Chapter 4. The Standard Template Library

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at *learnmodcppfinance@gmail.com*.

Introduction

What many informally call the Standard Template Library, commonly referred to as just "the STL", is a subset of C++'s Standard Library that houses a set of *container* classes, including `std::vector`, that will be discussed in this chapter. It also provides a group of algorithms applicable to those containers and other containers—including your own—that follow the same coding conventions.

"The STL", a name that will also be used in this chapter, is in fact a revolutionary design brought to the world by Alexander Stepanov, David Musser, and Meng Lee. It was officially integrated into the C++ Standard library in the 1990's **{1}**. This "library-within-a-library" combines algorithms and containers to form a whole that is significantly larger than the sum of its parts. Technically speaking, as Scott Meyers points out, "[t]here is no official definition of *the STL*, and different people mean different things when they use the term." **{2}**. However, it is used throughout "the literature" as well as in the C++ vernacular to represent the container classes, iterators, and algorithms to be presented in the current chapter, so we will also use it here.

You have already seen that a `vector` on its own is quite useful and versatile. It is in fact the workhorse of the STL and the container of choice for most financial modeling applications (as well as for many other application domains). For this reason alone, it is worth getting familiar with the `vector` class in more detail, but in doing so, you will also find it easier to then understand how other STL containers work.

What makes STL containers so powerful is their relationship with STL *algorithms*. Algorithms work on an abstraction called *iterators*, rather than on containers themselves. Containers define how data is organized, iterators describe how organized data can be traversed, and algorithms describe what you can do on the data. At a high level, algorithms let you traverse an STL container, and they can apply a function to each member, or a subset thereof, efficiently replacing iterative `for` and `while` statements in a single statement. Since algorithms are expressed in terms of iterators, not in terms of containers, you can write a single algorithm and leverage its impact on a whole set of containers at once.

STL containers such as a `vector` are generic in the sense that they can hold a homogeneous set of (almost) any arbitrary type—indicated as a template parameter—be it a plain numerical type such as `double`, or a class type such as a `VanillaOption`, as long as these types conform to the requirements of the functions we seek to apply to them. Such containers can also handle polymorphic objects as pointers to a common interface base class, in which case there is a convergence of three programming paradigms: functional -- with STL algorithms, generic -- using STL container class templates, and object-oriented, at least under some definitions of that term -- where the elements are pointers to polymorphic objects. In

modern C++, as we saw in Chapter Three, unique pointers provide a safe yet efficient alternative to the raw pointers of yore, and these can be held as elements in an STL container.

In this chapter, we will first work through an overview/review of templates, as these are the "T" in "STL". After this, STL containers, iterators, and algorithms will follow, demonstrating how iterative tasks can be expressed more efficiently and safely than loops. This will also include some new features in C++20 Standard that make them more intuitive to work with, particularly with *ranges* and *views*, to be introduced toward the end of this chapter. Most of the material presented will focus primarily on C++ iteself as opposed to applications in finance, but it will set us up for more applied examples in the remaining chapters.

Templates

Templates in C++ are like blueprints for functions and classes. They facilitate generic programming in C++, where a function or class can be designed for arbitrary types, as opposed to individual versions written for each type that can end up duplicating code to a large degree. Examples of a generic function and of a generic class will be presented here. Working through these, you should then be able to develop an intuition as to how templates work in C++.

To start, let us go back to the case of a `vector`, which is a common example of a class template. Like other containers, it can hold a homogeneous collection of an arbitrary type by specifying the type as a template parameter inside the angle brackets.

```
// A vector of real numbers:
vector<double> v{1.0, 2.0, 3.0};

// A vector of BlackScholes objects derived
// from their VanillaOption abstract interface:
```

```
vector<BlackScholes> opts
{
    { 100.0, 106.0, 0.04, 0.5, PayoffType::Call },
    { 110.0, 106.0, 0.04, 0.5, PayoffType::Put },
    { 95.0, 106.0, 0.032, 0.25, PayoffType::Call },
    { 105.0, 106.0, 0.032, 0.02, PayoffType::Put },
    { 115.0, 106.0, 0.045, 1.5, PayoffType::Put }
};
```

The `vector` class is generic in the sense that it doesn't care what type of object it is holding.

Going a step further, to better understand how templates work, we can next look at how to implement user-defined function and class templates.

Function Templates

Recalling the `Fraction` class from Chapter Two, we can add a definition of the `*` operator, and accessors for the numerator and denominator, in order to help demonstrate function templates. The previous inclusions of the `<` and spaceship operators will also be used shortly in an example involving class templates. The updated header file could be written as follows:

```
class Fraction
{
public:
    Fraction(unsigned n, unsigned d);
    bool operator == (const Fraction& rhs) const = default;
    std::strong_ordering operator <=> (const Fraction& rhs) const;

    // operator * has been added for current chapter:
    Fraction operator *(const Fraction& rhs) const;
```

```
    unsigned n() const;             // return n_ (numerator)
    unsigned d() const;             // return d_ (denominator)

private:
    unsigned n_, d_;

};
```

The two accessors returning the numerator and denominator are trivial, and for the `*` operator, for this demonstration we can just return the raw fraction product without reduction to simplest form:

```
Fraction Fraction::operator *(const Fraction& rhs) const
{
    return { n_ * rhs.n_, d_ * rhs.d_ };
}
```

Suppose now you need to square integer, real, and `Fraction` types. Proceeding naively, as C++ is a strongly typed language, you could write three separate functions:

```
// integers:
int int_square(int k)
{
  return k * k;
}

// real numbers (double):
double dbl_square(double x)
{
  return x * x;
}
```

```
// Fraction:
Fraction frac_square(const Fraction& f)
{
   return f * f;       // operator* on Fraction is defined
}
```

However, all three can be replaced by a single function template:

```
template <typename T>
T tmpl_square(const T& t)
{
   return t * t;
}
```

Squaring an integer or double precision value is trivial, as the numerical arguments imply their types, `int` and `double`, respectively:

```
int sq_int = tmpl_square(4);                // tmpl_square(const int&)
double sq_real = v(4.2);                    // tmpl_square(const double&)
```

Applying the function to a `Fraction` is more interesting, no longer having the built-in arithmetic operators as with numerical types. The object type needs to be specified somehow, and this can be done in a couple of different ways. One is to construct a `Fraction` object and pass it into the function:

```
Fraction f{ 2, 3 };
Fraction sq_frac = tmpl_square(f);          // tmpl_square(const Fraction&)
```

The second option is to use uniform initialization, where the constructor arguments alone suffice, but here we need to specify the template parameter; otherwise, the `tmpl_square(.)` function will have no way of knowing the type being passed:

```
auto sq_frac_unif = tmpl_square<Fraction>({ 2, 3 });
```

In each case, $\frac{4}{9}$ will be returned.

The function template does not care what the specific type `T` is, as long as the multiplication operator is defined. If it is not, such as in attempting to square a `Circle` object (Ch 3), the code would not compile:

```
// operator* not defined for Circle.
// Will not compile.
Circle circ{ 1.0 };
double area = tmpl_square(circ);
```

NOTE

Although it is generally a good thing when errors are detected by the compiler, with templated code, compiler error output can get long and somewhat cryptic, making it more painful to locate the source of the problem in the code, particularly in more realistic settings with expanded code complexity. Fortunately, there are new ways of alleviating this, particularly C++20 concepts, which will be presented in Chapter Nine.

Class Templates

A class template is syntactically similar to a function template except that a template parameter applies to the entire class. For example, here is a class template that will hold two private members of the same

type, and provide a public member function that computes the minimum of the two:

```
// Class Declaration:
template <typename T>
export class MyPair
{
public:
   MyPair(const T &first, const T &second) :a_(first), b_(second) {}
   T get_min() const;

private:
   T a_, b_;
};

// Class implementation:
template <typename T>
MyPair<T>::MyPair(T first, T second) :a_(first), b_(second) {}

template <typename T>
T MyPair<T>::get_min() const
{
   return a_ < b_ ? a_ : b_;
}
```

As long as the less-than operator `<` is defined for type `T`, then the code will compile. For two `int` types, the inequality definition is provided by the language, so no problem here:

```
MyPair<int> mp_int{ 10, 6 };
int min_int = mp_int.get_min();              // OK, returns 6
```

Using the simple `Fraction` class defined in Chapter Two, the inequality was defined within the class so we are OK here as well:

```
MyPair<Fraction> mp_frac{ {3, 2}, {5, 11} };
Fraction min_frac = mp_frac.get_min();          //  Returns 5/11
```

Attempting to call the `get_min()` member function where the template parameter type does not support the inequality, such as again with the `Circle` class, the result will be a compile-time error, eg:

```
// Will not compile:
MyPair<Circle> fail{ {1.5}, {3.0} };
fail.get_min(); // no operator< for Circle
```

NOTE

Note that we could have expressed the body of `MyPair<T>::get_min()` as

```
T retval;                       // (1)
retval = a < b? a : b;     // (2)
return retval;
```

However, this would have required that T expressed a copy assignment operator (comment (2)) as well as a default constructor (comment (1)), whereas our suggested implementation does not impose these requirements. When writing generic code, it is often useful to consider what we ask of the types for which our code will be instantiated and strive to ask only for what we really require, nothing more. This widens the set of types for which our code will be applicable.

Compiling Template Code

Templates are mostly written in header files. This was due to templates being just that: templates, guiding the compiler as to how code should be generated, not code that can be compiled on its own. A way to think about templates is like a pattern for a suit. Nothing will happen until the tailor selects the material to be used (analogous to the type), and then cuts it and stitches it together. A tailor could even say "in general, this is how I do things, but for you I have something special in mind", which would, for templates, be what we call a *specialization*.

For each type that a function or class template uses in its template parameter, a specialization is generated by the compiler, resulting in additional binary code for each. Suppose we write the same three lines of code as in the earlier example:

```
int sq_int = tmpl_square(4);
double sq_real = tmpl_square(4.2);
auto sq_frac_unif = tmpl_square<Fraction>({ 2, 3 });
```

This means putting these three lines in your code will cause the compiler to generate three different versions (specializations) of `tmpl_square(.)`, one for type `int`, for type `double` and one for `Fraction`.

Similarly, creating instances of `MyPair` objects as before will result in compiled code for each of the `int`, `double`, and `Fraction` specializations:

```
MyPair<int> mp_int{ 10, 6 };
MyPair<double> mp_dbl{4.22, 55.1115};
MyPair<Fraction> mp_frac{ {3, 2}, {5, 10} };
```

Of course, had we written these respective functions and classes individually with parameter type overloading, we would have arrived at the same solution. Templates save us the trouble of writing that code manually.

Additional benefits of templates include facilitating generic programming, as well as potentially improving run-time performance. The downside is this can increase build times quite significantly with template-heavy code bases, although this situation has improved since C++11. The introduction of modules in C++20 in some respects has also helped reduce build times, to be discussed in Chapter Nine.

There are also ways to use default template parameters in order to make user code simpler to express. Returning to the `MyPair` class that holds two elements of the same type `T`, we could default this template parameter to a `double` type:

```
template <typename T = double>
class MyPair
{
public:
	MyPair(T first, T second);
	T get_min() const;

private:
	T a_, b_;
};
```

Then, if we want a `MyPair` object to hold two `double` types, we can omit the explicit type name from inside the angle brackets:

```
MyPair<> real_pair{ 19.73, 10.26 };
double min_val = real_pair.get_min();		// 19.73
```

Class templates can involve more than one type. For example, the `MyPair` class template could have taken in two template parameters not necessarily of the same type:

```
template <typename T, typename U>
class MyPair
{
public:
	MyPair(T first, U second);
	// . . .

private:
	T a_;
	U b_;
};
```

Another example is the `std::map<K, V>` class template, an associative STL container to be covered shortly. Templates can also involve compile-time integral values, such as with the fixed-size sequential STL container `std::array<T, N>`, where `N` is a positive integer, which will also be discussed. Templates can even accept templates as parameters—which eventually can become challenging—as will be seen with expression templates introduced in Chapter Seven. However, we will not be able cover all possible permutations in this book, and thus we will mostly limit ourselves to providing a surface view of how templates can be written and how they can be used.

STL Containers and Iterators

We have already used `std::vector` in previous chapters, but other container classes exist, each of them allowing us to hold and organize data in different ways. Containers can be divided into at least two different categories:

- Sequential Containers, which emphasize sequential traversal of the elements. These containers include `std::vector`, along with `std::deque`, and `std::list`, each of which will be covered in this chapter.
- Associative Containers, such as `std::set` and `std::map`, which emphasize organizing the underlying storage in ways that make it easy to retrieve values once the data elements have been inserted.

We will start our informal overview with sequential containers.

Sequential Containers

Sequential containers hold elements that can be accessed sequentially. The following are sequential STL containers:

- `std::vector<T>`: A dynamic array container guaranteed to be allocated in contiguous memory, optimized for insertions and removals at the end of the container. Insertions and removal in other locations in the container are possible but are less efficient. It can be considered a no-cost and safe abstraction of a C-style dynamic array, safe in the sense that all heap memory allocation, management, and replacement is handled internally by the class. In practice, if used appropriately, a `vector` can actually be faster than a manually managed dynamic array since a `vector` does very careful and efficient resource management. This is the best known example of sequential container in C++, but it is not the only container of this family.
- `std::deque<T>`: This container offers functionality similar to a `vector`, but it also also provides efficient appending and removing of data elements to and and from the *front* of the container as well. Unlike a `vector`, however, its storage is not guaranteed to be in contiguous memory, and as a result traversal of that container will generally be less efficient as the same operation would be with a `vector`.
- `std::list<T>`: Data elements are organized as a doubly-linked list of nodes containing the value as well as pointers to neighboring (both previous and next) nodes. This makes it more efficient for insertions and deletions at arbitrary locations within the container. Unlike a `vector`, `deque`, or an

`array` (introduced next), however, it does not provide random access via the `[.]` operator or an `at(.)` member function. The only way to reach element `i` in a `list` is to start from the beginning and iterate through the first `i` elements.

- `std::array<T, N>` : A fixed-size array of `N` elements, the value of `N` must be explicitly known at compile time. Its contents are stored contiguously and are not dynamically allocated. As a result, an `array` can be the most efficient type of sequential container, but an explicit fixed size declaration requirement at compile time makes its practical use in financial applications highly limited.

In addition to the member functions on the `vector` class that you are now already familiar with, there are quite a few more that can be important in practice. We will start our coverage of sequential STL container with a `vector`, but you will eventually see that much of what you learn here will be relevant for other containers.

The `std::vector` Sequential Container

In previous chapters, in examples using a `vector`, the focus has primarily been on storing plain numeric types as elements, but as a generic container, objects of any valid type -- essentially any type that is at least copyable or movable -- can be used as its template parameter. For good reasons, a `vector` is seen by most as the default container in C++.

Most attention in this chapter will be on using a `vector`, as it is the Standard Library container you will almost surely be using the most as a financial developer. However, many of the member functions described for this container will carry over to other standard containers, so by learning about vectors well, you should be able to easily pick up what you need when using other containers.

Storing Objects on a `vector`

A common device used in financial modeling and actuarial science is a mythical but theoretically useful continuously-compounded bank account. Given an initial deposit B_0 and a fixed interest rate r, af-

ter a period of time t measured in units of years (or alternatively a *year fraction*), it will grow to an amount

$$B(t) = B_0 e^{rt}$$

A simple class that represents this can be written as follows (the reason for the default constructor and in class initialization will become apparent shortly):

```
#include <cmath>

class BankAccount
{
public:
	BankAccount(double init_value, double continuous_rate) :
		init_value_{ init_value }, continuous_rate_{ continuous_rate } {}

	BankAccount() = default;

	double value(double time) const
	{
		return init_value_ * std::exp(continuous_rate_ * time);
	}

private:
	double init_value_{ 1.0 }, continuous_rate_{ 0.0 };

};
```

Then, suppose we have three competing accounts with interest rates 2.1%, 2.2%, and 2.3% respectively, with initial deposits \$1000, \$2000, and \$3000 (pretend you get a higher interest rate with more money deposited), and that we need to place these objects in a `vector` container. This may seem trivial,

but there are some important consequences that depend on how this placement is done, as we are now dealing with objects rather than simple numerical types.

