# A Tour Of C++

Bjarne Stroustrup

May 9, 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
```

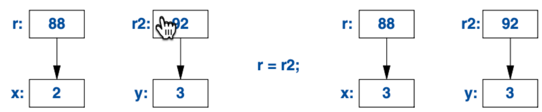
## 1.6 Tests

## 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_containers 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';
}
```