**Operators**

# Operators in C / C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.

Allow to perform different kinds of operations on operands.

# Types of operators

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Unary operator
7. Misc Operators

# Arithmetic Operators

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example (let A = 10 B = 30) |
| + | Adds two operands | C = A+B; // C = 40 |
| - | Subtracts second from first | C = B –A; // C = 20 |
| \* | Multiplies both | C = A\*B; // C = 300 |
|  | Divides numerator | C = B/A; // C = 3 |
| % | Remainder after an integer division | C = B%A; // C = 0 |
| ++ | Prefix increment operator Postfix increment operator | C = ++A; // C = 11 and A = 11 C= A++; // C = 10 and A = 11 |
| -- | Prefix decrement operator Postfix decrement operator | C = --B; // C = 29 and B = 29 C = B--; // C = 30 and B = 29 |

* Increment and decrement operator require l-value expression providing an r-value or a const qualified variable results in compilation error.
* Sign of left operand is appended to result in case of modulus operator (%) in C
* % works on integer types only not for floating types

fmod(x, y) should be used for floating point

## Pre-increment (or pre-decrement) in C++

In C++, pre-increment (or pre-decrement) can be used as l-value, but post-increment (or post-decrement) cannot be used as l-value.

#include <stdio.h>

int main() {

int a = 10;

++a = 20; // works

//a++ = 20; // CE: lvalue required as left operand of assignment

printf("a = %d", a);

return 0;

}

Output: a = 20

**How ++a is different from a++ as lvalue?**

It is because ++a returns an lvalue, which is basically a reference to the variable to which we can further assign — just like an ordinary variable. It could also be assigned to a reference as follows:

int &ref = ++a; // valid

int &ref = a++; // invalid

Whereas if you recall how a++ works, it doesn’t immediately increment the value it holds. So what basically happens is that a++ returns an rvalue, which is basically just a value like the value of an expression which is not stored. You can think of a++ = 20; as follows after being processed:

int a = 10;

// On compilation, a++ is replaced by the value of a which is an rvalue:

10 = 20; // Invalid

// Value of a is incremented

a = a + 1;

## Execution of printf with ++ operators

printf("%d %d %d", i, ++i, i++);

* This reuslts in undefined behaviour by referencing both ‘i’ and ‘i++’ or '--i' in the argument list.
* It is not defined in which order the arguments are evaluated.
* Different compilers may choose different orders. A single compiler can also choose different orders at different times.
* It is recommended not to do two or more than two pre or post increment operators in the same statement.

// All three printf() statements in this cause undefined behavior

#include <stdio.h>

int main() {

volatile int a = 10;

printf("\n %d %d", a, a++);

a = 10;

printf("\n %d %d", a++, a);

a = 10;

printf("\n %d %d %d ", a, a++, ++a);

return 0;

}

# Relational Operators

Used for comparison of two values

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example (let A = 10 and B = 20) |
| == | 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. |

In C, data type of result of comparison operations is int

In C++, data type of result of comparison operations is bool

printf("%zu", sizeof(x == y));

Output

4 // In C

1 // In C++

# Logical Operators

Used to combine two or more conditions/constraints

To complement the evaluation of the original condition in considerarion

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example (let A=1 and B=0) |
| && | 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. |

Although logical operators are commutative, their order is critical

Short-circuiting in logical operators

* In case of logical AND, second operand is not evaluated if first operand is false
* In case of logical OR, second operand is not evaluated if first operand is true

|  |  |  |
| --- | --- | --- |
| #include <stdio.h>  int main() {  int a = 10, b = 4;  bool res1 = ((a == b) && printf("Ratnesh"));  bool res2 = ((a != b) && printf("Tiwari"));  return 0;  }  Output:  Tiwari |  | #include <stdio.h>  int main() {  int a = 10, b = 4;  bool res1 = ((a == b) || printf("Ratnesh"));  bool res2 = ((a != b) || printf("Tiwari"));  return 0;  }  Output:  Ratnesh |

# Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation

|  |  |  |
| --- | --- | --- |
| Operator | Description | Example Let (A = 60 // 0011 1100 and  B = 13 // 0000 1101) |
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) = 12  // 0000 1100 |
| | | Binary OR Operator copies a bit if it exists in either operand. | (A | B) = 61  // 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) = 49  // 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) = -61  // 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 = 240  // 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 = 15  // 0000 1111 |

NOTE:

1. **Do not shift a negative number of bits or more bits than exist in the operand**

Result of << and >> is undefined behavior if any of the operand is a negative number

If number is shifted more than the size of integer, the behavior is undefined

1. **Bitwise XOR operator is the most useful operator**

* To find the odd occurring number where all elements occur even number of times except one number
* To find the Missing Number
* To swap two numbers without using a temporary variable
* To create a memory efficient doubly linked list
* To find the two non-repeating elements
* To find the two numbers with odd occurences in an unsorted-array
* To add two numbers without using arithmetic operators
* To swap bits in a given number/
* To count number of bits to be flipped to convert a to b
* To find the element that appears once
* To detect if two integers have opposite signs

1. **Bitwise operators should not be used in place of logical operators**

* result of logical operator is either 0 or 1 but bitwise operators return an integer value
* logical operators consider any non-zero operand as 1

int x = 2, y = 5;

(x & y) ? printf("True ") : printf("False ");

(x && y) ? printf("True ") : printf("False ");

Output:

False True

1. **left-shift is equivalent to multiplication by 2**

**left-shift is equivalent to division by 2**

Works only if numbers are positive

int x = 19;

printf("x << 1 = %d\n", x << 1);

printf("x >> 1 = %d\n", x >> 1);

Output:

x << 1 = 38

x >> 1 = 9

Your compiler has an optimizer in it that knows how to multiply as quickly as your target process architecture is capable so you always write your intent clearly (i\*2 rather than i<<1) and let it decide what the fastest assembly/machine code sequence is.

1. **& operator can be used to quickly check if a number is odd or even**

value of expression (x & 1) would be

non-zero only if x is odd, otherwise the value would be

zero if x is even

int x = 19;

(x & 1) ? printf("Odd") : printf("Even");

Output:

Odd

1. **~ operator should be used carefully**

Result of ~ operator on a small number can be a

big number if result is stored in a unsigned variable

-ve ,, ,, ,, ,, ,, ,, signed variable

(Assumption: -ve numbers are stored in 2’s complement form where leftmost bit is the sign bit)

## Bit mask

Reference: <https://www.learncpp.com/cpp-tutorial/bit-manipulation-with-bitwise-operators-and-bit-masks/>

A bit mask is a predefined set of bits that is used to select which specific bits will be modified by subsequent operations.

### **Defining bit masks in C++11 or earlier**

Because C++11 does not support binary literals, we have to use other methods to set the symbolic constants.

There are two good methods for doing this.

* **use hexadecimal:**

constexpr std::uint8\_t mask0{ 0x1 }; // hex for 0000 0001

constexpr std::uint8\_t mask1{ 0x2 }; // hex for 0000 0010

constexpr std::uint8\_t mask2{ 0x4 }; // hex for 0000 0100

constexpr std::uint8\_t mask3{ 0x8 }; // hex for 0000 1000

This can be a little hard to read. One way to make it easier is to use the

* **left-shift operator to shift a bit into the proper location:**

constexpr std::uint8\_t mask0{ 1 << 0 }; // 0000 0001

constexpr std::uint8\_t mask1{ 1 << 1 }; // 0000 0010

constexpr std::uint8\_t mask2{ 1 << 2 }; // 0000 0100

constexpr std::uint8\_t mask3{ 1 << 3 }; // 0000 1000

### **Defining bit masks in C++14**

Because C++14 supports binary literals, defining these bit masks is easy:

constexpr std::uint8\_t mask0{ 0b0000'0001 }; // represents bit 0

constexpr std::uint8\_t mask1{ 0b0000'0010 }; // represents bit 1

constexpr std::uint8\_t mask2{ 0b0000'0100 }; // represents bit 2

constexpr std::uint8\_t mask3{ 0b0000'1000 }; // represents bit 3

**Testing a bit (to see if it is on or off)**

std::uint8\_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

std::cout << "bit 0 is " << ((flags & mask0) ? "on\n" : "off\n");

std::cout << "bit 1 is " << ((flags & mask1) ? "on\n" : "off\n");

**Setting a bit**

std::uint8\_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

std::cout << "bit 1 is " << ((flags & mask1) ? "on\n" : "off\n");

flags |= mask1; // turn on bit 1

std::cout << "bit 1 is " << ((flags & mask1) ? "on\n" : "off\n");

**Resetting a bit**

std::uint8\_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");

flags &= ~mask2; // turn off bit 2

std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");

**Flipping a bit**

std::uint8\_t flags{ 0b0000'0101 }; // 8 bits in size means room for 8 flags

std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");

flags ^= mask2; // flip bit 2

std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");

flags ^= mask2; // flip bit 2

std::cout << "bit 2 is " << ((flags & mask2) ? "on\n" : "off\n");

**Bit masks and std::bitset**

std::bitset supports the full set of bitwise operators. So even though it’s easier to use the functions (test, set, reset, and flip) to modify individual bits, you can use bitwise operators and bit masks if you want.

For more info visit: <https://en.cppreference.com/w/cpp/utility/bitset>

# Assignment Operators

|  |  |  |
| --- | --- | --- |
| 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 |

# Unary operator

Unary operator are operators that act upon a single operand to produce a new value.

Types of unary operators:

1. Unary minus(-)
2. Increment(++)
3. Decrement(- -)
4. Logical NOT(!)
5. Addressof operator(&)
6. sizeof()

# Misc Operators

# sizeof() operator

* compile time unary operator (works at compile time)
* Operand of size of could be a type name or an expression
* used to compute the size of its operand
* result of sizeof is of unsigned integral type which is usually denoted by size\_t
* can be applied to any data-type (primitive types or user defined types)
* When operand is a data type it simply returns the amount of memory is allocated to that data type

printf("%d", sizeof(int)) // 4

* When operand is an expression, it returns size of the expression

int a = 0; double d = 10.21;

printf("%d", sizeof(a+d)) // 8

* Used to find out number of elements in an array

printf("Number of elements:%lu ", sizeof(arr) / sizeof(arr[0]));

* To allocate a block of memory dynamically

int\* ptr = (int\*)malloc(10 \* sizeof(int));

* an expression does not get evaluated inside size of operator

int i = 5, j = 10, k = 15;

printf("%d", sizeof(k /= (i+j))); // 4

printf("%d", k); // 15

as expression is not evaluated value of k will not be changed

size of operator returns sizeof(int) because the result of expression will be an integer

# conditional operator (? :)

* It’s a ternary operator

exp1 ? exp2 : exp3

* exp1 is evaluated

if it’s true

* + then exp2 is evaluated and becomes the value of entire ? : expression

else

* + then exp3 is evaluated and becomes the value of entire ? : expression
* can be used to replace if-else statements
* can be nested
  + exp1 ? (exp2 ? exp3 : exp4) : (exp5 ? exp6 : exp7)

The expression exp1 will be evaluated always. If the outcome of exp1 is non zero exp2 will be evaluated, otherwise exp3 will be evaluated.

**Side Effects**

Any side effects of exp1 will be evaluated and updated immediately before executing exp2 or exp3. In other words, there is sequence point after the evaluation of condition in the ternary expression. If either exp2 or exp3 have side effects, only one of them will be evaluated.

