operator overloading

Customizes the C++ operators for operands of user-defined types.

Syntax

Overloaded operators are functions with special function names:

operator *op*	(1)	
operator *type*	(2)	
operator new **operator new []**	(3)	
operator delete **operator delete []**	(4)	
operator "" *suffix-identifier*	(5)	(since C++11)

op - any of the following 38 operators: `+` `-` `*` `/` `%` `^` `&` `|` `~` `!` `=` `<` `>` `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<` `>>` `>>=` `<<=` `==` `!=` `<=` `>=` `&&` `||` `++` `--` `,` `->*` `->` `( )` `[ ]`

1) overloaded operator;
2) user-defined conversion function;
3) allocation function;
4) deallocation function;
5) user-defined literal.

Overloaded operators

When an operator appears in an expression, and at least one of its operands has a class type or an enumeration type, then overload resolution is used to determine the user-defined function to be called among all the functions whose signatures match the following:

Expression	As member function	As non-member function	Example
@a	(a).operator@ ( )	operator@ (a)	`!std::cin` calls `std::cin.operator!()`
a@b	(a).operator@ (b)	operator@ (a, b)	`std::cout << 42` calls `std::cout.operator<<(42)`
a=b	(a).operator= (b)	cannot be non-member	`std::string s; s = "abc";` calls `s.operator=("abc")`
a(b...)	(a).operator()(b...)	cannot be non-member	`std::random_device r; auto n = r();` calls `r.operator()()`
a[b]	(a).operator[](b)	cannot be non-member	`std::map<int, int> m; m[1] = 2;` calls `m.operator[](1)`
a->	(a).operator-> ( )	cannot be non-member	`auto p = std::make_unique<S>(); p->bar()` calls `p.operator->()`
a@	(a).operator@ (0)	operator@ (a, 0)	`std::vector<int>::iterator i = v.begin(); i++` calls `i.operator++(0)`
in this table, @ is a placeholder representing all matching operators: all prefix operators in @a, all postfix operators other than -> in a@, all infix operators other than = in a@b			

Note: for overloading user-defined conversion functions, user-defined literals, allocation and deallocation see their respective articles.

Overloaded operators (but not the built-in operators) can be called using function notation:

```
std::string str = "Hello, ";
str.operator+=("world");                       // same as str += "world";
operator<<(operator<<(std::cout, str) , '\n'); // same as std::cout << str << '\n';
                                               // (since C++17) except for sequencing
```

Restrictions

- The operators **::** (scope resolution), **.** (member access), **.*** (member access through pointer to member), and **?:** (ternary conditional) cannot be overloaded.
- New operators such as ******, **<>**, or **&|** cannot be created.
- The overloads of operators **&&** and **||** lose short-circuit evaluation.
- The overload of operator **->** must either return a raw pointer or return an object (by reference or by value), for which operator **->** is in turn overloaded.
- It is not possible to change the precedence, grouping, or number of operands of operators.

- **&&**, **||**, and **,** (comma) lose their special sequencing properties when overloaded and behave like regular function calls even when they are used without function-call notation. (until C++17)

Canonical implementations

Other than the restrictions above, the language puts no other constraints on what the overloaded operators do, or on the return type (it does not participate in overload resolution), but in general, overloaded operators are expected to behave as similar as possible to the built-in operators: `operator+` is expected to add, rather than multiply its arguments, `operator=` is expected to assign, etc. The related operators are expected to behave similarly (`operator+` and `operator+=` do the same addition-like operation). The return types are limited by the expressions in which the operator is expected to be used: for example, assignment operators return by reference to make it possible to write `a = b = c = d`, because the built-in operators allow that.

Commonly overloaded operators have the following typical, canonical forms:[1]

Assignment operator

The assignment operator (`operator=`) has special properties: see copy assignment and move assignment for details.

