Arithmetic operators

Returns the result of specific arithmetic operation.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
unary plus	`+a`	Yes	`T T::operator+() const;`	`T operator+(const T &a);`
unary minus	`-a`	Yes	`T T::operator-() const;`	`T operator-(const T &a);`
addition	`a + b`	Yes	`T T::operator+(const T2 &b) const;`	`T operator+(const T &a, const T2 &b);`
subtraction	`a - b`	Yes	`T T::operator-(const T2 &b) const;`	`T operator-(const T &a, const T2 &b);`
multiplication	`a * b`	Yes	`T T::operator*(const T2 &b) const;`	`T operator*(const T &a, const T2 &b);`
division	`a / b`	Yes	`T T::operator/(const T2 &b) const;`	`T operator/(const T &a, const T2 &b);`
modulo	`a % b`	Yes	`T T::operator%(const T2 &b) const;`	`T operator%(const T &a, const T2 &b);`
bitwise NOT	`~a`	Yes	`T T::operator~() const;`	`T operator~(const T &a);`
bitwise AND	`a & b`	Yes	`T T::operator&(const T2 &b) const;`	`T operator&(const T &a, const T2 &b);`
bitwise OR	`a | b`	Yes	`T T::operator|(const T2 &b) const;`	`T operator|(const T &a, const T2 &b);`
bitwise XOR	`a ^ b`	Yes	`T T::operator^(const T2 &b) const;`	`T operator^(const T &a, const T2 &b);`
bitwise left shift	`a << b`	Yes	`T T::operator<<(const T2 &b) const;`	`T operator<<(const T &a, const T2 &b);`
bitwise right shift	`a >> b`	Yes	`T T::operator>>(const T2 &b) const;`	`T operator>>(const T &a, const T2 &b);`

Notes

- All built-in operators return values, and most user-defined overloads also return values so that the user-defined operators can be used in the same manner as the built-ins. However, in a user-defined operator overload, any type can be used as return type (including `void`). In particular, stream insertion and stream extraction overloads of operator<< and operator>> return T&.
- T2 can be any type including T

Explanation

All arithmetic operators compute the result of specific arithmetic operation and returns its result. The arguments are not modified.

Conversions

If the operand passed to an arithmetic operator is integral or unscoped enumeration type, then before any other action (but after lvalue-to-rvalue conversion, if applicable), the operand undergoes integral promotion. If an operand has array or function type, array-to-pointer and function-to-pointer conversions are applied.

For the binary operators (except shifts), if the promoted operands have different types, additional set of implicit conversions is applied, known as *usual arithmetic conversions* with the goal to produce the *common type* (also accessible via the std::common_type type trait). If, prior to any integral promotion, one operand is of enumeration type and the other operand is of a floating-point type or a different enumeration type, this behavior is deprecated. (since C++20)

- If either operand has scoped enumeration type, no conversion is performed: the other operand and the return type must have the same type
- Otherwise, if either operand is `long double`, the other operand is converted to `long double`
- Otherwise, if either operand is `double`, the other operand is converted to `double`
- Otherwise, if either operand is `float`, the other operand is converted to `float`
- Otherwise, the operand has integer type (because `bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, and unscoped enumeration were promoted at this point) and integral conversions are applied to produce the common type, as follows:
  - If both operands are signed or both are unsigned, the operand with lesser *conversion rank* is converted to the operand with the greater integer conversion rank
  - Otherwise, if the unsigned operand's conversion rank is greater or equal to the conversion rank of the signed operand, the signed operand is converted to the unsigned operand's type.
  - Otherwise, if the signed operand's type can represent all values of the unsigned operand, the unsigned operand is converted to the signed operand's type
  - Otherwise, both operands are converted to the unsigned counterpart of the signed operand's type.

The *conversion rank* above increases in order `bool`, `signed char`, `short`, `int`, `long`, `long long`. The rank of any unsigned type is equal to the rank of the corresponding signed type. The rank of `char` is equal to the rank of `signed char` and `unsigned char`. The ranks of `char8_t`, `char16_t`, `char32_t`, and `wchar_t` are equal to the ranks of their underlying types.

