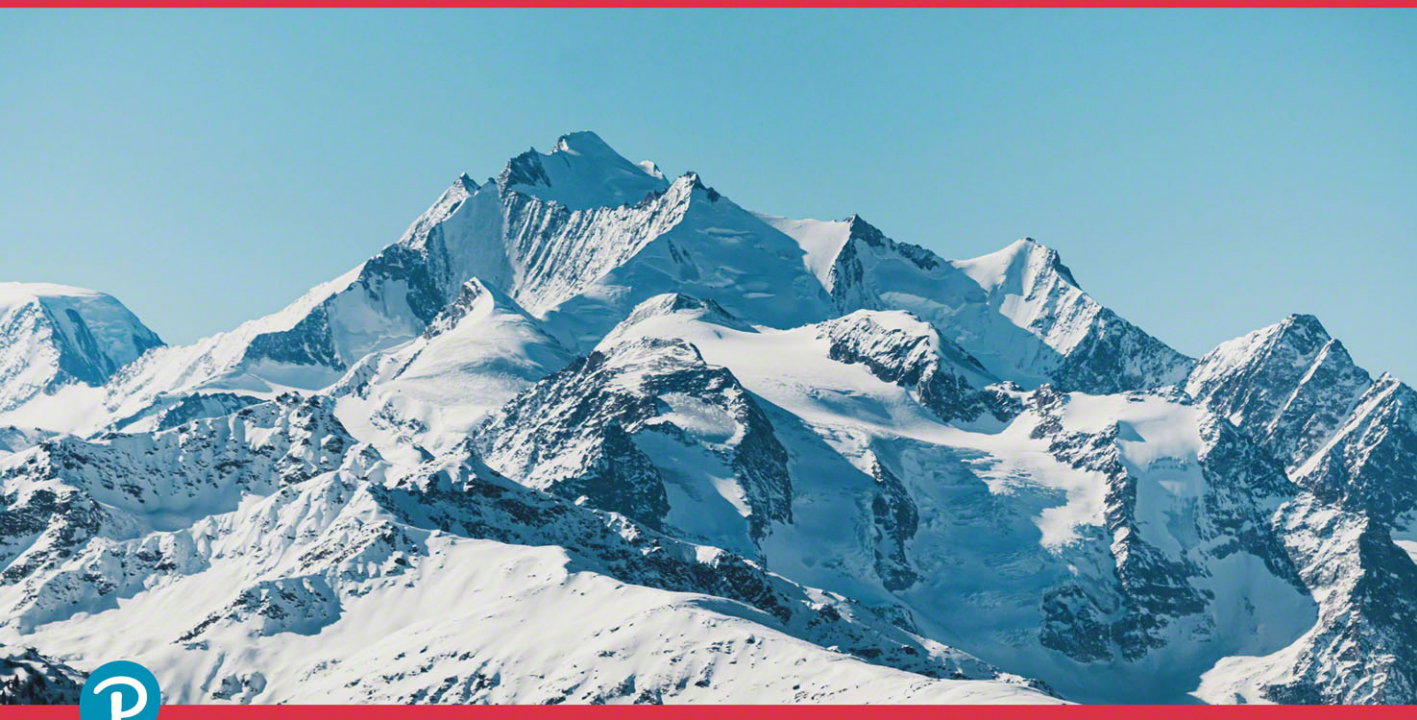
Updated for C++ 20

A Tour of C++

Third Edition

Bjarne Stroustrup



C++ In-Depth Series Bjarne Stroustrup

A Tour of C++
Third Edition

Alternatively, most operators can be defined as free-standing functions:

```
Matrix operator+(const Matrix& m1, const Matrix& m2);      // assign m1 to m2 and return the sum
```

It is conventional to define operators with symmetric operands as free-standing functions so that both operands are treated identically. To gain good performance from returning a potentially large object, such as a **Matrix**, we rely on move semantics (§6.2.2).

6.5 Conventional Operations

Some operations have conventional meanings when defined for a type. These conventional meanings are often assumed by programmers and libraries (notably, the standard library), so it is wise to conform to them when designing new types for which the operations make sense.

- Comparisons: **==**, **!=**, **<**, **<=**, **>**, **>=**, and **<=>** (§6.5.1)
- Container operations: **size()**, **begin()**, and **end()** (§6.5.2)
- Iterators and "smart pointers": **->**, *, **[]**, **++**, **--**, **+**, **-**, **+=**, and **-=** (§13.3, §15.2.1)
- Function objects: **()** (§7.3.2)
- Input and output operations: **>>** and **<<** (§6.5.4)
- **swap()** (§6.5.5)
- Hash functions: **hash<>** (§6.5.6)

6.5.1 Comparisons (Relational Operators)

The meaning of the equality comparisons (**==** and **!=**) is closely related to copying. After a copy, the copies should compare equal:

```
X a = something;
X b = a;
assert(a==b);   // if a!=b here, something is very odd (§4.5)
```

When defining **==**, also define **!=** and make sure that **a!=b** means **!(a==b)**.