A naive approach would be to create three `BankAccount` instances and push each back onto the end of a `vector`, say `ba_push`. When `ba_push` is created, it will not hold any elements, which can be verified by calling its `size()` method.

```
#include <iomanip>          // For std::setprecision and std::fixed,
                            // to fix account value output at two decimal places.

// . . .

vector<ZeroCouponBond> ba_push;
cout << format("ba_push contains {} elements.", ba_push.size())
    <<  "\n\n";             // size = 0
```

Now, create each bank account object and push it back on the vector:

```
BankAccount ba01{ 1000.00, 0.021 };
BankAccount ba02{ 2000.00, 0.022 };
BankAccount ba03{ 3000.00, 0.023 };

ba_push.push_back(ba01);
ba_push.push_back(ba02);
ba_push.push_back(ba03);
```

Then, you can check the number of elements again with the `size()` method, and you will find there are now three elements. You can also loop through to get the account values after one year on each stored object:

```
for (const auto& ba : ba_push)
{
      // This is where we need <iomanip>, to round to two decimal places:
      cout  << "Accumulated amount after 1 year = "
                  << std::setprecision(2) << std::fixed << ba.value(1.0) << "\n";
}
```

Rounded to two decimal places, the results are:

```
(Output)
Accumulated amount after 1 year = 1021.22
Accumulated amount after 1 year = 2044.49
Accumulated amount after 1 year = 3069.80
```

The point of this exercise is that if we create the `BankAccount` objects and use `push_back(.)`, it will require inefficient object copy. With `push_back`, an element in the vector will take in a `BankAccount` object with its copy constructor. This can be verified by setting the `BankAccount` copy constructor to `delete` and noticing the code will not compile. In a more realistic situation where a `vector` might contain thousands of larger object elements, the performance hit could add up significantly.

This problem was resolved in C++11 with the introduction of the `emplace_back(.)` member function. If the calling code knows which arguments to pass in order to construct an object but does not have the object on hand, calling `emplace_back(.)` with the constructor arguments will let the container create the object for us, saving one construction (and one destruction) with every call. For example, no `BankAccount` object copies are generated in the following code:

```
vector<BankAccount> ba_emplace;
```

```
ba_emplace.emplace_back(1000.00, 0.021);
ba_emplace.emplace_back(2000.00, 0.022);
ba_emplace.emplace_back(3000.00, 0.023);
```

Any performance gains to be had in this toy example would be minimal, since `BankAccount` is a tiny type, but pushing "heavier" objects, and a lot more of them, onto the end of a `vector`, the cost savings can become significant.

Returning to the example here, you can verify using console output that the `vector` size is three, and that the valuation results are the same as in the previous `push_back(.)` example. Note that `emplace_back(.)` is mostly useful if the object to add to the container has not been constructed yet. If you actually have a full object on hand, simply use `push_back(obj)`, or alternatively you can use `push_back(std::move(obj))` to avoid the costs of object copying, if `obj` is no longer needed outside the container.

Handling Polymorphic Objects

In financial C++ programming, there may be times where you will need to handle a collection of derived objects, determined dynamically, in a `vector`. Using modern C++, and in particular the related discussion in Chapter Three, this can be accomplished by defining a `vector` with a `std::unique_ptr` template parameter pointing to objects of derived classes via the (often abstract) base class type.

Suppose we might want to value a book of options on the same underlying equity. Each call and put payoff could be managed by a `vector` containing unique pointers to abstract `Payoff` types discussed in Chapter Three:

```
vector<std::unique_ptr<Payoff>> payoffs;
```

Then, we can push back each pointer onto the `vector` as exemplified here:

```
payoffs.push_back(std::make_unique<CallPayoff>(90.0));
payoffs.push_back(std::make_unique<CallPayoff>(95.0));
payoffs.push_back(std::make_unique<CallPayoff>(100.0));
payoffs.push_back(std::make_unique<PutPayoff>(100.0));
payoffs.push_back(std::make_unique<PutPayoff>(105.0));
payoffs.push_back(std::make_unique<PutPayoff>(110.0));
```

When the `payoffs` container goes out of scope, each `unique_ptr` cleans up after itself. In contrast, doing this with raw pointers, we would have:

```
std::vector<Payoff*> raw_ptr_payoffs;
raw_ptr_payoffs.push_back(new CallPayoff{ 90.0 });
raw_ptr_payoffs.push_back(new PutPayoff{ 105.0 });

// etc . . .
```

But now, once we are done with using `raw_ptr_payoffs`, we would have to iterate through the `vector` again to delete each element manually before exiting the active scope. This means:

- Risk of forgetting this step, leading to memory leaks
- More code to manage
- More that can go wrong

The destruction of each `unique_ptr` object will automate the destruction of its pointed-to object, doing implicitly what we would otherwise have been forced to do manually. As such, using unique pointers

should be preferred over raw pointers. Using this feature of modern C++ makes our lives so much easier, and at essentially no performance cost.

This example could also have used other sequential containers (to be discussed in due course), but it is shown here within the context of our discussion on `vector` containers, as these are again the most common container types you are likely to find in practice.

Before proceeding to the other STL sequential containers, there are various characteristics and useful member functions regarding `vector` containers that are covered here first.

Allocation and Contiguous Memory

A `vector` models a dynamic array that can grow when it is full. When an insertion (e.g., a call to `push_back()`) occurs, insertion into a vector can incur the costs of an object copy, but that cost can be higher when the container's *capacity* has been reached and the objects therein have to be copied or moved to a new and bigger storage location.

(Could maybe put a diagram here...not sure yet)

One needs to distinguish the *size* of a `vector`, exposed by its member function `size()`—which represents the number of elements stored in the container—from its capacity, exposed by the member function `capacity()`. The latter represents the number of objects that *can* be stored in the container, including those that are already present. At any time, for a `vector` named `v`, one can add up to `v.capacity() - v.size()` objects before `v` needs to increase its capacity.

Indeed, conceptually, appending an object to a `vector` can be illustrated as follows:

```
// Note: this is an oversimplified illustration
```

```
. . .

void push_back(const T &obj)
{
  // if full() // full() means size() == capacity()
  //    grow()
  // add obj at the end of the array
  // increment size
}

void grow()
{
  // determine a new capacity, bigger than the current one
  // allocate a new array of that new capacity
  // copy or move the elements from the old array to the new one
  // dispose of the old array
  // update capacity to the new capacity
}
// . . .
```

This process is usually quite fast, as a `vector` instance strives to keep its capacity bigger than its size in order to avoid reallocations, but when a reallocation needs to be done, then a number of copies (or a number of moves) will need to be performed.

There are ways to reduce the costs of these potentially costly moments. The `vector` class offers member functions that let user code decide when to change the size or the capacity of the underlying storage. One is `reserve(.)`, which changes its capacity but not its size (number of elements). Another is `resize(.)`, which changes both the size number of elements and the capacity by conceptually adding default values at the end (e.g. zero for fundamental types such as `int` or `double`, `nullptr` for pointers, or default-constructed objects for classes with such a constructor).

Thus, if you know at least approximately how many objects will be stored in a container, you can call these functions at selected (non-critical) moments in a program to ensure there will be enough space available when time comes to actually add the objects therein.

A `vector`, when resizing or reallocating space, will need to copy or move the objects from the old storage to the new storage. To do so, it will call each object's move constructor if these constructors have been marked as `noexcept`; otherwise, it will call each object's copy constructor otherwise. For a given object type, move constructors are faster (often significantly faster) than copy constructors.

NOTE

The `noexcept` specification, added to the Standard in C++11, prevents a function (including the move constructor special member function) from throwing an exception. In this book, as remarked in Chapter Two, we will only be concerned with using the default move constructor, which for our purposes can be assumed to be `noexcept`.

In the case where a user-defined move constructor is present, for the reasons provided in the point above, per the Core Guidelines, it should be declared `noexcept` **{3}** (*https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-noexcept*)

When a `vector` is created with its default constructor; eg,

```
std::vector<MyClass> v;
```

it contains no elements. This can be verified with the `empty()` member function on the `vector`,

```
cout << std::boolalpha << v.empty() << "\n";  // v.empty() == true
```

and/or with the `size()` method introduced earlier:

```
auto s = v.size();          // s = 0, return type technically is std::size_t
```

As a `vector` is guaranteed to store its data in contiguous memory, typically on the heap, with repeated `emplace_back(.)` or `push_back(.)` calls, the `vector` will attempt to place the next object in the memory block adjacent to the previous element, but if this block is already allocated to another resource, the vector will need to reallocate its memory to a different location and either move or copy all the existing data into this location. This can potentially become expensive, especially if the data is copied **{4}**.

We can control these costs if we know how many elements we will need (or a reasonable upper estimate). One way to achieve this is to proceed naively and place this value in the constructor of the vector, eg:

```
vector<MyClass> w(100'000);          // The apostrophe used as a proxy for a comma
                                     // was added to the Standard in C++14
```

This is not very appealing, however, because now the `MyClass` default constructor has been called 100,000 times, creating a default object in each location, plus we might very well need to copy a new non-default constructed object into each element. This option is reasonable if we do need the default `MyClass` objects and are expecting to perhaps replace only a portion of them later.

Fortunately, a viable and efficient alternative exists by specifying the number of elements to reserve in heap memory, using the `reserve(.)` on the `vector` container class:

```
std::vector <MyClass> u;
u.reserve(100'000);
```

Now, `u` now has *zero* objects (`u.size()==0`) but *has room* to add 100,000 ones through `push_back()` or `emplace_back()` without `u` ever having to reallocate memory.

The upshot here is if you have an estimate of how many objects will be stored in a `vector`, you can call the `reserve(.)` function after creating a `vector` container object but before appending additional objects with `emplace_back(.)` or `push_back(.)`. This will avoid forcing a (potentially costly) reallocation of contiguous memory.

This can be seen in a simple demonstration by reprising the `BankAccount` examples. The capacity is set to 5, so this will accommodate three elements and leave room for a couple more just in case.

```
// Before applying push_back(.):
vector<BankAccount> ba_push;
ba_push.reserve(5);                    // Space is reserved for 5 elements in memory,
                                       // but the size at this point is still zero.

// Same as before but included here for
// easier reference:
BankAccount ba01{ 1000.00, 0.021 };
BankAccount ba02{ 2000.00, 0.022 };
BankAccount ba03{ 3000.00, 0.023 };

ba_push.push_back(ba01);
ba_push.push_back(ba02);
ba_push.push_back(ba03);        // ba_push.size() now = 3,
                                       // but ba_push.capacity() still = 5.


// Similarly for emplace_back():
vector<BankAccount> ba_emplace;
ba_emplace.reserve(5);
```

```
ba_emplace.emplace_back(1000.00, 0.021);
ba_emplace.emplace_back(2000.00, 0.022);
ba_emplace.emplace_back(3000.00, 0.023); // ba_emplace.size() = 3, ba_emplace.capacity() = 5.
```

In real-world financial systems programming, portfolio valuation and risk calculations could apply to thousands of positions in multiple asset categories, and this could require loading each position object into a `vector`. In order to prevent a series of incremental memory reallocations for large `vector` containers, setting a sufficient capacity with the `reserve(.)` member function before processing the position data can help ensure better runtime performance. Also, the actual integral argument for the `reserve(.)` function will more likely be determined dynamically (based on some external input) rather than by a hard-coded value (eg `reserve(5)` or `reserve(100'000)`) as in the introductory examples here.

The `clear()` method

The `clear()` member function on the `vector` container class will destroy each object stored in the container and reset its size to zero. Define a trivial class as follows with the destructor implemented:

```
class MakeItClear
{
public:

	~MakeItClear()
	{
		std::cout << "MakeItClear destructor called" << "\n";
	}
};
```

Then, define a `vector` with three `MakeItClear` objects:

```
vector<MakeItClear> mic;
mic.reserve(3);
mic.emplace_back();    // Emplace MakeItClear object
mic.emplace_back();    // using default constructor - no argument.
mic.emplace_back();

cout << format("Size of vector<MakeItClear> = {}, capacity = {}\n\n",
      mic.size(), mic.capacity());
```

With the output to the screen, you can verify there are three elements, and the capacity is also 3, per the setting by the `reserve(.)` member function.

```
(Output)
Size of vector<MakeItClear> = 3, capacity = 3
```

Now, clear these elements, and output the updated size and capacity of the `vector` object `mic`:

```
mic.clear();

cout << format("\nSize of vector<MakeItClear> now = {}, capacity = {}\n\n",
      mic.size(), mic.capacity());
```

Then, the destructor is called three times, as indicated by the output stream in its body:

```
(Output)
MakeItClear destructor called
MakeItClear destructor called
MakeItClear destructor called
```

```
Size of vector<MakeItClear> now = 0, capacity = 3
```

Now, the `vector` size is zero, as its contents have been cleared. The `vector` capacity, however, has not changed. Since it can be costly to increase the capacity of a `vector`, the memory already held by that container remains held until explicit actions to the contrary are taken (or until the container is destroyed).

The `clear()` method is also defined for all the other STL containers covered in this chapter, except for `std::array()`.

`front()`, `back()`, and `pop_back()` methods

The first two of these member functions do what they say: return (a reference to) the first and the last element in a `vector`, respectively. As an example, let's go back to our `MyPair` class template, and create a `vector` of `MyPair<int>` elements. As an aside, note that a template argument can be a template itself. This is a mild example:

```
vector<MyPair<int>> pairs;
pairs.reserve(4);

pairs.emplace_back(1, 2);
pairs.emplace_back(1, -2);
pairs.emplace_back(3, 4);
pairs.emplace_back(3, -4);
```

We can then access the minimum element of the pair in the front (first) and back (last) elements of the `vector` container:

```
int front_min = pairs.front().get_min();          // front_min = 1
int back_min = pairs.back().get_min();                // back_min = -4
```

This will result in 1 and -4, respectively.

The `pop_back()` function will then remove the last element:

```
pairs.pop_back();
back_min = pairs.back().get_min();                    // back_min = 3
```

The value of `back_min` is now 3, the minimum value of the third (and now last) `MyPair` element.

`front()` and `back()` can also be used as mutators:

```
pairs.front() = { 10, 6 };                            // pairs[0] now = (10, 6)
front_min = pairs.front().get_min();              // front_min = 6
```

Utilizing `back()` for this purpose would be similar.

One could say that `pop_back()` is the opposite of `push_back()`. For a `vector`, there is neither a `push_front()` nor a `pop_front()`, but these do exist for other STL container classes. The reason for this is the Standard Library tends to offer only those member functions that can be implemented efficiently. A function like `push_back()` takes constant time on a `vector` (unless a call leads to a reallocation, which in practice should be infrequent by using its `reserve(.)` function).

If there were a `push_front(.)` member function on the `vector` container class, it would be a very costly operation since a `vector` models a dynamic array, and inserting at the beginning would mean copying or

moving every existing element “one position to the right”, a significantly costly operation if the number of elements is large. Containers that have a `push_front(.)` member function (eg `std::deque`, `std::list`, etc) are those for which that function can be reasonably efficient.

Random Access in a `vector`: `at(.)` vs `[.]`

A `vector` of size `n` allows random access of an element by its index $0, 1, \dots, n-1$, using either the `[.]` operator or the `at(.)` member function. The difference between these two is that `at(.)` provides bounds checking by throwing exceptions in the case where an attempt is made to access an invalid index. Should one seek to obtain an error message, it is made available by the overridden `what()` function on this standard exception class, `std::out_of_range` **{5}**:

```
#include <stdexcept>  // for exception class std::out_of_range
// . . .

vector<int> v = { 0, 1, 2, 3, 4, 5 };     // v.size() == 5

try
{
      int n = v.at(10);    // at(.) throws out_of_range exception
}
catch (const std::out_of_range& e)
{
      std::cout << e.what() << "\n\n";
}
```

Attempting to access an element at a negative index will also through an `out_of_range` exception.

There is no such protection with the square bracket operator. Replacing `at(.)` with `[.]` would result in a runtime error, resulting possibly in a program crash or, even worse, the error being silently ignored and

propagating itself where it could do even more harm. Technically, anything can happen if one accesses objects out of bounds since this leads to undefined behavior, meaning that the code is not "playing by the rules", and that the program is broken. For this reason, if you believe it is possible that an index could end up out of bounds, you can consider using `at(.)` instead of `[]` in order to identify the bugs in your code, and fix them. Of course, there is a cost to checking the bounds on every access, so only use `at(.)` when there is a doubt with respect to the validity of the indices.

