A Tour Of C++

Bjarne Stroustrup

November 7, 2022

Contents

1 The Basics

1.1 Introduction

The operator `<<` ("put to") writes its second argument onto its first

A function declaration gives the name of the function, the type of the value returned (if any), and the number and types of the arguments that must be supplied in a call

If two functions are defined with the same name, but with different argument types, the compiler will choose the most appropriate function to invoke for each call.

Defining multiple functions with the same name is known as function **overloading** and is one of the essential parts of generic programming

1.2 Types, Variables and Arithemtic

A **declaration** is a statement that introduces an entity into the program. It specifies a type for the entity:

- A **type** defines a set of possible values and a set of operations (for an object)
- An **object** is some memory that holds a value of some type.

- A **value** is a set of bits interpreted according to a type.
- A **variable** is a named object.

Unfortunately, conversions that lose information, **narrowing conversions**, such as `double` to `int` and `int` to `char`, are allowed and implicitly applied when you use `=` (but not when you use `{}`)

When defining a variable, you don't need to state its type explicitly when it can be deduced from the initializer:

```
auto b = true;     // a bool
auto ch = 'x';     // a char
auto i = 123;      // an int
auto d = 1.2;      // a double
auto z = sqrt(y); // z has the type of whatever
                   // sqrt(y) returns
auto bb {true};    //bb is a bool
```

With `auto`, we tend to use the `=` because there is no potentially troublesome type conversion involved, but if you prefer to use `{}` initialization consistently, you can do that instead.

1.3 Scope and Lifetime

- **Local scope**: A name declared in a function or lambda is called a local name. Its scope extends from its point of declaration to the end of the block in which its declaration occurs. A **block** is delimited by a `{ }` pair. Function argument names are considered local names.
- **Class scope**: A name is called a **member name** (or a **class member name**) if it is defined in a class , outside any function , lambda, or enum class. Its scope extends from the opening `{` of its enclosing declaration to the end of that declaration.
- **Namespace scope**: A name is called a **namespace member name** if it is defined in a namespace outside any function, lambda, class, or enum class. Its scope extends from the point of declaration to the end of its namespace.

```
vector<int> vec; // vec is global
struct Record {
    string name; // name is a member of Record
// ...
};
void fct(int arg) { // fct is global (a global function)
```

```
                    // arg is local (an integer argument)
    string motto {"Who dares wins"}; // motto is local
    auto p = new Record{"Hume"};
    // p points to an unnamed Record (created by new)
    // ...
}
```

1.4 Constants

C++ supports two notions of immutability:

- `const`: meaning roughly "I promise not to change this value." This is used primarily to specify interfaces so that data can be passed to functions using pointers and references without fear of it being modified. The compiler enforces the promise made by `const`. The value of a `const` can be calculated at run time.
- `constexpr`: meaning roughly "to be evaluated at compile time." This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted), and for performance. The value of a `constexpr` must be calculated by the compiler.

For example

```
constexpr int dmv = 17;            // dmv is a named constant
int var = 17;                      // var is not a constant
const double sqv = sqrt(var);      // sqv is a named constant,
                                   // possibly computed at run time
double sum(const vector<double>&);// sum will not modify
                                   // its argument
vector<double> v {1.2, 3.4, 4.5}; // v is not a constant
const double s1 = sum(v);          // OK: sum(v) is evaluated at
                                   // run time
constexpr double s2 = sum(v);      // error: sum(v) is not a
                                   // constant expression
```

For a function to be usable in a **constant expression**, that is, in an expression that will be evaluated by the compiler, it must be defined `constexpr`. For example:

```
constexpr double square(double x) { return x*x; }
constexpr double max1 = 1.4*square(17);
// OK 1.4*square(17) is a constant expression
constexpr double max2 = 1.4*square(var);
// error: var is not a constant expression
const double max3 = 1.4*square(var);
// OK, may be evaluated at run time
```

A constexpr function can be used for non-constant arguments, but when that is done the result is not a constant expression. We allow a constexpr function to be called with non-constant-expression arguments in contexts that do not require constant expressions. That way, we don't have to define essentially the same function twice: once for constant expressions and once for variables.

To be constexpr, a function must be rather simple and cannot have side effects and can only use information passed to it as arguments. In particular, it cannot modify non-local variables, but it can have loops and use its own local variables. For example:

```
constexpr double nth(double x, int n) // assume 0<=n {
{
    double res = 1;
    int i = 0;
    while (i<n) {
        res*=x;
        ++i;
    }
    return res;
}
```

1.5 Pointers, Arrays, and References

```
char* p = &v[3];
char x = *p;
```

in an expression, prefix unary * means "contents of" and prefix unary & means "address of"

If we didn't want to copy the values from v into the variable x, but rather just have x refer to an element, we could write:

```
void increment() {
    int v[] = {0,1,2,3,4,5,6,7,8,9};
    for (auto& x : v) // add 1 to each x in v
        ++x;
    // ...
}
```

In a declaration, the unary suffix & means "reference to." A reference is similar to a pointer, except that you don't need to use a prefix * to access the value referred to by the reference. Also, a reference cannot be made to refer to a different object after its initialization.

References are particularly useful for specifying function arguments. For example:

```
void sort(vector<double>& v); // sort v
                              // v is a vector of doubles
```

By using a reference, we ensure that for a call `sort(vec)`, we do not copy `vec` and that it really is `vec` that is sorted and not a copy of it.

When used in declarations, operators (such as `&`, `*`, and `[]`) are called declarator operators:

```
T a[n] // T[n]: a is an array of n Ts
T* p   // T*: p is a pointer to T
T& r   // T&: r is a reference to T
T f(A) // T(A): f is a function taking an argument of type A
       // returning a result of type T
```

We try to ensure that a pointer always points to an object so that dereferencing it is valid. When we don't have an object to point to or if we need to represent the notion of "no object available" (e.g., for an end of a list), we give the pointer the value `nullptr` ("the null pointer"). There is only one `nullptr` shared by all pointer types:

```
double* pd = nullptr;
Link<Record>* lst = nullptr; // pointer to a Link to a Record
int x = nullptr; // error: nullptr is a pointer not an integer
```

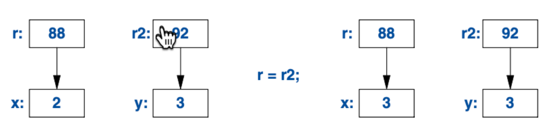
1.6 Tests

1.7 Mapping to Hardware

An assignment of a built-in type is a simple machine copy operation.

A reference and a pointer both refer/point to an object and both are represented in memory as a machine address. However, the language rules for using them differ. Assignment to a reference does not change what the reference refers to but assigns to the referenced object:

```
int x = 2;
int y = 3;
int& r = x; // r refers to x
int& r2 = y; // now r2 refers to y
r = r2; // read through r2, write through r: x becomes 3
```



2 User-Defined Types

2.1 Introduction

Types built out of other types using C++'s abstraction mechanisms are called **user-defined types**. They are referred to as **classes** and **enumerations**.

2.2 Structures

The `new` operator allocates memory from an area called the **free store** (also known as **dynamic memory** and **heap**). Objects allocated on the free store are independent of the scope from which they are created and "live" until they are destroyed using the `delete` operator

2.3 Classes

The language mechanism for that is called a **class**. A class has a set of **members**, which can be data, function, or type members. The interface is defined by the `public` members of a class, and `private` members are accessible only through that interface.

```
class Vector {
    public:
        Vector(int s) :elem{new double[s]}, sz{s} { }
        double& operator[](int i) { return elem[i]; }
        int size() { return sz; }
    private:
        double* elem; // pointer to the elements
        int sz; // the number of elements
};
```

`Vector(int)` defines how objects of type `Vector` are constructed. The constructor initializes the `Vector` members using a member initializer list:

```
:elem{new double[s]}, sz{s}
```

That is, we first initialize `elem` with a pointer to `s` elements of type `double` obtained from the free store. Then, we initialize `sz` to `s`

Access to elements is provided by a subscript function, called `operator[]`. It returns a reference to the appropriate element (a `double&` allowing both reading and writing)

There is no fundamental difference between a `struct` and a `class`; a `struct` is simply a class with members `public` by default.

