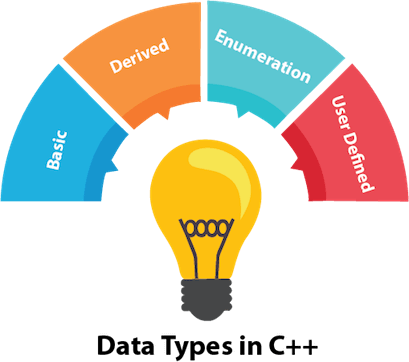
C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.



There are 4 types of data types in C++ language.

|  |  |
| --- | --- |
| **Types** | **Data Types** |
| Basic Data Type | int, char, float, double, etc |
| Derived Data Type | array, pointer, etc |
| Enumeration Data Type | enum |
| User Defined Data Type | structure |

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

Let's see the basic data types. It size is given according to 32 bit OS.

|  |  |  |
| --- | --- | --- |
| **Data Types** | **Memory Size** | **Range** |
| char | 1 byte | -128 to 127 |
| signed char | 1 byte | -128 to 127 |
| unsigned char | 1 byte | 0 to 127 |
| short | 2 byte | -32,768 to 32,767 |
| signed short | 2 byte | -32,768 to 32,767 |
| unsigned short | 2 byte | 0 to 32,767 |
| int | 2 byte | -32,768 to 32,767 |
| signed int | 2 byte | -32,768 to 32,767 |
| unsigned int | 2 byte | 0 to 32,767 |
| short int | 2 byte | -32,768 to 32,767 |
| signed short int | 2 byte | -32,768 to 32,767 |
| unsigned short int | 2 byte | 0 to 32,767 |
| long int | 4 byte |  |
| signed long int | 4 byte |  |
| unsigned long int | 4 byte |  |
| float | 4 byte |  |
| double | 8 byte |  |
| long double | 10 byte |  |

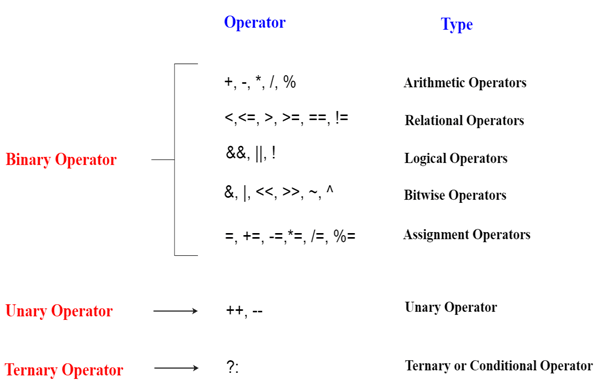
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| C++ Keywords A keyword is a reserved word. You cannot use it as a variable name, constant name etc. **A list of 32 Keywords in C++ Language which are also available in C language are given below.**   |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | auto | break | case | char | const | continue | default | do | | double | else | enum | extern | float | for | goto | if | | int | long | register | return | short | signed | sizeof | static | | struct | switch | typedef | union | unsigned | void | volatile | while |   **A list of 30 Keywords in C++ Language which are not available in C language are given below.**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | asm | dynamic\_cast | namespace | reinterpret\_cast | bool | | explicit | new | static\_cast | false | catch | | operator | template | friend | private | class | | this | inline | public | throw | const\_cast | | delete | mutable | protected | true | try | | typeid | typename | using | virtual | wchar\_t |   Next Topic[C++ Operators](https://www.javatpoint.com/cpp-operators) |

# C++ Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C language.

* Arithmetic Operators
* Relational Operators
* Logical Operators
* Bitwise Operators
* Assignment Operator
* Unary operator
* Ternary or Conditional Operator
* Misc Operator



Precedence of Operators in C++

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below:

1. **int** data=5+10\*10;

The "data" variable will contain 105 because \* (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C++ operators is given below:

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

Next Topic[C++ Identifiers](https://www.javatpoint.com/cpp-identifiers)

# C++ Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

* **Constants**
* **Variables**
* **Functions**
* **Labels**
* **Defined data types**

**Some naming rules are common in both C and C++. They are as follows:**

* Only alphabetic characters, digits, and underscores are allowed.
* Char first\_name[20];
* The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
* In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
* A declared keyword cannot be used as a variable name.

**For example,** suppose we have two identifiers, named as 'FirstName', and 'Firstname'. Both the identifiers will be different as the letter 'N' in the first case in uppercase while lowercase in second. Therefore, it proves that identifiers are case-sensitive.