A Potential Pitfall with the meaning of `()` and `{}` with a `vector`

This is something you may have noticed already, but in order to keep the flow of the discussion above moving, an explanation has been deferred to this point. As you now know, in modern C++, constructor arguments—including case of the default constructor, where there are no arguments—are often preferred in modern code bases using uniform initializations using braces (`{. . .}`), as opposed to the still-legal "traditional" definition using round brackets. For example, reprising the `Fraction` class that has two `unsigned` integer parameters in its constructor, you could write the following two definitions to get the same results:

```
Fraction braces{ 1, 2 };
Fraction round_brackets( 1, 2 );
```

In both cases, the numerator value will be 1, and the denominator value 2.

However, there can be cases with a `vector` (and other STL containers) where the results can be different. For example, if we attempt to use braces for the constructor argument for `vector` length, specifically for a vector of `int` values, say

```
std::vector<int> v{ 10 };                // size() == 1
```

this would initialize a `vector` of size 1 with a single element equal to 10. If we wanted to specify a size of 10, we would need to use the round bracket version:

```
vector<int> v_round_brackets(10);    // size() == 10
```

This might seem like an oddity, but there is a reason for this situation, which is detailed in the following sidebar. As for the upshot on how to resolve this inconsistency, an explanation follows afterward, below.

In the `vector` class, one will find a number of constructors, including the following two:

```
template <class T /* ... this is a simplification */>
  class vector
  {
    // ...
  public:
    vector(size_type, const T&); // number of elements, initial value
    vector(std::initializer_list<T>); // braces with values of type T
    // ...
  };
```

For this reason, given a `vector<int>` type and two values of integral types, both constructors are viable:

```
// 10 ints of value -1 or two ints valued 10 and -1?
vector<int> v1{ 10, -1 };          // two ints valued 10 and -1
vector<int> v2(10, -1);            // 10 ints of value -1
```

In C++, when using braces in a constructor, if there's a choice to be made between a constructor with an `initializer_list` **{6}** and another constructor, and both are viable, then the one that accepts an `initializer_list` is preferred, whereas when using parentheses, the one accepting an `initializer_list` is not considered (by definition). This leads to a surprising dichotomy, but it is part of the rules of the language (it would definitely be less annoying if it did not happen with `vector<int>` which is arguably the poster-child of containers used when teaching the language).

Suppose we added a default constructor to the `Fraction` class (and, say, initialized the numerator to 0 and the denominator to 1). Then, creating a `vector` of `Fraction` objects would not exhibit this difference,

as there is no way to initialize a `Fraction` with a *single* `unsigned` value. The following two lines will each result in creating a `vector<Fraction>` object with 10 `Fraction` elements:

```
vector<Fraction> v_frac_braces{ 10 };      // size() == 10
vector<Fraction> v_frac_rb(10);            // size() == 10
```

However, to make your head spin, suppose we add another `Fraction` constructor that takes in the numerator only (and initializes the denominator again to 1). Now, we're back to the original issue where the the former has a size of 1, and the latter has size = 10, because now we *can* initialize a `Fraction` with a single `unsigned` value.

```
// In the Fraction class, a constructor taking in a single int argument
// (the numerator) is added:
Fraction(unsigned n) : n_{n}            // numerator only, denominator = 1

// . . .

// the sizes will now be different when creating vector instances:
vector<Fraction> v_frac_braces{ 10 };      // size() == 1
vector<Fraction> v_frac_rb(10);            // size() == 10
```

Because of this potential pitfall, and to avoid complications, there are coding style requirements in wide use that explicitly require adhering to the C++98 method of round brackets for `vector` constructor arguments. A well-known example is the Google style guidelines **{7}**. In this book, we will use braces to indicate initialization of a `vector`, and take the round bracket version to dictate the number of elements. This way, the ambiguity is avoided.

Similar behavior also applies to other STL containers, so the same style guideline will be adopted in these cases as well.

C-arrays and the `data()` Member Function

A `vector` essentially is a no-cost abstraction of an encapsulated dynamic C array; in fact, it can be considered to be a negative cost abstraction for this very idea as it performs extremely efficient memory management, probably much better than any casual programmer would be able to implement in a reasonable amount of time. Its data is stored in contiguous memory, usually on the heap just like its C counterpart, but like a unique pointer, it cleans up after itself, plus it carries a robust set of member functions, many of which have been discussed above.

The `data()` method on `vector`, introduced in C++11, returns the memory address of its first element. It should be rarely, or ever, necessary to use it in a modern C++ context, but there are legacy numerical libraries written and C—such as the GNU Scientific Library for C—plus plenty of other legacy code bases that require interfacing with a raw pointer, and the `data()` member function makes things more convenient.

The interest behind `data()` is that it is safer than the alternatives. Consider the following C function that one might want to call, and suppose this function takes as argument the beginning of a sequence of `n` consecutive objects of type `double`:

```
double compute_sum(const double *buf, std::size_t n);
```

Now, suppose one seeks to call this function passing the buffer stored inside a `std::vector<double>` as argument. One could go about this in at least two ways:

```
double call_compute_sum_A(const std::vector<double> &v)
{
  return compute_sum(v.data(), v.size()); // this is OK
}

double call_compute_sum_B(const std::vector<double> &v)
{
  return compute_sum(&v.front(), v.size()); // this is risky
}
```

As you can see from the comments, one version (using `v.data()`) as it is well-defined in all cases, even in the case where `v` is empty as in that case `v.data()` will yield a null pointer (which is OK as long as `compute_sum` handles this case of course). On the other hand, calling `v.front()` yields a reference to the first element of `v`, an operation with undefined behavior when `v` is empty (an undesirable outcome to say the least).

The `data()` member function will return in Chapter Seven, in practical linear algebra examples.

The `std::deque` Sequential Container

As mentioned initially, most applications will use `std::vector` in most of their use cases, but no container is good at everything, and knowing the strengths and weaknesses of each standard container will help you make informed choices.

A `std::deque` is functionally very similar to a `vector`, with the added feature of being able to push elements onto, and pop elements from, the beginning of the container as well as the end. These member functions are not surprisingly named as follows:

- `push_front()`: Appends an element to the front of a `deque`

- `pop_front()` : Removes the first element from a `deque`

Put in diagram of vector vs deque, one-ended vs double-ended appending

As a first example, create a `deque` containing integers, and apply the member functions at both the front and the back of the container:

```
std::deque<int> on_deque{ 0, 1, 2, 3 };

// Push new elements onto the front:
on_deque.push_front(-1);
on_deque.push_front(-2);

// Can also push onto the back like a vector;
on_deque.push_back(4);
for (int k : on_deque)
{
	cout << k << " "; // on_deque now contains: -2 -1 0 1 2 3 4
}

// Remove both 1st and last element:
on_deque.pop_front();
on_deque.pop_back();
for (int k : on_deque)
{
	cout << k << " "; // on_deque now contains: -1 0 1 2 3
}
```

In addition to `emplace_back(.)` , there is also an `emplace_front(.)` function that obviates copying when appending an object to the front of a deque container. Suppose we have a `Rectangle` class with a constructor that accepts two `double` arguments and exposes an `area()` member function:

```
std::deque<Rectangle> recs;
recs.emplace_front(3.0, 2.0);
recs.emplace_front(4.0, 3.0);
recs.emplace_front(5.0, 4.0);
```

Then, iterating through the container to get the area values:

```
for (const auto& elem : recs)
{
      cout << elem.area()) << " ";
}
```

Note that due to the fact that we performed the insertions at the beginning, not at the end, the values displayed will be:

```
(Output)
20 12 6
```

Like a `vector`, a `deque` also supports random access using the `at(.)` function and the `[.]` operator, with `at(.)` again checking the bounds and throwing exceptions in cases of invalid index values. In fact, with only a few differences, almost all of the member functions defined on a vector will also work for a `deque`. The reason why a `vector` is usually preferred over a `deque` is that the common case of iterating through the elements in order will be faster on a vector, which models an array of contiguous values, than on a `deque` which models a sequence of contiguous chunks. To get an intuition as to the reasons behind this, note that modern computers are equipped with different levels of cache memory and benefit quite a lot from linear traversals of objects arranged in contiguous memory.

There may still be occasions in financial applications, such as storing values from newest to oldest in a time series, where appending to the front of a container would be convenient, or when computing returns from market prices, as will be seen shortly.

The fragmented memory storage does have one potential advantage over a vector in that new elements can either be appended into a new heap memory block, or by reallocating just a portion of the of the memory, rather than reallocation of the entire contiguous block. However, as we saw with a `vector`, the capacity can be set at the outset with the `reserve(.)` member function to prevent reallocation altogether. This is not an option with a `deque`.

The `std::list` Sequential Container

As mentioned above, the `vector` and `deque` containers share much of the same functionality, and for appending new data at the end of either, or at the front of a `deque`, these operations are very efficient. Where they fall short, however, is in cases where data needs to be inserted somewhere in between.

The internal structure of a `list` is very different from that of a `vector` or `deque`. Without going into a lot of detail, an element in a `list` is stored in a node, which is an object that contains the element and knows the address of its predecessor and succeeding nodes. Thus, technically speaking, the container is a "doubly-linked list".

Note that this structure means that for containers holding `n` elements, the overall memory consumption associated with a `list` will be higher than with a `vector` For example, a `vector<MyClass>` container object will contain some data to represent the size, capacity and location of the underlying storage, but its data will be a contiguous sequence of `MyClass` objects, nothing else. For that reason, the space consumed by the objects in a `vector` is the number of elements multiplied by the size of an element. A `list<MyClass>` container object, on the other hand, will contain a sequence of nodes, each containing a `MyClass` object and two pointers—so each node occupies space that is greater than the space occupied by its element—and each node will be allocated separately leading to slower traversals in practice.

When a new element needs to be inserted, rather than shifting elements in memory to provide space, a new node is created, the links between two existing elements are broken, and the new node is inserted in the list by establishing new links between the two that were previously connected. This means that the effect of an insertion (or of a removal) in a list is "local" in the sense that it only impacts the immediate neighboring nodes.

Show diagram?

Inserting a new element in a sequential STL container requires a discussion of STL iterators, which will follow shortly. For now, the main point is insertion at an arbitrary location in a `list` will be faster than in a `vector` or a `deque`, but accessing a particular element will be less efficient. Unlike a `vector` or a `deque`, a `list` does not support random access by index using the `at(.)` function or the `[.]` operator, since there would be no way to implement these operations efficiently (see [NOTE] below), but it does have many of the same member functions such as `push_back(.)`, `emplace_back(.)`, `front()`, `back()`, etc, plus it can be used in a range-based `for` loop like other standard containers.

```
std::list<int> franz{ 0, 1, 2, 3 };
	for (int elem : franz)
	{
		cout << elem << "\n";
	}

	franz.push_front(-1);
	franz.push_back(4);
	franz.push_back(5);
	franz.pop_back();

	for (int elem : franz)
	{
		cout << elem << "\n";
```

```
    }

//    int sum = franz.at(0) + franz[1]; // Compiler error!
                                                // Neither at(.) nor [.] is defined for a list
```

The results are again not surprisingly:

```
(Output)
 0 1 2 3
-1 0 1 2 3 4
```

NOTE

Since nodes in a `list` only know their immediate neighbors, the only way to reach the nth element in a `list` is to start from the beginning and iterate `n` times. With a `vector`, accessing the `n` th element is a simple matter of constant-time arithmetic, computing the address of the `n` th element past the beginning of the underlying storage.

As most data in finance is temporal, you will probably not need to insert data within the interior of a container very often—if at all—so the advantages of a `list` will probably not apply all that often in financial applications. With contemporary hardware, a `list` is mostly used when many node insertions and removal have to be performed at arbitrary locations in the container, when the fact that these operations have local impact can make a difference (which can be the case for multithreaded programs), or when the elements stored in the `list` can be very costly to copy or move (which can hurt in a vector when either a reallocation or an insertion—or a removal—occurs somewhere else than at the end).

Fixed Length `std::array`

Fixed-length arrays were introduced in C++11 formally as an STL container class under the name of `std::array`. Contrary to a `vector` which models a dynamic array and allocates on the heap, an `array` does not allocate in and of itself, and it has a fixed size (and capacity, which amounts to the same thing with this container). Because its size must be provided at compile time, and because the amount of financial data needed for a typical task will not be known a priori, you will also probably find it of limited use. Still, if your code uses raw arrays of fixed size on occasion, then an `array` will provide an exact equivalent (same memory consumption, same speed characteristics) but with additional services.

Indeed, an `array` supports member functions `size()`, `at(.)`, `front()`, `back()`, and `empty()`, and the `[.]` operator, but not `push_back(.)`, `push_front(.)`, `pop_front(.)`, or `pop_back(.)`, as the size of an `array` cannot be modified once it is defined.

In the following code, use of `arr_0` and `arr_1` are strictly equivalent:

```
int arr_0[]{ 0, 1, 2, 3 };
int sum_0 = 0;
for (int x : arr_0)
{
	sum_0 += x;
}

// sum_0 = 6

std::array<int, 4> arr_1{ 0, 1, 2, 3 };
int sum_1 = 0;
for (int elem : arr_1)
{
	sum_1 += elem;
}
```

```
// sum_1 = 6
```

When in Doubt, Use a `vector`

An obvious question at this point would be "which sequential container class should I use?". As mentioned, the use of `array` is quite limited, so that essentially leaves us with a `vector`, `deque`, or `list`. The answer is that unless there is a compelling reason to use one of the latter two, you should choose a `vector` by default. This is because a `vector` is, except for the very efficient but more limited `std::array`. As described in Sutter and Alexandrescu **{8}**:

- *Guaranteed to have the lowest space overhead of any container*
- *Guaranteed to have the fastest access speed to contained elements of any (dynamic) container*

The primary reason for this is its storage is guaranteed to be in contiguous memory, providing it with the efficiency of a dynamic C-style array. It is also equipped with the same convenience of random access as in a C array. The storage for the other two dynamically sized sequential containers may be fragmented, and `list` does not support random access.

The `vector` container truly is the workhorse of the standard C++ library. Getting to know its capabilities and details will make you a more effective quantitative developer. Understanding its use within STL algorithms (to follow) will also transfer to cases utilizing any of the other STL container classes.

Associative Containers

In C++, there are two primary associative containers that automatically keep their contents sorted, irrespective of the order the data is inserted. These containers also enforce the uniqueness of their elements. These two containers are:

- `std::set<T>` which stores unique elements of type `T`, and re-orders them whenever a new element is inserted.
- `std::map<K, T>` which stores pairs of elements of types `K` and `T`, where `K` is a key value upon which the container is sorted, and which is used to obtain its associated `T` value. This container ensures each key is unique but allows more than one key to have the same value.

For example:

```
#include <set>
#include <map>

// . . .

std::set<int> some_set{ 5, 1, 2, 3, 4, 3 };
for (int n : some_set)
{
	cout << n << " ";
}

cout << "\n\n";

std::map<std::string, int> some_map
{
	{ "five", 5 }, { "one", 1 }, { "two", 2 },
	{ "three", 3 }, { "four", 4 }, { "three", -3 }
};

for (auto [k, v] : some_map)
{
	cout << k << ": " << v << "; ";
}
```

If you run this code, you will see for output

`1 2 3 4 5`

displayed on the first line (note that there is only one occurrence of number 3). Then, recalling that `std::string` has inequality operators defined based on lexicographic ordering, eg,

$ ab < ac $

$ abc < abd $

you will see

```
(Output)
five:5; four:4; one:1; three:3; two:2;
```

on the second line. Note that there are no duplicate keys, for key "three" only the first insertion has worked and that the keys are sorted in alphabetical order.

Individual values can be obtained by using the key value in the square bracket operator, for example:

```
int four = some_map["four"];     // four = 4 (value)
```

Note, however, we will run into a problem with a `set` of `Circle` objects (from Chapter Three). The following will not compile, because there is no ordering defined on the class.

```
// Compiler error!
```

```
std::set<Circle> circles{ {5.5}, {3.2}, {8.4} };
```

The same issue would arise if attempting to use a type not supporting inequality operators as the key in a `map`.

There are two additional versions of these containers that *do* allow duplicates:

- `std::multiset<T>` which accepts multiple occurrences of the same value
- `std::multimap<K, T>` which accepts multiple pairs with the same key

For example:

```
// std::multiset and std::multimap are defined
// in the Standard Library <set> and <map> headers
// . . .

std::multiset<int> some_multiset{ 5, 1, 2, 3, 4, 3 };

for (int n : some_multiset)
{
      cout << n << " ";
}

cout << "\n\n";

std::multimap<std::string, int> some_multimap
{
      { "five", 5 }, { "one", 1 }, { "two", 2 },
      { "three", 3 }, { "four", 4 }, { "three", -3 }
};
```

```
for (auto [k, v] : some_multimap)
{
      cout << k << ": " << v << "; ";
}
// ...
```

If you run this code, you will see (output)

```
1 2 3 3 4 5
```

displayed on the first line (note that there are two occurrences of number 3), and from the `map` example you will see

```
(Output)
five:5; four:4; one:1; three:3; three:-3; two:2;
```

on the second line. Note that there are two entries with key "three", even though their values are different and that the keys are sorted in alphabetical order.