2.4 Unions

A `union` is a `struct` in which all members are allocated at the same address so that the `union` occupies only as much space as its largest member. Naturally, a `union` can hold a value for only one member at a time.

```
union Value {
    Node* p;
    int i;
};
```

The language doesn't keep track of which kind of value is held by a union, so the programmer must do that:

```
enum Type { ptr, num }; // a Type can hold values ptr and num

struct Entry {
    string name;
    Type t;
    Value v; // use v.p if t==ptr; use v.i if t==num
};

void f(Entry* pe) {
    if (pe->t == num)
        cout << pe->v.i;
    // ...
}
```

Maintaining the correspondence between a **type field** (here, `t`) and the type held in a `union` is error-prone.

The standard library type, `variant`, can be used to eliminate most direct uses of unions. A `variant` stores a value of one of a set of alternative types.

```
struct Entry {
    string name;
    variant<Node*,int> v;
};

void f(Entry* pe) {
if (holds_alternative<int>(pe->v))
    // does *pe hold an int?
    cout << get<int>(pe->v);
    // get the int
    // ...
}
```

For many uses, a `variant` is simpler and safer to use than a `union`

2.5 Enumerations

```
enum class Color { red, blue, green };
enum class Traffic_light { green, yellow, red };
Color col = Color::red;
Traffic_light light = Traffic_light::red;
```

Note that enumerators (e.g., `red`) are in the scope of their `enum class`, so that they can be used repeatedly in different `enum classes` without confusion. For example, `Color::red` is `Color` 's `red` which is different from `Traffic_light::red`.

Enumerations are used to represent small sets of integer values. They are used to make code more readable and less error-prone than it would have been had the symbolic (and mnemonic) enumerator names not been used.

The `class` after the `enum` specifies that an enumeration is strongly typed and that its enumerators are scoped.

```
Color x = red; // error : which red?
Color y = Traffic_light::red;
// error: that red is not a Color
Color z = Color::red; // OK
```

Similarly, we cannot implicitly mix `Color` and integer values:

```
int i = Color::red; // error: Color::red is not an int
Color c = 2; // initialization error: 2 is not a Color
```

By default, an `enum class` has only assignment, initialization, and comparisons. However, an enumeration is a user-defined type, so we can define operators for it:

```
Traffic_light& operator++(Traffic_light& t)
{ // prefix increment: ++
        switch (t) {
            case Traffic_light::green:
                return t=Traffic_light::yellow;
            case Traffic_light::yellow:
                return t=Traffic_light::red;
            case Traffic_light::red:
                return t=Traffic_light::green;
}
}
Traffic_light next = ++light;
// next becomes Traffic_light::green
```

If you don't want to explicitly qualify enumerator names and want enumerator values to be ints (without the need for an explicit conversion), you can remove the `class` from `enum class` to get a "plain" `enum`. The enumerators from a "plain" `enum` are entered into the same scope as the name of their enum and implicitly converts to their integer value

```
enum Color { red, green, blue };
int col = green;
```

Here `col` gets the value 1. By default, the integer values of enumerators start with `0` and increase by one for each additional enumerator.

3 Modularity

3.1 Introduction

A **declaration** specifies all that's needed to use a function or a type. For example:

```
double sqrt(double);
// the square root function takes a double and returns a double
class Vector {
    public:
        Vector(int s);
        double& operator[](int i); int size();
    private:
        double* elem; // elem points to an array of
                      // sz doubles int sz;
};
```

The key point here is that the function bodies, the function **definitions**, are "elsewhere"

The definition of `sqrt()` will look like this:

```
double sqrt(double d) // definition of sqrt()
{
    // ... algorithm as found in math textbook ...
}
```

For `vector`, we need to define

```
Vector::Vector(int s) // definition of the constructor
    :elem{new double[s]}, sz{s}
     // initialize members
{
}
```

```
double& Vector::operator[](int i) {
    // definition of subscripting
    return elem[i];
}
int Vector::size() {
    // definition of size()
    return sz;
}
```

3.2 Separate Compilation

C++ supports a notion of separate compilation where user code sees only declarations of the types and functions used. The definitions of those types and functions are in separate source files and are compiled separately.

This can be used to organize a program into a set of semi-independent code fragments. Such separation can be used to minimize compilation times and to strictly enforce sepa- ration of logically distinct parts of a program (thus minimizing the chance of errors). A library is often a collection of separately compiled code fragments (e.g., functions).

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. Example:

```
// Vector.h:
class Vector {
    public:
        Vector(int s);
        double& operator[](int i); int size();
    private:
        double* elem;
        int sz;
};
```

This declaration would be placed in a file `Vector.h`. Users then **include** that file, called a **header file**, to access that interface. For example:

```
// user.cpp:
#include "Vector.h" // get Vector's interface
#include <cmath> // get the standard-library
                 // math function interface including sqrt()
double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]);
    return sum;
}
```

To help the compiler ensure consistency, the `.cpp` file providing the implementation of `Vector` will also include the .h file providing its interface:

```
// Vector.cpp:
#include "Vector.h" // get Vector's interface

Vector::Vector(int s)
    :elem{new double[s]}, sz{s}
{
}
double& Vector::operator[](int i)
{
    return elem[i];
}
int Vector::size()
{
    return sz;
}
```

The code in `user.cpp` and `Vector.cpp` shares the `Vector` interface information presented in `Vector.h`, but the two files are otherwise independent and can be separately compiled.

A `.cpp` file that is compiled by itself (including the h files it `#includes`) is called a **translation unit**. A program can consist of many thousand translation units.

3.3 Modules (C++20)

The use of `#includes` is a very old, error-prone, and rather expensive way of composing programs out of parts. If you `#include header.h` in 101 translation units, the text of `header.h` will be processed by the compiler 101 times. If you `#include header1.h` before `header2.h` the declarations and macros in `header1.h` might affect the meaning of the code in `header2.h`. If instead you `#include header2.h` before `header1.h`, it is `header2.h` that might affect the code in `header1.h`. Obviously, this is not ideal, and in fact it has been a major source of cost and bugs since 1972 when this mechanism was first introduced into C.

Consider how to express the `Vector` and `sqrt_sum()` example from §3.2 using `modules`:

```
// file Vector.cpp:
module; // this compilation will define a module
// ... here we put stuff that Vector might
// need for its implementation ...
export module Vector; // defining the module called "Vector"
```

```
export class Vector {
    public:
        Vector(int s);
        double& operator[](int i); int size();
    private:
        double* elem; // elem points to an array of sz doubles
        int sz;
};

Vector::Vector(int s)
:elem{new double[s]}, sz{s}
{
}

double& Vector::operator[](int i)
{
return elem[i];
}

int Vector::size()
{
return sz;
}

export int size(const Vector& v) { return v.size(); }
```

This defines a module called `Vector`, which exports the class Vector, all its member functions, and the non-member function `size()`

The way we use this module is to `import` it where we need it. For example:.

```
// file user.cpp:
//
import Vector; // get Vector's interface
#include <cmath>

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]);
    return sum;
}
```

The differences between headers and modules are not just syntactic. • A module is compiled once only (rather than in each translation unit in which it is used). • Two modules can be `imported` in either order without changing their meaning. • If you import something into a module, users of your module do not implicitly gain access to (and are not bothered by) what you imported: `import` is not transitive.

3.4 Namespaces

C++ offers **namespaces** as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names

```
namespace My_code {
    class complex {
        // ...
    };
    complex sqrt(complex);
    // ...
    int main();
}

int My_code::main()
{
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
}

int main()
{
    return My_code::main();
}
```

By putting my code into the namespace `My_code`, I make sure that my names do not conflict with the standard-library names in namespace `std`

If repeatedly qualifying a name becomes tedious or distracting, we can bring the name into a scope with a `using`-declaration:

```
void my_code(vector<int>& x, vector<int>& y)
{
    using std::swap; // ...
    swap(x,y);
    other::swap(x,y); // ...
}
```

To gain access to all names in the standard-library namespace, we can use a `using`-directive:

```
using namespace std;
```

3.5 Error Handling

3.5.1 Exceptions

Consider again the `Vector` example.

