Chapter 11
Iterators and the Standard Template Library

11.1 The Standard Template Library (STL)

The **standard template library (STL)** is a well-documented library that provides high-quality, fast implementations of data structures and algorithms that can be used to operate on data. All of these implementations can be found in public headers that are available for anyone to see.

The STL is the reason why you do not have to implement everything from scratch when you write a program! If you want to store your data in a dynamic array that grows with the size of the data, you do not have to implement the container; you can just `#include <vector>` and instantiate a `std::vector<>` in your program. If you want to use a doubly-linked list, you do not need to implement your own `Node` class and list methods; you can just `#include <list>` and instantiate a `std::list<>`. If you want to sort the contents of a container, you do not have to implement your own sorting function; you can just `#include <algorithm>` and call `std::sort()`. Because the STL is built for speed and was written by some of the best programmers around the world, you are guaranteed to have an efficient, bug-free implementation available to you when you use an STL container or algorithm.

What exactly does the STL provide? The STL consists of four primary components: containers, algorithms, functions, and iterators. The STL provides implementations of different *containers* that can be used to store and retrieve data in different ways. For example, vectors, lists, deques, stacks, queues, and priority queues are all containers that are provided by the STL. The STL also provides implementations of *algorithms* that can be used to operate on data in containers. If you want to use an algorithm in your program, there is a chance that it is already implemented in the STL! As mentioned, `std::sort()` can be called to sort a container of data, and other functions such as `std::min_element()` and `std::max_element()` can be used to find the minimum and maximum value in a range of data. Many STL algorithms can be found in the algorithm library, which can be accessed with `#include <algorithm>`. We will cover this library in a later section of this chapter.

In addition to algorithms, the STL also provides implementations for utilities (e.g., simple operations like `std::swap()` in the `<utility>` library), function objects (e.g., comparators such as `std::less<>` and `std::greater<>` that can be used to compare two values), and memory allocators (which will not be covered in this class).

The STL is very extensive, with entire books written on its contents. These notes will not go through every detail on all the different libraries that C++ offers. Instead, this chapter will contain a sample of certain STL concepts that you might see again in future classes or in your career. You will gain most of your exposure to the functionalities of the STL through programming experience and searching for resources online.

Remark: You will not be required to memorize everything from the STL in this class! Even experienced programmers have to look things up every once in a while. However, there are certain things that you should be comfortable with, such as iterators, pairs, and function objects. The other sections are just here as an optional reference for your coding assignments. If a concept was never mentioned in class, you will not need to understand it for this class (but it still might be helpful to know!).

11.2 Pairs

A **pair** is a data type that groups together two values and treats them as a single unit. The two objects in a pair may be of different types. Many different container classes rely on the pair construct to manage their elements (such as the `std::map<>` and `std::unordered_map<>` containers, which will be covered in later chapters). To instantiate a pair, you should declare an object of type `std::pair<T1, T2>`, where `T1` and `T2` are the types of the two objects in the pair.

The two values that make up a `std::pair<>` can be accessed using its public members, `first` and `second`. This is very similar to how a `struct` with two members would behave (in fact, a `std::pair<>` is implemented as a templated `struct` with two members `first` and `second`). In addition to storing two values together, pairs can also be copied, assigned, swapped, and compared. Thus, if you want to bind two data types together, a `std::pair<>` will allow you to do so without requiring you to redefine how simple operations work.

The following table lists some common pair operations. The list is not comprehensive, but it gives a sufficient overview of a few things you can do with pair objects.

`template <typename T1, typename T2>` `std::pair<T1, T2> p;`
Default constructor, initializes `p` as a pair with the default constructed values of types `T1` and `T2`.
`template <typename T1, typename T2>` `std::pair<T1, T2> p(const T1& v1, const T2& v2);`
Creates a pair `p` with two values, one of type `T1` and one of type `T2`, that are initialized to `v1` and `v2`.
`template <typename T1, typename T2>` `std::pair<T1, T2> p2(const std::pair<T1, T2>& p1);`
Copy constructor, initializes `p2` to be a copy of `p1`.
`p2 = p1;`
Assignment operator, assigns a pair `p2` to the value of `p1`.
`p1 == p2;`
Checks if both `first` and `second` are identical for two pairs `p1` and `p2`.
`p1 < p2;`
Checks if `p1` is less than `p2`, first by comparing their `first` values, and then by their `second` values if the `first` values are equal. This process works similarly for >, <=, and >=.
`p.first;`
The first element of the pair `p`.
`p.second;`
The second element of the pair `p`.
`std::make_pair(v1, v2);`
Creates a pair with `first` initialized to `v1` and `second` initialized to `v2`.