Overflows

Unsigned integer arithmetic is always performed *modulo* 2^n where n is the number of bits in that particular integer. E.g. for `unsigned int`, adding one to UINT_MAX gives `0`, and subtracting one from `0` gives UINT_MAX.

When signed integer arithmetic operation overflows (the result does not fit in the result type), the behavior is undefined, — the possible manifestations of such an operation include:

- it wraps around according to the rules of the representation (typically 2's complement),
- it traps – on some platforms or due to compiler options (e.g. -ftrapv in GCC and Clang),
- it saturates to minimal or maximal value (on many DSPs),
- it is completely optimized out by the compiler (http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html) .

Floating-point environment

If #pragma STDC FENV_ACCESS is supported and set to ON, all floating-point arithmetic operators obey the current floating-point rounding direction and report floating-point arithmetic errors as specified in math_errhandling unless part of a static initializer (in which case floating-point exceptions are not raised and the rounding mode is to nearest)

Floating-point contraction

Unless #pragma STDC FP_CONTRACT is supported and set to OFF, all floating-point arithmetic may be performed as if the intermediate results have infinite range and precision, that is, optimizations that omit rounding errors and floating-point exceptions are allowed. For example, C++ allows the implementation of `(x*y) + z` with a single fused multiply-add CPU instruction or optimization of `a = x*x*x*x;` as `tmp = x *x; a = tmp*tmp`.

Unrelated to contracting, intermediate results of floating-point arithmetic may have range and precision that is different from the one indicated by its type, see FLT_EVAL_METHOD

Formally, the C++ standard makes no guarantee on the accuracy of floating-point operations.

Unary arithmetic operators

The unary arithmetic operator expressions have the form

+ *expression*	(1)
- *expression*	(2)

1) unary plus (promotion).

For the built-in operator, *expression* must have arithmetic, unscoped enumeration, or pointer type. Integral promotion is performed on the operand if it has integral or unscoped enumeration type and determines the type of the result.

2) unary minus (negation).

For the built-in operator, *expression* must have arithmetic or unscoped enumeration type. Integral promotion is performed on the operand and determines the type of the result.

The built-in unary plus operator returns the value of its operand. The only situation where it is not a no-op is when the operand has integral type or unscoped enumeration type, which is changed by integral promotion, e.g, it converts `char` to `int` or if the operand is subject to lvalue-to-rvalue, array-to-pointer, or function-to-pointer conversion.

The builtin unary minus operator calculates the negative of its promoted operand. For unsigned a, the value of -a is 2^b-a, where b is the number of bits after promotion.

In overload resolution against user-defined operators, for every cv-unqualified promoted arithmetic type A and for every type T, the following function signatures participate in overload resolution:

```
A operator+(A)
T* operator+(T*)
A operator-(A)
```

Run this code

```cpp
#include <iostream>
int main()
{
    char c = 0x6a;
    int n1 = 1;
    unsigned char n2 = 1;
    unsigned int n3 = 1;
    std::cout << "char: " << c << " int: " << +c << '\n'
              << "-1, where 1 is signed: " << -n1 << '\n'
              << "-1, where 1 is unsigned char: " << -n2 << '\n'
              << "-1, where 1 is unsigned int: " << -n3 << '\n';
    char a[3];
    std::cout << "size of array: " << sizeof a << '\n'
              << "size of pointer: " << sizeof +a << '\n';
}
```

Possible output:

```
char: j int: 106
-1, where 1 is signed: -1
-1, where 1 is unsigned char: -1
-1, where 1 is unsigned int: 4294967295
size of array: 3
size of pointer: 8
```

Additive operators

The binary additive arithmetic operator expressions have the form

lhs **+** *rhs*	(1)
lhs **-** *rhs*	(2)

1) addition

For the built-in operator, *lhs* and *rhs* must be one of the following:

- both have arithmetic or unscoped enumeration type. In this case, the usual arithmetic conversions are performed on both operands and determine the type of the result.
- one is a pointer to completely-defined object type, the other has integral or unscoped enumeration type. In this case, the result type has the type of the pointer.

