Assignment operators

Assignment operators modify the value of the object.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
simple assignment	`a = b`	Yes	`T& T::operator =(const T2& b);`	N/A
addition assignment	`a += b`	Yes	`T& T::operator +=(const T2& b);`	`T& operator +=(T& a, const T2& b);`
subtraction assignment	`a -= b`	Yes	`T& T::operator -=(const T2& b);`	`T& operator -=(T& a, const T2& b);`
multiplication assignment	`a *= b`	Yes	`T& T::operator *=(const T2& b);`	`T& operator *=(T& a, const T2& b);`
division assignment	`a /= b`	Yes	`T& T::operator /=(const T2& b);`	`T& operator /=(T& a, const T2& b);`
modulo assignment	`a %= b`	Yes	`T& T::operator %=(const T2& b);`	`T& operator %=(T& a, const T2& b);`
bitwise AND assignment	`a &= b`	Yes	`T& T::operator &=(const T2& b);`	`T& operator &=(T& a, const T2& b);`
bitwise OR assignment	`a |= b`	Yes	`T& T::operator |=(const T2& b);`	`T& operator |=(T& a, const T2& b);`
bitwise XOR assignment	`a ^= b`	Yes	`T& T::operator ^=(const T2& b);`	`T& operator ^=(T& a, const T2& b);`
bitwise left shift assignment	`a <<= b`	Yes	`T& T::operator <<=(const T2& b);`	`T& operator <<=(T& a, const T2& b);`
bitwise right shift assignment	`a >>= b`	Yes	`T& T::operator >>=(const T2& b);`	`T& operator >>=(T& a, const T2& b);`

Notes

- All built-in assignment operators return `*this`, and most user-defined overloads also return `*this` 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`).
- T2 can be any type including T

Explanation

copy assignment operator replaces the contents of the object a with a copy of the contents of b (b is not modified). For class types, this is a special member function, described in copy assignment operator.

move assignment operator replaces the contents of the object a with the contents of b, avoiding copying if possible (b may be modified). For class types, this is a special member function, described in move assignment operator. (since C++11)

For non-class types, copy and move assignment are indistinguishable and are referred to as *direct assignment*.

compound assignment operators replace the contents of the object a with the result of a binary operation between the previous value of a and the value of b.

Builtin direct assignment

The direct assignment expressions have the form

lhs **=** *rhs*	(1)	
lhs **= {}**	(2)	(since C++11)
lhs **= {** *rhs* **}**	(3)	(since C++11)

For the built-in operator, *lhs* may have any non-const scalar type and *rhs* must be implicitly convertible to the type of *lhs*.

The direct assignment operator expects a modifiable lvalue as its left operand and an rvalue expression or a *braced-init-list* (since C++11) as its right operand, and returns an lvalue identifying the left operand after modification.

For non-class types, the right operand is first implicitly converted to the cv-unqualified type of the left operand, and then its value is copied into the object identified by left operand.

When the left operand has reference type, the assignment operator modifies the referred-to object.

If the left and the right operands identify overlapping objects, the behavior is undefined (unless the overlap is exact and the type is the same)

If the right operand is a *braced-init-list* (since C++11)

- if the expression E1 has scalar type,
  - the expression `E1 = {}` is equivalent to `E1 = T{}`, where T is the type of E1.
  - the expression `E1 = {E2}` is equivalent to `E1 = T{E2}`, where T is the type of E1.
- if the expression E1 has class type, the syntax `E1 = {args...}` generates a call to the assignment operator with the *braced-init-list* as the argument, which then selects the appropriate assignment operator following the rules of overload resolution. Note that, if a non-template assignment operator from some non-class type is available, it is preferred over the copy/move assignment in `E1 = {}` because `{}` to non-class is an identity conversion, which outranks the user-defined conversion from `{}` to a class type.