The canonical copy-assignment operator is expected to perform no action on self-assignment (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#c62-make-copy-assignment-safe-for-self-assignment) , and to return the lhs by reference:

```
// assume the object holds reusable storage, such as a heap-allocated buffer mArray
T& operator=(const T& other) // copy assignment
{
    if (this != &other) { // self-assignment check expected
        if (other.size != size) {         // storage cannot be reused
            delete[] mArray;              // destroy storage in this
            size = 0;
            mArray = nullptr;             // preserve invariants in case next line throws
            mArray = new int[other.size]; // create storage in this
            size = other.size;
        }
        std::copy(other.mArray, other.mArray + other.size, mArray);
    }
    return *this;
}
```

The canonical move assignment is expected to leave the moved-from object in valid state (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#c64-a-move-operation-should-move-and-leave-its-source-in-a-valid-state) (that is, a state with class invariants intact), and either do nothing (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#c65-make-move-assignment-safe-for-self-assignment) or at least leave the object in a valid state on self-assignment, and return the lhs by reference to non-const, and be noexcept:

```
T& operator=(T&& other) noexcept // move assignment
{
    if(this != &other) { // no-op on self-move-assignment (delete[]/size=0 also ok)
        delete[] mArray;                               // delete this storage
        mArray = std::exchange(other.mArray, nullptr); // leave moved-from in valid state
        size = std::exchange(other.size, 0);
    }
    return *this;
}
```

In those situations where copy assignment cannot benefit from resource reuse (it does not manage a heap-allocated array and does not have a (possibly transitive) member that does, such as a member `std::vector` or `std::string`), there is a popular convenient shorthand: the copy-and-swap assignment operator, which takes its parameter by value (thus working as both copy- and move-assignment depending on the value category of the argument), swaps with the parameter, and lets the destructor clean it up.

```
T& T::operator=(T arg) noexcept // copy/move constructor is called to construct arg
{
    swap(arg); // resources are exchanged between *this and arg
    return *this;
} // destructor of arg is called to release the resources formerly held by *this
```

This form automatically provides strong exception guarantee, but prohibits resource reuse.

Stream extraction and insertion

The overloads of `operator>>` and `operator<<` that take a `std::istream&` or `std::ostream&` as the left hand argument are known as insertion and extraction operators. Since they take the user-defined type as the right argument (b in *a@b*), they must be implemented as non-members.

```
std::ostream& operator<<(std::ostream& os, const T& obj)
{
    // write obj to stream
    return os;
}
std::istream& operator>>(std::istream& is, T& obj)
{
    // read obj from stream
    if( /* T could not be constructed */ )
        is.setstate(std::ios::failbit);
    return is;
}
```

These operators are sometimes implemented as friend functions.

Function call operator

When a user-defined class overloads the function call operator, `operator()`, it becomes a `FunctionObject` type. Many standard algorithms, from `std::sort` to `std::accumulate` accept objects of such types to customize behavior. There are no particularly notable canonical forms of `operator()`, but to illustrate the usage

```
struct Sum
{
    int sum;
    Sum() : sum(0) { }
    void operator()(int n) { sum += n; }
};
Sum s = std::for_each(v.begin(), v.end(), Sum());
```

Increment and decrement

When the postfix increment and decrement appear in an expression, the corresponding user-defined function (`operator++` or `operator--`) is called with an integer argument `0`. Typically, it is implemented as `T operator++(int)`, where the argument is ignored. The postfix increment and decrement operator is usually implemented in terms of the prefix version:

```
struct X
{
    X& operator++()
    {
        // actual increment takes place here
        return *this;
    }
    X operator++(int)
    {
        X tmp(*this); // copy
        operator++(); // pre-increment
        return tmp;   // return old value
    }
};
```

Although canonical form of pre-increment/pre-decrement returns a reference, as with any operator overload, the return type is user-defined; for example the overloads of these operators for `std::atomic` return by value.

Binary arithmetic operators

Binary operators are typically implemented as non-members to maintain symmetry (for example, when adding a complex number and an integer, if `operator+` is a member function of the complex type, then only `complex+integer` would compile, and not `integer+complex`). Since for every binary arithmetic operator there exists a corresponding compound assignment operator, canonical forms of binary operators are implemented in terms of their compound assignments:

```
class X
{
 public:
  X& operator+=(const X& rhs) // compound assignment (does not need to be a member,
  {                           // but often is, to modify the private members)
    /* addition of rhs to *this takes place here */
    return *this; // return the result by reference
  }

  // friends defined inside class body are inline and are hidden from non-ADL lookup
  friend X operator+(X lhs,        // passing lhs by value helps optimize chained a+b+c
                     const X& rhs) // otherwise, both parameters may be const references
  {
    lhs += rhs; // reuse compound assignment
    return lhs; // return the result by value (uses move constructor)
  }
};
```