Finally, starting in C++11, unordered (aka hashed) versions of each of these associative containers were added to the Standard Library. They have the capability of making searches faster than their ordered counterparts (but often consume more space in memory), although this is not guaranteed. These are:

- `std::unordered_set<T>`
- `std::unordered_multiset<T>`
- `std::unordered_map<K, T>`
- `std::unordered_multimap<K, T>`

Technically speaking, the unordered versions are no longer “associative” containers, but they are variations on the themes of `set` and `map`. For this reason, as here in the present discussion, they are often combined into the same category **{9}**, although there is an argument for a separate “unordered containers” classification **{10}**. In any event, as their names show, these containers do not keep their data or keys sorted but can make search operations very fast.

Out of these eight associative containers, one that is particularly useful in financial modeling applications is a `map`, for managing model input data and output results. Its unordered counterpart is seen more often in applications that operate under very low latency constraints, such as in high-frequency trading systems. In addition, by learning how to use the `map` container well, using the others will become straightforward, as they work similarly.

`std::map` as a Model Data Container

Enum classes can be very useful as key values in a `map` for inputs to and outputs from a financial model. To see an example of this, consider an option deal, where our model computes an option price and associated “Greek” (risk) values: delta (Δ), gamma (Γ), vega, rho (ρ), and theta (Θ). Define an enum class called `RiskValues`:

```
enum class RiskValues
{
      Delta,
      Gamma,
      Vega,
      Rho,
      Theta
};
```

Returning to our `BlackScholes` class in Chapter Two, we could include `RiskValues` and add a private member function `risk_values_(.)` to compute each "Greek" value. In addition, as we would need to compute the Standard Normal cumulative distribution function (CDF) for both the option price and risk values, it is also refactored into a private member function, `norm_cdf_(.)`.

```
#include <map>

// . . .

class BlackScholes
{
public:
    BlackScholes(double strike, double spot, double rate,
        double time_to_exp, PayoffType pot);

    double operator()(double vol);
    std::map<RiskValues, double> risk_values(double vol);    // New

private:
    void compute_norm_args_(double vol);        // d_1 and d_2
    double norm_cdf_(double x) const;           // New - refactor out original lambda expression

    double strike_, spot_, rate_, time_to_exp_;
    PayoffType pot_;

    double d1_{ 0.0 }, d2_{ 0.0 };
```

The risk values can be computed using the closed-form formulae derived from Black-Scholes theory **{11}** (with the simplifying assumption of no dividend in this example):

$$\Delta = \varphi N(\varphi d_1) \Gamma = \frac{N'(d_1)}{X\sigma\sqrt{T-t}} = S^2 \Gamma \sigma T \rho = \varphi T X e^{-rT} N\left(d_2\right)$$
$$\Theta = \varphi S_0 N\left(\varphi d_1\right) - \varphi r X e^{-rT} N\left(\varphi d_2\right) - S_0 N'\left(d_1\right) \frac{\sigma}{2\sqrt{T}}$$

where $N'(x)$ is the Standard Normal probability density function (PDF):

$$N'\left(x\right) = \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}}$$

The results are computed and then placed in the `map` by using its `insert(.)` member function. This is similar to `push_back(.)` for a sequential STL container, except that each pair is ordered automatically by its key value. A lambda expression has been added for the Standard Normal PDF:

```
std::map<RiskValues, double> BlackScholes::risk_values(double vol)
{
	std::map<RiskValues, double> results;
	compute_norm_args_(vol);
	int phi = static_cast<int>(pot_);

	double nd_1 = norm_cdf_(phi * d1_);		// N(d1)
	double nd_2 = norm_cdf_(phi * d2_);		// N(d2)
	double disc_fctr = std::exp(-rate_ * time_to_exp_);

	// N'(x): Standard Normal pdf:
	auto norm_pdf = [](double x) -> double
	{
		return (1.0 / std::numbers::sqrt2) * std::exp(-x);
	};

	double delta = phi * nd_1;
	double gamma = norm_pdf(d1_) / (spot_ * vol * std::sqrt(time_to_exp_));
	double vega = spot_ * spot_ * gamma * vol * time_to_exp_;
	double rho = phi * time_to_exp_ * strike_ * disc_fctr * nd_2;
	double theta = 1.0;		// Placeholder for now
```

```
    // DELTA, GAMMA, VEGA, RHO, THETA
    results.insert({ RiskValues::Delta, delta });
    results.insert({ RiskValues::Gamma, gamma });
    results.insert({ RiskValues::Vega, vega });
    results.insert({ RiskValues::Rho, rho });
    results.insert({ RiskValues::Theta, theta });

    return results;
}
```

This makes storing the option value and risk calculations to a database more foolproof. The following *pseudocode* shows an example. Suppose a database of positions is represented by an interface object `database`, and that it contains a table called `Options`, with column names `PRICE` for the option valuation, and the respective "Greek" values in all caps. These align with the respective results from the C++ code.

```
double strike = 75.0;
auto corp = corp = PayoffType::Put;
double spot = 100.0;
double rate = 0.05;
double vol = 0.25;
double time_to_exp = 0.3;
BlackScholes bsp_otm_tv{ strike, spot, rate, time_to_exp, corp };

// (This is pseudocode, NOT real {cpp} code!)
DBRecord record = database.table("Options").next_record();
record.column("PRICE") = bsp_otm_tv(vol);
record.column("DELTA") = risk_values[RiskValues::Delta];
record.column("GAMMA") = risk_values[RiskValues::Gamma];
record.column("VEGA") = risk_values[RiskValues::Vega];
```

```
record.column("RHO") = risk_values[RiskValues::Rho];
record.column("THETA") = risk_values[RiskValues::Theta];
```

An `unordered_map` could be substituted for the `map` in this example if desired, although for this small number of elements, any performance gain would probably be minimal.

STL Iterators

Iterators are objects that can iterate over elements of a sequence. Writing algorithms on iterators rather than writing them on containers make algorithms more general and typically more useful.

Syntactically, iterators expose their services "via a common interface that is adapted from ordinary pointers" **{12}**, which has the nice side-effect of making ordinary pointers to contiguous sequences of objects work with Standard Library algorithms. For example, given an iterator `iter`, one moves to the next element in the sequence by (preferably) applying the increment iterator: `++iter` (preferable), or alternatively `iter++`. One then obtains a reference to the object currently referred-to by `iter` through `*iter`.

There are six categories of iterator types that are described in the usual C++ textbooks, namely output, input, forward, bidirectional, random-access and contiguous iterators. Note that these categories are not really classes; they are more like families conceptually grouped together by the operations they allow. Technically:

- An input iterator is a single-pass tool that can be used for such things as consuming data from a stream (including such fleeting things as keyboard input)
- A forward iterator can do anything an input iterator can do, but allows you to make more than one pass over the same sequence (if you don't alter the elements along the way), but only lets you go forward one element at a time

- A bidirectional iterator can do anything a forward iterator can do, but also lets you go backward one element at a time
- A random access iterator can do anything a bidirectional iterator can do, but lets you go forward or backward `n` elements at a time efficiently
- A contiguous iterator is a random-access iterator with the added requirement that the sequence is placed contiguously in memory (e.g., iterators on a `vector` or an `array`, but not a `deque`)

In this section, we will primarily focus on iterators for a `vector`, which fall into the *contiguous* category. Obtaining an appropriate iterator for other STL containers is similar, but the category to which an iterator belongs depends on the actual container. For example, in a `std::list` —which models a doubly-linked list —one can only efficiently move an iterator to the next or previous element of the sequence, so an iterator for a `list` is a bidirectional iterator.

One could define an iterator on a `vector<int>` container as follows:

```
// Create the iterator:
vector<int>::iterator pos;          // pos = "position"

// Also create a vector of integer values:
vector<int> v = { 17, 28, 12, 33, 13, 10 };
```

Next, we set the iterator to the *position* of the first element, using the `begin()` member function on a `vector`. Note that the word “position” here is not used in the way one would talk about an index in the same array. Because an iterator acts syntactically like a pointer, the actual element in the first position is accessed by dereferencing the iterator.

Suppose we have a `print_this(.)` function template:

```
#include <iostream>

template<typename T>
void print_this(T t)
{
      std::cout << t << " ";
}
```

The `begin()` member function returns an iterator to the first position of `v`:

```
pos = v.begin();              // Sets iterator to point at 1st position of the vector v
int first_elem = *pos;        // Dereferencing pos returns 17
print_this(first_elem);
```

Dereferencing the iterator `pos` at this position, the following would display `17` to the screen:

To advance the iterator to the next position, use the increment operator:

```
++pos;
print_this(*pos);             // *pos = 28
```

We can also reassign the value to the element found at this position just as we would with a pointer:

```
*pos = 19;                          // Now, *pos = 19, but the iterator
print_this(*pos);             // still points to the 2nd position
```

In cases where the container supports random access, such as a `vector` or a `deque`, an iterator can also be moved ahead by adding to it the number of positions to advance:

```
pos += 3;
print_this(*pos);              // *pos = 13
```

Because in this example `pos` is a random-access iterator associated with a `vector` container, it is also a bidirectional iterator and is able to move backward. Again, as with pointers, this can be done by with the decrement operator:

```
// Move back two positions:
--pos;
--pos;
print_this(*pos);              // *pos = 12
```

Subtraction assignment also works in the case of a random-access iterator (`-=`).

```
pos -= 2;             // *pos =  17
print_this(*pos);     // Back to initial position
```

STL iterators are made to be very fast and generally will not validate whether you go out of bounds or not, so you will need to be careful. For example, the code above which moves backwards two positions in a sequence will lead the iterator out of bounds if `pos` initially points to the first element of the sequence, a behavior that in turn leads the call to `print(*pos)` to perform what the C++ standard calls *undefined behavior*, meaning behavior for which no guarantees are provided (essentially, this means your code is broken, so do not do that).

The `end()` member function of a container returns an iterator that conceptually points to "one past the last element". The metaphor expressed by C++ Standard Library iterators is that the beginning of the sequence points to the first element and the end of the sequence points to where an iterator will go if advanced past the last element. More succinctly, C++ iterators on a container define a half-open range of the form $[$*begin*, *end*$)$, and understanding this is essential to using algorithms and containers correctly. Technically, some container `v` is said to be empty if `v.begin() == v.end()`. To access the last element of a non-empty vector, subtract one from the latter result and dereference the iterator again:

```
pos = v.end() - 1;
print_this(*pos);               // *pos = 10
```

Show diagram, with element values, a la SL 2E

Use `auto` to Reduce Verbosity

Both the `begin()` and `end()` functions on a Standard Library container will return an iterator for the sequence defined therein, so we can make our code less verbose while saving ourselves some typing by just using the `auto` keyword:

```
auto auto_pos = v.begin();        // vector v
```

Here, if `v` is of type `std::vector<int>`, then `auto` replaces having to type `std::vector<int>::iterator`, which greatly reduces the noise in this declaration.

Constant Iterators

Much like preferring `const` member functions on a class by default, it is often desirable to prevent elements of a container from being modified. This can be done by defining a `const` iterator, using the `cbegin()` member function rather than `begin()`:

```
vector<int> w{ 17, 28, 12, 33, 13, 10 };
auto cpos = w.cbegin();      // cpos = const iterator
```

A `const` iterator is an iterator that only provides non-mutating operations on the elements. On a `const` container, the `begin()` and `end()` member functions will yield `const` iterators, whereas on a non-`const` container, the `begin()` and `end()` member functions will yield non-`const` iterators. The `cbegin()` and `cend()` member functions are useful if you need a `const` iterator over a non-`const` container.

Applying this to the previous example, everything will compile and run just fine except for the reassignment. With a `const` iterator, this will result in a compile-time error:

```
++cpos;
*cpos = 19;          // Compiler error!
```

This is precisely the motivation for using a `const` iterator and helps prevent undefined behavior.

Iterators or Indices?

You have probably noticed by now we used neither operator `[.]` nor member function `at(.)`. as both of these requires indices as arguments, not iterators. In previous examples, traversing a `vector` was performed by using iterators alone. This is convenient because iterators can be used on any STL container,

but indices cannot, something you can verify yourself by trying to use indices on a container such as a `std::list`. In fact, since we used iterators, we could replace the `vector` with either a `list` or a `deque` in the previous examples, and they would compile and run successfully, yielding the same output.

Another useful result is the range-based `for` loop, first introduced in Chapter One, will also work with *any* STL container, and even user-defined or third party containers, as long as they offer the appropriate `begin()` and `end()` member functions. This is again because a range-based `for` loop is built on iterators and not on indices. Indices only make sense when one can advance an iterator efficiently `n` positions at once, or access the `n` th element of a container just as efficiently. Through iterators, we have one consistent and reliable way to iterate over any STL container with the familiarity of a `for` loop.

NOTE

Prior to the introduction of range-based `for` loops in C++11, one would need to use iterators explicitly to achieve the same generality over all standard containers. In this case, the loop would place an iterator at the beginning of the sequence and increment the iterator up to – but not including – the end of that sequence. For example:

```
std::vector<int>  v{ 17, 28, 12, 33, 13, 10 };

for (std::vector<int>::iterator pos = v.begin();
    pos != v.end(); ++pos)
{
      print_this(*pos);
}
```

One can still write loops this way in C++ today, and there are sometimes reasons to do so, but range-based `for` loops are so much simpler and more readable that they are generally preferred.

Iterators on Associative Containers

As alluded to above, associative containers also expose iterators which let us traverse the elements of that container. In such containers, the elements typically are key-value pairs. We will illustrate this with the following `toy_map` container object. First, we will create a simple `int` / `double` map object:

```
std::map<int, double> toy_map;
toy_map = { {5, 92.6}, {2, 42.4}, {1, 10.6}, {4, 3.58}, {3, 33.3} };
```

Initialize a new iterator `pos` by pointing it at the first element

```
auto pos = toy_map.begin();
```

Display the first pair by dereferencing the iterator:

```
// Display 1st pair:
print_this(pos->first);
cout << ":  ";
print_this(pos->second);
```

The first pair is displayed by accessing its key (`first`) and value (`second`) by dereferencing the iterator.

```
(Output)
1 :  10.6
```

Note that in a `std::map` keys are `const` and cannot be modified, but values are not (unless the container itself is `const`, of course). Suppose we advance one position and modify the value

```
// Advance to next position and modify:
++pos;
pos->second = 100.0; // In a std::map keys are const but values
                                    // (pos->second) can be modified
```

A range-based `for` loop will display each key/value pair `pr`, including the updated value in the second position:

```
// Display modified map:
for (const auto& pr : toy_map)
{
      cout << format("{} : {}\n", pr.first, pr.second);
}
```

However, as seen earlier, this can be written more succinctly in terms of the actual `[k, v]` pairs:

```
for (auto& [k, v] : toy_map)
{
      cout << format("{} : {}\n", k, v);
}
```

The range-based `for` loop in each case displays sequence of key/value pairs, ordered by the key value:

```
(Output)
1 :  10.6
```

```
2 :  100
3 :  33.3
4 :  3.58
5 :  92.6
```

STL algorithms

STL algorithms, in a single command, can replace operations on sequences of values that would otherwise typically require `for` or `while` loop blocks with multiple lines of code. In addition to making code more easily maintainable, they often accomplish the same tasks more efficiently than had they been implemented with handwritten loops, and they clearly document user intent, making the code self-documenting.

Most STL algorithms use a pair of iterators to traverse across the range of a container (or more generally a subset thereof), and many of them then invoke on each element an *auxiliary function* that is supplied as an argument to the algorithm.

