotient. However, if either divisor or dividend is a floating-point number, we will get the result in decimals.

In C++,

7/2 is 3

7.0 / 2 is 3.5

7 / 2.0 is 3.5

7.0 / 2.0 is 3.5

**% Modulo Operator**

The modulo operator % computes the remainder. When a = 9 is divided by b = 4, the remainder is **1**.

**Note:** The % operator can only be used with integers.

### Increment and Decrement Operators

C++ also provides increment and decrement operators: ++ and -- respectively. ++ increases the value of the operand by **1**, while -- decreases it by **1**.

For example,

int num = 5;

// increasing num by 1

++num;

Here, the value of num gets increased to **6** from its initial value of **5**.

### Example 2: Increment and Decrement Operators

// Working of increment and decrement operators

#include <iostream>

using namespace std;

int main() {

int a = 10, b = 100, result\_a, result\_b;

// incrementing a by 1 and storing the result in result\_a

result\_a = ++a;

cout << "result\_a = " << result\_a << endl;

// decrementing b by 1 and storing the result in result\_b

result\_b = --b;

cout << "result\_b = " << result\_b << endl;

return 0;

}

**Output**

result\_a = 11

result\_b = 99

In the above program, we used ++ and -- operator as **prefixes**. We can also use these operators as **postfix**.

There is a slight difference when these operators are used as a prefix versus when they are used as a postfix.

To learn more about these operators, visit [increment and decrement operators](https://www.programiz.com/article/increment-decrement-operator-difference-prefix-postfix).

## 2. C++ Assignment Operators

In C++, assignment operators are used to assign values to variables. For example,

// assign 5 to a

a = 5;

Here, we have assigned a value of 5 to the variable a.

| Operator | Example | Equivalent to |
| --- | --- | --- |
| = | a = b; | a = b; |
| += | a += b; | a = a + b; |
| -= | a -= b; | a = a - b; |
| \*= | a \*= b; | a = a \* b; |
| /= | a /= b; | a = a / b; |
| %= | a %= b; | a = a % b; |

### Example 2: Assignment Operators

#include <iostream>

using namespace std;

int main() {

int a, b, temp;

// 2 is assigned to a

a = 2;

// 7 is assigned to b

b = 7;

// value of a is assigned to temp

temp = a; // temp will be 2

cout << "temp = " << temp << endl;

// assigning the sum of a and b to a

a += b; // a = a +b

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

return 0;

}

**Output**

temp = 2

a = 9

## 3. C++ Relational Operators

A relational operator is used to check the relationship between two operands. For example,

// checks if a is greater than b

a > b;

Here, > is a relational operator. It checks if a is greater than b or not.

If the relation is **true**, it returns **1** whereas if the relation is **false**, it returns **0**.

|  |  |  |
| --- | --- | --- |
| Operator | Meaning | Example |
| == | Is Equal To | 3 == 5 gives us **false** |
| != | Not Equal To | 3 != 5 gives us **true** |
| > | Greater Than | 3 > 5 gives us **false** |
| < | Less Than | 3 < 5 gives us **true** |
| >= | Greater Than or Equal To | 3 >= 5 give us **false** |
| <= | Less Than or Equal To | 3 <= 5 gives us **true** |

### Example 4: Relational Operators

#include <iostream>

using namespace std;

int main() {

int a, b;

a = 3;

b = 5;

bool result;

result = (a == b); // false

cout << "3 == 5 is " << result << endl;

result = (a != b); // true

cout << "3 != 5 is " << result << endl;

result = a > b; // false

cout << "3 > 5 is " << result << endl;

result = a < b; // true

cout << "3 < 5 is " << result << endl;

result = a >= b; // false

cout << "3 >= 5 is " << result << endl;

result = a <= b; // true

cout << "3 <= 5 is " << result << endl;

return 0;

}

**Output**

3 == 5 is 0

3 != 5 is 1

3 > 5 is 0

3 < 5 is 1

3 >= 5 is 0

3 <= 5 is 1

**Note**: Relational operators are used in decision making and loops.

## 4. C++ Logical Operators

Logical operators are used to check whether an expression is **true** or **false**. If the expression is **true**, it returns **1** whereas if the expression is **false**, it returns **0**.

|  |  |  |
| --- | --- | --- |
| Operator | Example | Meaning |
| && | expression1 **&&** expression2 | Logical AND. True only if all the operands are true. |
| || | expression1 **||** expression2 | Logical OR. True if at least one of the operands is true. |
| ! | **!**expression | Logical NOT. True only if the operand is false. |

In C++, logical operators are commonly used in decision making. To further understand the logical operators, let's see the following examples,