The `std::make_pair()` function can be used to construct a pair. A call to the function `std::make_pair(x, y)` constructs and returns a pair with `first` set to `x` and `second` set to `y`. For instance, the statement

```
std::pair<std::string, int32_t> myPair = std::make_pair("EECS", 281);
```

creates a pair with a `first` value of `"EECS"` and a `second` value of `281`. Curly braces may also be used for the same result:

```
std::pair<std::string, int> myPair = {"EECS", 281};
```

The `std::make_pair()` function can be used to pass a pair as an argument into a function. For example, consider the following function that takes in a `std::pair<std::string, int>` as a parameter:

```
int32_t return_class_difficulty(std::pair<std::string, int32_t> course);
```

You can call `std::make_pair()` in the function argument itself to build a pair and pass it in:

```
return_class_difficulty(std::make_pair("EECS", 281));
```

Curly braces can also be used to pass a pair into a function:

```
return_class_difficulty({"EECS", 281});
```

The `std::pair<>` also has built-in comparison methods. Two pairs are equal if and only if they have identical `first` and `second` values. When comparing pairs using < and >, the `first` value is compared first. If the `first` values of two pairs differ, the comparison between these `first` values determines which pair is lesser or greater. For example, the pair `{12, 45}` is *greater* than the pair `{10, 953}` because 12 is greater than 10. However, if the `first` values are equal, the comparison of the `second` values determines which pair is larger. For instance, the pair `{10, 45}` is less than the pair `{10, 953}`. Because these two pairs share identical `first` values, their `second` values are compared, and 953 is larger than 45.

11.3 Tuples

A pair groups together two values and treats them as a single entity. A **tuple** does the exact same thing, but with *any* number of elements. To use tuples and tuple operations in your program, you must `#include <tuple>`. To instantiate a tuple, create an object of type `std::tuple<>`.

A table of common tuple operations is shown below. This is not comprehensive, but it provides an overview of things you can do with tuple objects. Note that much of this behavior is analogous to that of a pair.

`template <typename T1, typename T2, typename T3, typename... OtherTypes>`
`std::tuple<T1, T2, T3, OtherTypes...> t;`
Default constructor, initializes a tuple `t` with the default constructed values of all its component types.

`template <typename T1, typename T2, typename T3, typename... OtherTypes>`
`std::tuple<T1, T2, T3, OtherTypes...> t(const T1& v1, const T2& v2, const T3& v3, OtherTypes...);`
Creates a tuple `t` with values of types `T1`, `T2`, `T3`, …, that are initialized to the values `v1`, `v2`, `v3` …

`template <typename T1, typename T2>`
`std::tuple<T1, T2> t(const std::pair<T1, T2>& p);`
Given a pair `p`, this initializes a two-element tuple `t` with the contents of `p`.

`t2 = t1;` Assignment operator, assigns the value of `t1` to `t2`.
`t1 == t2;` Checks if `t1` is less than `t2` by comparing their first values, then their second values, then their third values, and so on.
`std::make_tuple(v1, v2, v3, ...);` Creates a tuple that is initialized with all the values that are passed in as arguments.

Similar to pairs, you can create tuples using the `std::make_tuple()` function or curly braces. For instance, the following creates a tuple `t` that is initialized with the contents `"EECS"`, `281`, `"PI"`, and `3.14`:

```
std::tuple<std::string, int32_t, std::string, double> t = std::make_tuple("EECS", 281, "PI", 3.14);
```

Tuples also support copying, assignment, and comparison. The process of comparing tuples is similar to that of comparing pairs: the first component is compared first, followed by the second, then the third, and so on.

To access data members of a tuple, you can use the `std::get<>` method. Given a tuple `t`, `std::get<i>(t)` can be used to retrieve the component at position `i` in the tuple, using zero indexing. For the above tuple:

```
std::get<0>(t) == "EECS"
std::get<1>(t) == 281
std::get<2>(t) == "PI"
std::get<3>(t) == 3.14
```

A limitation with `std::get<>`, however, is that the value of the index `i` must be known at compile time. That is, you have to know what the value of `i` is *before* you compile the program. Passing an index in at runtime is not possible; thus, the following code would *not* compile:

```
for (int32_t i = 0; i < 4; ++i) {
  std::cout << get<i>(t) << '\n';  // does not compile!
} // for i
```

Furthermore, `std::get<>` checks if the index is valid before compiling. If you attempt to access a component of a tuple that does not exist, you will get a compile error.