Assuming that out-of-range access is a kind of error that we want to recover from, the solution is for the `Vector` implementer to detect the attempted out-of-range access and tell the user about it. The user can then take appropriate action. For example, `Vector::operator[]()` can detect an attempted out-of-range access and throw an `out_of_range` exception:

```
double& Vector::operator[](int i)
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

The `throw` transfers control to a handler for exceptions of type `out_of_range` in some function that directly or indirectly called `Vector::operator[]()`. To do that, the implementation will **unwind** the function call stack as needed to get back the context of that caller. That is, the exception handling mechanism will exit scopes and functions as needed to get back to a caller that has expressed interest in handling that kind of exception, invoking destructors (§4.2.2) along the way as needed. For example:

```
void f(Vector& v) {
// ...
    try { // exceptions here are handled by
          // the handler defined below
        v[v.size()] = 7; // try to access beyond the end of v
    }
    catch (out_of_range& err) {
    // ... handle range error ...
        cerr << err.what() << '\n';
    }
    // ...
}
```

We put code for which we are interested in handling exceptions into a `try`-block. The attempted assignment to `v[v.size()]` will fail. Therefore, the `catch`-clause providing a handler for exceptions of type `out_of_range` will be entered. The `out_of_range` type is defined in the standard library (in `<stdexcept>`) and is in fact used by some standard-library container access functions.

The main technique for making error handling simple and systematic (called **Resource Acquisition Is Initialization**; RAII) is explained in §4.2.2. The basic idea behind RAII is for a constructor to acquire all resources necessary for a class to operate and have the destructor release all resources, thus making resource release guaranteed and implicit.

A function that should never throw an exception can be declared `noexcept`. For example:

```
void user(int sz) noexcept {
    Vector v(sz);
    iota(&v[0],&v[sz],1); // fill v with 1,2,3,4...
    // ...
}
```

3.5.2 Invariants

The use of exceptions to signal out-of-range access is an example of a function checking its argument and refusing to act because a basic assumption, a **precondition**, didn't hold

```
Vector::Vector(int s)
{
    if (s<0)
        throw length_error{"Vector constructor: negative size"};
    elem = new double[s];
    sz = s;
}
```

If operator `new` can't find memory to allocate, it throws a `std::bad_alloc`.

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error& err) {
// handle negative size
    }
    catch (std::bad_alloc& err) {
// handle memory exhaustion
    }
}
```

Often, a function has no way of completing its assigned task after an exception is thrown. Then, "handling" an exception means doing some minimal local cleanup and rethrowing the exception.

```
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error&) {
        // do something and rethrow
        cerr << "test failed: length error\n";
        throw; // rethrow
    }
    catch (std::bad_alloc&) {
        // Ouch! this program is not designed to handle memory exhaustion
        std::terminate(); // terminate the program
    }
}
```

3.5.3 Error-Handling Alternatives

Throwing an exception is not the only way of reporting an error that cannot be handled locally. A function can indicate that it cannot perform its allotted task by: • throwing an exception • somehow return a value indicating failure • terminating the program (by invoking a function like `terminate()`, `exit()`, or `abort()`).

One way to ensure termination is to add `noexcept` to a function so that a `throw` from anywhere in the function's implementation will turn into a `terminate()`.

3.5.4 Contracts

The standard library offers the debug macro, `assert()`, to assert that a condition must hold at run time. For example:

```
void f(const char* p)
{
    assert(p!=nullptr);
    // p must not be the nullptr
}
```

If the condition of an `assert()` fails in "debug mode", the program terminates

3.5.5 Static Assertions

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so.

The `static_assert` mechanism can be used for anything that can be expressed in terms of constant expressions

```
constexpr double C = 299792.458; // km/s
void f(double speed)
{
    constexpr double local_max = 160.0/(60*60); // 160 km/h == 160.0/(60*60) km/s
    static_assert(speed<C,"can't go that fast"); // error: speed must be a constant
    static_assert(local_max<C,"can't go that fast"); // OK
    // ...
}
```

In general, `static_assert(A,S)` prints `S` as a compiler error message if `A` is not `true`. If you don't want a specific message printed, leave out the `S` and the compiler will supply a default message:

4 Classes

4.1 Introduction

4.2 Concrete Types

The basic idea of **concrete classes** is that they behave "just like built-in types."

4.2.1 An Arithmetic Type

```
class complex {
    double re, im; // representation: two doubles
    public:
        // construct complex from two scalars
        complex(double r, double i) :re{r}, im{i} {}
        // construct complex from one scalar
        complex(double r) :re{r}, im{0} {}
        // default complex: {0,0}
        complex() :re{0}, im{0} {}

        double real() const { return re; }
        void real(double d) { re=d; }
        double imag() const { return im; }
        void imag(double d) { im=d; }

        complex& operator+=(complex z) {
            re+=z.re; // add to re and im im+=z.im;
            return *this; // and return the result
        }
        complex& operator-=(complex z) {
```

```
            re-=z.re;
            im-=z.im;
            return *this;
        }
        complex& operator*=(complex); // defined out-of-class somewhere
        complex& operator/=(complex); // defined out-of-class somewhere
};
```

`complex` must be efficient or it will remain unused. This implies that simple operations must be inlined. That is, simple operations (such as constructors, `+=`, and `imag()`) must be implemented without function calls in the generated machine code. **Functions defined in a class are inlined by default**. It is possible to explicitly request inlining by preceding a function declaration with the keyword `inline`

A constructor that can be invoked without an argument is called a **default constructor**.

The `const` specifiers on the functions returning the real and imaginary parts indicate that these functions do not modify the object for which they are called. A `const` member function can be invoked for both `const` and non-`const` objects, but a non-`const` member function can only be invoked for non-`const` objects. stackexchange

```
complex z = {1,0};
const complex cz {1,3};
z = cz; // OK: assigning to a non-const variable
cz = z; // error: complex::operator=() is a non-const member function double
x = z.real(); // OK: complex::real() is a const member function
```

Many useful operations do not require direct access to the representation of complex, so they can be defined separately from the class definition:

```
complex operator+(complex a, complex b) { return a+=b; }
complex operator-(complex a, complex b) { return a-=b; }
complex operator-(complex a) { return {-a.real(), -a.imag()}; }
complex operator*(complex a, complex b) { return a*=b; }
complex operator/(complex a, complex b) { return a/=b; }
```

The compiler converts operators involving complex numbers into appropriate function calls. For example, `c!=b` means operator `!=(c,b)` and `1/a` means operator `/(complex{1},a)`.

User-defined operators ("overloaded operators") should be used cautiously and conventionally. The syntax is fixed by the language, so you can't define a unary `/`. Also, it is not possible to change the meaning of an operator for built-in types, so you can't redefine `+` to subtract `ints`.

4.2.2 A Container

A **container** is an object holding a collection of elements.

We need a mechanism to ensure that the memory allocated by the constructor is deallocated; that mechanism is a **destructor**

```
class Vector { public:
        Vector(int s) :elem{new double[s]}, sz{s}
        // constructor: acquire resources
        {
            // initialize elements
            for (int i=0; i!=s; ++i)
                elem[i]=0;
        }
        // destructor: release resources
        ~Vector() { delete[] elem; }

        double& operator[](int i);
        int size() const;

    private:
        double* elem; // elem points to an array of sz doubles
        int sz;
};
```

`Vector`'s constructor allocates some memory on the free store (also called the **heap** or **dynamic store**) using the `new` operator. The destructor cleans up by freeing that memory using the `delete[]` operator. Plain `delete` deletes an individual object, `delete[]` deletes an array.

The technique of acquiring resources in a constructor and releasing them in a destructor, known as **Resource Acquisition Is Initialization** or **RAII**, allows us to eliminate "naked `new` operations", that is, to avoid allocations in general code and keep them buried inside the implementation of well-behaved abstractions.