A First Example

Suppose you need to count the number of odd integers in an STL container. One approach would be to just use a `for` loop, check if an element is odd, and increment a counter.

```
std::vector<int> int_vec{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int count = 0 ;
for (int k : int_vec)
{
      if (k % 2 != 0)
```

```
    {
        ++count;
    }
}
```

To clarify intent, one could define an `is_odd(.)` *predicate*—a function returning true or false—that takes an integer argument and returns `true` only if that integer is odd, and then call it in a loop:

```
bool is_odd(int n)
{
    return n % 2 != 0;
}

// ...

count = 0;
for (int k : int_vec)
{
    if (is_odd(k))
    {
        ++count;
    }
}
```

To make the code clearer (and potentially faster), we could use the `is_odd(.)` predicate as an auxiliary function and pass it as an argument to the `count_if(.)` STL algorithm, which will implement the algorithm optimally for us:

```
#include <algorithm>        // For most STL algos (including count_if)
//. . .
```

```
// is_odd(.) becomes our auxiliary function:


// Apply the count_if algorithm:
auto num_odd = std::count_if(int_vec.begin(), int_vec.end(), is_odd);

// num_odd is 5 (unsigned type)
```

The `count_if(.)` algorithm will traverse the sequence determined by the two iterators passed as argument, starting from the first position (`int_vec.begin()`) and ending at the last position (`int_vec.end()`). Remember this iterator pair forms a half-open range such that the `end` is an iterator positioned just after the last element. The `is_odd(.)` function will be applied to each element, counting the number of times it returns `true`.

As seen from the calling code, the algorithm replaces the handwritten loop and counter with a single line, and the `is_odd(.)` function is decoupled from the loop so that it can be reused elsewhere if desired.

NOTE

Many algorithms use a *predicate* as an auxiliary function. As noted previously, a predicate is a function that takes one argument and returns a `bool` or something that implicitly converts to `bool`. Above, `is_odd(.)` is a predicate on an `int`.

Well-behaved predicates used with STL algorithms should not mutate the elements they are applied to. Clearly, `is_odd(.)` is well-behaved in that sense as it takes its argument by value.

There can also be cases where a predicate takes in more than one argument. For example, the `std::sort` algorithm takes as argument two iterators and a predicate that accepts two arguments, and it returns `true` only if they are already ordered as expected. This lets the calling code control the order in which elements in a sequence will be sorted. If no predicate is passed, the algorithm sorts elements in increasing order.

NOTE

To understand how an algorithm such as `std::count_if` works, it can be useful to look at a naïve implementation (real implementations are more sophisticated):

```
template <class It, class Pred>
int naive_count_if(It b, It e, Pred pred)
{
  int n = 0;
  for(; b != e; ++b)
  {
    if(pred(*b))
    {
      ++n;
    }
  }
  return n;
}
```

Here, `It` is the type of the iterators, `b` and `e` are objects of type `It` passed by value such that the function can modify them without altering the objects passed by the calling code, and `pred` is a function (or functor, or lambda) that takes an element of the sequence and returns a `bool`, The algorithm applies `pred` to each element in the half-open range $[$ `b`, `e` $)$ and counts the number of calls that yield `true`.

At worst, calling an algorithm will give you results as good as if you had written a very good loop yourself but will lead to better documented code. At best, your code will leverage optimizations that your Standard Library vendor has implemented to leverage your types and your machine and will achieve its objectives faster while still being better documented. Using algorithms, you can only win.

One more advantage is that algorithms such as `count_if(.)` are reusable and can be used with arbitrary auxiliary functions of the appropriate signature, including lambda expressions. With `count_if(.)`, for ex-

ample, one could use a function that checks if a value is odd or even, congruent modulo n to some other value, located with an inclusive interval, etc.

```
// Assume the is_odd(.) function is as defined above...
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto num_odd = std::count_if(v.begin(), v.end(), is_odd);


// We could also count the even numbers in the sequence using a lambda
// that returns true if an int element is not odd
auto num_even = std::count_if(v.begin(), v.end(),
                    [](int n) { return !is_odd(n); });

// Count the number of elements congruent modulo 3 to
// some other int value:
const int other = 4;
auto num_congruent_mod_3 =
  std::count_if(v.begin(), v.end(),
            [](int n) { return n % 3 == other; });

// count the number of elements in [3,6]
auto num_within_interval =
  std::count_if(v.begin(), v.end(),
            [low = 3, high = 6](int n) { return low <= n && n <= high; });
```

We can already see how lambda expressions make it easy to use and benefit from STL algorithms. In order to count the even numbers in a sequence, for example, we leveraged the existing `is_odd(.)` function we wrote and used a lambda to return `true` for the numbers that are not odd. To count the number of elements whose value is in `[low, high]` where `low` is 3 and `high` is 6, we used the capture block of a

lambda to fix the bounds of the interval and we used the function call operator of that lambda to implement the test given some integer `n`.

Another benefit of STL algorithms is they can be applied on *any* half-open range from any STL container as long as the iterators used are of an appropriate category. The example above could have applied the exact same logic to a `std::list` or a `std::set`, for example.

```
std::list<int> int_list{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto num_odd = std::count_if(int_list.begin(), int_list.end(), is_odd);

std::set<int> int_set{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
num_odd = std::count_if(int_set.begin(), int_set.end(), is_odd);
```

To see why that is the case, you can go back to the naïve implementation of `naive_count_if` proposed previously. You will see that the implementation does not know about the particular container type it is being applied to, and it only requires that you can move forward one element using operator `++` on the iterators, which means that the algorithm could be used with any any algorithm that at least supports the input iterator category.

For map-type containers such as `map` and `unordered_map`, we could also get the number of odd elements with the `count_if` algorithm, but because each element is a `std::pair`, we would need to extract the value from each pair first, which is easily accomplished with a separate auxiliary function that dispatches the value to the original `is_odd(.)` predicate:

```
#include <utility>       // std::pair
// . . .

bool is_odd_value(const std::pair<unsigned, int>& a_pair)
{
```

```
    return is_odd(a_pair.second);
}
```

And then, apply the `count_if` algorithm again:

```
std::map<unsigned, int> int_map{ {1, 9}, {2, 8}, {3, 7}, {4, 6}, {5, 5}, {6, 4},
  {7, 3}, {8, 2}, {9, 1} };

num_odd = std::count_if(int_map.begin(), int_map.end(), is_odd_value);

std::unordered_map<unsigned, int> int_unord_map{ {1, 9}, {2, 8}, {3, 7}, {4, 6},
  {5, 5}, {6, 4}, {7, 3}, {8, 2}, {9, 1} };

num_odd =
  std::count_if(int_unord_map.begin(), int_unord_map.end(), is_odd_value);
```

The result will still count five odd elements in each case.

Finally, algorithms are not limited to containers with plain numerical types. They can be applied to class types as well, provided the auxiliary function is defined for that type.

More information on applying `count_if` on a `map` can be found on Stack Overflow **{13}**

A First Example with Ranges

The new Ranges library in C++20 provides abstractions that are more intuitive than specifying the `begin` and `end` iterator positions every time a container is to have an STL algorithm applied. Traditional STL algorithms ask you to pass the beginning and the end of a sequence separately, which is useful to operate

on the whole sequence or on sub-sequences thereof. Ranges take into account the fact that the common case is to operate on the whole sequence and make that case simpler to use.

A formal definition of a range "is a single object that represents a sequence of values" **{14}**. As such, any STL container would itself be a range, and passing a range into an STL algorithm results in a more concise and expressive statement compared to passing in iterator positions.

With the `count_if` example previously, you can just pass in the container and the auxiliary function to be applied, as shown next. The `<ranges>` header must be included, and the range version of an algorithm is scoped with the `std::ranges` namespace:

```
#include <ranges>
// . . .

std::vector<int> int_vec . . .;
std::list<int> int_list . . .;

num_odd = std::ranges::count_if(int_vec, is_odd);
num_odd = std::ranges::count_if(int_list, is_odd);
```

This is a welcome addition to the Standard Library, compared with the earlier and more verbose iterator versions:

```
num_odd = std::count_if(int_vec.begin(), int_vec.end(), is_odd);
num_odd = std::count_if(int_list.begin(), int_list.end(), is_odd);
```

The impact of ranges becomes even more noticeable for cases where a sequence of algorithms is to be applied to values since ranges are highly composable. Still, it is important to understand how to work

with pre-C++20 algorithm forms, due to several reasons:

- Some STL algorithms have not yet been updated for ranges
- Parallel execution policies are not yet available with ranges (to be covered in Chapter Five)
- There exist a lot of libraries and code bases current as of C++14 and C++17 that are still considered "modern"
- STL-compliant containers defined in external libraries (outside of the Standard Library) that require iterators are also quite common (a few of which we will cover later)

These points are understandable given that C++20 is still relatively new, so both iterator and range forms will be covered in this chapter.

Some Commonly-Used Algorithms

Two commonly used STL algorithms are `std::for_each`, and `std::transform`. `for_each(b, e, op)` applies a function `op` to each element in $[$ `b`, `e` $)$. A simple example might be where `op` prints out a single element to the screen. In contrast, `transform(b, e, d, f)` takes each element in $[$ `b`, `e` $)$, applies a function `f` to that element, and *stores the result* in the sequence beginning at `d`. At this stage, the first of these is of limited use, but this can change with lambda expressions supplied as auxiliary functions. `std::transform`, on the other hand, will have some immediate use cases. There are also useful algorithms that have their functionality built in -- for example sorting elements -- rather than requiring an auxiliary function.

The `for_each` Algorithm

The `for_each` algorithm applies a unary function – that is, taking one argument – to each element of a single container.

One immediate use is to display the contents of a sequence to the screen. Using our function template `print_this(.)`, we can print out any type that has an overloaded stream operator. By default, numerical types and `std::string` objects have this property.

`for_each` simply iterates over a container and passes each element to the `print_this(.)` function. As `print_this(.)` is a function template, if it is used as an auxiliary function, the particular template parameter must be included, as shown here:

```
#include <string>
// . . .

std::deque<int> q{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::vector<string> s{ "the", "sun", "is", "a", "lightbulb" };

std::for_each(q.begin(), q.end(), print_this<int>);
std::for_each(s.begin(), s.end(), print_this<string>);
```

The reason why we need to spell out `print_this(.)<int>` or `print_this(.)<std::string>` is that at the point where the call to `std::for_each` is made, the compiler cannot know which `print_this(.)` function it will be using. It works, but it's unsatisfying.

Here (again), lambdas can make our lives easier:

```
// . . .

auto prn = [](const auto &x) { print_this(x); };        // -> void ?

std::for_each(q.begin(), q.end(), prn);
std::for_each(s.begin(), s.end(), prn);
```

That might look surprising, but it works wonderfully well. Our `prn` lambda is generic (the `const auto &` argument can take an object of any type by reference-to-const), and it calls `print_this(.)` for an argument of that type which deduces which `print_this(.)` function is required. It is simpler, and it is just as fast!

Equivalent but cleaner code can be written using ranges. Reusing the `prn` lambda above, we get:

```
std::ranges::for_each(q, prn);
std::ranges::for_each(s, prn);
```

Alternatively, a lambda wrapper around a function template can be substituted in for the auxiliary function, eg `tmpl_square<.>`, as implemented in subsequent examples.

NOTE

The `for_each` algorithm can be seen as somewhat equivalent to the range-based for loop. The main differences between the two are:

- The range-based for loop is...a loop, so you can use `break`, `continue` or even `return` in such a loop in the same way you could with other loops
- The `for_each` algorithm is a function itself—not a loop—that calls another, namely an auxiliary function
- As will be seen in Chapter Five, `for_each` can be executed in parallel simply by adding an argument at the call site, which can increase throughput in your code when processing important amounts of data

In many cases, the range-based `for` loop will be sufficient for what you want, but there are cases where `std::for_each` will be better for you, so it is useful to know about these differences.

The `transform` Algorithm

The algorithm examples up to now have not done anything to modify the target containers, but there are cases in finance where you will need to:

- Apply a function across a container and replace the elements with the results
- Apply a function across a container and place the results in a separate container

This is where the `transform` algorithm comes in handy.

Modify and Replace Elements in the Same Container

Recalling the `tmpl_square(.)` function template used as an earlier example, and wrapping it in a lambda as the auxiliary function, we can apply the `transform` algorithm on a `vector` of integers, square each element, and place the result in the same position in the original container:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::transform(v.begin(), v.end(), v.begin(), [](int k) { return tmpl_square(k); });
```

Note that the source requires both iterators returned by `begin()` and `end()`, but the target -- in this case the same vector `v` —only requires the beginning position. This is typical of most of STL algorithms where a target container is present. You are responsible for ensuring that the destination container is at least large enough to hold all elements of the source sequence (in this example, it is obviously the case).

Using ranges, however, the syntax is more elegant, with the source and target alone sufficing:

```
std::ranges::transform(v, v.begin(), [](int k) { return tmpl_square(k); });
```

NOTE

Beginning with C++11, the functions `std::begin(.)` and `std::end(.)` were added to the Standard Library. They are advantageous in that they provide a generalization with C-style arrays for which of course STL container member functions `begin(.)` and `end(.)` are not defined.

For example, suppose we need to copy elements of a C-style array to a `vector`. Instead of writing

```
int c_array_ints[] = { 10, 20, 30, 45, 50, 60 };
vector v_ints(c_array_ints + 0, c_array_ints + 6);
```

we can replace this clunky syntax with a cleaner and more expressive version using `std::begin()` and `std::end()`:

```
vector v_ints(std::begin(c_array_ints), std::end(c_array_ints));
```

Note also, as an aside, the use of CTAD here.

Similarly, when applying STL algorithms on C-style arrays, the syntax can be made consistent with STL containers. Suppose next we want to square each value in `c_array_ints` and `v_ints`. Applying the `transform` algorithm is then the same form for each:

```
std::transform(std::begin(c_array_ints), std::end(c_array_ints), [](int k) { return tmpl_square(k); });
std::transform(std::begin(v_ints), std::end(v_ints), [](int k) { return tmpl_square(k); });
```

`const` analogs of `cbegin()` and `cend()` are also available: `std::cbegin(.)` and `std::cend(.)`.

In this chapter, the focus is on using STL containers, so the examples just use the member function versions, but if you are working with legacy code containing C-style arrays, this is an option for you.

Modify Elements and Place in Separate Container

Next, suppose we have the same vector `v`, and again we want to square each element, but in this case put the results in a different container `w`. This will often be useful when the transformation applied to each element yields a different type that the type of the elements in the source sequence (e.g., transforming a sequence of customers into a sequence of Social Security numbers).

Start by again initializing `v`, plus a deque `w`. Note again that since algorithms operate on iterators and not on containers, the source and target container types can be different. In this case, the source data type is `int`, but the target is `double`, with a constant `double` value of 0.5 added to the square of each value in `v`.

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::deque<double> dq;
```

Now, the `transform` algorithm will traverse `v`, square each element and add 0.5 (to show that the destination type can be different from the source type), and then push it onto the back of `w` using the `std::back_inserter(.)` function. This is equivalent to calling the `push_back(.)` member function on `w`. `back_inserter(.)` is declared in the Standard Library `iterator` header:

```
#include <iterator>            // std::back_inserter
// . . .

std::transform(v.begin(), v.end(),
      std::back_inserter(dq), [](int n)
      {
            return tmpl_square(n) + 0.5;
      });
```

The ranges version again dispenses with the verbose `begin` and `end` terms and expresses what it is doing more clearly:

```
std::ranges::transform(v,
        std::back_inserter(dq), [](int n)
        {
                return tmpl_square(n) + 0.5;
        });
```

A Closer Look at Algorithms with `std::transform`

For a given container, iterators express how one can go from one element to another and access the value of individual elements. As we saw earlier, an iterator is syntactically similar to a pointer. It provides operations to move from element to element (e.g.: `++iter` to move iterator `iter` to the next element in the sequence), allowing access to the element to which the iterator actively points (e.g. `*iter` to access the value pointed-to by iterator `iter`), compare two iterators with `==` or `!=` to know if they point to the same element, etc.

For example, a naïve implementation of algorithm `std::transform` might look like the following:

```
// IIt & OIt Input/Output iterators, F for function object (lambda or operator())
template <typename IIt, typename OIt, typename F>
void transform(IIt begin_src, IIt end_src, OIt begin_dest, F fcn)
{
   for(; begin_src != end_src; ++begin_src, ++begin_dest)
      *begin_dest = f(*begin_src);              // (1)
}
```

This algorithm requires that one can traverse the source range defined by the half-open interval [`begin_src` , `end_src`) (thus `begin_src` is included and `end_src` is excluded), traverse the destination range starting at `begin_dest` and write to that range (which has to be at least as large as the source range) and apply function `fcn` to each of the elements along the way (`fcn` has to be able to take as its argument an element from the source range and has to produce as result a value that can be written to the destination range). A correct call to that algorithm would be:

```
// . . .

int square(int n) { return n * n; }

// . . .

std::array<int, 5> vals{ 2, 3, 5, 7, 11 };
std::array<int, 5>  dest; // size as large as vals
transform(vals.begin(), vals.end(), dest.begin(), square);          (2)
// after this call, dest contains 4, 9, 25, 49, 121
```

In the `transform()` example above, `fcn` is an auxiliary function.