Another interesting method is `std::tuple_cat()`. The `std::tuple_cat()` function allows you to concatenate two tuples together. As an example, suppose you had two tuples, each containing three components, as shown below:

```
std::tuple<std::string, int32_t, double> t1 = std::make_tuple("EECS", 281, 3.14);
std::tuple<double, std::string, int32_t, char> t2 = std::make_tuple(1.618, "EECS", 370, 'Q');
```

If you ran `std::tuple_cat()` on these two tuples:

```
std::tuple<std::string, int32_t, double, double, std::string, int32_t, char> combined =
   std::tuple_cat(t1, t2);
```

You would end up with a tuple that has seven components:

```
std::get<0>(combined) == "EECS"
std::get<1>(combined) == 281
std::get<2>(combined) == 3.14
std::get<3>(combined) == 1.618
std::get<4>(combined) == "EECS"
std::get<5>(combined) == 370
std::get<6>(combined) == 'Q'
```

The `std::tie()` function can be used to unpack the contents of a tuple. For example, suppose we wanted to unpack the contents of the following tuple:

```
std::tuple<std::string, int32_t, std::string, double> t = std::make_tuple("EECS", 281, "PI", 3.14);
```

We can use `std::tie()` to retrieve the components of the tuple `t` and store them into individual variables:

```
1  std::string v1;
2  int32_t v2;
3  std::string v3;
4  double v4;
5  std::tie(v1, v2, v3, v4) = t;   // stores tuple values in variables v1, v2, ...
6  std::cout << v1 << '\n';        // prints "EECS"
7  std::cout << v2 << '\n';        // prints 281
8  std::cout << v3 << '\n';        // prints "PI"
9  std::cout << v4 << '\n';        // prints 3.14
```

You can use `std::ignore` within a call to `std::tie()` to ignore certain components of the tuple:

```
1  std::string v1;
2  int32_t v2;
3  double v4;
4  std::tie(v1, v2, std::ignore, v4) = t;   // ignores third item in tuple ("PI")
5  std::cout << v1 << '\n';                 // prints "EECS"
6  std::cout << v2 << '\n';                 // prints 281
7  std::cout << v4 << '\n';                 // prints 3.14
```

Lastly, because pairs are simply two-element tuples, tuple operations can also be used on pairs. Therefore, you can also use `std::get<>`, `std::tuple_cat()`, and `std::tie()` when working with pairs.

Remark: C++17 introduced the **structured binding** declaration, which allows you to bind variables to the subcomponents of another object. This can be quite useful when working with pairs and tuples! Consider the following:

```
1  std::pair<std::string, int> p = {"EECS", 281};   // p stores {"EECS", 281}
2  auto [str, num] = p;                             // structured binding
3  std::cout << str << '\n';                        // prints "EECS"
4  std::cout << num << '\n';                        // prints 281
```

Here, line 2 is a structured binding that creates two variables, `str` and `num`, whose values are derived from the contents of the pair. You can also declare a reference to the components of the pair using `auto& [str, num]`. This is not required knowledge for the course, but it is a very neat feature that is worthwhile to know. Structured bindings will be covered in a bit more detail in the last section of this chapter.

11.4 Iterators

※ 11.4.1 *Iterator Definitions*

In the past few chapters, we discussed different approaches that can be used to implement different containers. If we used a pure object-oriented approach to implement our containers (e.g., like what we did in section 6.8), each container would be defined as a class, with algorithms defined as member functions.

```
class Vector {
  ...  // data
public:
  ...  // algorithms
};
```

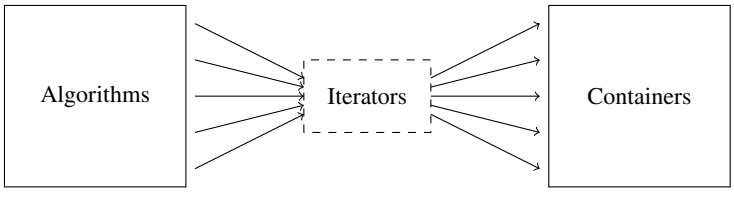
```
class List {
  ...  // data
public:
  ...  // algorithms
};
```

```
class Deque {
  ...  // data
public:
  ...  // algorithms
};
```

However, there is an issue with this approach: each algorithm must be implemented multiple times, once for each container class! For instance, if we wanted to write an algorithm that can be used to find the smallest element in a container, we would have to implement it once for the vector, once for the list, once for the deque, etc. In general, if we had A algorithms and C containers, we would have to write $A \times C$ implementations to ensure that all A algorithms are supported by all C containers.