Using an lvalue of volatile-qualified non-class type as left operand of built-in direct assignment operator is deprecated, unless the assignment expression appears in an unevaluated context or is a discarded-value expression. (since C++20)

In overload resolution against user-defined operators, for every type T, the following function signatures participate in overload resolution:

```
T*& operator=(T*&, T*);
T*volatile & operator=(T*volatile &, T*);
```

For every enumeration or pointer to member type T, optionally volatile-qualified, the following function signature participates in overload resolution:

```
T& operator=(T&, T );
```

For every pair A1 and A2, where A1 is an arithmetic type (optionally volatile-qualified) and A2 is a promoted arithmetic type, the following function signature participates in overload resolution:

```
A1& operator=(A1&, A2);
```

Example

Run this code

```
#include <iostream>
int main()
{
    int n = 0;  // not an assignment
    n = 1;      // direct assignment
    std::cout << n << ' ';
    n = {};     // zero-initialization, then assignment
    std::cout << n << ' ';
    n = 'a';    // integral promotion, then assignment
    std::cout << n << ' ';
    n = {'b'};   // explicit cast, then assignment
    std::cout << n << ' ';
    n = 1.0;    // floating-point conversion, then assignment
    std::cout << n << ' ';
//    n = {1.0}; // compiler error (narrowing conversion)

    int& r = n;  // not an assignment
    int* p;

    r = 2;       // assignment through reference
    std::cout << n << '\n';
    p = &n;      // direct assignment
    p = nullptr; // null-pointer conversion, then assignment

    struct {int a; std::string s;} obj;
    obj = {1, "abc"}; // assignment from a braced-init-list
    std::cout << obj.a << ':' << obj.s << '\n';
}
```

Output:

```
1 0 97 98 1 2
1:abc
```

Builtin compound assignment

The compound assignment expressions have the form

lhs op rhs	(1)	
lhs *op* {}	(2)	(since C++11)
lhs *op* {rhs}	(3)	(since C++11)

op - one of `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`

lhs - for the built-in operator, *lhs* may have any arithmetic type, except when *op* is `+=` or `-=`, which also accept pointer types with the same restrictions as + and -

rhs - for the built-in operator, *rhs* must be implicitly convertible to *lhs*

The behavior of every builtin compound-assignment expression `E1 op= E2` (where E1 is a modifiable lvalue expression and E2 is an rvalue expression or a *braced-init-list* (since C++11)) is exactly the same as the behavior of the expression `E1 = E1 op E2`, except that the expression E1 is evaluated only once and that it behaves as a single operation with respect to indeterminately-sequenced function calls (e.g. in `f(a += b, g())`, the += is either not started at all or is completed as seen from inside `g()`).

Using an lvalue of volatile-qualified non-class type as left operand of built-in compound assignment operator is deprecated. (since C++20)

In overload resolution against user-defined operators, for every pair A1 and A2, where A1 is an arithmetic type (optionally volatile-qualified) and A2 is a promoted arithmetic type, the following function signatures participate in overload resolution:

```
A1& operator*=(A1&, A2);
A1& operator/=(A1&, A2);
A1& operator+=(A1&, A2);
A1& operator-=(A1&, A2);
```

For every pair I1 and I2, where I1 is an integral type (optionally volatile-qualified) and I2 is a promoted integral type, the following function signatures participate in overload resolution:

```
I1& operator%=(I1&, I2);
I1& operator<<=(I1&, I2);
I1& operator>>=(I1&, I2);
I1& operator&=(I1&, I2);
I1& operator^=(I1&, I2);
I1& operator|=(I1&, I2);
```

For every optionally cv-qualified object type T, the following function signatures participate in overload resolution:

```
T*& operator+=(T*&, std::ptrdiff_t);
T*& operator-=(T*&, std::ptrdiff_t);
T*volatile & operator+=(T*volatile &, std::ptrdiff_t);
T*volatile & operator-=(T*volatile &, std::ptrdiff_t);
```

Example

This section is incomplete
Reason: no example

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`
						`? :`

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 **`Assignment operators`**