NOTE

When using `std::transform`, or for that matter any other STL algorithm requiring an auxiliary function, suppose we were to attempt applying a `<cmath>` function, such as `std::log(.)`. For example, if we wrote:

```
vector<double> dbl_vals{2.0, 3.0, 5.0, 7.0, 11.0};
vector<double> dbl_dest(dbl_vals.size());

// Using the naive version of transform:
transform(dbl_vals.begin(), dbl_vals.end(), dbl_dest.begin(), std::log);      (2)
```

a compiler error would result. The reason is similar to the issue discussed previously as with the `print_this(.)` function, except the `std::log(.)` function, rather than being a function template, is overloaded for several different floating point types. Referring back to our naive version of `transform`, the `double` type would not would not be recognized as desired overload until the actual `fcn` function call inside the `transform` algorithm:

```
*begin_dest = fcn(*begin_src);          // (1)
```

That is, the call site is the `transform` instance being generated, which takes place before `f` is called. The knowledge you're hoping the compiler would use is only known when `fcn` is called above in line (1), but at the call site (2) it is just one option among many, and the compiler cannot guess the programmer's intent at this stage.

One way to avoid this ambiguity is again to wrap `std::log(.)` inside a lambda, but which explicitly sets its argument type as `double`:

```
transform(dbl_vals.begin(), dbl_vals.end(), dbl_dest.begin(), [](double x) { return sqrt(x); });
```

This will be the case later when we use the `transform` algorithm and `std::log(.)` to calculate log returns from a set of asset prices. This will also be an issue with any STL algorithm when the intent is to use a `<cmath>` function as its auxiliary function.

Function Objects as Auxiliary Functions

Suppose we want to apply a quadratic function with arbitrary coefficients to the elements of a sequence using an STL algorithm. By implementing the quadratic as a stateful functor and storing the coefficients as member variables, the coefficient-based computation can be expressed as an auxiliary function, something that would be unpleasant to do with something stateless such as a standalone function.

```
class Quadratic
{
public:
	Quadratic(double a, double b, double c):
		a_{ a }, b_{ b }, c_{ c } {}

	double operator()(double x) const
	{
		return (a_ * x + b_) * x + c_;
	}
private:

	double a_, b_, c_;

};
```

Next, if we want to apply a quadratic function to a `vector` of real values and store it in, say, a `deque` containing their images, all we need to do is construct a `Quadratic` object with its coefficients as mem-

bers, and then pass it as a function object into the `std::transform` algorithm. The result is straightforward:

```
Quadratic q{ 2.0, 4.0, 2.0 };

std::vector<double> v{-1.4, -1.3, -1.2, -1.1, 0.0, 1.1, 1.2, 1.3, 1.4};
std::deque<double> y;
std::ranges::transform(v, std::back_inserter(y), q);
```

Class Member Functions as Auxiliary Functions

You may also run into situations where you will need to invoke a member function on an object as an auxiliary function. The easiest way to do this is to simply wrap the member function in a lambda. For example, suppose in the `Quadratic` class, there were a `value(.)` function rather than a functor (`operator ()`) to evaluate the function for a given value of `x` :

```
class Quadratic
{
public:
	// . . .

	double value(double x) const
	{
		return (a_ * x + b_) * x + c_;
	}

	// . . .

};
```

In this case, you can just define a lambda taking in the `Quadratic` instance `q` in its capture by reference, thus preserving its state, holding the same coefficients `a_`, `b_`, and `c_`, and calling its `value(.)` member function on each value in the container `v`:

```
Quadratic q{ 2.0, 4.0, 2.0 };

std::vector<double> v{-1.4, -1.3, -1.2, -1.1, 0.0, 1.1, 1.2, 1.3, 1.4};
std::deque<double> y;

auto quad = [&q](double x) -> double
{
      return q.value(x);
};

std::ranges::transform(v, std::back_inserter(y), quad);
```

Algorithms not (necessarily) Requiring Auxiliary Functions

There are additional STL algorithms available that locate particular elements in a container, sort the elements, identify unique elements, and copy or move elements to a separate and possibly different container type. In these cases, an auxiliary function is not necessarily required. However, it should be noted that some of these might be special cases of more general algorithms with an `_if` extension that will use a predicate. As an example, compare the `std::count` algorithm with the more general `std::count_if`, the latter of which we used in the example at the outset of the section on algorithms:

```
vector<int> v{ 1, 2, 3, 4, 2, 5, 2, 6, 7 };     // three occurrences of value 2

// n0 == 3 (no auxiliary function)
int n0 = std::count(v.begin(), v.end(), 2);
```

```
// n1 == 3 (equivalent result by taking in a predicate)
int n1 = std::count_if(v.begin(), v.end(), [](int n) { return n == 2; });
```

Another example is `std::find`, a special case of `std::find_if`, which will be presented shortly. We will start, however, by examining a few cases where an auxiliary function is not required, namely locating maximum and minimum elements of a container, and sorting elements in a container, which is also useful for filtering out non-unique elements.

Maximum and Minimum Elements

The algorithms here are pretty straightforward, with no auxiliary functions required, but you do need to keep in mind each algorithm returns an *iterator* to the maximum and minimum element positions, *not the actual value* itself:

```
std::vector<int> v = { 6, 7, 3, 5, 4, 1, 2, 9, 8 };
auto max_elem = std::max_element(v.begin(), v.end());
auto min_elem = std::min_element(v.begin(), v.end());
```

To obtain the actual values, you need to dereference the iterators, viz:

```
cout << format("max = {}, min = {}", *max_elem, *min_elem);
```

The range versions are again more to the point:

```
max_elem = std::ranges::max_element(v);
min_elem = std::ranges::min_element(v);
```

One important point here is neither of these algorithms modifies either the contents or the order of the original container. That is not the case for the next two examples.

Sorting and Unique Values

Two more tasks sometimes found in financial programming are sorting elements and determining the unique values. For associative containers `map` and `set`, these are automatically enforced by the objects themselves (more precisely, by default, this is enforced by operator `<` applied to the key elements), but for sequential containers, the algorithms `std::sort` and `std::unique` can be used.

Sorting in increasing order is easy to do. Examples of both iterator and ranges versions are shown here:

```
std::deque<int> dq{ 6, 7, 3, 5, 4, 1, 2, 9, 8 };
std::sort(dq.begin(), dq.end());

// With ranges:
std::vector<int> dq_ranges{ 6, 7, 3, 5, 4, 1, 2, 9, 8 };
std::ranges::sort(dq_ranges);
```

More generally, sorting according to other criteria is also easy to do once one understands the basics of algorithms and lambda expressions. Sorting can be done in decreasing order by now *including* the predicate auxiliary function as shown. Examples of both the pre-C++20 and ranges versions are provided:

```
std::deque<int> dq_aux{ 6, 7, 3, 5, 4, 1, 2, 9, 8 };
std::vector<int> dq_ranges_aux{ 6, 7, 3, 5, 4, 1, 2, 9, 8 };

// sort in decreasing order
auto le = [](int a, int b) { return b < a; };
```

```
std::sort(dq_aux.begin(), dq_aux.end(), le);
std::ranges::sort(dq_ranges_aux, le);
```

Note that the `sort` algorithm modifies the order of the elements, so it is considered a *modifying* algorithm.

Determining the unique values in a container, however, is a bit more complicated. First, in order for `std::unique` to work, the elements need to be in sorted order first.

```
#include <iterator>             // for std::distance(.) -- see below

//. . .

std::vector<int> u{ 5, 3, 9, 9, 8, 10, 15, 20, 0, 20, 15, 3, 6, 9, 12};
std::vector<int> save_u = u;       // Take a backup (will this use momentarily)
std::ranges::sort(u);
```

The `unique` algorithm will rearrange a sorted container so that the unique elements are sorted up front and then followed by the remaining redundant values. It returns an iterator to the position of the first redundant element.

```
auto first_redundant = std::unique(u.begin(), u.end());
for (int k : u) { print_this(k); }
cout << std::format("\nNumber of non-redundant elements: {}\n\n",
      std::distance(u.begin(), first_redundant));
```

The screen output will show these results. The first redundant value can be seen here, namely the value of `12` following the last unique value, `20`. This second occurrence of `12` is pointed to by the

`first_redundant` iterator returned by `std::unique`. The `std::distance(.)` function, declared in the `<iterator>` Standard Library header, will return the number of positions between two iterators `u.begin()` and `first_redundant` **{15}**.

```
(Output)
 0 3 5 6 8 9 10 12 15 20 12 15 15 20 20
Number of non-redundant elements: 10
```

This means we need to get rid of the redundant elements if the result is to be truly unique. This can be done by applying the `erase()` member function on the container, starting from the first redundant position through the `end` of `u`:

```
u.erase(first_redundant, u.end());
for (int k : u) { print_this(k); }
```

Now, `u` contains only its unique elements:

```
(Output)
0 3 5 6 8 9 10 12 15 20
```

It is possible to consolidate the steps above, which would lead to the same result.

```
u = save_u;                         // Restore from backup
std::ranges::sort(u);     // Ranges version of the sort algorithm
x.erase(std::unique(u.begin(), u.end()), u.end());
```

Searching for Elements in Containers

Programs will often need to search for a value in a container at least on occasion. As with most algorithms, it is not necessarily difficult to write a loop that does this correctly, but there are a number of optimizations that can be performed based on the types involved that no one actually takes the time to write, but that the providers of your Standard Library will have implemented for you.

And of course, using STL algorithms means you do not need to think about the microdetails of writing a loop, and you can concentrate instead on what you are trying to find.

With unsorted data, the two best-known algorithms made available to you would be:

- `std::find` which returns an iterator to the first occurrence of a value in the sequence. Comparisons are made with `operator ==` on the elements and the value you are trying to find, so this would not be the best option for floating point values
- `std::find_if` which returns an iterator to the first element for which a predicate yields `true` in the sequence

These algorithms return the end of the sequence when no element matching the search criterion is found (remember, this is the position one past the last element).

A usage example for each algorithm is shown here:

```
vector<int> ints{ 747, 377, 707, 757, 727, 787, 777, 717, 247, 737, 767 };
int n = 757;

// No auxiliary function:
auto ipos = std::find(ints.begin(), ints.end(), n);
if (ipos != ints.end())
{
```

```
        cout << format("Found value {} at index {}\n",
              n, std::distance(ints.begin(), ipos));
}


vector<double> reals{0.5, 1.6, -2.3, 0.85, -3.2, 2.5, 1.8, -0.72};
// Look for the first occurrence of a negative real value x in reals.
// An auxiliary function is employed in the case of find_if:
auto rpos = std::find_if(reals.begin(), reals.end(), [](double x)
        {
              return x < 0.0;
        });

if (rpos != reals.end())
{
        std::cout << std::format("First negative value is {}\n", *rpos);
}
```

The `std::distance(.)` function, in this case, will return the number of positions between the first element of `ints`, and the first matched value. In the second case, the first negative value is returned:

```
(Output)
Found value 757 at index 3
First negative value is -2.3
```

A `ranges` version is also available, for example:

```
ipos = std::ranges::find(ints, n);
if (ipos != ints.end())
{
```

```
        cout << format("Found value {} at index {} (with ranges version)\n",
            n, std::distance(ints.begin(), ipos));
    }
```

`std::find` and `std::find_if` perform a linear search through the sequence, and as such will perform a number of checks that (on average) grows linearly with the size of the input.

If data within your container is sorted, and you are trying to know if a given value is in the container actually contains a value—not where it is—you can also use `std::binary_search`. Note that this algorithm returns a `bool`, not an iterator, and as such it is more limited than its linear counterparts seen previously, but it is also much faster, requiring at most $O(log_2 n)$ comparisons for a container of `n` elements. Using the same `ints` container as before, an example is as follows:

```
// Sort the data first:
std::sort(ints.begin(), ints.end());

if (std::binary_search(ints.begin(), ints.end(), n))
{
    cout << std::format("Found value {}\n", n);
}
```

As the `std::binary_search` algorithm requires a sorted sequence, and as sorting takes time, you will get more from this algorithm if your data does not change often and you search through it frequently.

NOTE

Remember that iterator pairs in C++ form a $[$ `begin` , `end` $)$ half-open range. For that reason, algorithms that return an iterator (e.g.: `std::find` , `std::find_if` , `std::max_element` , etc) return the `end` iterator when no element is found:

```
bool contains(const std::vector<int> &v, int value)
{
  return std::find(v.begin(), v.end(), value)
         != v.end();
}
```

Also note that due to that fact, given iterators `b` and `e` such that $[$ `b` , `e` $)$

Copying and Moving Elements

Sometimes, you will want to copy or move elements from a container to another. There can be many reasons for this, but two common cases are:

- Wanting to use a container with different strengths than your usual preferred tool, e.g.: you usually use a `std::vector` but will need to perform a number of operations at the front for a while, and you think using a `std::deque` instead will be beneficial
- Wanting to make a copy of existing data in order to modify it without altering the original, etc.

There are many ways to achieve the objective of copying or moving elements from one container to another to cover in this book, so what follows is simply an overview.

The simplest way to make a copy of a container is to use its copy constructor, of course. C++ distinguishes itself from many other popular languages that claim to be object-oriented by giving programmers direct access to objects. Consider the following:

```
std::vector<int> v{ 1, 2, 3, 4, 5 };
std::vector<int> w = v;       // copy constructor (w did not exist before)
std::transform(w.begin(), w.end(), w.begin(), [](int n) { return -n; });
for(int n : v) // 1 2 3 4 5
{
  std::cout << n << ' ';
}
std::cout << '\n';
for(int n : w) // -1 -2 -3 -4 -5
{
  std::cout << n << ' ';
}
std::cout << '\n';
// ...
```

This will modify elements in `w` but have no effect on the original elements of `v`, as both are distinct objects.

Along with the copy constructor is the assignment operator, which clears the object on the left hand and replaces its contents with a copy of the elements on the right hand, but it will only work with STL containers of the same type containing elements of the same type:

```
// vector a is populated with data:
a.push_back(5);
a.push_back(4);
a.push_back(3);
a.push_back(2);
a.push_back(1);
```

```
// a is cleared, then makes a copy of v by assignment, so a is now { 1, 2, 3, 4, 5 }
a = v;
```

Note that assignment applies to the act of *replacing* the contents of an existing object, whereas construction happens when the object comes into existence.

Containers of the same type and holding the same type of elements can also be moved easily. For example:

```
vector<int> u = std::move(w);
```

Just remember (from Chapter Two) that a move should only be done if `w` is no longer needed.

Now, what if you wished to temporarily place elements of `v` in a `deque`, to take advantage of manipulating elements at the front of the container? In this case, neither the copy constructor nor the move constructor would apply. That is, none of the following would compile:

```
// Will not compile!
vector<unsigned> uv = v;          // v is vector<int>, not vector <unsigned>
std::deque<int> dc = v;                  // v is a vector, dc is a deque
std::deque<int> dm = std::move(u); // v is a vector, dc is a deque
```

Instead, if you want to copy the elements of a container into a different type of container, and/or into a container whose elements are of a different type, STL containers support sequence constructors which take a pair of iterators as argument. Note that both versions of `begin()` and `end()` —as container member functions, and their standalone alternatives—are valid:

```
vector<unsigned> uv(v.begin(), v.end());
std::deque<int> dc(std::begin(v), std::end(v));
std::list<unsigned> list_us(v.begin(), v.end());
```

To move container elements under similar situations—different element types and/or different container types—the `std::move` algorithm moves elements from a source sequence to a destination determined by an iterator. There is also a ranges version:

```
vector<unsigned> vm;
std::deque<unsigned> dm;
std::list<unsigned> list_usm;

std::move(v.begin(), v.end(), std::back_inserter(vm));
std::ranges::move(vm, std::back_inserter(dm));
std::ranges::move(dm, std::back_inserter(list_usm));
```

Remember that here as well, after the original elements are moved, they are left in a valid-yet-unspecified state and should not be reused unless reassigned to.

There are also analogous `std::copy` and `std::ranges::copy` algorithms, but you will probably find the earlier sequence constructors preferable.