This is not ideal, since many algorithms tend to have similar implementations. If we rewrite each algorithm implementation C times, we can run into a lot of issues (e.g., code duplication, bugs, etc.). If possible, we would like to be able to write each algorithm once and generalize it to multiple containers.

This is where iterators come into play. In the STL, iterators serve as an interface between containers and algorithms. With iterators, an algorithm's implementation does not need to worry about the specific container type or the data layout in memory; it can operate independently of the container type! Any algorithm that we write will only have to deal with the actual data values in a container, which we can access using the iterators that the container provides.

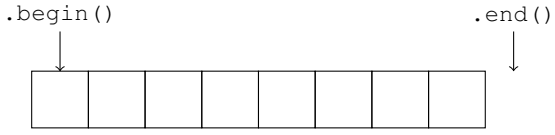


An **iterator** is an object that can be used to traverse through elements of a container. Iterators are similar to pointers in that they can be dereferenced (`operator*`), incremented (`operator++`), compared (`operator==`), and assigned (`operator=`). However, they are better than pointers in that they provide an interface that is adapted to their underlying container. When you increment a pointer, you are incrementing the address the pointer is pointing to. However, there is no guarantee that elements in an STL container are contiguous in memory. With an iterator, you can move through the elements of a container without having to worry about how the data of the container is stored in memory. Incrementing an iterator, for instance, would point it to the next element in the container, regardless of whether memory is contiguous or not.

Because different containers store data differently, an iterator's internal behavior depends on its underlying container. An iterator for a vector is implemented differently from an iterator for a list, for instance, because the elements in a vector are contiguous in memory while the elements in a list are not. However, even though the implementation may be different, *the interface that an iterator provides is the same across different containers*. An algorithm that uses iterators can be used across many different containers to accomplish the same task. This removes the need to write multiple versions of the same algorithm to support every type of STL container, since a single implementation of a function can be used for multiple data structures.

With the exception of container adaptors (such as stacks and queues, which do not support iteration), STL container classes provide the following member functions that return iterators to the container:

- `.begin()` returns an iterator to the first element in the container.
- `.end()` returns an iterator that points *one past* the last element in the container.



The `.begin()` and `.end()` iterators can be used to iterate through the contents of a container in a loop. This is done by incrementing an iterator through the container until the end iterator is reached. An example is shown below:

```
std::vector<int32_t> vec = {1, 2, 3, 4};
for (std::vector<int32_t>::iterator it = vec.begin(); it != vec.end(); ++it) {
  std::cout << *it << " ";
} // for it
```

The following code does the same thing, but with a `while` loop:

```
std::vector<int32_t> vec = {1, 2, 3, 4};
std::vector<int32_t>::iterator it = vec.begin();
while (it != vec.end()) {
  std::cout << *it++ << " ";
} // while
```

Here, the term `*it++` dereferences the iterator *before* incrementing the iterator's position (postfix incrementation).

In addition to `.begin()` and `.end()`, the member functions `.cbegin()` and `.cend()` can be used to return `const` iterators to the first and one past the last position of a container. Dereferencing a `const` iterator returns a reference to a `const` value that cannot be modified.

It should be noted that the type of the iterator (e.g., `std::vector<int>::iterator`) does *not* need to be explicitly written out. Instead, the `auto` keyword can be used to automatically deduce an iterator's type.

Remark: In C++11, the C++ standard library introduced the `std::begin()` and `std::end()` functions. These two functions do the same thing as `.begin()` and `.end()`, but they make it easier to write generic code that can be applied to many different container types. These two functions can also be used on C-style arrays, which do not have their own `.begin()` and `.end()` member functions:

```
int32_t arr[5] = {1, 2, 3, 4, 5};
auto start = std::begin(arr);
while (start != std::end(arr)) {
  std::cout << *start++ << " ";     // prints 1 2 3 4 5
} // while
```

Given any container `c` that supports iterators, `std::begin(c)` can be used instead of `c.begin()`, and `std::end(c)` can be used instead of `c.end()`. Even though `std::begin(c)` and `std::end(c)` provide more flexibility, you are more likely to see `c.begin()` and `c.end()` just because the alternative is newer and not everyone knows about it (or just do not use out of habit, since `.begin()` and `.end()` have been the go-to standard for years).