4.2.3 Initializing Containers

- **Initializer-list constructor**: Initialize with a list of elements.
- `push_back()`: Add a new element at the end of (at the back of) the sequence.

```
class Vector {
    public:
        // initialize with a list of doubles
```

```
        Vector(std::initializer_list<double>);
        // ...
        // add element at end, increasing the size by one
        void push_back(double);
        // ...
};
```

The `push_back()` is useful for input of arbitrary numbers of elements

```
Vector read(istream& is) {
    Vector v;
    for (double d; is>>d; ) // read floating-point values into d
        v.push_back(d); // add d to v return v;
}
```

The input loop is terminated by an end-of-file or a formatting error.
The way to provide Vector with a move constructor, so that returning a potentially huge amount of data from read() is cheap

```
Vector v = read(cin); // no copy of Vector elements here
```

The `std::initializer_list` used to define the initializer-list constructor is a standard-library type known to the compiler: when we use a `{}`-list, such as `{1,2,3,4}`, the compiler will create an object of type `initializer_list` to give to the program. So, we can write:

```
 // v1 has 5 elements Vector
Vector v1 = {1,2,3,4,5};
// v2 has 4 elements
v2 = {1.23, 3.45, 6.7, 8};
```

`Vector`'s initializer-list constructor might be defined like this:

```
Vector::Vector(std::initializer_list<double> lst) // initialize with a list
    :elem{new double[lst.size()]}, sz{static_cast<int>(lst.size())}
{
    copy(lst.begin(),lst.end(),elem); // copy from lst into elem (§12.6)
}
```

Unfortunately, the standard-library uses `unsigned` integers for sizes and subscripts, so I need to use the ugly `static_cast` to explicitly convert the size of the initializer list to an `int`

A `static_cast` does not check the value it is converting; the programmer is trusted to use it correctly.

Other casts are `reinterpret_cast` for treating an object as simply a sequence of bytes and `const_cast` for "casting away `const`."

4.3 Abstract Types

an **abstract type** is a type that completely insulates a user from implementation details

First, we define the interface of a class `Container`, which we will design as a more abstract version of our `Vector`:

```
class Container {
    public:
        // pure virtual function
        virtual double& operator[](int) = 0;
        // const member function (§4.2.1)
        virtual int size() const = 0;
        // destructor (§4.2.2)
        virtual ~Container() {}
};
```

The word `virtual` means “may be redefined later in a class derived from this one”, and a function declared `virtual` is called a **virtual function**.

A class derived from `Container` provides an implementation for the `Container` interface. The curious `=0` syntax says the function is **pure virtual**; that is, some class derived from Container must define the function. Thus, it is not possible to define an object that is just a Container. For example:

```
Container c; // error: there can be no objects of an abstract class
Container* p = new Vector_container(10); // OK: Container is an interface
```

A `Container` can only serve as the interface to a class that implements its `operator[]()` and `size()` functions. A class with a pure virtual function is called an **abstract class**.

This `Container` can be used like this:

```
void use(Container& c) {
    const int sz = c.size();
    for (int i=0; i!=sz; ++i)
        cout << c[i] << '\n';
}
```

Note how `use()` uses the `Container` interface in complete ignorance of implementation details. It uses `size()` and `[]` without any idea of exactly which type provides their implementation. A class that provides the interface to a variety of other classes is often called a **polymorphic type**.

As is common for abstract classes, `Container` does not have a constructor. After all, it does not have any data to initialize. On the other hand, `Container` does have a destructor and that destructor is `virtual`, so that classes derived from `Container` can provide implementations. Again, that is common for abstract classes because they tend to be manipulated through references or pointers, and someone destroying a `Container` through a pointer has no idea what resources are owned by its implementation;

For `Container` to be useful, we have to implement a container that implements the functions required by its interface. For that, we could use the concrete class `Vector`:

```
class Vector_container : public Container {
    // Vector_container implements Container
    public:
        Vector_container(int s) : v(s) { } // Vector of s elements
        ~Vector_container() {}

        double& operator[](int i) override { return v[i]; }
        int size() const override { return v.size(); }
    private:
        Vector v;
};
```

The `:public` can be read as "is derived from" or "is a subtype of." Class `Vector_container` is said to be **derived** from class `Container`, and class `Container` is said to be a **base** of class `Vector_container`. An alternative terminology calls `Vector_container` and `Container` **subclass** and **superclass**, respectively. The derived class is said to inherit members from its base class, so the use of base and derived classes is commonly referred to as **inheritance**.

The members `operator[]()` and `size()` are said to **override** the corresponding members in the base class `Container`. I used the explicit `override` to make clear what's intended. The use of `override` is optional, but being explicit allows the compiler to catch mistakes, such as misspellings of function names or slight differences between the type of a `virtual` function and its intended overrider. The explicit use of `override` is particularly useful in larger class hiearchies where it can otherwise be hard to know what is supposed to override what.

The destructor (`~Vector_container()`) overrides the base class destructor (`~Container()`). Note that the member destructor (`~Vector()`) is implicitly invoked by its class's destructor (`~Vector_container()`).

For a function like `use(Container&)` to use a Container in complete ignorance of implementation details, some other function will have to make an object on which it can operate. For example:

```
void g() {
    Vector_container vc(10); // ... fill vc ...
```

```
    use(vc);
}
```

Since `use()` doesn't know about `Vector_container`s but only knows the `Container` interface, it will work just as well for a different implementation of a Container. For example:

```
class List_container : public Container {
        // List_container implements Container
        public:
        List_container() { } // empty List
        List_container(initializer_list<double> il) : ld{il} { }
        ~List_container() {}
        double& operator[](int i) override;
        int size() const override { return ld.size(); }
    private:
        std::list<double> ld; // (standard-library) list of doubles
};
double& List_container::operator[](int i) {
    for (auto& x : ld) {
        if (i==0)
            return x;
        --i;
    }
    throw out_of_range{"List container"};
}
```

A function can create a `List_container` and have `use()` use it:

```
void h() {
    List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    use(lc);
}
```

The point is that `use(Container&)` has no idea if its argument is a `Vector_container`, a `List_container`, or some other kind of container; it doesn't need to know. It can use any kind of `Container`. It knows only the interface defined by `Container`. Consequently, `use(Container&)` needn't be recompiled if the implementation of `List_container` changes or a brand-new class derived from Container is used.

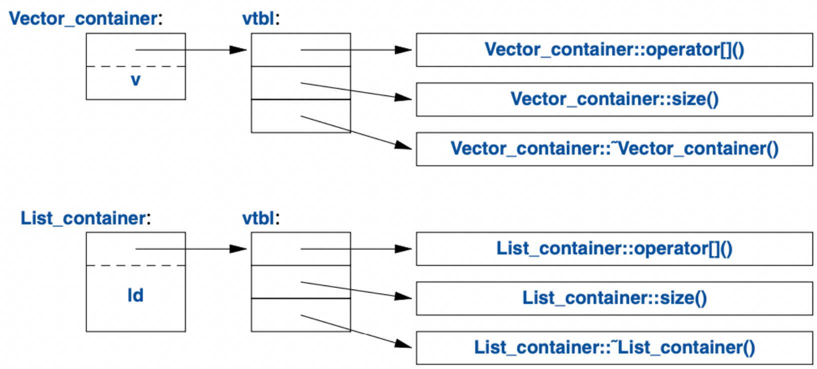
4.4 Virtual Functions

Consider again the use of `Container`:

```
void use(Container& c) {
    const int sz = c.size();
    for (int i=0; i!=sz; ++i) cout << c[i] << '\n';
}
```

How is the call `c[i]` in `use()` resolved to the right `operator[]()`?

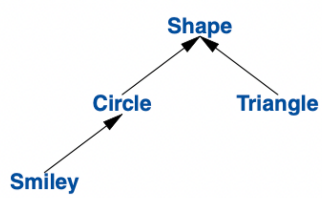
When `h()` calls `use()`, `List_container`'s `operator[]()` must be called. When `g()` calls `use()`, `Vector_container`'s `operator[]()` must be called. To achieve this resolution, a `Container` object **must** contain information to allow it to select the right function to call at run time. The usual implementation technique is for the compiler to convert the name of a virtual function into an index into a table of pointers to functions. That table is usually called the **virtual function table** or simply the `vtbl`. Each class with virtual functions has its own `vtbl` identifying its virtual functions



The implementation of the caller needs only to know the location of the pointer to the `vtbl` in a `Container` and the index used for each virtual function. This virtual call mechanism can be made almost as efficient as the "normal function call" mechanism (within 25%).