Both `copy` and `move` can be applied to nontrivial class types. For example, we could first copy `Fraction` objects stored in a `vector` to a `deque`:

```
std::vector<Fraction> vec_fracs{ { 1, 2 }, { 5, 8 }, { 3, 4 }, { 2, 3 }, { 1, 9 } };
std::deque<Fraction> dq_fracs(vec_fracs.begin(), vec_fracs.end());
```

We could also move `Fraction` objects stored in a `deque` to a `list`:

```
std::list<Fraction> list_fracs;
std::ranges::move(dq_fracs, std::back_inserter(list_fracs));
```

NOTE

There is one caveat in this case. We are OK applying the `move` algorithm here because `Fraction` has a default move constructor that is `noexcept`, in line with the assumptions for this book, as noted earlier. In general, however, when applying it to a container holding elements of a nontrivial class type, you should ensure that said class has a move constructor declared `noexcept`. This is especially the case where a move constructor is user-defined.

Numeric Algorithms

A separate set of algorithms that perform mathematical operations such as summations and dot products are also available in C++. The algorithms are defined in a different Standard Library header, `<numeric>`, rather than `<algorithm>`. Those that can be particularly useful are as follows:

- `std::iota`: Generates a sequence of incremented values
- `std::accumulate`: Computes the sum of elements in a container
- `std::inner_product`: Computes the dot product from two containers
- `std::adjacent_difference`: Computes the difference of each element and its predecessor
- `std::partial_sum`: Computes the cumulative sums at each element in a container

The first algorithm, `iota`, is useful for testing, or perhaps generating a fixed schedule of time increments. In the next two two cases, the result will be a scalar value. For the remaining algorithms, the result will also be a container, either the original with elements replaced with modified values, or a separate container in which the differences or partial sums are inserted.

It should also be noted that the last four algorithms have generalizations beyond their default behavior listed here. Some of these variations will be presented in this section.

NOTE

As mentioned above, `numeric` algorithms have not yet been updated with their `range` counterparts as of C++20.

Generating Incremented Values with `std::iota`

The `iota` algorithm takes a specified value, increments it, and places it in successive elements of a container, up to the size of the container. For this reason, the container size needs to be specified at its construction.

```
#include <numeric>
//. . .

vector<int> v(6);
vector<double> w(6);

std::iota(v.begin(), v.end(), 101);
std::iota(w.begin(), w.end(), -2.5);
```

The first example will start with 101, and then increment it and successive values using the `++` operator, and populate the vector `v` with the results. In the second case, the starting value is -2.5. The results, if `v` and `u` are output to the console, would be

```
(Output)
101 102 103 104 105 106
```

```
-2.5 -1.5 -0.5 0.5 1.5 2.5
```

This is probably not all that useful for practical financial applications, but it does make initializing containers for testing more convenient, as opposed to manually typing out each value:

```
vector<int> v{101, 102, 103, 104, 105, 106};
vector<double> w{-2.5, -1.5, -0.5, 0.5, 1.5, 2.5};
```

`std::iota` could also be used for generating time increments, eg years, months, days, hours, or minutes. For example, to generate a time period of 30 days in the month of November, we could write:

```
vector<unsigned> days_in_november(30);
std::iota(days_in_november.begin(), days_in_november.end(), 1);

// days = 1, 2, . . ., 30
```

`std::accumulate` for Sums and Generalized Accumulations

The default behavior of `accumulate` simply sums all elements in a sequence, plus an initial value. The type of the result of that computation will be the type of that initial value, so choose it accordingly. Using `v` and `w` as in the previous section, the sum of these numerical elements can be computed in a one-liner:

```
int sum_of_ints = std::accumulate(v.begin(), v.end(), 0);
// 0 + 101 + 102 + 103 + 104 + 105 + 106 == 621
```

```
double sum_of_reals = std::accumulate(w.begin(), w.end(), 0.0);
// 0.0 + -2.5 + -1.5 + -0.5 + 0.5 + 1.5 + 2.5 == 0.0
```

One gotcha to be aware of, however, is how the type of the zero constant is interpreted by the compiler. For example, adding the integral typ `0` to the `double` elements in `sum_of_reals` can result in incorrect results.

Other binary operations can also be used. For example, to compute the product of all the elements, use an initial value of one for the multiplicative identity, and the `<functional>` operator `std::multiplies`. Note that the constant is now the multiplicative identity `1.0`:

```
#include <functional>        // std::multiplies, etc
//. . .

auto prod = std::accumulate(w.begin(), w.end(), 1.0, std::multiplies<double>());
```

This returns

$1.0 \times -2.5 \times -1.5 \times \dots \times 2.5 = -3.515625$

Element-by-element subtraction, and division are also available in the form of `std::minus<T>` and `std::divides<T>`, respectively. Addition can be explicitly indicated if desired with `std::plus<T>`.

In addition, `<functional>` representations of modulus, negation, and inequality comparisons can also be employed. For a full list, you can refer to cppreference.com **{16}**, and for detailed descriptions and examples, the Josuttis (Standard Library 2E) text is again an excellent source **{17}**.

Computing Dot Products and Generalizations with `std::inner_product`

The default behavior of the `inner_product` algorithm computes the dot product of two containers. Using the integer `v` vector as before, with a second vector of six elements defined by

```
std::vector<int> u(6);
std::iota(u.begin(), u.end(), -3);        // -3, -2, . . ., 2
```

the dot product can be easily computed as follows:

```
int dot_prod = std::inner_product(v.begin(), v.end(), u.begin(), 0 );
```

The value of zero in the last argument position is again an initial value that is added to the dot product, giving us -293.

The `inner_product` algorithm can also be abstracted for any two operations. Again using the same real vector `w` defined previously, and a second vector `y` with six elements from 1.5 to 6.5 (incremented by one, generated with `std::iota`), suppose we want to calculate the sum of the element by element differences:

$(-2.5 - 1.5) + (-1.5 - 2.5) + \dots + (1.5 - 6.5)$

We can use `inner_product` with the addition and subtraction operators appended in its argument. Note that the first operator refers to the outside addition operation, and the second to the inside subtraction operation.

```
vector<double> y(w.size());              // 6 elements
std::iota(y.begin(), y.end(), 1.5);
```

```
double sum_diffs = std::inner_product(w.begin(), w.end(), y.begin(), 0.0,
      std::plus<double>(), std::minus<double>());
```

This yields

$-4 +(-4) \dots + (-4) = -24$

Pairwise Differences and Generalizations with `std::adjacent_difference`

For a container with elements, say

$\{x_0, x_1, x_2, \dots, x_{n-1}, x_n\}$

the `std::adjacent_difference` algorithm by default will compute

$y_i=x_i-x_{i-1}$,

for each $i = 1, \dots, n$, and place the results

$\{x_0,y_1,y_2, \dots,y_{n-1},y_n\}$

in either a new container or in the original by replacing the elements found there. Note that the size of the differenced result is the same as the original, with x_0 remaining in the front position. For this reason, you might consider using a `deque` as the target container in order to pop x_0 off the front of the container and store only the differenced values:

```
std::vector<int> v{ 100, 101, 103, 106, 110, 115, 121 };
std::deque<int> adj_diffs;
```

```
std::adjacent_difference(v.begin(), v.end(),
      std::back_inserter(adj_diffs));          // adj_diffs = 100 1 2 3 4 5 6

adj_diffs.pop_front();   // adj_diffs now = 1 2 3 4 5 6 (differenced values only)
```

As with the `accumulate` algorithm, it is possible to specify an operation other than subtraction on each pair in the container. The form is also the same, explicitly indicating the binary operation, for example addition, as the auxiliary function. In the code snippet below, the results replace the original elements of `adj_sums`:

```
std::deque<double> adj_sums{ -2.5, -1.5, -0.5, 0.5, 1.5, 2.5 };
std::adjacent_difference(adj_sums.begin(), adj_sums.end(),
      adj_sums.begin(), std::plus<double>());
adj_sums.pop_front();        // adj_sums now = -4 -2 0 2 4
```

Compute Partial Sums of Finite Series with `std::partial_sum`

This algorithm does what it says, namely compute the partial sum for each prior sequence of elements in a container, and store the results in a different container. As with other numerical algorithms, the summation can be replaced by a different operator, such as multiplication that computes partial products.

An example of default partial sum calculations is shown here:

```
vector<int> z{ 10, 11, 13, 16, 20, 25, 31 };
vector<int> part_sums;
part_sums.reserve(z.size());
std::partial_sum(z.begin(), z.end(), std::back_inserter(part_sums));
// Result: part_sums  = 10 21 34 50 70 95 126
```

The `partial_sum` algorithm can also be abstracted for other operations, such as partial products. Note in this case, the alternative approach (also valid for the prior examples) of allocating the memory for the target `vector` is used, and instead of `std::back_inserter(.)`, we can just use the iterator `part_prods.begin()`. The usual caveats regarding performance would apply, however, for larger containers, and especially also where the contained type is a class (with default constructor).

```
vector<int> part_prods(z.size());
std::partial_sum(z.begin(), z.end(), part_prods.begin(), std::multiplies<int>());
// Result: part_prods = 10 110 1430 22880 457600 11440000 354640000
```

Some Financial Applications of Numeric STL Algorithms

As examples of where we might apply some of these numeric algorithms, consider two use cases. The first will be computing the daily Volume-Weighted Average Price (VWAP) of an equity, and for the second, we will look at two possible approaches for calculating the log returns from an equity or traded fund.

Daily VWAP is typically calculated by first dividing the trading day into equal time intervals, say N, and extracting the trading volume v_i for an equity. The total daily volume is thus the sum

$\sum\limits^{N}_{i=1} v_i$

To obtain the weighted average price, we would also need to take the sum of products of the volume over each interval times a specific price p_i over each interval, and divide by the total volume. That is,

$VWAP = \frac{\sum\limits^{N}_{i=1} v_i p_i}{\sum\limits^{N}_{i=1} v_i}$

where

$p_i = \frac{High + Low + Close}{3}$

over each time period i.

Suppose the observed trading volumes and average prices at the end of each 30-minute interval from 10:00 am to 4:00 pm are (for this exercise) placed in the following `vector` containers `v` and `p`, respectively:

```
vector v
{
      376000, 365000, 344000, 346000, 345000, 336000, 335000,
      339000, 340000, 340000, 343000, 367000, 37400
};

vector p
{
      208.59, 206.93, 207.75, 209.21, 208.58, 208.63, 207.92,
      208.87, 208.16, 209.49, 208.53, 209.12, 209.05
};
```

The sum comprising the total daily volume can be easily obtained by utilizing the `accumulate` algorithm:

```
double daily_volume = std::accumulate(v.cbegin(), v.cend(), 0.0);
```

And, the value for the numerator is also a one-liner using the `inner_product` algorithm:

```
double raw_wgt_price = std::inner_product(v.cbegin(), v.cend(), p.cbegin(), 0.0);
```

The daily VWAP value is then just the quotient

```
double vwap = raw_wgt_price / daily_volume;
```

whose result is \$208.48.

The log returns for a set of equity or ETF prices

$$\{S_0, S_1, \cdots, S_{n-1}, S_n\}$$

is comprised of the set of natural logs of the successive quotients:

$$r_i = log \frac{S_i}{S_{i-1}}, i = 1, \cdots, n$$

There are two approaches we can take. First, we could compute each quotient and apply the natural log function. Alternatively, we could apply the natural log function to each price, and then take the differences of each successive pair:

$$r_i = log\left(S_i\right) - log\left(S_{i-1}\right), i = 1, \cdots, n$$

Suppose we have a set of daily share prices for a particular listing stored in a `vector`:

```
vector<double> prices
{
    25.5, 28.0, 30.5, 31.0, 27.5, 31.0, 29.5, 28.5, 37.5,
    33.5, 25.5, 31.5, 26.5, 29.5, 32.5, 34.5, 28.5, 35.5,
    28.5, 29.0, 32.0, 23.5, 27.5, 33.5, 28.0, 28.0, 32.5,
    31.5, 29.0, 33.0, 32.5, 29.5, 34.5
};
```

For the first method, we can apply the `adjacent_difference` algorithm, but with the division operation applied to each pair, and the result stored in the same `vector` of `prices`. Once this is done, we can apply the natural log function to each, and place the results in a `deque`, the reason being we easily can pop the first element(the original first price, not a quotient) off of the front of the container:

```
std::adjacent_difference(prices.begin(), prices.end(), prices.begin(), std::divides<double>());
std::deque<double> log_rtns(prices.size());
std::ranges::transform(prices, log_rtns.begin(), [](double x) {return std::log(x); });
```

The resulting contents in the `log_rtns` container are then:

```
(Output)
0.0935 0.0855 0.0163 -0.12 0.12 -0.0496 -0.0345 0.274
-0.113 -0.273 0.211 -0.173 0.107 0.0968 0.0597 -0.191
0.22 -0.22 0.0174 0.0984 -0.309 0.157 0.197 -0.179
0 0.149 -0.0313 -0.0827 0.129 -0.0153 -0.0968 0.157
```

NOTE

You might notice we used a lambda inside the `transform` algorithm and not the `std::log(.)` function itself. This is because writing

```
std::ranges::transform(prices, log_rtns.begin(), std::log);
```

would result in a compiler error. The reason for this is essentially the same as the problem with auxiliary function templates we saw earlier, except that the `std::log(.)` function is overloaded for different numerical types, as opposed to being a function template. This again results in ambiguity, as the type is not known at the call site. Wrapping it in a lambda first, as shown in the example, will give us the desired results.

For an alternative method, we could go back to the original `prices` container—before the adjacent quotients have been computed—and calculate the natural log of each price. Then, we can apply the default version of the `adjacent_difference` algorithm that will take the difference of each log value. The results are again stored in a `deque`, so that we can remove the first price value and keep the returns alone.

```
std::ranges::transform(prices, prices.begin(), [](double x) {return std::log(x); });
std::deque<double> log_rtns;
std::adjacent_difference(prices.begin(), prices.end(), std::back_inserter(log_rtns));
```

The results will be the same as before.

Views, Range Adaptors, and Functional Programming

In addition to the advantages provided by ranges in terms of clarity, they also provide a mechanism for composing a sequence of algorithms without the overhead from copying data and generating temporary

container objects. This is facilitated by a functional approach using *pipelines* of lightweight forms of ranges called *views*, sometimes referred to more specifically as *range views* **{18}**, as the term *view* has a more general meaning, as will be seen in Chapter Seven.

A view in the general sense allows certain operations to be performed using the elements of a container without incurring the cost of copying data from the container. In other (technical) words, this means a view is said to be *non-owning*. Additional properties of a view in C++ are as follows:

- A view is copyable
- A view acts as a reference to some data, in that if you copy the view, both the copy and the original will conceptually point to the same data
- Creation, copy, and destruction of a view are all $O(1)$ (“cheap”) in the number of elements to which the object refers
- If a view refers to data, it does not allocate, construct, destroy, or deallocate the data

Also in general, a non-owning entity such as a view can be mutating or non-mutating (read-only), and you can use it as long as the underlying entity (what you are viewing) exists.

Returning to range views in particular, what makes them interesting and more useful than it would seem at first glance is their functional behavior, allowing composition of multiple functions in a manner similar to piping in Linux scripting. These pipelines of operations are syntactically similar to piping in Linux scripting and are performed *lazily*. This means you can combine operations on views in such a way that these operations will be deferred and actually performed only when needed. This tends to lead to fewer loops, less copying of data, and overall better throughput.

Further enhancements are set for release in C++23, but the functional programming direction of Standard Library-related programming is quite clear with the content so far in C++20. This opens up some exciting new capabilities for C++ quantitative development that have up to now been present in other popular

languages used for financial modeling, such as R, Python, Haskell, OCaml, and F#. In the next section, we will work through an example.

A Line-by-Line Introduction of Range Views

The following introduction is very much a "textbook" example similar to those you can find in recently published books and on Internet references, but we will go into perhaps a bit more detail by first examining it as a series of separate tasks and explaining each step. After that, we will see how the same tasks can be chained together as a composition of several functions.

The views we will examine are as follows.

A *range adaptor* is used to create a range view. Four examples of which are noted here:

- `std::views::take` - Takes the first `n` elements of a view and discards the rest
- `std::views::filter` - Filters a set of elements according to a predicate
- `std::views::transform` - Modifies elements of a view based on an auxiliary function (this only modifies the values on which the computation is being performed, not the original values in the source range)
- `std::views::drop` - Removes the first `n` elements of a view

These may look similar to STL algorithms, but their behavior is different in that they provide views of subsets of the data without generating copies of the data or new instances of STL containers.