※ 11.4.2 *Iterator Categories*

Iterators in the STL can be placed into five different categories based on the operations they support. These five categories are:

- *Input iterators:* read only, forward moving (single pass only)
- *Output iterators:* write only, forward moving (single pass only)
- *Forward iterators:* forward moving (multiple passes allowed)
- *Bidirectional iterators:* forward and backward moving
- *Random access iterators:* provides random access

An **input iterator** *reads* values with forward movement (decrementation using −− is *not* allowed). This type of iterator

- can be dereferenced (⋆), incremented (++), and compared (== and !=).
- does *not* support multiple passes (once you increment the iterator, the previous position is no longer guaranteed to be dereferenceable, even if you attempt to store an input iterator to the previous element).
- can be dereferenced multiple times, but only for the current element — once you increment an input iterator, you can no longer dereference values that it pointed to before.

Comparisons using input iterators are usually done to determine whether the iterator being incremented has reached the end of an iterator range.

An **output iterator** *writes* values with forward movement (decrementation using −− is *not* allowed). This type of iterator

- can be written to (using ⋆) and incremented (++), but *cannot* be compared.
- does *not* support multiple passes (once you increment the iterator, the previous position is no longer guaranteed to be dereferenceable).
- does *not* support multiple writes to the same element (once you dereference and write, you cannot dereference again without incrementing the iterator).

A **forward iterator** supports both reading and writing in the forward direction (decrementation using −− is *not* allowed). This type of iterator

- can be dereferenced (⋆), incremented (++), and compared (== and !=).
- supports multiple passes.
- supports multiple reads and writes to the same element.

A **bidirectional iterator** can do everything a forward iterator can, but is also supports backward movement (i.e., you can decrement a bidirectional iterator using −−).

A **random access iterator** can do everything a bidirectional iterator can, but it also supports pointer arithmetic and pointer comparisons. Given a random access iterator, you are allowed to use mathematical expressions such as +, +=, −, and −= to move the iterator's position, as well as comparison operators such as <, <=, >, and >= to compare the relative positions of different random access iterators.

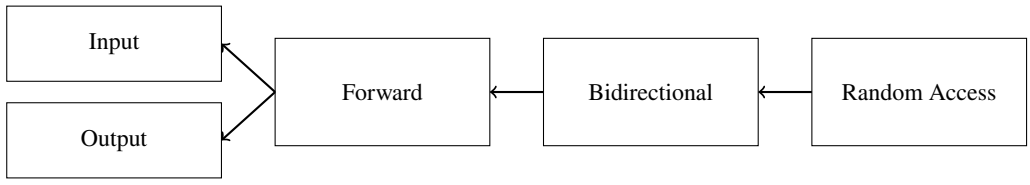
	Input	Output	Forward	Bidirectional	Random
Supports dereference (⋆) and read	✓		✓	✓	✓
Supports dereference (⋆) and write		✓	✓	✓	✓
Supports forward movement (++)	✓	✓	✓	✓	✓
Supports backward movement (−−)				✓	✓
Supports multiple passes			✓	✓	✓
Supports == and !=	✓		✓	✓	✓
Supports pointer arithmetic (+, −, etc.)					✓
Supports pointer comparison (<, >, etc.)					✓

Of the containers we have covered so far, vector and deque iterators are random access iterators, while list iterators are bidirectional iterators. A summary of different STL containers and their iterator categories is shown below (we will cover many of these containers in later chapters):

Container Type	Iterator Category
`std::vector<>`	Random Access
`std::deque<>`	Random Access
`std::string<>`	Random Access
`std::list<>`	Bidirectional
`std::set<>`	Bidirectional
`std::multiset<>`	Bidirectional
`std::map<>`	Bidirectional
`std::multimap<>`	Bidirectional
`std::unordered_set<>`	Forward
`std::unordered_multiset<>`	Forward
`std::unordered_map<>`	Forward
`std::unordered_multimap<>`	Forward
`std::forward_list<>`	Forward

The type of iterator that a container supports is important for determining which algorithms can be used on that container. Some algorithms only support certain iterator types (for instance, `std::sort()` in the algorithm library only accepts random access iterators, so this function cannot be used on a list — instead, lists have their own sort function that uses bidirectional iterators).