**Return Type (VERY IMPORTANT)**

It is another interesting fact. The ternary operator has return type. The return type depends on exp2, and convertibility of exp3 into exp2 as per usual overloaded conversion rules. If they are not convertible, the compiler throws an error.

The following program compiles without any error. The return type of ternary expression is expected to be float (as that of exp2) and exp3 (i.e. literal zero – int type) is implicitly convertible to float.

#include <iostream>

using namespace std;

int main() {

int test = 0;

float fvalue = 3.111f;

cout << (test ? fvalue : 0) << endl;

return 0;

}

The following program **may compile**, or but fails at runtime. The return type of ternary expression is bounded to type (char \*), yet the expression returns int, hence the program fails. Literally, the program tries to print string at 0th address at runtime.

#include <iostream>

using namespace std;

int main() {

int test = 0;

cout << (test ? "A String" : 0) << endl;

return 0;

}

# Comma in C and C++

## Comma as a separator

Comma acts as a separator when used with function calls and definitions

/\* comma as a separator \*/

int a = 1, b = 2;

void fun(x, y);

/\* Comma acts as a separator here and doesn't enforce any sequence.

Therefore, either f1() or f2() can be called first \*/

void fun(f1(), f2());

## Comma operator in place of a semicolon

Comma operator is also used to terminate the statement after satisfying the following rules:

* The variable declaration statements must be terminated with semicolon.
* The statements after declaration statement can be terminated by comma operator.
* The last statement of the program must be terminated by semicolon.

#include <iostream>

using namespace std;

int main() {

cout << "First Line\n",

cout << "Second Line\n",

cout << "Third Line\n",

cout << "Last line";

return 0;

}

Output:

First Line

Second Line

Third Line

Last line

## Comma as an operator

The comma (,) operator is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).

The comma operator has the lowest precedence of any C operator, and acts as a sequence point.

/\* 10 is assigned to i\*/

int i = (5, 10);

/\* f1() is called (evaluated) first followed by f2(). The returned value of f2() is assigned to j \*/

int j = (f1(), f2());

## Result of comma operator as l-value in C and C++

In C, The result of a comma expression is not an lvalue.

In C++, the result is an lvalue if the right operand is an lvalue.

The following statements are equivalent:

r = (a,b,...,c);

a; b; r = c;

The difference is that the comma operator may be suitable for expression contexts, such as loop control expressions.

Similarly, the address of a compound expression can be taken if the right operand is an lvalue.

&(a, b)

a, &b

Any number of expressions separated by commas can form a single expression because the comma operator is associative. The use of the comma operator guarantees that the subexpressions will be evaluated in left-to-right order, and the value of the last becomes the value of the entire expression. In the following example, if omega has the value 11, the expression increments delta and assigns the value 3 to alpha:

alpha = (delta++, omega % 4);

The primary use of the comma operator is to produce side effects in the following situations:

* Calling a function
* Entering or repeating an iteration loop
* Testing a condition
* Other situations where a side effect is required but the result of the expression is not immediately needed
* The following table gives some examples of the uses of the comma operator.

|  |  |
| --- | --- |
| Statement | Effects |
| for (i=0; i<2; ++i, f() ); | A for statement in which i is incremented and f() is called at each iteration. |
| if ( f(), ++i, i>1 )  { /\* ... \*/ } | An if statement in which function f() is called, variable i is incremented, and variable i is tested against a value. The first two expressions within this comma expression are evaluated before the expression i>1. Regardless of the results of the first two expressions, the third is evaluated and its result determines whether the if statement is processed. |
| func( ( ++a, f(a) ) ); | A function call to func() in which a is incremented, the resulting value is passed to a function f(), and the return value of f() is passed to func(). The function func() is passed only a single argument, because the comma expression is enclosed in parentheses within the function argument list. |