Relational operators

Standard algorithms such as `std::sort` and containers such as `std::set` expect `operator<` to be defined, by default, for the user-provided types, and expect it to implement strict weak ordering (thus satisfying the `Compare` concept). An idiomatic way to implement strict weak ordering for a structure is to use lexicographical comparison provided by `std::tie`:

```
struct Record
{
    std::string name;
    unsigned int floor;
    double weight;
    friend bool operator<(const Record& l, const Record& r)
    {
        return std::tie(l.name, l.floor, l.weight)
             < std::tie(r.name, r.floor, r.weight); // keep the same order
    }
};
```

Typically, once `operator<` is provided, the other relational operators are implemented in terms of `operator<`.

```
inline bool operator< (const X& lhs, const X& rhs){ /* do actual comparison */ }
inline bool operator> (const X& lhs, const X& rhs){ return rhs < lhs; }
inline bool operator<=(const X& lhs, const X& rhs){ return !(lhs > rhs); }
inline bool operator>=(const X& lhs, const X& rhs){ return !(lhs < rhs); }
```

Likewise, the inequality operator is typically implemented in terms of `operator==`:

```
inline bool operator==(const X& lhs, const X& rhs){ /* do actual comparison */ }
inline bool operator!=(const X& lhs, const X& rhs){ return !(lhs == rhs); }
```

When three-way comparison (such as `std::memcmp` or `std::string::compare`) is provided, all six relational operators may be expressed through that:

```
inline bool operator==(const X& lhs, const X& rhs){ return cmp(lhs,rhs) == 0; }
inline bool operator!=(const X& lhs, const X& rhs){ return cmp(lhs,rhs) != 0; }
inline bool operator< (const X& lhs, const X& rhs){ return cmp(lhs,rhs) <  0; }
inline bool operator> (const X& lhs, const X& rhs){ return cmp(lhs,rhs) >  0; }
inline bool operator<=(const X& lhs, const X& rhs){ return cmp(lhs,rhs) <= 0; }
inline bool operator>=(const X& lhs, const X& rhs){ return cmp(lhs,rhs) >= 0; }
```

Array subscript operator

User-defined classes that provide array-like access that allows both reading and writing typically define two overloads for `operator[]`: const and non-const variants:

```
struct T
{
          value_t& operator[](std::size_t idx)       { return mVector[idx]; }
    const value_t& operator[](std::size_t idx) const { return mVector[idx]; }
};
```

If the value type is known to be a built-in type, the const variant should return by value.

Where direct access to the elements of the container is not wanted or not possible or distinguishing between lvalue `c[i] = v;` and rvalue `v = c[i];` usage, operator[] may return a proxy. see for example `std::bitset::operator[]`.

To provide multidimensional array access semantics, e.g. to implement a 3D array access `a[i][j][k] = x;`, operator[] has to return a reference to a 2D plane, which has to have its own operator[] which returns a reference to a 1D row, which has to have operator[] which returns a reference to the element. To avoid this complexity, some libraries opt for overloading `operator()` instead, so that 3D access expressions have the Fortran-like syntax `a(i, j, k) = x;`

Bitwise arithmetic operators

User-defined classes and enumerations that implement the requirements of `BitmaskType` are required to overload the bitwise arithmetic operators `operator&`, `operator|`, `operator^`, `operator~`, `operator&=`, `operator|=`, and `operator^=`, and may optionally overload the shift operators `operator<<` `operator>>`, `operator>>=`, and `operator<<=`. The canonical implementations usually follow the pattern for binary arithmetic operators described above.

Boolean negation operator

The operator `operator!` is commonly overloaded by the user-defined classes that are intended to be used in boolean contexts. Such classes also provide a user-defined conversion function `explicit operator bool()` (see `std::basic_ios` for the standard library example), and the expected behavior of `operator!` is to return the value opposite of `operator bool`.