Similarly, if you define **<**, also define **<=**, **>**, **>=** to make sure that the usual equivalences hold:

- **a<=b** means **(a<b)||(a==b)** and **!(b<a)**.
- **a>b** means **b<a**.
- **a>=b** means **(a>b)||(a==b)** and **!(a<b)**.

To give identical treatment to both operands of a binary operator, such as **==**, it is best defined as a free-standing function in the namespace of its class. For example:

```
namespace NX {
        class X {
                // ...
        };
        bool operator==(const X&, const X&);
        // ...
};
```

The "spaceship operator," **<=>** is a law onto itself; its rules differ from those for all other operators. In particular, by defining the default **<=>** the other relational operators are implicitly defined:

```
class R {
        // ...
        auto operator<=>(const R& a) const = default;
};

void user(R r1, R r2)
{
        bool b1 = (r1<=>r2) == 0; // r1==r2
        bool b2 = (r1<=>r2) < 0;  // r1<r2
        bool b3 = (r1<=>r2) > 0;  // r1>r2

        bool b4 = (r1==r2);
        bool b5 = (r1<r2);
}
```

Like C's **strcmp()**, **<=>** implements a three-way-comparison. A negative return value means less-than, 0 means equal, and a positive value means greater-than.

If **<=>** is defined as non-default, **==** is not implicitly defined, but **<** and the other relational operators are! For example:

```
struct R2 {
        int m;
        auto operator<=>(const R2& a) const { return a.m == m ? 0 : a.m < m ? –1 : 1; }
};
```

Here, I used the expression form of the **if**-statement : **p?x:y** is an expression that evaluates the condition **p** and if it is true, the value of the **?:** expression is **x** otherwise **y**.

```
void user(R2 r1, R2 r2)
{
        bool b4 = (r1==r2);  // error: no non-default ==
        bool b5 = (r1<r2);   // OK
}
```

This leads to this pattern of definition for nontrivial types:

```
struct R3 { /* ... */ };

auto operator<=>(const R3& a,const R3& b) { /* ... */ }

bool operator==(const R3& a, const R3& b) { /* ... */ }
```

Most standard-library types, such as **string** and **vector**, follow that pattern. The reason is that if a type has more than one element taking part in a comparison, the default **<=>** examines them one at a time yielding a lexicographical order. In such case, it is often worthwhile to provide a separate optimized **==** in addition because **<=>** has to examine all elements to determine all three alternatives. Consider comparing character strings:

```
string s1 = "asdfghjkl";
string s2 = "asdfghjk";

bool b1 = s1==s2;          // false
bool b2 = (s1<=>s2)==0;  // false
```

Using a conventional `==` we find that the strings are not equal by looking at the number of characters. Using `<=>`, we have to read all the characters of `s2` to find that it is less than `s1` and therefore not equal.

There are many more details to operator `<=>`, but those are primarily of interest to advanced implementors of library facilities concerned with comparisons and sorting beyond the scope of this book. Older code does not use `<=>`.

6.5.2 Container Operations

Unless there is a really good reason not to, design containers in the style of the standard-library containers (Chapter 12). In particular, make the container resource safe by implementing it as a handle with appropriate essential operations (§6.1.1, §6.2).

The standard-library containers all know their number of elements and we can obtain it by calling `size()`. For example:

```
for (size_t i = 0; i!=c.size(); ++i)    // size_t is the name of the type returned by a standard-library size()
    c[i] = 0;
```

However, rather than traversing containers using indices from `0` to `size()`, the standard algorithms (Chapter 13) rely on the notion of *sequence*s delimited by pairs of *iterator*s:

```
for (auto p = c.begin(); p!=c.end(); ++p)
    *p = 0;
```

Here, `c.begin()` is an iterator pointing to the first element of `c` and `c.end()` points one-beyond-the-last element of `c`. Like pointers, iterators support `++` to move to the next element and `*` to access the value of the pointed-to element.

These `begin()` and `end()` functions are also used by the implementation of the range-`for`, so we can simplify loops over a range:

```
for (auto& x : c)
    x= 0;
```

Iterators are used to pass sequences to standard-library algorithms. For example:

```
sort(v.begin(),v.end());
```

This *iterator model* (§13.3) allows for great generality and efficiency. For details and more container operations, see Chapter 12 and Chapter 13.

The `begin()` and `end()` can also be defined as free-standing functions; see §7.2. The versions of `begin()` and `end()` for `const` containers are called `cbegin()` and `cend()`.

6.5.3 Iterators and "smart pointers"

User-defined iterators (§13.3) and "smart pointers" (§15.2.1) implement the operators and aspects of a pointer desired for their purpose and often add semantics as needed.