|  |  |  |
| --- | --- | --- |
| #include <stdio.h>  int main() {  int a = 1,2,3;  printf("%d", a);  return 0;  }  // C CE: expected identifier or ‘(’ before numeric constant  // C++ CE: expected unqualified-id before numeric constant | #include <stdio.h>  int main() {  int a;  a = 1,2,3;  printf("%d", a);  return 0;  }  Output: 1 | #include <stdio.h>  int main() {  int a = (1,2,3);  printf("%d", a);  return 0;  }  Output: 3 |
| As a separator  Compilation error  -------------  Compiler fails to create integer variable 2 because 2 is not a valid identifier | As an operator  Assignment operator takes precedence over comma | As an operator  Brackets are used so comma operator is executed first and we get the output as 3 |

# Dot (.) and Arrow (->) operator

* Member access operator
* Used to reference individual members of classes, structures and unions
* Dot operator is applied to the actual object
* Arrow operator is used with a pointer to an object

Example

struct employee {

Char first\_name[16];

Int age;

} emp, \*p\_emp;

strcpy(emp.first\_name, “Zara”);

strcpy(p\_emp->first\_name, “Zara”);

# Pointer Operator (& and \*)

Address of operator (&) and Indirection operator (\*)

## & Operator

* Unary operator
* Returns the memory address of its operand
* Same precedence as the other unary operators
* Right to left associativity

## \* Opeartor

* Unary operator
* Returns the value of variable located at the address specified by its operand
* It’s the complement of &

int var = 3000;

int \*ptr = &var;

int val = \*ptr;

cout << "var " << var << endl;

cout << "ptr " << ptr << endl;

cout << "val " << val << endl;

var 3000

ptr 0x7ffe97f2f108

val 3000

# type cast operator

unary operator

same precedence as any other unary operartor

forces one data type to be converted into another

(type\_name) expression; // C – notation

type\_name (expression); // C++ - notation only if type name is an identifier

(type\_name) expression; // C++ - notation if type name is not an identifier

Other casting operators:

const\_cast <type> (expr);

dynamic\_cast <type> (expr);

reinterpret\_cast<type> (expr);

static\_cast <type> (expr);

Check document related to these individual cast operator.

# Other operators

|  |  |
| --- | --- |
| Operator | Description |
| :: (Scope resolution operator) | To uncover a hidden variable |
| ::\* (Pointer-to-member declarator) | To declare a pointer to a member of a class |
| ->\* (Pointer-to-member operator) | To access a member using a pointer to the object and a pointer to that member |
| .\* (Pointer-to-member operator) | To access a member using an object and a pointer to that member |
| \* | To access a member using object name and a pointer to that member |
| new (memory allocation operator) | To allocate memory |
| delete (memory release operator) | To release memory |