Rarely overloaded operators

The following operators are rarely overloaded:

- The address-of operator, `operator&`. If the unary & is applied to an lvalue of incomplete type and the complete type declares an overloaded `operator&`, the behavior is undefined (until C++11) it is implementation-defined whether the overloaded operator is used (since C++11). Because this operator may be overloaded, generic libraries use `std::addressof` to obtain addresses of objects of user-defined types. The best known example of a canonical overloaded operator& is the Microsoft class CComPtr (https://msdn.microsoft.com/en-us/library/31k6d0k7(v=vs.140).aspx) . An example of its use in EDSL can be found in boost.spirit (http://www.boost.org/doc/libs/release/libs/spirit/doc/html/spirit/qi/reference/operator/and_predicate.html) .
- The boolean logic operators, `operator&&` and `operator||`. Unlike the built-in versions, the overloads cannot implement short-circuit evaluation. Also unlike the built-in versions, they do not sequence their left operand before the right one. (until C++17) In the standard library, these operators are only overloaded for `std::valarray`.
- The comma operator, `operator,`. Unlike the built-in version, the overloads do not sequence their left operand before the right one. (until C++17) Because this operator may be overloaded, generic libraries use expressions such as `a,void(),b` instead of `a,b` to sequence execution of expressions of user-defined types. The boost library uses `operator,` in boost.assign (http://www.boost.org/doc/libs/release/libs/assign/doc/index.html#intro) , boost.spirit (https://github.com/boostorg/spirit/blob/develop/include/boost/spirit/home/qi/string/symbols.hpp#L317) , and other libraries. The database access library SOCI (http://soci.sourceforge.net/doc.html) also overloads `operator,`.
- The member access through pointer to member `operator->*`. There are no specific downsides to overloading this operator, but it is rarely used in practice. It was suggested that it could be part of smart pointer interface (http://www.aristeia.com/Papers/DDJ_Oct_1999.pdf) , and in fact is used in that capacity by actors in boost.phoenix (http://www.boost.org/doc/libs/release/libs/phoenix/doc/html/phoenix/modules/operator.html#phoenix.modules.operator.member_pointer_operator) . It is more common in EDSLs such as cpp.react (https://github.com/schlangster/cpp.react/blob/master/include/react/Signal.h#L557) .

Example

Run this code

```cpp
#include <iostream>

class Fraction
{
    int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
    int n, d;
public:
    Fraction(int n, int d = 1) : n(n/gcd(n, d)), d(d/gcd(n, d)) { }
    int num() const { return n; }
    int den() const { return d; }
    Fraction& operator*=(const Fraction& rhs)
    {
        int new_n = n * rhs.n/gcd(n * rhs.n, d * rhs.d);
        d = d * rhs.d/gcd(n * rhs.n, d * rhs.d);
        n = new_n;
        return *this;
    }
};
std::ostream& operator<<(std::ostream& out, const Fraction& f)
{
   return out << f.num() << '/' << f.den() ;
}
bool operator==(const Fraction& lhs, const Fraction& rhs)
{
    return lhs.num() == rhs.num() && lhs.den() == rhs.den();
}
bool operator!=(const Fraction& lhs, const Fraction& rhs)
{
    return !(lhs == rhs);
}
Fraction operator*(Fraction lhs, const Fraction& rhs)
{
    return lhs *= rhs;
}

int main()
{
   Fraction f1(3, 8), f2(1, 2), f3(10, 2);
   std::cout << f1 << " * " << f2 << " = " << f1 * f2 << '\n'
             << f2 << " * " << f3 << " = " << f2 * f3 << '\n'
             <<  2 << " * " << f1 << " = " <<  2 * f1 << '\n';
}
```

Output:

```
3/8 * 1/2 = 3/16
1/2 * 5/1 = 5/2
```

```
2 * 3/8 = 3/4
```

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 1458 (http://open-std.org/JTC1/SC22/WG21/docs/cwg_defects.html#1458)	C++11	taking address of incomplete type that overloads address-of was undefined behavior	the behavior is only unspecified

See Also

- Operator precedence
- Alternative operator syntax

Common operators

assignment	increment decrement	arithmetic	logical	comparison	member access	other
a = b	++a	+a	!a	a == b	a[b]	a(...)
a += b	--a	-a	a && b	a != b	*a	a, b
a -= b	a++	a + b	a || b	a < b	&a	? :
a *= b	a--	a - b		a > b	a->b	
a /= b		a * b		a <= b	a.b	
a %= b		a / b		a >= b	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				

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)

References

1. ↑ Operator Overloading (http://stackoverflow.com/questions/4421706/operator-overloading/4421708#4421708) on StackOverflow C++ FAQ

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/language/operators&oldid=94797"