All of the iterator types can be organized into the following hierarchy:



The random access iterator is highest in the hierarchy, while input and output iterators are lowest in the hierarchy. An iterator that is higher on the hierarchy supports all of the features of iterators lower on the hierarchy. Because of this, if an iterator is expected in an algorithm, any iterator that is higher on the hierarchy may be used in its place. For instance, a random access iterator supports all the functionalities of bidirectional, forward, input, and output iterators, and thus can be used anywhere one of these iterators is expected. Similarly, a bidirectional iterator supports all the functionalities of forward, input, and output iterators, but do not support all the functionalities of a random access iterator. Thus, a bidirectional iterator can be used in place of an input, output, or forward iterator, but it cannot be used in place of a random access iterator.

11.5 Iterator Adaptors

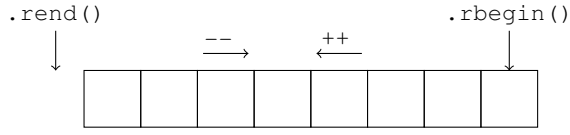
The C++ `<iterator>` library provides several **iterator adaptors**, which build upon the interface of the five categories of iterators to provide additional functionality. There are several types of iterator adaptors, but we will focus on three: the reverse, stream, and insert iterators.

※ 11.5.1 *Reverse Iterators*

A **reverse iterator**, much like its name implies, can be used to iterate through a container in reverse order. Incrementing a reverse iterator *decrements* the relative position of the iterator in the container.

All containers that support bidirectional iterators or random access iterators also support reverse iterators. Reverse iterators for these containers can be retrieved using the following member functions:

- `.rbegin()` returns an iterator pointing to the last element in the container (`std::rbegin()` can also be used).
- `.rend()` returns an iterator pointing *one before* the first element in the container (`std::rend()` can also be used).



The member functions `.crbegin()` and `.crend()` can be used to return `const` reverse iterators to a container. Much like standard `const` iterators, dereferencing a `const` reverse iterator returns a value that cannot be modified.

Reverse iterators are useful in that they allow algorithms to operate on a container's data in reverse order without having to modify the ordering of elements in the container itself. Passing reverse iterators into an algorithm is akin to passing in a range of data in backwards order. The algorithm would run normally, but with the order of elements flipped — the algorithm would treat the last element as if it were the first element in the container. AsTo show off an example, the following code uses reverse iterators:

```
1  std::vector<int32_t> vec = {1, 2, 3, 4, 5};
2  for (auto it = vec.rbegin(); it != vec.rend(); ++it) {
3    std::cout << *it << " ";   // prints 5 4 3 2 1
4  } // for it
5
6  // range constructor uses reverse iterators, so vec_copy holds the
7  // contents of vec, but in reverse order: vec_copy == {5, 4, 3, 2, 1}
8  std::vector<int32_t> vec_copy(vec.rbegin(), vec.rend());
9  // iterators can be used to copy data to a different container as well
10 std::deque<int32_t> deq(vec.rbegin(), vec.rend());   // deq == {5, 4, 3, 2, 1}
11 std::list<int32_t> lst(vec.rbegin(), vec.rend());    // lst == {5, 4, 3, 2, 1}
```

※ 11.5.2 *Stream Iterators* (✻)

A **stream iterator** is another iterator adaptor. Stream iterators can be used to traverse through the contents of a stream, as if the stream were a container. There are two classes of stream iterators, input stream iterators (`std::istream_iterator<>`) and output stream iterators (`std::ostream_iterator<>`). An `std::istream_iterator<>` is an input iterator that reads in the contents of a stream. Since input stream iterators are input iterators, they only support a single pass of a stream.

There are two different forms that an `std::istream_iterator<>` can take on:

- Initializing an `std::istream_iterator<>` with a stream object creates an iterator that reads in the first object in the stream. For example, `std::istream_iterator<T> it{std::cin}` creates an iterator `it` that reads in the first object in the stream `std::cin`. This behavior is similar to calling `std::cin >> val`, where `val` is of type `T`.
- Initializing an `std::istream_iterator<>` that is default constructed creates an *end-of-stream* iterator, or an iterator that indicates that the stream has been entirely read (similar to the `end` iterator of a standard container). This end-of-stream iterator can be compared with another `std::istream_iterator<>` to determine if it has reached the end of the stream.