# Operator Precedence in C++

<https://en.cppreference.com/w/cpp/language/operator_precedence>

|  |  |  |  |
| --- | --- | --- | --- |
| Precedence | Operator | Description | Associativity |
| 1 | :: | [Scope resolution](https://en.cppreference.com/w/cpp/language/identifiers#Qualified_identifiers) | Left-to-right |
| 2 | a++   a−− | Suffix/postfix [increment and decrement](https://en.cppreference.com/w/cpp/language/operator_incdec) |
| *type*()   *type*{} | [Functional cast](https://en.cppreference.com/w/cpp/language/explicit_cast) |
| a() | [Function call](https://en.cppreference.com/w/cpp/language/operator_other#Built-in_function_call_operator) |
| a[] | [Subscript](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_subscript_operator) |
| .   -> | [Member access](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_member_access_operators) |
| 3 | ++a   −−a | Prefix [increment and decrement](https://en.cppreference.com/w/cpp/language/operator_incdec) | Right-to-left |
| +a   −a | Unary [plus and minus](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Unary_arithmetic_operators) |
| !   ~ | [Logical NOT](https://en.cppreference.com/w/cpp/language/operator_logical) and [bitwise NOT](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) |
| (*type*) | [C-style cast](https://en.cppreference.com/w/cpp/language/explicit_cast) |
| \*a | [Indirection](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_indirection_operator) (dereference) |
| &a | [Address-of](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_address-of_operator) |
| sizeof | [Size-of](https://en.cppreference.com/w/cpp/language/sizeof)[[note 1]](https://en.cppreference.com/w/cpp/language/operator_precedence#cite_note-1) |
| co\_await | await-expression (C++20) |
| new   new[] | [Dynamic memory allocation](https://en.cppreference.com/w/cpp/language/new) |
| delete   delete[] | [Dynamic memory deallocation](https://en.cppreference.com/w/cpp/language/delete) |
| 4 | .\*   ->\* | [Pointer-to-member](https://en.cppreference.com/w/cpp/language/operator_member_access#Built-in_pointer-to-member_access_operators) | Left-to-right |
| 5 | a\*b   a/b   a%b | [Multiplication, division, and remainder](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Multiplicative_operators) |
| 6 | a+b   a−b | [Addition and subtraction](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Additive_operators) |
| 7 | <<   >> | Bitwise [left shift and right shift](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_shift_operators) |
| 8 | <=> | [Three-way comparison operator](https://en.cppreference.com/w/cpp/language/operator_comparison#Three-way_comparison) (since C++20) |
| 9 | <   <= | For [relational operators](https://en.cppreference.com/w/cpp/language/operator_comparison) < and ≤ respectively |
| >   >= | For [relational operators](https://en.cppreference.com/w/cpp/language/operator_comparison) > and ≥ respectively |
| 10 | ==   != | For [relational operators](https://en.cppreference.com/w/cpp/language/operator_comparison) = and ≠ respectively |
| 11 | & | [Bitwise AND](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) |
| 12 | ^ | [Bitwise XOR](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) (exclusive or) |
| 13 | | | [Bitwise OR](https://en.cppreference.com/w/cpp/language/operator_arithmetic#Bitwise_logic_operators) (inclusive or) |
| 14 | && | [Logical AND](https://en.cppreference.com/w/cpp/language/operator_logical) |
| 15 | || | [Logical OR](https://en.cppreference.com/w/cpp/language/operator_logical) |
| 16 | a?b:c | [Ternary conditional](https://en.cppreference.com/w/cpp/language/operator_other#Conditional_operator)[[note 2]](https://en.cppreference.com/w/cpp/language/operator_precedence#cite_note-2) | Right-to-left |
| throw | [throw operator](https://en.cppreference.com/w/cpp/language/throw) |
| co\_yield | yield-expression (C++20) |
| = | [Direct assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_direct_assignment) (provided by default for C++ classes) |
| +=   −= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by sum and difference |
| \*=   /=   %= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by product, quotient, and remainder |
| <<=   >>= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by bitwise left shift and right shift |
| &=   ^=   |= | [Compound assignment](https://en.cppreference.com/w/cpp/language/operator_assignment#Builtin_compound_assignment) by bitwise AND, XOR, and OR |
| 17 | , | [Comma](https://en.cppreference.com/w/cpp/language/operator_other#Built-in_comma_operator) | Left-to-righ |

* Best thing is to use brackets, especially for less commonly used operators
* Associativity is only used when there are two or more operators of same precedence
* All operators with same precedence have same associativity necessary for compiler to decide evaluation order of an expression
* There is no chaining of comparison operators in C and C++
* When parsing an expression, an operator which is listed on some row of the table above with a precedence will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it with a lower precedence. For example, the expressions std::cout << a & b and \*p++ are parsed as (std::cout << a) & b and \*(p++), and not as std::cout << (a & b) or (\*p)++.
* Operators that have the same precedence are bound to their arguments in the direction of their associativity. For example, the expression a = b = c is parsed as a = (b = c), and not as (a = b) = c because of right-to-left associativity of assignment, but a + b − c is parsed (a + b) − c and not a + (b − c) because of left-to-right associativity of addition and subtraction.
* Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (delete ++\*p is delete(++(\*p))) and unary postfix operators always associate left-to-right (a[1][2]++ is ((a[1])[2])++). Note that the associativity is meaningful for member access operators, even though they are grouped with unary postfix operators: a.b++ is parsed (a.b)++ and not a.(b++).
* Operator precedence is unaffected by operator overloading. For example, std::cout << a ? b : c; parses as (std::cout << a) ? b : c; because the precedence of arithmetic left shift is higher than the conditional operator.