### **Valid Identifiers**

**The following are the examples of valid identifiers are:**

1. Result
2. Test2
3. \_sum
4. power

### **Invalid Identifiers**

**The following are the examples of invalid identifiers:**

1. Sum-1   // containing special character '-'.
2. 2data    // the first letter is a digit.
3. **break**    // use of a keyword.

#### **Note: Identifiers cannot be used as the keywords. It may not conflict with the keywords, but it is highly recommended that the keywords should not be used as the identifier name. You should always use a consistent way to name the identifiers so that your code will be more readable and maintainable.**

The major difference between C and C++ is the limit on the length of the name of the variable. ANSI C considers only the first 32 characters in a name while ANSI C++ imposes no limit on the length of the name.

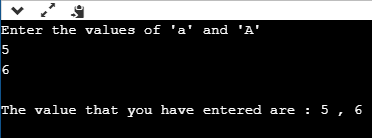
Constants are the identifiers that refer to the fixed value, which do not change during the execution of a program. Both C and C++ support various kinds of literal constants, and they do have any memory location. For example, 123, 12.34, 037, 0X2, etc. are the literal constants.

**Let's look at a simple example to understand the concept of identifiers.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a;
6. **int** A;
7. cout<<"Enter the values of 'a' and 'A'";
8. cin>>a;
9. cin>>A;
10. cout<<"\nThe values that you have entered are : "<<a<<" , "<<A;
11. **return** 0;
12. }

In the above code, we declare two variables 'a' and 'A'. Both the letters are same but they will behave as different identifiers. As we know that the identifiers are the case-sensitive so both the identifiers will have different memory locations.

**Output**



### **What are the keywords?**

Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes.

**Differences between Identifiers and Keywords**

**The following is the list of differences between identifiers and keywords:**

|  |  |
| --- | --- |
| **Identifiers** | **Keywords** |
| Identifiers are the names defined by the programmer to the basic elements of a program. | Keywords are the reserved words whose meaning is known by the compiler. |
| It is used to identify the name of the variable. | It is used to specify the type of entity. |
| It can consist of letters, digits, and underscore. | It contains only letters. |
| It can use both lowercase and uppercase letters. | It uses only lowercase letters. |
| No special character can be used except the underscore. | It cannot contain any special character. |
| The starting letter of identifiers can be lowercase, uppercase or underscore. | It can be started only with the lowercase letter. |
| It can be classified as internal and external identifiers. | It cannot be further classified. |
| Examples are test, result, sum, power, etc. | Examples are 'for', 'if', 'else', 'break', etc. |

Next Topic[C++ Expression](https://www.javatpoint.com/cpp-expression)

# C++ Expression

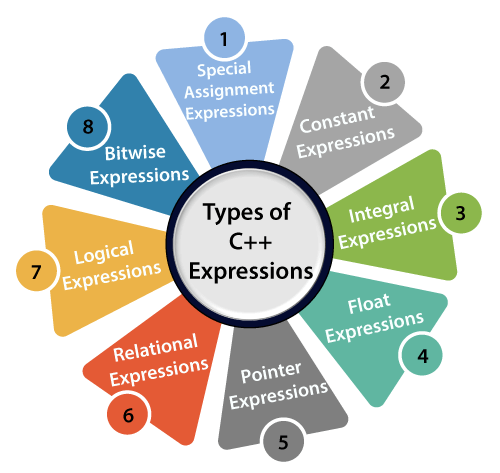
C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values. An expression can consist of one or more operands, zero or more operators to compute a value. Every expression produces some value which is assigned to the variable with the help of an assignment operator.

**Examples of C++ expression:**

1. (a+b) - c
2. (x/y) -z
3. 4a2 - 5b +c
4. (a+b) \* (x+y)

### **An expression can be of following types:**

* Constant expressions
* Integral expressions
* Float expressions
* Pointer expressions
* Relational expressions
* Logical expressions
* Bitwise expressions
* Special assignment expressions



If the expression is a combination of the above expressions, such expressions are known as compound expressions.

### **Constant expressions**

A constant expression is an expression that consists of only constant values. It is an expression whose value is determined at the compile-time but evaluated at the run-time. It can be composed of integer, character, floating-point, and enumeration constants.

Constants are used in the following situations:

* It is used in the subscript declarator to describe the array bound.
* It is used after the case keyword in the switch statement.
* It is used as a numeric value in an **enum**
* It specifies a bit-field width.
* It is used in the pre-processor **#if**

In the above scenarios, the constant expression can have integer, character, and enumeration constants. We can use the static and extern keyword with the constants to define the function-scope.