2) subtraction

For the built-in operator, *lhs* and *rhs* must be one of the following:

- both have arithmetic or unscoped enumeration type. In this case, the usual arithmetic conversions are performed on both operands and determine the type of the result.
- *lhs* is a pointer to completely-defined object type, *rhs* has integral or unscoped enumeration type. In this case, the result type has the type of the pointer.
- both are pointers to the same completely-defined object types, ignoring cv-qualifiers. In this case, the result type is `std::ptrdiff_t`.

With operands of arithmetic or enumeration type, the result of binary plus is the sum of the operands (after usual arithmetic conversions), and the result of the binary minus operator is the result of subtracting the second operand from the first (after usual arithmetic conversions), except that, if the type supports IEEE floating-point arithmetic (see `std::numeric_limits::is_iec559`),

- if one operand is NaN, the result is NaN
- infinity minus infinity is NaN and FE_INVALID is raised
- infinity plus the negative infinity is NaN and FE_INVALID is raised

If any of the operands is a pointer, the following rules apply:

- A pointer to non-array object is treated as a pointer to the first element of an array with size 1.
- If the pointer P points to the ith element of an array, then the expressions P+n, n+P, and P-n are pointers of the same type that point to the i+nth, i+nth, and i-nth element of the same array, respectively. The result of pointer addition may also be a one-past-the-end pointer (that is, pointer P such that the expression P-1 points to the last element of the array). Any other situations (that is, attempts to generate a pointer that isn't pointing at an element of the same array or one past the end) invoke undefined behavior.
- If the pointer P points to the ith element of an array, and the pointer Q points at the jth element of the same array, the expression P-Q has the value `i-j`, if the value fits in `std::ptrdiff_t`. Both operands must point to the elements of the same array (or one past the end), otherwise the behavior is undefined. If the result does not fit in `std::ptrdiff_t`, the behavior is undefined.
- In any case, if the pointed-to type is different from the array element type, disregarding cv qualifications, at every level if the elements are themselves pointers, the behavior of pointer arithmetic is undefined. In particular, pointer arithmetic with pointer to base, which is pointing at an element of an array of derived objects is undefined.
- If the value `0` is added or subtracted from a pointer, the result is the pointer, unchanged. If two pointers point at the same object or are both one past the end of the same array, or both are null pointers, then the result of subtraction is equal to `(std::ptrdiff_t)0`.

These pointer arithmetic operators allow pointers to satisfy the *LegacyRandomAccessIterator* requirements.

In overload resolution against user-defined operators, for every pair of promoted arithmetic types L and R and for every object type T, the following function signatures participate in overload resolution:

`LR operator+(L, R)`
`LR operator-(L, R)`
`T* operator+(T*, std::ptrdiff_t)`
`T* operator+(std::ptrdiff_t, T*)`
`T* operator-(T*, std::ptrdiff_t)`
`std::ptrdiff_t operator-(T*, T*)`

where LR is the result of usual arithmetic conversions on L and R

Run this code

```
#include <iostream>
int main()
{
    char c = 2;
    unsigned int un = 2;
    int  n = -10;
    std::cout <<  " 2 + (-10), where 2 is a char    = " << c + n << '\n'
              <<  " 2 + (-10), where 2 is unsigned  = " << un + n << '\n'
              <<  " -10 - 2.12  = " << n - 2.12 << '\n';
    char a[4] = {'a', 'b', 'c', 'd'};
    char* p = &a[1];
    std::cout << "Pointer addition examples: " << *p << *(p + 2)
              << *(2 + p) << *(p - 1) << '\n';
    char* p2 = &a[4];
    std::cout << "Pointer difference: " << p2 - p << '\n';
}
```

Output:

```
 2 + (-10), where 2 is a char    = -8
 2 + (-10), where 2 is unsigned  = 4294967288
 -10 - 2.12  = -12.12
Pointer addition examples: bdda
Pointer difference: 3
```

Multiplicative operators

The binary multiplicative arithmetic operator expressions have the form

lhs `*` *rhs*	(1)
lhs `/` *rhs*	(2)
lhs `%` *rhs*	(3)

1) multiplication
   For the built-in operator, *lhs* and *rhs* must both have arithmetic or unscoped enumeration type.
2) division
   For the built-in operator, *lhs* and *rhs* must both have arithmetic or unscoped enumeration type.
3) remainder
   For the built-in operator, *lhs* and *rhs* must both have integral or unscoped enumeration type

For all three operators, the usual arithmetic conversions are performed on both operands and determine the type of the result.