Notes

* Precedence and associativity are compile-time concepts and are independent from order of evaluation, which is a runtime concept.
* The standard itself doesn't specify precedence levels. They are derived from the grammar.
* const\_cast, static\_cast, dynamic\_cast, reinterpret\_cast, typeid, sizeof..., noexcept and alignof are not included since they are never ambiguous.
* In C, the ternary conditional operator has higher precedence than assignment operators. Therefore, the expression e = a < d ? a++ : a = d, which is parsed in C++ as e = ((a < d) ? (a++) : (a = d)), will fail to compile in C due to grammatical or semantic constraints in C. See the corresponding C page for details.

e = a < d ? a++ : a = d;

In C, CE: lvalue required as left operand of assignment

e = a < d ? a++ : a = d;

^

# Evaluation order of operands

#include <stdio.h>

int x = 0;

int f1() {

x = 5; return x;

}

int f2() {

x = 10; return x;

}

int main() {

int p = f1() + f2();

printf("p: %d, x: %d", p, x);

return 0;

}

Output: p: 15, x: 10

* The output is undefined as the order of evaluation of f1() + f2() is not mandated by standard.
* The compiler is free to first call either f1() or f2().
* Only when equal level precedence operators appear in an expression, the associativity comes into picture. For example, f1() + f2() + f3() will be considered as (f1() + f2()) + f3(). But among first pair, which function (the operand) evaluated first is not defined by the standard.

# Difference between ++\*p, \*p++ and \*++p

Precedence of prefix ++ and \* is same. Associativity of both is right to left.

Precedence of postfix ++ is higher than both \* and prefix ++. Associativity of postfix ++ is left to right.

++\*p, two operators of same precedence (associativity of operators is right to left)

++\*p is treated as ++(\*p)

\*p++, precedence of postfix ++ is higher than \*

\*p++ is treated as \*(p++)

\*++p, two operators of same precedence (associativity of operators is right to left)

\*++p is treated as \*(++p)

#include <stdio.h>

int main(void) {

int arr[] = {10, 20};

int \*p = arr;

++\*p;

printf("arr[0] = %d, arr[1] = %d, \*p = %d\n", arr[0], arr[1], \*p);

arr[0] = 10, arr[1] = 20, p = arr;

\*p++;

printf("arr[0] = %d, arr[1] = %d, \*p = %d\n", arr[0], arr[1], \*p);

arr[0] = 10, arr[1] = 20, p = arr;

\*++p;

printf("arr[0] = %d, arr[1] = %d, \*p = %d\n", arr[0], arr[1], \*p);

return 0;

}

Output:

arr[0] = 11, arr[1] = 20, \*p = 11

arr[0] = 10, arr[1] = 20, \*p = 20

arr[0] = 10, arr[1] = 20, \*p = 20

# new and delete operators in C++

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap.

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory.

C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

## new operator

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

**Syntax:**

pointer-variable = new data-type;

**Initialize memory:** We can also initialize the memory using new operator:

pointer-variable = new data-type(value);

**Allocate block of memory:** new operator is also used to allocate a block(an array) of memory of type data-type.

pointer-variable = new data-type[size];

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type **std::bad\_alloc**, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer.