4.5 Class Hierarchies

A **class hierarchy** is a set of classes ordered in a lattice created by derivation (e.g., `:public`)



```
class Shape {
    public:
        virtual Point center() const =0;
        virtual void move(Point to) =0;
        virtual void draw() const = 0;
        virtual void rotate(int angle) = 0;
        virtual ~Shape() {}
        // ...
};

class Circle : public Shape {
    public:
        Circle(Point p, int rad);
        Point center() const override
        {
            return x;
        }
        void move(Point to) override
        {
            x = to;
        }
        void draw() const override;
        void rotate(int) override {}
    private:
        Point x; // center
        int r; // radius
};

class Smiley : public Circle {
        Smiley(Point p, int rad) : Circle{p,rad}, mouth{nullptr} { }
        ~Smiley()
        {
            delete mouth;
            for (auto p : eyes)
                delete p;
        }

        void move(Point to) override;

        void draw() const override;
        void rotate(int) override;

        void add_eye(Shape* s)
        {
            eyes.push_back(s);
        }
        void set_mouth(Shape* s);
        virtual void wink(int i);
```

```
    private:
        vector<Shape*> eyes; // usually two eyes
        Shape* mouth;
};
```

We can now define `Smiley::draw()` using calls to `Smiley`'s base and member `draw()`s:

```
void Smiley::draw() const {
    Circle::draw();
    for (auto p : eyes)
        p->draw(); mouth->draw();
}
```

4.5.1 Benefits from Hierarchies

- **Interface inheritance**: An object of a derived class can be used wherever an object of a base class is required. That is, the base class acts as an interface for the derived class. The `Container` and `Shape` classes are examples. Such classes are often abstract classes.

- **Implementation inheritance**: A base class provides functions or data that simplifies the implementation of derived classes. `Smiley`'s uses of `Circle`'s constructor and of `Circle::draw()` are examples. Such base classes often have data members and constructors.

Classes in class hierarchies are different: we tend to allocate them on the free store using `new`, and we access them through pointers or references. For example, consider a function that reads data describing shapes from an input stream and constructs the appropriate `Shape` objects:

```
enum class Kind { circle, triangle, smiley };
Shape* read_shape(istream& is) // read shape descriptions from input stream is
{
    // ... read shape header from is and find its Kind k ...
    switch (k) {
        case Kind::circle:
            // read circle data {Point,int} into p and r
            return new Circle{p,r};
        case Kind::triangle:
            // read triangle data {Point,Point,Point} into p1, p2, and p3
            return new Triangle{p1,p2,p3};
        case Kind::smiley:
            // read smiley data {Point,int,Shape,Shape,Shape} into p, r, e1, e2, and m
```

```
            Smiley* ps = new Smiley{p,r};
            ps->add_eye(e1);
            ps->add_eye(e2);
            ps->set_mouth(m);
            return ps;
    }
}

void user() {
    std::vector<Shape*> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v); // call draw() for each element
    rotate_all(v,45); // call rotate(45) for each element
    for (auto p : v) // remember to delete elements
        delete p;
}
```

`user()` has absolutely no idea of which kinds of shapes it manipulates.

Note that there are no pointers to the shapes outside `user()`, so `user()` is responsible for deallocating them. This is done with the `delete` operator and relies critically on `Shape`'s virtual destructor. Because that destructor is virtual, `delete` invokes the destructor for the most derived class. In this case, a `Smiley` deletes its `eyes` and `mouth` objects. Once it has done that, it calls `Circle`'s destructor. Objects are constructed "bottom up" (base first) by constructors and destroyed "top down" (derived first) by destructors

4.5.2 Hierarchy Navigation

The `read_shape()` function returns `Shape*` so that we can treat all `Shapes` alike. However, what can we do if we want to use a member function that is only provided by a particular derived class, such as `Smiley`'s `wink()`? We can ask "is this `Shape` a kind of `Smiley`?" using the `dynamic_cast` operator:

```
Shape* ps {read_shape(cin)};

if (Smiley* p = dynamic_cast<Smiley*>(ps)) { // ... does ps point to a Smiley? ...
    // ... a Smiley; use it
}
else {
    // ... not a Smiley, try something else ...
}
```

If at run time the object pointed to by the argument of `dynamic_cast` (here, `ps`) is not of the expected type (here, `Smiley`) or a class derived from the expected type, `dynamic_cast` returns `nullptr`

We use dynamic_cast to a pointer type when a pointer to an object of a different derived class is a valid argument. We then test whether the result is nullptr. This test can often conveniently be placed in the initialization of a variable in a condition.

When a different type is unacceptable, we can simply dynamic_cast to a reference type. If the object is not of the expected type, dynamic_cast throws a bad_cast exception:

```
Shape* ps {read_shape(cin)};
Smiley& r {dynamic_cast<Smiley&>(*ps)}; // somewhere, catch std::bad_cast
```

4.5.3 Avoiding Resource Leaks

- The implementer of Smiley may fail to delete the pointer to mouth.
- A user of read_shape() might fail to delete the pointer returned.
- The owner of a container of Shape pointers might fail to delete the objects pointed to.

In that sense, pointers to objects allocated on the free store is dangerous: a ''plain old pointer'' should not be used to represent ownership. For example:

```
void user(int x) {
    Shape* p = new Circle{Point{0,0},10};
    // ...
    if (x<0) throw Bad_x{};  // potential leak
    if (x==0) return;        // potential leak
    // ...
    delete p;
}
```

This will leak unless x is positive. Assigning the result of new to a "naked pointer" is asking for trouble.

One simple solution to such problems is to use a standard-library unique_ptr rather than a "naked pointer" when deletion is required:

```
class Smiley : public Circle {
    // ...
    private:
        vector<unique_ptr<Shape>> eyes; // usually two eyes
        unique_ptr<Shape> mouth;
};
```

As a pleasant side effect of this change, we no longer need to define a destructor for `Smiley`. The compiler will implicitly generate one that does the required destruction of the `unique_ptr`s in the vector. The code using `unique_ptr` will be exactly as efficient as code using the raw pointers correctly.

```
unique_ptr<Shape> read_shape(istream& is) // read shape descriptions from input stream is
{
    // read shape header from is and find its Kind k
    switch (k) {
        case Kind::circle:
            // read circle data {Point,int} into p and r
            return unique_ptr<Shape>{new Circle{p,r}};
    }
}
void user() {
    vector<unique_ptr<Shape>> v;
    while (cin)
        v.push_back(read_shape(cin));
    draw_all(v);
    rotate_all(v,45);
} // all Shapes implicitly destroyed
```

Now each object is owned by a `unique_ptr` that will `delete` the object when it is no longer needed, that is, when its `unique_ptr` goes out of scope.

4.6 Advice

1. avoid “naked” `new` and `delete`
2. use `override` to make overriding explicit in large class hierarchies
3. use `dynamic_cast` where class hierarchy navigation is unavoidable
4. use `dynamic_cast` to a reference type when failure to find the required class is considered a failure
5. use `dynamic_cast` to a pointer type when failure to find the required class is considered a valid alternative
6. use `unique_ptr` or `shared_ptr` to avoid forgetting to `delete` objects created using `new`

5 Essential Operations

5.1 Introduction

5.1.1 Essential Operations

Constructors, destructors, and copy and move operations for a type are not logically separate. We must define them as a matched set or suffer logical or performance problems. If a class `X` has a destructor that performs a nontrivial task, such as free-store deallocation or lock release, the class is likely to need the full complement of functions

```
class X { public:
  X(Sometype);              // "ordinar y constructor": create an object
  X();                      // default constructor
  X(const X&);              // copy constructor
  X(X&&);                   // move constructor
  X& operator=(const X&);   // copy assignment: clean up target and copy
  X& operator=(X&&);        // move assignment: clean up target and move
  ~X();                     // destructor: clean up
  // ...
};
```

There are five situations in which an object can be copied or moved

- as the source of an assignment
- as an object initializer
- as a function argument
- as a function return value
- as an exception

An assignment uses a copy or move assignment operator. In principle, the other cases use a copy or move constructor. Hence, a copy or move constructor invocation is often optimized away by constructing the object used initialize right in the target object. For example:

```
X make(Sometype
X x = make(value)
```

Here a compiler will typically construct the `X` from `make()` directly in `x`; thus eliminating a copy

In addition to the initialization of named objects and of objects on the free store, constructors are used to initialize temporary objects and to implement explicit type conversion.

Except for the "ordinary constructor", these special member functions will be generated by the compiler as needed. If you want to explicit about generating default implementations, you can:

```
class Y { public:
  Y(Sometype);
  Y(const Y&) = default; // I really do want the default copy constructor
  Y(Y&&) = default;      // and the default move constructor
  // ...
};
```

If you are explicit about some defaults, other default definitions will not be generated.