Going forward, in order to lighten the terminology, the term *view* will specifically refer to a *range view* for the remainder of the chapter, unless otherwise noted. Like ranges, views require inclusion of the `<ranges>` Standard Library header, and they are scoped with the either the `std::views` or `std::ranges::views` namespace, depending on your preference, as they are aliases of each other.

First, we will start with a `vector` of real numbers, and then apply the first range adaptor, `std::views::take`.

```
#include <ranges>
//. . .

std::vector<double> w(10);
std::iota(w.begin(), w.end(), -5.5);

auto take_five = std::views::take(w, 5);
for(double x : take_five) { print_this(x); }
```

Starting with a vector of 10 elements, $-5.5, -4.5, \dots, 4.5$, application of this range adaptor will yield the `take_five` view of the first five elements. Note that no copy of the elements has occured: `take_five` is at this point an object which can be traversed to obtain these elements, but it does not own the elements (they are owned by `w`). Elements controlled by the view `take_five` are then displayed on the screen, leaving us with

$$-5.5, -4.5, -3.5, -2.5, -1.5$$

The key idea to understand here is `take_five` does not *contain* copies of these five values, but rather, as its type suggests, it allows us to *view* these elements. Furthermore, application of the `take` range adaptor will not modify the original vector `w`. In general, a view is non-owning, and a range adaptor is non-mutating. This is conceptually similar to a database view that displays the results of applying a query, but without physically containing any data rows.

A related point is that the return type of a view can be a very long and ghastly template type. It is not a container; rather, it is something that, when iterated over, eventually yields values resulting from whatever combination of operations that view represents. For example, the type for `take_five` would be

`std::ranges::take_view<std::ranges::ref_view<std::vector<double, std::allocator<double>>>>`

Thankfully, the `auto` keyword comes to the rescue here. We need not care what the particular type is in order to get the job done, only that it is a view.

In the next step, applying the `filter` range adaptor on `take_five` will lead to another view (a view of a view!) yielding only elements that satisfy the conditions of a predicate. For example, to choose those elements in `take_five` that are strictly less than -2, a simple lambda expression can be applied:

```
auto two_below = std::views::filter(take_five, [](double x) { return x < -2.0; });
```

When you observe the "contents" of `two_below`, you will see this now leaves us with:

$$-5.5, -4.5, -3.5, -2.5$$

Next, we will square each value of this new view by applying the `transform` range adaptor. This is the view equivalent of the `transform` algorithm, and the mechanics are the same by applying an auxiliary function -- again in this case a lambda -- that will return the square of each element.

```
auto squares = std::views::transform(two_below, [](double x) { return x * x; });
```

When observing the values provided by this new view, we now have

$$30.25, 20.25, 12.25, 6.25$$

Finally, we will remove the first two elements of `squares` by applying the `drop` range adaptor, essentially the complement of the `take` adaptor used at the outset.

```
auto drop_two = std::views::drop(squares, 2);
```

This leaves us with:

12.25, 6.25

Restating the code example as a whole, and writing a lambda with a *range*-based `for` loop to print each interim result, we now have:

```
#include <ranges>
//. . .

auto take_five = std::views::take(w, 5);
auto two_below = std::views::filter(take_five, [](double x) {return x < -2.0; });
auto squares = std::views::transform(two_below, [](double x) {return x * x; });
auto drop_two = std::views::drop(squares, 2);

auto print_range = [](auto rng) -> void
{
	for (double x : rng) { print_this(x); }
};

print_range(take_five);
print_this("\n");
print_range(two_below);
print_this("\n");
print_range(squares);
print_this("\n");
print_range(drop_two);
```

The output, taken altogether, is then:

```
(Output)
-5.5 -4.5 -3.5 -2.5 -1.5
-5.5 -4.5 -3.5 -2.5
30.25 20.25 12.25 6.25
12.25 6.25

Resulting sum = 18.5
```

As we will see next, this can be written in a more elegant manner through a composition of views.

For now, this begs two questions you might have at this point:

- Can we use the results in an STL algorithm? The simple answer to this question is that containers generally are ranges, but ranges are not necessarily containers (in particular, views are *definitely* not containers). Thus, it is possible to use algorithms from the `std::ranges` namespace on containers, as we have seen previously, but the converse is less simple and in many cases counterproductive.
- Can we store the results in an STL container? The answer to this question is "of course, let's see how" and will be addressed shortly.

Chaining for Functional Composition

This is where views become very elegant and powerful. The example above, which was broken up into independent steps, can instead be expressed as a composition of range adaptors, with the result of one taken in as input to the next. This is done with the pipe operator ('|'), syntactically similar to Linux shell scripting methods.

```
auto drop_two = w | std::views::take(5)
      | std::views::filter([](double x) { return x < -2.0; })
      | std::views::transform([](double x) { return x * x; })
      | std::views::drop(2);
```

This construct makes no copy of the elements; what it does is build a view in which the elements will be the result of the sequence of transformations that have been composed together. Very cool.

Compare this with the previous step-by-step version, placed in a single code block:

```
auto take_five = std::views::take(w, 5);
auto two_below =
      std::views::filter(take_five, [](double x) { return x < -2.0; });
auto squares =
      std::views::transform(two_below, [](double x) { return x * x; });
auto drop_two = std::views::drop(squares, 2);
auto sum_result =
      std::accumulate(drop_two.begin(), drop_two.end(), 0.0);
```

The composed version eliminates the need for intermediate variable assignments and reintroduction of each as an argument in each subsequent step. This makes the logic clearer and the code cleaner.

Views, Containers, and Range-Based `for` Loops

As mentioned in the introduction, views only use data residing in a container without incurring a copy penalty, but there might be times when you will want to store the result independently in another container, for example to return the results to an interface to a database.

As views themselves are ranges, they can be used as range expressions in range-based `for` loops. A simple example is to copy the results from the `take_five` view into a `vector`, as shown here:

```
std::vector<double> u;
u.reserve(5);
for (auto x : take_five)
{
	u.push_back(x);
}
```

Going a step further, we can start with a container, such as `w` in the introductory example, and first apply the range adaptor that selects the first five elements to set up the view (a range expression) for the following range-based `for` loop. As a simple example here, we just output the results to the screen:

```
for (auto y : w | std::views::take(5))
{
	print_this(y);  // -5.5 -4.5 -3.5 -2.5 -1.5
}
```

Finally, multiple chained views can also be used as the range expression, for example combining `take` with `transform`:

```
for (auto y : w | std::views::take(5)
		| std::views::transform([](double x) { return x * x; }))
{
	print_this(y);  // 30.25 20.25 12.25 6.25 2.25
}
```

A `ranges::to` function that will enable implicit conversion from views to containers has been adopted for C++23, but in the meantime, the simplest way is to just use a range-based `for` loop as described here.

NOTE

The working codebase for next-generation versions of the Ranges library, *range-v3*, is available on GitHub **{19}**. The ReadMe.md file also contains useful information about the existing release currently in C++20, as well as links to the original proposals, related documentation, and videos of conference talks.

Parallel STL algorithms

Thanks to a new feature introduced in C++17, a large subset of STL algorithms can be instructed to run in parallel by simply setting an additional parameter in the algorithm argument. When running on modern multicore hardware, the performance improvements can be quite significant.

It will be more straightforward to present some meaningful examples, however, by having the ability to randomly generate values to populate large containers, but this involves a separate discussion of random number distributions that will be covered in the next chapter. For this reason, this topic will be deferred until then.

Summary

STL container classes are broadly classified as sequential or associative containers, although in reality containers such as `unordered_map` are not actually associative and could arguably be categorized separately as "unordered containers". Still, even among all of the containers available, a set of common mem-

ber functions allow for a degree of interchangeability, as well as when used as arguments in STL algorithms.

The following member functions are common to all STL containers. This is not an exhaustive list, but it shows methods that you will very likely encounter and find useful in practice:

Table 4-1. Member Functions that Apply to all STL Containers

Member Function	Description
`begin()`	returns an iterator that points to the position of the first element
`end()`	returns an iterator that points to the first position *after* the last element
`cbegin()`	returns a `const` iterator that points to the position of the first element
`cend()`	returns a `const` iterator that points to the first position *after* the last element
`size()`	returns the number of elements in a container
`empty()`	returns `true` if a container is empty
`clear()`	clears all elements from a container

NOTE

The `begin()` and `end()` member functions will each return a non-`const` iterator if the container is non-`const`, and a `const` iterator if the the container itself is `const`. `cbegin()` and `cend()` will each return a `const` iterator if the container is non-`const`, and (trivially) as well for a `const` container. It is *not* the case that `cbegin()` and `cend()` are required for a `const` container.

Table 4-2. Member Functions that Apply to All Sequential Containers (`vector` , `deque` , `list` , `array`)

Member Function	Description
`front()`	accesses a reference to the first element of a non-empty container
`back()`	accesses a reference to the last element of a non-empty container

Table 4-3. Member Functions that Apply to `vector` , `deque` , and `array` Containers Only

Member Function/Operator	Description
`at(.)`	bounds-checked random accessor of the element at a specific position
`[.]`	same as `at(.)` , but without bounds checking

Table 4-4. Member Functions that Apply to `vector` , `deque` , and `list` Containers Only

Member Function	Description
`push_back(.)`	appends an element to the back of a container
`pop_back(.)`	removes the last element of a container

Table 4-5. Member Functions that Apply to `deque` , and `list` Containers Only

Member Function	Description
`push_front(.)`	adds an element to the front of a container
`pop_front(.)`	removes the first element of a container

Table 4-6. Member Functions that Apply to a `vector` Container Only

Member Function	Description
`reserve(n)`	reserves enough continguous memory to hold `n` elements but does not construct objects of the element type
`capacity(.)`	returns the number of elements that may be stored in contiguous memory

STL iterators provide the means to traverse the elements of an STL container. There is a variety of iterator types in the STL, as described in the section on STL iterators in the chapter, but usually in practice, the easiest way to access an iterator is to just call the generic `begin()` or `cbegin()` method on the container. Then, you can iterate across the container in either direction, and access the actual data in a particular position using the dereferencing `*` operator, similar to a pointer:

```
std::vector<int> x{1, . . ., 10};
std::map<unsigned, int> m{ {1, 100}, {2, 200}, . . ., {10, 1000} };

auto v_iter = x.begin();
auto m_iter = m.cbegin();

int v_elem = *v_iter;           // v_elem = 1
++v_iter;
++v_iter;
*v_iter = -3;                   // Third element is now -3
--v_iter;
v_elem = *v_iter;         // v_elem = 2


++m_iter;
++m_iter;
```

```
unsigned key = (*m_iter).first;          // key = 3
int val = (*m_iter).second;              // val = 300

// Alternatively can access as, say, [key_alt, value_alt] pair:
++m_iter;
auto [key_alt, value_alt] = *m_iter;
cout << format("key (alt) at 4th position = {}, val (alt) at 4th position = {}\n\n",
      key_alt, value_alt);
```

STL algorithms provide cleaner and efficient means of accessing, manipulating (eg ordering), and/or applying functions to each element in a container, as opposed to more error-prone user-defined multiline implementations in a `for` loop. Scott Meyers in fact sums this up nicely, stating that STL algorithms are usually going to be better options compared to hand-written loops for the following reasons **{20}**:

- ***Efficiency***: *Algorithms are often more efficient than the loops programmers produce*.
- ***Correctness***: *Writing loops is more subject to errors than is calling algorithms*.
- ***Maintainability***: *Algorithm calls often yield code that is clearer and more straightforward than the corresponding explicit loops*.

For algorithms expecting an auxiliary function, these arguments can be conveniently supplied as function objects and lambda expressions. Recalling the case using a lambda, the following shows the same auxiliary function can be with the `transform` algorithm but with different STL container types:

```
#include <algorithm>
// . . .

std::array<int, 5> a{ 1, 2, 3, 4, 5 };
std::deque<int> d{ 1, 2, 3, 4, 5 };
std::set<int> s{ 1, 2, 3, 4, 5 };
```

```
std::vector<int> v(a.size());

auto square_that = [](int x) {return x * x; };

std::transform(a.begin(), a.end(), a.begin(), square_that);
std::ranges::transform(d, d.begin(), square_that);
std::transform(s.cbegin(), s.cend(), v.begin(), square_that);

// Results:
// a = { 1, 4, 9, 16, 25 }
// d = { 1, 4, 9, 16, 25 }
// s = { 1, 2, 3, 4, 5 }          (s not modified)
// v = { 1, 4, 9, 16, 25 }        (v contains the squares of the elems of s)
```

A set of separate numeric algorithms is also available, such as `accumulate` and `inner_product`, which as we saw can be useful, for example, in computing the log returns from traded assets. These require inclusion of a separate header, `<numeric>`. Also, range versions of numeric algorithms are not yet available at the present time.

Finally, as part of the `ranges` addition to C++20, non-owning views can be used to extract, filter, and/or transform data in STL containers (and more generally ranges) without the overhead of copying the data. Furthermore, views similar to STL algorithms—such as `transform`—can be elegantly chained together in a functional programming approach, using the pipe operator (`|`) in a manner similar to Linux shell scripting. In addition, with operations deferred due to lazy evaluation, efficiency gains can be realized. A wider set of views covering similar functionality as that existing in STL algorithms is due for release in C++23.

References

{1} Bjarne Stroustrup, C++ in 2005, pp 5-8

{2} Scott Meyers, Effective STL, Introduction (*https://learning.oreilly.com/library/view/effective-stl/9780321545183/*)

{3} Core Guidelines, `noexcept`, *https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-noexcept*

{4} (See for reference): Bjarne Stroustrup, The C++ Programming Language (4E), *The strong guarantee for key operations*, Section 13.2, *Exception Guarantees*, (earning.oreilly.com/library/view/the-c-programming/9780133522884/ch13.xhtml#ch13lev1sec2)

{5} For a review of Standard Library exceptions, see Sec 4.3.1 of Josuttis, C++ Std Lib, 2E

{6} Josuttis, The C++ Standard Library (2E), Sec 3.1.3 *Uniform Initialization and Initializer Lists*

{7} Google C++ Coding Style Guidelines, constructor form for `std::vector` and other STL containers

{8} Herb Sutter and Andrei Alexandrescu, C++ Coding Standards, Item 76, Addison-Wesley (2004) (*https://learning.oreilly.com/library/view/c-coding-standards/0321113586/ch77.html*)

{9} Unordered map (*https://en.cppreference.com/w/cpp/container/unordered_map*) _

{10} Josuttis, The C++ Standard Library (2E), Sec 6.2, p 168 (*https://learning.oreilly.com/library/view/c-standard-library/9780132978286/ch06.html#ch06lev1sec2*)

{11} Peter James, Option Theory, Black-Scholes/Greeks

{12} Josuttis, C++ Std Lib, 2E, Sec 9.2 (*https://learning.oreilly.com/library/view/c-standard-library/9780132978286/ch09.html*)

{13} Stack Overflow: *https://stackoverflow.com/questions/27837893/how-to-count-the-number-of-a-given-value-in-a-c-map*

{14} Josuttis C++20, Ch 6

{15} *https://en.cppreference.com/w/cpp/iterator/distance*

{16} *https://en.cppreference.com/w/cpp/header/functional*

{17} Josuttis, C++ Std Lib (2E), Sec 10.2, p 486 (functional)

{18'} Range-v3 User Manual *https://ericniebler.github.io/range-v3/index.html#tutorial-quick-start*

{19} Eric Niebler, range-v3 Repository, *https://github.com/ericniebler/range-v3* (MOVED TO END OF SECTION)

{20} Scott Meyers, Effective STL, Ch 7, Item 43 (*https://learning.oreilly.com/library/view/effective-stl/9780321545183/ch08.html*)

Additional reference on exception handling: *https://en.cppreference.com/w/cpp/language/exceptions*, *Exception Safety*, item 2: *Strong exception guarantee*

Additional background/intro to views: **{Timur}** *https://timur.audio/how-to-make-a-container-from-a-c20-range*.

Related CppCon 2023 Talks (will be on YouTube by end of year):

Nicolai Josuttis, *Back to Basics: Iterators*, *https://cppcon2023.sched.com/event/1QtdZ*

Klaus Iglberger, *Back to Basics: Algorithms*, *https://cppcon2023.sched.com/event/1QtdQ*

Jeff Garland, *Effective Ranges: A Tutorial for Using C++2x Ranges*, *https://cppcon2023.sched.com/event/1QtgT*

Talk on STL algorithms from CppCon 2015 (or 2016)