## delete operator

Since it is programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

**Syntax**

delete pointer-variable;

# Scope resolution operator vs this pointer

Scope resolution operator is for accessing static or class members

This pointer is for accessing object members when there is a local variable with same name

#include <iostream>

using namespace std;

class test {

int a;

static int sa;

public:

void fun1(int a) { cout << (this->a); }

void fun2(int a) { cout << (test::a); }

};

int test::sa = 5;

int main() {

test obj;

int k = 3;

obj.fun1(k);

return 0;

}

# Stringizing(#) and Token-pasting(##) operator

## Stringizing operator (#)

This operator causes the corresponding actual argument to be enclosed in double quotation marks.

The # operator, which is generally called the stringize operator, turns the argument it precedes into a quoted string.

#include <stdio.h>

#define mkstr(s) #s

int main(void) {

printf(mkstr(geeksforgeeks));

return 0;

}

Output: geeksforgeeks

// preprocessor turns the line printf(mkstr(geeksforgeeks)); into printf(“geeksforgeeks”);

## Token-pasting operator (##)

Allows tokens used as actual arguments to be concatenated to form other tokens. It is often useful to merge two tokens into one while expanding macros. This is called token pasting or token concatenation.

The ‘##’ pre-processing operator performs token pasting. When a macro is expanded, the two tokens on either side of each ‘##’ operator are combined into a single token, which then replaces the ‘##’ and the two original tokens in the macro expansion.

#include <stdio.h>

#define concat(a, b) a##b

int main(void) {

int xy = 30;

printf("%d", concat(x, y));

return 0;

}

Output: 30

# Sequence Point

|  |  |  |
| --- | --- | --- |
| #include <stdio.h>  int f1() {  printf ("Geeks");  return 1;  }  int f2() {  printf ("forGeeks");  return 1;  }    int main() {  int p = f1() + f2();  return 0;  } | #include <stdio.h>  int x = 20;  int f1() { x = x+10;  return x;  }  int f2() { x = x-5;  return x;  }    int main() {  int p = f1() + f2();  printf ("p = %d", p);  return 0;  } | #include <stdio.h>  int main() {  int i = 8;  int p = i++\*i++;  printf("%d\n", p);  return 0;  } |

The output of all of the above programs is undefined or unspecified.

The output may be different with different compilers and different machines.

In PROGRAM 1, the operator ‘+’ doesn’t have standard defined order of evaluation for its operands. Either f1() or f2() may be executed first.

Evaluation of an expression may also produce side effects.

In PROGRAM 2, the final values of p is ambiguous. Depending on the order of expression evaluation, if f1() executes first, the value of p will be 55, otherwise 40.

In PROGRAM 3, The subexpression i++ causes a side effect, it modifies i’s value, which leads to undefined behavior since i is also referenced elsewhere in the same expression.

## Sequence Point

<https://en.wikipedia.org/wiki/Sequence_point>

A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed.

They are often mentioned in reference to C and C++, because they are a core concept for determining the validity and, if valid, the possible results of expressions.

Adding more sequence points is sometimes necessary to make an expression defined and to ensure a single valid order of evaluation.

## Sequence points in C and C++

* Between evaluation of the left and right operands of the && (logical AND), || (logical OR) (as part of short-circuit evaluation), and comma operators.
* Between the evaluation of the first operand of the ternary "question-mark" operator and the second or third operand.
* At the end of a full expression.
* Before a function is entered in a function call.
* At a function return, after the return value is copied into the calling context.
* At the end of an initializer.
* Between each declarator in each declarator sequence.
* After each conversion associated with an input/output format specifier.

# References

<https://en.cppreference.com/w/cpp/language/operators>

<https://en.cppreference.com/w/cpp/language/operator_precedence>

<https://en.wikipedia.org/wiki/Sequence_point>

<https://www.learncpp.com/cpp-tutorial/bit-manipulation-with-bitwise-operators-and-bit-masks/>

# END