The binary operator * performs multiplication of its operands (after usual arithmetic conversions), except that, for floating-point multiplication,

- multiplication of a NaN by any number gives NaN
- multiplication of infinity by zero gives NaN and FE_INVALID is raised

The binary operator / divides the first operand by the second (after usual arithmetic conversions).

For integral operands, it yields the algebraic quotient.

The quotient is rounded in implementation-defined direction.	(until C++11)
The quotient is truncated towards zero (fractional part is discarded).	(since C++11)

If the second operand is zero, the behavior is undefined, except that if floating-point division is taking place and the type supports IEEE floating-point arithmetic (see std::numeric_limits::is_iec559), then:

- if one operand is NaN, the result is NaN
- dividing a non-zero number by ±0.0 gives the correctly-signed infinity and FE_DIVBYZERO is raised
- dividing 0.0 by 0.0 gives NaN and FE_INVALID is raised

The binary operator % yields the remainder of the integer division of the first operand by the second (after usual arithmetic conversions; note that the operand types must be integral types). If the quotient a/b is representable in the result type, `(a/b)*b + a%b == a`. If the second operand is zero, the behavior is undefined. If the quotient a/b is not representable in the result type, the behavior of both a/b and a%b is undefined (that means `INT_MIN%-1` is undefined on 2's complement systems)

Note: Until C++11, if one or both operands to binary operator % were negative, the sign of the remainder was implementation-defined, as it depends on the rounding direction of integer division. The function std::div provided well-defined behavior in that case.

Note: for floating-point remainder, see std::remainder and std::fmod.

In overload resolution against user-defined operators, for every pair of promoted arithmetic types LA and RA and for every pair of promoted integral types LI and RI the following function signatures participate in overload resolution:

```
LRA operator*(LA, RA)
LRA operator/(LA, RA)
LRI operator%(LI, RI)
```

where LRx is the result of usual arithmetic conversions on Lx and Rx

Run this code

```
#include <iostream>
int main()
```

```
{
    char c = 2;
    unsigned int un = 2;
    int  n = -10;
    std::cout <<  "2 * (-10), where 2 is a char    = " << c * n << '\n'
              <<  "2 * (-10), where 2 is unsigned  = " << un * n << '\n'
              <<  "-10 / 2.12  = " << n / 2.12 << '\n'
              <<  "-10 / 21  = " << n / 21 << '\n'
              <<  "-10 % 21  = " << n % 21 << '\n';
}
```

Output:

```
2 * (-10), where 2 is a char    = -20
2 * (-10), where 2 is unsigned  = 4294967276
-10 / 2.12  = -4.71698
-10 / 21  = 0
-10 % 21  = -10
```

Bitwise logic operators

The bitwise arithmetic operator expressions have the form

~ *rhs*	(1)
lhs **&** *rhs*	(2)
lhs **|** *rhs*	(3)
lhs **^** *rhs*	(4)

1) bitwise NOT
2) bitwise AND
3) bitwise OR
4) bitwise XOR

For the built-in operators, *lhs* and *rhs* must both have integral or unscoped enumeration type. Usual arithmetic conversions are performed on both operands and determine the type of the result.

The result of operator~ is the bitwise NOT (one's complement) value of the argument (after promotion). The result of operator& is the bitwise AND value of the operands (after usual arithmetic conversions). The result of operator| is the bitwise OR value of the operands (after usual arithmetic conversions). The result of operator^ is the bitwise XOR value of the operands (after usual arithmetic conversions)

In overload resolution against user-defined operators, for every pair of promoted integral types L and R the following function signatures participate in overload resolution:

R operator~(R)
LR operator&(L, R)
LR operator^(L, R)
LR operator|(L, R)

where LR is the result of usual arithmetic conversions on L and R

Run this code

```
#include <iostream>
#include <iomanip>
#include <bitset>
int main()
{
    uint16_t mask = 0x00f0;
    uint32_t x0 = 0x12345678;
    uint32_t x1 = x0 | mask;
    uint32_t x2 = x0 & ~mask;
    uint32_t x3 = x0 & mask;
    uint32_t x4 = x0 ^ mask;
    uint32_t x5 = ~x0;
    using bin16 = std::bitset<16>;
    using bin32 = std::bitset<32>;
    std::cout << std::hex << std::showbase
              << "Mask: " << mask << std::setw(49) << bin16(mask) << '\n'
              << "Value: " << x0 << std::setw(42) << bin32(x0) << '\n'
              << "Setting bits: " << x1 << std::setw(35) << bin32(x1) << '\n'
              << "Clearing bits: " << x2 << std::setw(34) << bin32(x2) << '\n'
              << "Selecting bits: " << x3 << std::setw(39) << bin32(x3) << '\n'
              << "XOR-ing bits: " << x4 << std::setw(35) << bin32(x4) << '\n'
              << "Inverting bits: " << x5 << std::setw(33) << bin32(x5) << '\n';
}
```

Output:

```
Mask: 0xf0                                0000000011110000
Value: 0x12345678          00010010001101000101011001111000
Setting bits: 0x123456f8   00010010001101000101011011111000
Clearing bits: 0x12345608  00010010001101000101011000001000
Selecting bits: 0x70       00000000000000000000000001110000
XOR-ing bits: 0x12345688   00010010001101000101011010001000
Inverting bits: 0xedcba987 11101101110010111010100110000111
```

Bitwise shift operators

The bitwise shift operator expressions have the form

lhs **<<** *rhs*	(1)
lhs **>>** *rhs*	(2)

1) left shift of *lhs* by *rhs* bits

2) right shift of *lhs* by *rhs* bits

For the built-in operators, *lhs* and *rhs* must both have integral or unscoped enumeration type. Integral promotions are performed on both operands.

The return type is the type of the left operand after integral promotions.

For unsigned a, the value of a << b is the value of $a * 2^b$, reduced modulo 2^N where N is the number of bits in the return type (that is, bitwise left shift is performed and the bits that get shifted out of the destination type are discarded).

For signed and non-negative a, if $a * 2^b$ is representable in the unsigned version of the return type, then that value, converted to signed, is the value of a << b (this makes it legal to create INT_MIN as 1<<31); otherwise the behavior is undefined.

For negative a, the behavior of a << b is undefined.

For unsigned a and for signed and non-negative a, the value of a >> b is the integer part of $a/2^b$.

For negative a, the value of a >> b is implementation-defined (in most implementations, this performs arithmetic right shift, so that the result remains negative).

(until C++20)

The value of a << b is the unique value congruent to $a * 2^b$ modulo 2^N where N is the number of bits in the return type (that is, bitwise left shift is performed and the bits that get shifted out of the destination type are discarded).

The value of a >> b is $a/2^b$, rounded down (in other words, right shift on signed a is arithmetic right shift).

(since C++20)

In any case, if the value of the right operand is negative or is greater or equal to the number of bits in the promoted left operand, the behavior is undefined.

In overload resolution against user-defined operators, for every pair of promoted integral types L and R, the following function signatures participate in overload resolution:

```
L operator<<(L, R)
L operator>>(L, R)
```

Run this code

```
#include <iostream>
enum {ONE=1, TWO=2};
int main()
{
    std::cout << std::hex << std::showbase;
    char c = 0x10;
    unsigned long long ull = 0x123;
    std::cout << "0x123 << 1 = " << (ull << 1) << '\n'
              << "0x123 << 63 = " << (ull << 63) << '\n' // overflow in unsigned
              << "0x10 << 10 = " << (c << 10) << '\n';   // char is promoted to int
    long long ll = -1000;
    std::cout << std::dec << "-1000 >> 1 = " << (ll >> ONE) << '\n';
}
```

Output:

```
0x123 << 1 = 0x246
0x123 << 63 = 0x8000000000000000
0x10 << 10 = 0x4000
-1000 >> 1 = -500
```

Standard library

Arithmetic operators are overloaded for many standard library types.