When a class has a pointer member, it is usually a good idea to be explicit about copy and move operations. The reason is that a pointer may point to something that the class needs to `delete`, in which case the default memberwise copy would be wrong. Alternatively, it might point to something that the class must *not* `delete`.

A good rule of thumb is to either define all of the essential operations or none (using the default for all). For example

```
struct Z {
  Vector v;
  string s;
};

Z z1;    // default initialize z1.v and z1.s Z
z2 = z1; // default copy z1.v and z1.s
```

To complement `=default`, we have `=delete` to indicate that an operation is not to be generated. A base class in a class hierarchy is the classical example where we don't want to allow a memberwise copy. For example:

```
class Shape {
 public:
  Shape(const Shape&) =delete;           // no copy operations
  Shape& operator=(const Shape&) =delete;
  // ...
};
void copy(Shape& s1, const Shape& s2) {
  s1 = s2; // error : Shape copy is deleted
}
```

A `=delete` makes an attempted use of the deleted function a compile-time error; `=delete` can be used to suppress any function, not just essential member functions.

Using the default copy or move for a class in a hierarchy is typically a disaster: given only a pointer to a base, we simply don't know what members the derived class has, so we can't know how to copy them. So, the best thing to do is usually to delete the default copy and move operations, that is, to eliminate the default definitions of those two operations:

(The C++ Programming Language - Bjarne Stroustrup)

5.1.2 Conversions

A constructor taking a single argument defines a conversion from its argument type. For example, `complex` (4.2.1) provides a constructor from a `double`

```
complex z1 = 3.14; // z1 becomes {3.14,0.0}
complex z2 = z1*2; // z2 becomes z1*{2.0,0} == {6.28,0.0}
```

This implicit conversion is sometimes ideal, but not always. For example, `Vector` (2.3) provides a constructor from an `int`:

```
Vector v1 = 7; // OK: v1 has 7 elements
```

This is typically considered unfortunate, and the standard-library `vector` does not allow this `int`-to-`vector` conversion

The way to avoid this problem is to say that only explicit "conversion" is allowed; that is, we can define the constructor like this:

```
class Vector { public:
  explicit Vector(int s); // no implicit conversion from int to Vector
  // ...
};
```

5.1.3 Member Initializers

When a data member of a class is defined, we can supply a default initializer called a **default member initializer**. Consider a revision of complex (4.2.1):

```
class complex {
  double re = 0;
  double im = 0; // representation: two doubles with default value 0.0 public:
  complex(double r, double i) :re{r}, im{i} {} // construct complex from two scalars: {r,i}
  complex(double r) :re{r} {}                  // construct complex from one scalar: {r,0}
  complex() {}                                 // default complex: {0,0}
  // ...
}
```

5.2 Copy and Move

By default, objects can be copied. This is true for objects of user-defined types as well as for built-in types. The default meaning of copy is memberwise copy: copy each member.

When we design a class, we must always consider if and how an object might be copied. For simple concrete types, memberwise copy is often exactly the right semantics for copy. For some sophisticated concrete types, such as `Vector`, memberwise copy is not the right semantics for copy; for abstract types it almost never is.

5.2.1 Copying Containers

When a class is a **resource handle** – that is, when the class is responsible for an object accessed through a pointer – the default memberwise copy is typically a disaster. Memberwise copy would violate the resource handle's invariant. For example, the default copy would leave a copy of a `Vector` referring to the same elements as the original:

```
void bad_copy(Vector v1) {
  Vector v2 = v1;    // copy v1's representation into v2
  v1[0] = 2;         // v2[0] is now also 2!
  v2[1] = 3;         // v1[1] is now also 3!
}
```

Copying of an object of a class is defined by two members: a **copy constructor** and a **copy assignment**:

```
class Vector { private:
  double* elem; // elem points to an array of sz doubles
  int sz;
 public:
  Vector(int s);                    // constructor: establish invariant, acquire resources
  ~Vector() { delete[] elem; }      // destructor: release resources

  Vector(const Vector& a);          // copy constructor
  Vector& operator=(const Vector& a);// copy assignment
  double& operator[](int i);
  const double& operator[](int i) const;
  int size() const;
};
```

A suitable definition of a copy constructor for Vector allocates the space for the required number of elements and then copies the elements into it so that after a copy each `Vector` has its own copy of the elements:

```
Vector::Vector(const Vector& a) // copy constructor
    :elem{new double[a.sz]},    // allocate space for elements
     sz{a.sz}
{
  for (int i=0; i!=sz; ++i) // copy elements
    elem[i] = a.elem[i];
}

Vector& Vector::operator=(const Vector& a) { // copy assignment
  double* p = new double[a.sz];
  for (int i=0; i!=a.sz; ++i)
    p[i] = a.elem[i];
  delete[] elem;        // delete old elements
  elem = p;
  sz = a.sz;
  return *this;
}
```

5.2.2 Moving Containers

We can control copying by defining a copy constructor and a copy assignment, but copying can be costly for large containers. We avoid the cost of copying when we pass objects to a function by using references, but we can't return a reference to a local object as the result (the local object would be destroyed by the time the caller got a chance to look at it). Consider:

```
Vector operator+(const Vector& a, const Vector& b) {
  if (a.size()!=b.size())
    throw Vector_size_mismatch{};

  Vector res(a.size());
  for (int i=0; i!=a.size(); ++i)
    res[i]=a[i]+b[i];
  return res;
}
```

Returning from a `+` involves copying the result out of the local variable `res` and into some place where the caller can access it. We might use this `+` like this

```
void f(const Vector& x, const Vector& y, const Vector& z)
{
  Vector r; // ...
  r = x+y+z; // ...
}
```

That would be copying a `Vector` at least twice (one for each use of the `+` operator).

We want to **move** a `Vector` rather than *copy* it.

```
class Vector { // ...
  Vector(const Vector& a);              // copy constructor
  Vector& operator=(const Vector& a); // copy assignment
  Vector(Vector&& a);                   // move constructor
  Vector& operator=(Vector&& a);       // move assignment
};
```

Given that definition, the compiler will choose the *move constructor* to implement the transfer of the return value out of the function. This means that `r=x+y+z` will involve no copying of `Vector`s. Instead, `Vector`s are just moved.

```
Vector::Vector(Vector&& a)
    :elem{a.elem},      // "grab the elements" from a
     sz{a.sz}
{
  a.elem = nullptr;     // now a has no elements
  a.sz = 0;
}
```

The `&&` means "rvalue reference" and is a reference to which we can bind an rvalue. The word "rvalue" is intended to complement "lvalue" which roughly means "something that can appear on the left-hand side of an assignment". So an rvalue is - to a first approximation - a value that you can't assign to, such as an integer returned by a function call. Thus, an rvalue reference is a reference to something that **nobody else** can assign to, so we can safely "steal" its value.

A move constructor does *not* take a `const` argument. A **move assignment** is defined similarly.

A move operation is applied when an rvalue reference is used as an initializer or as the right- hand side of an assignment.

After a move, a moved-from object should be in a state that allows a destructor to be run. Typically, we also allow assignment to a moved-from object. The standard-library algorithms (Chapter 12) assumes that. Our `Vector` does that.

Where the programmer knows that a value will not be used again, but the compiler can't be expected to be smart enough to figure that out, the programmer can be specific:

```
Vector f() {
  Vector x(1000);
  Vector y(2000);
  Vector z(3000);
```

```
  z = x;              // we get a copy (x might be used later in f())
  y = std::move(x);  // we get a move (move assignment)
  // ... better not use x here ...
  return z; // we get a move
}
```

The standard-library function `move()` doesn't actually move anything. Instead, it returns a reference to its argument from which we may move - an **rvalue reference**

5.3 Resource Management

By defining constructors, copy operations, move operations, and a destructor, a programmer can provide complete control of the lifetime of a contained resource

Consider a standard-library `thread` representing a concurrent activity and a `Vector` of a million doubles. We can't copy the former and don't want to copy the latter.

```
std::vector<thread> my_threads;

Vector init(int n)
{
  thread t {heartbeat};              // run heartbeat concurrently (in a separate thread)
  my_threads.push_back(std::move(t)); //move t into my_threads
  // ... more initialization ...

  Vector vec(n);
  for (int i=0; i!=vec.size(); ++i)
    vec[i] = 777;
  return vec;                        // move vec out of init()
}
auto v = init(1'000'000);            // start heartbeat and initialize v
```

In very much the same way that `new` and `delete` disappear from application code, we can make pointers disappear into resource handles. In both cases, the result is simpler and more maintainable code, without added overhead. In particular, we can achieve **strong resource safety**; that is, we can eliminate resource leaks for a general notion of a resource. Examples are `vector`s holding memory, `thread`s holding system threads, and `fstream`s holding file handles.