The following table shows the expression containing constant value:

|  |  |
| --- | --- |
| **Expression containing constant** | **Constant value** |
| x = (2/3) \* 4 | (2/3) \* 4 |
| extern int y = 67 | 67 |
| int z = 43 | 43 |
| static int a = 56 | 56 |

Let's see a simple program containing constant expression:

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** x;        // variable declaration.
6. x=(3/2) + 2;  // constant expression
7. cout<<"Value of x is : "<<x;  // displaying the value of x.
8. **return** 0;
9. }

In the above code, we have first declared the 'x' variable of integer type. After declaration, we assign the simple constant expression to the 'x' variable.

**Output**

Value of x is : 3

### **Integral Expressions**

An integer expression is an expression that produces the integer value as output after performing all the explicit and implicit conversions.

**Following are the examples of integral expression:**

1. (x \* y) -5
2. x + **int**(9.0)
3. where x and y are the integers.

**Let's see a simple example of integral expression:**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** x;  // variable declaration.
6. **int** y;  // variable declaration
7. **int** z;  // variable declaration
8. cout<<"Enter the values of x and y";
9. cin>>x>>y;
10. z=x+y;
11. cout<<"\n"<<"Value of z is :"<<z; //  displaying the value of z.
12. **return** 0;
13. }

In the above code, we have declared three variables, i.e., x, y, and z. After declaration, we take the user input for the values of 'x' and 'y'. Then, we add the values of 'x' and 'y' and stores their result in 'z' variable.

**Output**

Enter the values of x and y

8

9

Value of z is :17

**Let's see another example of integral expression.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
6. **int** x;   // variable declaration
7. **int** y=9;    // variable initialization
8. x=y+**int**(10.0);    // integral expression
9. cout<<"Value of x : "<<x;   // displaying the value of x.
10. **return** 0;
11. }

In the above code, we declare two variables, i.e., x and y. We store the value of expression (y+int(10.0)) in a 'x' variable.

**Output**

Value of x : 19

### **Float Expressions**

A float expression is an expression that produces floating-point value as output after performing all the explicit and implicit conversions.

The following are the examples of float expressions:

1. x+y
2. (x/10) + y
3. 34.5
4. x+**float**(10)

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
6. **float** x=8.9;      // variable initialization
7. **float** y=5.6;      // variable initialization
8. **float** z;             // variable declaration
9. z=x+y;
10. std::cout <<"value of z is :"  << z<<std::endl;  // displaying the value of z.

13. **return** 0;
14. }

**Output**

value of z is :14.5

**Let's see another example of float expression.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **float** x=6.7;    // variable initialization
6. **float** y;      // variable declaration
7. y=x+**float**(10);   // float expression
8. std::cout <<"value of y is :"  << y<<std::endl;  // displaying the value of y
9. **return** 0;
10. }

In the above code, we have declared two variables, i.e., x and y. After declaration, we store the value of expression (x+float(10)) in variable 'y'.

**Output**

value of y is :16.7

### **Pointer Expressions**

A pointer expression is an expression that produces address value as an output.

**The following are the examples of pointer expression:**

1. &x
2. ptr
3. ptr++
4. ptr-

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
6. **int** a[]={1,2,3,4,5};  // array initialization
7. **int** \*ptr;       // pointer declaration
8. ptr=a;    // assigning base address of array to the pointer ptr
9. ptr=ptr+1;   // incrementing the value of pointer
10. std::cout <<"value of second element of an array : "  << \*ptr<<std::endl;
11. **return** 0;
12. }

In the above code, we declare the array and a pointer ptr. We assign the base address to the variable 'ptr'. After assigning the address, we increment the value of pointer 'ptr'. When pointer is incremented then 'ptr' will be pointing to the second element of the array.

**Output**

value of second element of an array : 2

### **Relational Expressions**

A relational expression is an expression that produces a value of type bool, which can be either true or false. It is also known as a boolean expression. When arithmetic expressions are used on both sides of the relational operator, arithmetic expressions are evaluated first, and then their results are compared.

**The following are the examples of the relational expression:**

1. a>b
2. a-b >= x-y
3. a+b>80

**Let's understand through an example**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=45;    // variable declaration
6. **int** b=78;    // variable declaration
7. **bool** y= a>b;   // relational expression
8. cout<<"Value of y is :"<<y;  // displaying the value of y.
9. **return** 0;
10. }

In the above code, we have declared two variables, i.e., 'a' and 'b'. After declaration, we have applied the relational operator between the variables to check whether 'a' is greater than 'b' or not.