Unary arithmetic operators

operator+ **operator-**	implements unary + and unary - (public member function of std::chrono::duration<Rep,Period>)
operator+ **operator-**	applies unary operators to complex numbers (function template)
operator+ **operator-** **operator~** **operator!**	applies a unary arithmetic operator to each element of the valarray (public member function of std::valarray<T>)

Additive operators

operator+ **operator-** (C++11)	performs add and subtract operations involving a time point (function template)
operator+ **operator-** **operator*** (C++11) **operator/** **operator%**	implements arithmetic operations with durations as arguments (function template)
operator+ **operator-** (C++20)	adds or subtracts a year_month_day and some number of years or months (function)
operator+	concatenates two strings or a string and a char (function template)
operator+ **operator-**	advances or decrements the iterator (public member function of std::reverse_iterator<Iter>)
operator+ **operator-**	advances or decrements the iterator (public member function of std::move_iterator<Iter>)
operator+ **operator-** **operator*** **operator/**	performs complex number arithmetics on two complex values or a complex and a scalar (function template)
operator+ **operator-** **operator*** **operator/** **operator%** **operator&** **operator|** **operator^** **operator<<** **operator>>** **operator&&** **operator||**	applies binary operators to each element of two valarrays, or a valarray and a value (function template)

Multiplicative operators

operator+ **operator-** **operator*** (C++11) **operator/** **operator%**	implements arithmetic operations with durations as arguments (function template)
operator+ **operator-** **operator*** **operator/**	performs complex number arithmetics on two complex values or a complex and a scalar (function template)
operator+ **operator-** **operator*** **operator/** **operator%** **operator&** **operator|** **operator^** **operator<<** **operator>>** **operator&&** **operator||**	applies binary operators to each element of two valarrays, or a valarray and a value (function template)

Bitwise logic operators

operator&= **operator|=** **operator^=** **operator~**	performs binary AND, OR, XOR and NOT (public member function of std::bitset<N>)
	performs binary logic operations on bitsets (function template)

operator& **operator|** **operator^**	
operator~	applies a unary arithmetic operator to each element of the valarray (public member function of std::valarray<T>)
operator^ **operator&** **operator|**	applies binary operators to each element of two valarrays, or a valarray and a value (function template)

Bitwise shift operators

operator<< **operator>>**	applies binary operators to each element of two valarrays, or a valarray and a value (function template)
operator<< **operator>>**	performs binary shift left and shift right (public member function of std::bitset<N>)

Stream insertion/extraction operators

Throughout the standard library, bitwise shift operators are commonly overloaded with I/O stream (`std::ios_base&` or one of the classes derived from it) as both the left operand and return type. Such operators are known as *stream insertion* and *stream extraction* operators:

operator>>	extracts formatted data (public member function of std::basic_istream<CharT,Traits>)
operator>>(std::basic_istream)	extracts characters and character arrays (function template)
operator<<	inserts formatted data (public member function of std::basic_ostream<CharT,Traits>)
operator<<(std::basic_ostream)	inserts character data or insert into rvalue stream (function template)
operator<< **operator>>**	serializes and deserializes a complex number (function template)
operator<< **operator>>**	performs stream input and output of bitsets (function template)
operator<< **operator>>**	performs stream input and output on strings (function template)
operator<< **operator>>** (C++11)	performs stream input and output on pseudo-random number engine (function template)
operator<< **operator>>** (C++11)	performs stream input and output on pseudo-random number distribution (function template)

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 1457 (https://cplusplus.github.io/CWG/issues/1457.html)	C++98	shifting the leftmost 1 bit of a positive signed value into the sign bit was UB	made well-defined

See also

Operator precedence

Operator overloading

Common operators						
assignment	increment decrement	**arithmetic**	logical	comparison	member access	other

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
				`a <=> b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`a ? b : c`

Special operators
`static_cast` converts one type to another related type `dynamic_cast` converts within inheritance hierarchies `const_cast` adds or removes cv qualifiers `reinterpret_cast` converts type to unrelated type C-style cast converts one type to another by a mix of `static_cast`, `const_cast`, and `reinterpret_cast` `new` creates objects with dynamic storage duration `delete` destructs objects previously created by the new expression and releases obtained memory area `sizeof` queries the size of a type `sizeof...` queries the size of a parameter pack (since C++11) `typeid` queries the type information of a type `noexcept` checks if an expression can throw an exception (since C++11) `alignof` queries alignment requirements of a type (since C++11)

C documentation for **Arithmetic operators**

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/language/operator_arithmetic&oldid=136183"