Before resorting to garbage collection, systematically use resource handles: let each resource have an owner in some scope and by default be released at the end of its owners scope.

5.4 Conventional Operations

5.4.1 Comparisons

5.5 Advice

1. By default, declare single-argument constructors `explicit`
2. If a class member has a reasonable default value, provide it as a data member initializer
3. Far large operands, use `const` reference argument types

6 Template

6.1 Parameterized Types

We can generalize our vector-of-doubles type to a vector-of-anything type by making it a `template` and replacing the specific type `double` with a type parameter. For example

```
template<typename T>
class Vector {
 private:
  T* elem;  // elem points to an array of sz elements of type T
  int sz;
 public:
  explicit Vector(int s);           // constructor: establish invariant, acquire resources
  ~Vector() { delete[] elem; }      // destructor: release resources
  // ... copy and move operations ...
  T& operator[](int i);             // for non-const Vectors
  const T& operator[](int i) const; // for const Vectors
  int size() const { return sz; }
};
```

The `template<typename T>` prefix makes `T` a parameter of the declaration it prefixes. It is C++'s version of the mathematical "for all T" or more precisely "for all types T."

The member functions might be defined similarly

```
template<typename T>
Vector<T>::Vector(int s)
{
  if (s<0)
    throw Negative_size{};
  elem = new T[s];
  sz = s;
```

```
}
template<typename T>
const T& Vector<T>::operator[](int i) const {
  if (i<0 || size()<=i)
    throw out_of_range{"Vector::operator[]"};
  return elem[i];
}
```

Given these definitions, we can define `Vector`s like this:

```
Vector<char> vc(200);      // vector of 200 characters
Vector<string> vs(17);     // vector of 17 strings
Vector<list<int>> vli(45); // vector of 45 lists of integers
```

We can use `Vector`s like this

```
void write(const Vector<string>& vs) { // Vector of some strings
  for (int i = 0; i!=vs.size(); ++i)
    cout << vs[i] << '\n';
}
```

To support the range-`for` loop for our `Vector`, we must define suitable `begin()` and `end()` functions:

```
template<typename T>
T* begin(Vector<T>& x)
{
  return x.size() ? &x[0] : nullptr;   // pointer to first element or nullptr
}

template<typename T>
T* end(Vector<T>& x) {
  return x.size() ? &x[0]+x.size() : nullptr; // pointer to one-past-last element
}
```

Given those, we can write

```
void f2(Vector<string>& vs)    // Vector of some strings
{
  for (auto& s : vs)
    cout << s << '\n';
}
```

Templates are a compile-time mechanism, so their use incurs no run-time overhead compared to hand-crafted code.

A template plus a set of template arguments is called an **instantiation** or a **specialization**. Late in the compilation process, at **instantiation time**, code is generated for each instantiation used in a program

6.1.1 Contrained Template Arguments (C++20)

Most often, a template will make sense only for template arguments that meet certain criteria. For example, a `Vector` typically offers a copy operation, and if it does, it must require that its elements must be copyable. That is, we must require that `Vector`'s template argument is not just a `typename` but an `Element` where "`Element`" specifies the requirements of a type that can be an element:

```
template<Element T>
class Vector {
 private:
  T* elem; // elem points to an array of sz elements of type T
  int sz;
  // ...
};
```

This `template<Element T>` prefix is C++'s version of mathematic's "for all T such that `Element(T)`"; that is, `Element` is a predicate that checks whether T has all the properties that a `Vector` requires. Such a predicate is called a **concept**. A template argument for which a concept is specified is called a **constrained argument** and a template for which an argument is constrained is called a **constrained template**.

6.1.2 Value Template Arguments

In addition to type arguments, a template can take value arguments. For example

```
template<typename T, int N>
struct Buffer
{
  using value_type = T;
  constexpr int size() { return N; }
  T[N];
  // ...
};
```

The alias (`value_type`) and the `constexpr` function are provided to allow users access to the template arguments.

Value arguments are useful in many contexts. For example, `Buffer` allows us to create arbitrarily sized buffers with no use of the free store (dynamic memory):

6.1.3 Template Argument Deduction

Consider using the standard-library template `pair`

```
pair<int,double> p = {1,5.2};
```

Many have found the need to specify the template argument types tedious, so the standard library offers a function, `make_pair()`, that deduces the template arguments of the `pair` it returns from its function arguments:

```
auto p = make_pair(1,5.2); // p is a pair<int,double>
```

This leads to the obvious question "Why can't we just deduce template parameters from constructor arguments?" So, in C++17, we can. That is:

```
pair p = {1,5.2}; // p is a pair<int,double>
```

Consider a simple example:

```
template<typename T>
class Vector {
 public:
  Vector(int);
  Vector(initializer_list<T>); // initializer-list constructor
  // ...

};

Vector v1 {1,2,3}; // deduce v1's element type from the initializer element type
Vector v2 = v1; // deduce v2's element type from v1's element type

auto p = new Vector{1,2,3}; // p points to a Vector<int>

Vector<int> v3(1); // here we need to be explicit about the
                   // element type (no element type is mentioned)
```

Clearly, this simplifies notation and can eliminate annoyances caused by mistyping redundant template argument types. However, deduction can cause surprises

```
Vector<string> vs1 {"Hello", "World"}; // Vector<string>
Vector vs {"Hello", "World"};          // deduces to Vector<const char*> (Surprise?)
Vector vs2 {"Hello"s, "World"s};       // deduces to Vector<string>
Vector vs3 {"Hello"s, "World"};        // error: the initializer list is not homogenous
```

The type of a C-style string literal is `const char*`. If that was not what was intended, use the `s` suffix to make it a proper `string`.

When a template argument cannot be deduced from the constructor arguments, we can help by providing a **deduction guide**. Consider

```
template<typename T>
class Vector2 {
 public:
  using value_type = T;
  // ...
  Vector2(initializer_list<T>);   // initializer-list constructor

  template<typename Iter>
  Vector2(Iter b, Iter e);        // [b:e) range constructor
  // ...
};

Vector2 v1 {1,2,3,4,5};           // element type is int
Vector2 v2(v1.begin(),v1.begin()+2);
```

Obviously, v2 should be a `Vector2<int>`, but without help, the compiler cannot deduce that. The code only states that there is a constructor from a pair of values of the same type. Without language support for concepts, the compiler cannot assume anything about the types. To allow deduction, we can add a **deduction guide** after the declaration of `Vector2`:

```
emplate<typename Iter>
Vector2(Iter,Iter) -> Vector2<typename Iter::value_type>;
```

6.2 Parameterized Operations

6.2.1 Function Templates

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v)
{
  for (auto x : s)
    v+=x;
  return v;
}
```

6.2.2 Function Objects

One particularly useful kind of template is the **function object** (sometimes called a **functor**), which is used to define objects that can be called like functions.

```
template<typename T>
class Less_than {
  const T val; // value to compare against
 public:
  Less_than(const T& v) :val{v} { }
```

```
  bool operator()(const T& x) const { return x<val; } // call operator
};
```

The function called `operator()` implements the "function call", "call" or "application" operator `()`.

We can define named variables of type `Less_than` for some argument type:

```
Less_than lti {42};              // lti(i) will compare i to 42 using < (i<42)
Less_than lts {"Backus"s};       // lts(s) will compare s to "Backus" using < (s<"Backus")
Less_than<string> lts2 {"Naur"}; // "Naur" is a C-style string, so we need <string>
                                 // to get the right <
```

Such function objects are widely used as arguments to algorithms. For example, we can count the occurrences of values for which a predicate returns `true`:

```
template<typename C, typename P>
// requires Sequence<C> && Callable<P,Value_type<P>>
int count(const C& c, P pred) {
  int cnt = 0;
  for (const auto& x : c)
    if (pred(x))
      ++cnt;
  return cnt;
}
```

A **predicate** is something that we can invoke to return `true` or `false`. For example:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s)
{
  cout << "number of values less than " << x << ": " << count(vec,Less_than{x}) << '\n';
  cout << "number of values less than " << s << ": " << count(lst,Less_than{s}) << '\n';
}
```

The beauty of these function objects is that they carry the value to be compared against with them.