- Access: `*`, `->` (for a class), and `[]` (for a container)
- Iteration/navigation: `++` (forward), `--` (backward) , `+=`, `-=`, `+`, and `-`
- Copy and/or move: `=`

6.5.4 Input and Output Operations

For pairs of integers, `<<` means left-shift and `>>` means right-shift. However, for `iostreams`, they are the output and input operators, respectively (§1.8, Chapter 11). For details and more I/O operations, see Chapter 11.

6.5.5 swap()

Many algorithms, most notably `sort()`, use a `swap()` function that exchanges the values of two objects. Such algorithms generally assume that `swap()` is very fast and doesn't throw an exception. The standard-library provides a `std::swap(a,b)` implemented as three move operations (§16.6). If you design a type that is expensive to copy and could plausibly be swapped (e.g., by a sort function), then give it move operations or a `swap()` or both. Note that the standard-library containers (Chapter 12) and `string` (§10.2.1) have fast move operations.

6.5.6 hash<>

The standard-library `unordered_map<K,V>` is a hash table with `K` as the key type and `V` as the value type (§12.6). To use a type `X` as a key, we must define `hash<X>`. For common types, such as `std::string`, the standard library defines `hash<>` for us.

6.6 User-Defined Literals

One purpose of classes was to enable the programmer to design and implement types to closely mimic built-in types. Constructors provide initialization that equals or exceeds the flexibility and efficiency of built-in type initialization, but for built-in types, we have literals:

- `123` is an `int`.
- `0xFF00u` is an `unsigned int`.
- `123.456` is a `double`.
- `"Surprise!"` is a `const char[10]`.

It can be useful to provide such literals for a user-defined type also. This is done by defining the meaning of a suitable suffix to a literal, so we can get

- `"Surprise!"s` is a `std::string`.
- `123s` is `second`s.
- `12.7i` is `imaginary` so that `12.7i+47` is a `complex` number (i.e., `{47,12.7}`).

In particular, we can get these examples from the standard library by using suitable headers and namespaces:

Standard-Library Suffixes for Literals		
<chrono>	std::literals::chrono_literals	h, min, s, ms, us, ns
<string>	std::literals::string_literals	s
<string_view>	std::literals::string_literals	sv
<complex>	std::literals::complex_literals	i, il, if

Literals with user-defined suffixes are called *user-defined literals* or *UDL*s. Such literals are defined using *literal operators*. A literal operator converts a literal of its argument type, followed by a subscript, into its return type. For example, the `i` for `imaginary` suffix might be implemented like this:

```
constexpr complex<double> operator""i(long double arg)    // imaginary literal
{
    return {0,arg};
}
```

Here

- The `operator""` indicates that we are defining a literal operator.
- The `i` after the *literal indicator*, `""`, is the suffix to which the operator gives a meaning.
- The argument type, `long double`, indicates that the suffix (`i`) is being defined for a floating-point literal.
- The return type, `complex<double>`, specifies the type of the resulting literal.

Given that, we can write

```
complex<double> z = 2.7182818+6.283185i;
```

The implementation of the `i` suffix and the `+` are both `constexpr`, so the computation of `z`'s value is done at compile time.

6.7 Advice

[1] Control construction, copy, move, and destruction of objects; §6.1.1; [CG: R.1].
[2] Design constructors, assignments, and the destructor as a matched set of operations; §6.1.1; [CG: C.22].
[3] Define all essential operations or none; §6.1.1; [CG: C.21].
[4] If a default constructor, assignment, or destructor is appropriate, let the compiler generate it; §6.1.1; [CG: C.20].
[5] If a class has a pointer member, consider if it needs a user-defined or deleted destructor, copy and move; §6.1.1; [CG: C.32] [CG: C.33].
[6] If a class has a user-defined destructor, it probably needs user-defined or deleted copy and move; §6.2.1.
[7] By default, declare single-argument constructors `explicit`; §6.1.2; [CG: C.46].
[8] If a class member has a reasonable default value, provide it as a data member initializer; §6.1.3; [CG: C.48].
[9] Redefine or prohibit copying if the default is not appropriate for a type; §6.1.1; [CG: C.61].

[10] Return containers by value (relying on copy elision and move for efficiency); §6.2.2; [CG: F.20].
[11] Avoid explicit use of **std::copy()**; §16.6; [CG: ES.56].
[12] For large operands, use **const** reference argument types; §6.2.2; [CG: F.16].
[13] Provide strong resource safety; that is, never leak anything that you think of as a resource; §6.3; [CG: R.1].
[14] If a class is a resource handle, it needs a user-defined constructor, a destructor, and non-default copy operations; §6.3; [CG: R.1].
[15] Manage all resources – memory and non-memory – resources using RAII; §6.3; [CG: R.1].
[16] Overload operations to mimic conventional usage; §6.5; [CG: C.160].
[17] If you overload an operator, define all operations that conventionally work together; §6.1.1, §6.5.
[18] If you define **<=>** for a type as non-default, also define **==**; §6.5.1.
[19] Follow the standard-library container design; §6.5.2; [CG: C.100].