Suppose,

a = 5

b = 8

Then,

(a > 3) && (b > 5) evaluates to true

(a > 3) && (b < 5) evaluates to false

(a > 3) || (b > 5) evaluates to true

(a > 3) || (b < 5) evaluates to true

(a < 3) || (b < 5) evaluates to false

!(a == 3) evaluates to true

!(a > 3) evaluates to false

### Example 5: Logical Operators

#include <iostream>

using namespace std;

int main() {

bool result;

result = (3 != 5) && (3 < 5); // true

cout << "(3 != 5) && (3 < 5) is " << result << endl;

result = (3 == 5) && (3 < 5); // false

cout << "(3 == 5) && (3 < 5) is " << result << endl;

result = (3 == 5) && (3 > 5); // false

cout << "(3 == 5) && (3 > 5) is " << result << endl;

result = (3 != 5) || (3 < 5); // true

cout << "(3 != 5) || (3 < 5) is " << result << endl;

result = (3 != 5) || (3 > 5); // true

cout << "(3 != 5) || (3 > 5) is " << result << endl;

result = (3 == 5) || (3 > 5); // false

cout << "(3 == 5) || (3 > 5) is " << result << endl;

result = !(5 == 2); // true

cout << "!(5 == 2) is " << result << endl;

result = !(5 == 5); // false

cout << "!(5 == 5) is " << result << endl;

return 0;

}

**Output**

(3 != 5) && (3 < 5) is 1

(3 == 5) && (3 < 5) is 0

(3 == 5) && (3 > 5) is 0

(3 != 5) || (3 < 5) is 1

(3 != 5) || (3 > 5) is 1

(3 == 5) || (3 < 5) is 0

!(5 == 2) is 1

!(5 == 5) is 0

**Explanation of logical operator program**

* (3 != 5) && (3 < 5) evaluates to **1** because both operands (3 != 5) and (3 < 5) are **1** (true).
* (3 == 5) && (3 < 5) evaluates to **0** because the operand (3 == 5) is **0** (false).
* (3 == 5) && (3 > 5) evaluates to **0** because both operands (3 == 5) and (3 > 5) are **0** (false).
* (3 != 5) || (3 < 5) evaluates to **1** because both operands (3 != 5) and (3 < 5) are **1** (true).
* (3 != 5) || (3 > 5) evaluates to **1** because the operand (3 != 5) is **1** (true).
* (3 == 5) || (3 > 5) evaluates to **0** because both operands (3 == 5) and (3 > 5) are **0** (false).
* !(5 == 2) evaluates to **1** because the operand (5 == 2) is **0** (false).
* !(5 == 5) evaluates to **0** because the operand (5 == 5) is **1** (true).

## 5. C++ Bitwise Operators

In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside char and int data types.

|  |  |
| --- | --- |
| Operator | Description |
| & | Binary AND |
| | | Binary OR |
| ^ | Binary XOR |
| ~ | Binary One's Complement |
| << | Binary Shift Left |
| >> | Binary Shift Right |

To learn more, visit [C++ bitwise operators](https://www.programiz.com/cpp-programming/bitwise-operators).

Apart from the operators discussed above, there are a few other operators, such as sizeof, ?, ., &, etc., that cannot be neatly classified into one or another type. We will learn more about these operators in later tutorials.

**C++ Comments**

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

C++ comments are hints that a programmer can add to make their code easier to read and understand. They are completely ignored by C++ compilers.

There are two ways to add comments to code:

// - Single Line Comments

/\* \*/ -Multi-line Comments

**Single Line Comments**

In C++, any line that starts with // is a comment. For example,

// declaring a variable

int a;

// initializing the variable 'a' with the value 2

a = 2;

Here, we have used two single-line comments:

* // declaring a variable
* // initializing the variable 'a' with the value 2

We can also use single line comment like this:

int a; // declaring a variable

**Multi-line comments**

In C++, any line between /\* and \*/ is also a comment. For example,

/\* declaring a variable

to store salary to employees

\*/

int salary = 2000;

This syntax can be used to write both single-line and multi-line comments.

**Using Comments for Debugging**

Comments can also be used to disable code to prevent it from being executed. For example,

#include <iostream>

using namespace std;

int main() {

cout << "some code";

cout << ''error code;

cout << "some other code";

return 0;

}

If we get an error while running the program, instead of removing the error-prone code, we can use comments to disable it from being executed; this can be a valuable debugging tool.

#include <iostream>

using namespace std;

int main() {

cout << "some code";

// cout << ''error code;

cout << "some other code";

return 0;

}

**Pro Tip:** Remember the shortcut for using comments; it can be really helpful. For most code editors, it's Ctrl + / for Windows and Cmd + / for Mac.

**Why use Comments?**

If we write comments on our code, it will be easier for us to understand the code in the future. Also, it will be easier for your fellow developers to understand the code.

**Note:** Comments shouldn't be the substitute for a way to explain poorly written code in English. We should always write well-structured and self-explanatory code. And, then use comments.

As a general rule of thumb, use comments to explain **Why** you did something rather than **How** you did something, and you are good.