1. We don't have to write a separate function for each value (and each type),
2. we don't have to introduce nasty global variables to hold values.
3. for a simple function object like `Less_than`, inlining is simple, so a call of `Less_than` is far more efficient than an indirect function call. The ability to carry data plus their efficiency makes function objects particularly useful as arguments to algorithms.

Function objects used to specify the meaning of key operations of a general algorithm are often referred to as **policy objects**.

6.2.3 Lambda Expression

The notation `[&](int a){ return a<x; }` is called a lambda expression. It generates a function object exactly like `Less_than<int>{x}`. The `[&]` is a **capture list** specifying that all local names used in the lambda body (such as `x`) will be accessed through references.

Had we wanted to "capture" only `x`, we could have said so: `[&x]`. Had we wanted to give the generated object a copy of `x`, we could have said so: `[=x]`. Capturing nothing is `[]`, capture all local names by reference is `[&]`, and capture all local nmaes used by value is `[=]`.

Like a function, a lambda can be generic. For example:

```
template<class S>
void rotate_and_draw(vector<S>& v, int r) {
  for_all(v,[](auto& s){ s->rotate(r); s->draw(); });
}
```

Here, like in variable declarations, `auto` means that any type is accepted as an initializer (an argument is considered to initialize the formal parameter in a call). This makes a lambda with an `auto` parameter a template, a `generic lambda`. For reasons lost in standards committee politics, this use of `auto` is not currently allowed for function arguments.

6.3 Template Mechanisms

6.3.1 Variable Templates

When we use a type, we often want constants and values of that type. This is of course also the case when we use a class template: when we define a `C<T>`, we often want constants and variables of type `T` and other types depending on `T`.

```
template <class T>
constexpr T viscosity = 0.4;

template <class T>
constexpr space_vector<T> external_acceleration = { T{}, T{-9.8}, T{} };

auto vis2 = 2*viscosity<double>;
auto acc = external_acceleration<float>;
```

Here space_vector is a three-dimensional vector.

Naturally, we can use arbitrary expressions of suitable type as initializers. Consider:

```
template<typename T, typename T2>
constexpr bool Assignable = is_assignable<T&,T2>::value;
// is_assignable is a type trait

template<typename T> void testing()
{
  static_assert(Assignable<T&,double>, "can't assign a double");
  static_assert(Assignable<T&,string>, "can't assign a string");
}
```

6.3.2 Alias

It is very common for a parameterized type to provide an alias for types related to their template arguments. For example:

```
template<typename T> class Vector {
 public:
  using value_type = T;
  // ...
};
```

In fact, every standard-library container provides value_type as the name of its value type. This allows us to write code that will work for every container that follows this convention.

```
template<typename C>
using Value_type = typename C::value_type; // the type of C's elements
template<typename Container>
void algo(Container& c)
{
  Vector<Value_type<Container>> vec; // keep results here
  // ...
}
```

The aliasing mechanism can be used to define a new template by binding some or all template arguments. For example:

```
template<typename Key, typename Value>
class Map {
  // ...
};
template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m; // m is a Map<string,int>
```

6.3.3 Compile-Time `if`

Consider writing an operation that can use one of two operations `slow_and_safe(T)` or `simple_and_fast(T)`. The traditional solution is to write a pair of overloaded functions and select the most appropriate based on a trait (**??**), such as the standard-library `is_pod`. If a class hierarchy is involved, a base class can provide the `slow_and_safe` general operation and a derived class can override with a `simple_and_fast` implementation.

In C++17, we can use a compile-time `if`:

```
template<typename T>
void update(T& target) {
  // ...
  if constexpr(is_pod<T>::value)
      simple_and_fast(target); // for "plain old data"
  else
    slow_and_safe(target);
  // ...
}
```

The `is_pod<T>` is a type trait that tells us whether a type can be trivially copied

Only the selected branch of an `if constexpr` is instantiated.

Importantly, an `if constexpr` is not a text-manipulation mechanism and cannot be used to break the usual rules of grammar, type and scope.

6.4 Advice

1. There is no separate compilation of templates: `#include` template definitions in every translation unit that uses them

7 Concepts and Generic Programming

7.1 Concepts (C++20)

Consider the `sum()`

```
template<typename Seq, typename Num>
Num sum(Seq s, Num v)
{
  for (const auto& x : s) v+=x;
  return v;
}
```

`sum()` requires that its first template argument is some kind of sequence and its second template argument is some kind of number. We call such requirements **concepts**.

7.1.1 Use of Concepts

Consider the `sum()` again.

```
template<Sequence Seq, Number Num>
Num sum(Seq s, Num v)
{
  for (const auto& x : s)
    v+=x;
  return v;
}
```

Once we have defined what the concepts `Sequence` and `Number` mean, the compiler can reject bad calls by looking at `sum()`'s interface only, rather than looking at its implementation. This improves error reporting.

However, the specification of `sum()`'s interface is not complete: we should be able to add elements of a `Sequence` to a `Number`. We can do that

```
template<Sequence Seq, Number Num>
requires Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n);
```

The `Value_type` of a sequence is the type of the elements in the sequence. `Arithmetic<X,Y>` is a concept specifying that we can do arithmetic with numbers of types `X` and `Y`. This saves us from accidentally trying to calculate the `sum()` of a `vector<string>` or a `vector<int*>` while still accepting `vector<int>` and `vector<complex<double>>`.

Unsurprisingly, `requires Arithmetic<Value_type<Seq>,Num>` is called a `requirements`-clause. The `template<Sequence Seq>` notation is simply a shorthand for an explicit use of `requires Sequence<Seq>`. If I liked verbosity, I could equivalently have written

```
template<typename Seq, typename Num>
requires Sequence<Seq> && Number<Num> && Arithmetic<Value_type<Seq>,Num>
Num sum(Seq s, Num n);
```

On the other hand, we could also use the equivalence between the two notations to write:

```
template<Sequence Seq, Arithmetic<Value_type<Seq>> Num>
Num sum(Seq s, Num n);
```

Where we cannot yet use concepts, we have to make do with naming conventions and comments.

```
template<typename Sequence, typename Number>
// requires Arithmetic<Value_type<Sequence>,Number>
Numer sum(Sequence s, Number n);
```

7.1.2 Concept-based Overloading

Once we have properly specified templates with their interfaces, we can overload based on their properties, much as we do for functions. Consider a slightly simplified standard-library function advance() that advances an iterator

8 Library Overview

9 Utilities

9.1 Resource Management

9.1.1 `unique_ptr` and `shared_ptr`

The basic use of these "smart pointers" is to prevent memory leaks caused by careless programming

The `shared_ptr` is similar to `unique_ptr` except that `shared_ptr` s are copied rather than moved.

10 Concurrency

10.1 Waiting for Events

Consider the classical example of two `thread`s communicating by passing messages through a `queue`.

```
class Message {

};

queue<Message> mqueue;
condition_variable mcond;
mutex mmutex;

void consumer() {
```

```
  while (true) {
    unique_lock lck{mmutex};  //acquire mmutex
    mcond.wait(lck, []{return !mqueue.empty();});
    // release lck and wait
    // re-acquire lck upon wakeup
    // don't wake up unless mqueue is non-empty
    auto m = mqueue.front();
    mqueue.pop();
    lck.unlock();
  }
}
```

I used a `unique_lock` rather than a `scoped_lock` for two reasons:

- we need to pass the lock to the `condition_variable`'s `wait()`. A `scoped_lock` cannot be copied, but a `unique_lock` can be
- we want to unlock the `mutex` protecting the condition variable before processing the message. A `unique_lock` offers operations, such as `lock()` and `unlock()`, for low-level control of synchronization.

On the other hand, `unique_lock` can only handle a single `mutex`

```
void producer() {
  while (true) {
    Message m;

    scoped_lock lck{mmutex};
    mqueue.push(m);
    mcond.notify_one();
  }
}
```

11 Problems

ref	status
6.3.1	