**Output**

Value of y is :0

**Let's see another example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=4;     // variable declaration
6. **int** b=5;     // variable declaration
7. **int** x=3;     // variable declaration
8. **int** y=6;    // variable declaration
9. cout<<((a+b)>=(x+y));   // relational expression
10. **return** 0;
11. }

In the above code, we have declared four variables, i.e., 'a', 'b', 'x' and 'y'. Then, we apply the relational operator (>=) between these variables.

**Output**

1

### **Logical Expressions**

A logical expression is an expression that combines two or more relational expressions and produces a bool type value. The logical operators are '&&' and '||' that combines two or more relational expressions.

The following are some examples of logical expressions:

1. a>b && x>y
2. a>10 || b==5

**Let's see a simple example of logical expression.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=2;
6. **int** b=7;
7. **int** c=4;
8. cout<<((a>b)||(a>c));
9. **return** 0;
10. }

**Output**

0

### **Bitwise Expressions**

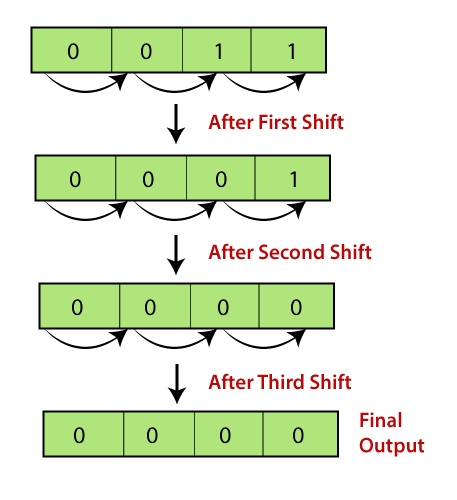
A bitwise expression is an expression which is used to manipulate the data at a bit level. They are basically used to shift the bits.

For example:

x=3

x>>3 // This statement means that we are shifting the three-bit position to the right.

In the above example, the value of 'x' is 3 and its binary value is 0011. We are shifting the value of 'x' by three-bit position to the right. Let's understand through the diagrammatic representation.



**Let's see a simple example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** x=5;   // variable declaration
6. std::cout << (x>>1) << std::endl;
7. **return** 0;
8. }

In the above code, we have declared a variable 'x'. After declaration, we applied the bitwise operator, i.e., right shift operator to shift one-bit position to right.

**Output**

2

**Let's look at another example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** x=7;   // variable declaration
6. std::cout << (x<<3) << std::endl;
7. **return** 0;
8. }

In the above code, we have declared a variable 'x'. After declaration, we applied the left shift operator to variable 'x' to shift the three-bit position to the left.

**Output**

56

### **Special Assignment Expressions**

Special assignment expressions are the expressions which can be further classified depending upon the value assigned to the variable.

* **Chained Assignment**

Chained assignment expression is an expression in which the same value is assigned to more than one variable by using single statement.

**For example:**

1. a=b=20
2. or
3. (a=b) = 20

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
5. **int** a;   // variable declaration
6. **int** b;   // variable declaration
7. a=b=80;  // chained assignment
8. std::cout <<"Values of 'a' and 'b' are : " <<a<<","<<b<< std::endl;
9. **return** 0;
10. }

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we have assigned the same value to both the variables using chained assignment expression.

**Output**

Values of 'a' and 'b' are : 80,80

#### **Note: Using chained assignment expression, the value cannot be assigned to the variable at the time of declaration. For example, int a=b=c=90 is an invalid statement.**

* **Embedded Assignment Expression**

An embedded assignment expression is an assignment expression in which assignment expression is enclosed within another assignment expression.

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a;  // variable declaration
6. **int** b;  // variable declaration
7. a=10+(b=90);  // embedded assignment expression
8. std::cout <<"Values of 'a' is " <<a<< std::endl;
9. **return** 0;
10. }

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we applied embedded assignment expression (a=10+(b=90)).

**Output**

Values of 'a' is 100

* **Compound Assignment**

A compound assignment expression is an expression which is a combination of an assignment operator and binary operator.

**For example,**

1. a+=10;
2. a=a+10

In the above statement, 'a' is a variable and '+=' is a compound statement.

**Let's understand through an example.**

1. #include <iostream>
2. **using** **namespace** std;
3. **int** main()
4. {
5. **int** a=10;   // variable declaration
6. a+=10;    // compound assignment
7. std::cout << "Value of a is :" <<a<< std::endl; // displaying the value of a.
8. **return** 0;
9. }

In the above code, we have declared a variable 'a' and assigns 10 value to this variable. Then, we applied compound assignment operator (+=) to 'a' variable, i.e., a+=10 which is equal to (a=a+10). This statement increments the value of 'a' by 10.

**Output**

Value of a is :20