Incrementing an `std::istream_iterator<>` reads in the next object in the stream. Note that an object is read in when the iterator is incremented, *not* when it is dereferenced! Dereferencing only returns a copy of the object that was most recently read in. The following examples make use of the `std::istream_iterator<>`:

```
1  // 'it' is an iterator that can be used to
2  // traverse the contents of the cin stream
3  std::istream_iterator<int32_t> it{std::cin};
4  // 'eof' is an iterator that identifies
5  // the end of the stream (default initialized)
6  std::istream_iterator<int32_t> eof;
7  // iterate until 'it' reaches eof
8  while (it != eof) {
9    std::cout << *it++ << " ";
10 } // while
```

input.txt

```
1 2
3 4 5
6 7 8 9
```

Output of program:

```
1 2 3 4 5 6 7 8 9
```

```
1  std::istream_iterator<std::string> it{std::cin};
2  std::istream_iterator<std::string> eof;
3  while (it != eof) {
4    std::cout << *it++ << " ";
5  } // while
```

input.txt

```
EECS 281
is fun
```

Output of program:

```
EECS 281 is fun
```

An `std::ostream_iterator<>` is an output iterator that writes objects to a stream. Since this type of iterator is an output iterator, it only supports a single pass in the forward direction. An `std::ostream_iterator<>` can be quite useful for writing the contents of a container to an output stream (like `std::cout`):

```
1  std::vector<int32_t> vec = {1, 2, 3, 4, 5};
2  std::ostream_iterator<int32_t> out{std::cout};
3  // copy contents of vector to cout using std::copy()
4  std::copy(vec.begin(), vec.end(), out);  // prints 12345
```

The `std::ostream_iterator<>` can also be constructed with a delimiter argument. This delimiter is used to separate the elements that are written. The delimiter is optional, and without it, no separation is added between the data.

```
1  std::vector<int32_t> vec = {1, 2, 3, 4, 5};
2  // use a space to separate values in output
3  std::ostream_iterator<int32_t> out{std::cout, " "};
4  // copy contents of vector to cout using std::copy()
5  std::copy(vec.begin(), vec.end(), out);  // prints 1 2 3 4 5
```

In fact, you can condense the code above into the following (this writes the contents of an iterator range to `std::cout` in just one line of code, where elements are separated by spaces):

```
1  std::vector<int32_t> vec = {1, 2, 3, 4, 5};
2  std::copy(vec.begin(), vec.end(), std::ostream_iterator<int32_t>{std::cout, " "});
```

This method can be quite useful for efficiently printing out the contents of any iterator range to an output stream.

※ 11.5.3 *Insert Iterators* (✻)

The last iterator adaptor that we will discuss in this section is the **insert iterator**. By default, output iterators operate in overwrite mode. If you give an output iterator to an algorithm that writes data, the algorithm will likely overwrite the data located at the position of the output iterator.

Consider the `std::copy()` function in the `<algorithm>` library, which takes in two input iterators and one output iterator and copies the data in the iterator range `[first, last)` to the range beginning at `result`:

```
std::copy(InputIterator first, InputIterator last, OutputIterator result);
```

The following code copies the data from `orig` into `dest`. However, since output iterators overwrite the data at their position, the final contents of `dest` end up being `{5, 6, 7, 8}`, since `{1, 2, 3, 4}` was overwritten.

```
1  std::vector<int32_t> orig = {5, 6, 7, 8};
2  std::vector<int32_t> dest = {1, 2, 3, 4};
3  std::copy(orig.begin(), orig.end(), dest.begin());
```

An insert iterator is a special type of output iterator that *inserts* values at a given position instead of overwriting them. If you write a value to an insert iterator, the insert iterator inserts the value at that position of the container instead of overwriting the data that was originally there. There are three different types of insert iterators: the back-insert iterator, the front-insert iterator, and the generic insert iterator.

The **back-insert iterator** is an insert iterator that allows algorithms to insert new elements at the back of the underlying container (after the last element). When a back-insert iterator is dereferenced and assigned to, a `.push_back()` operation is called on the container. To construct a back-insert iterator for a container, pass the container into the `std::back_inserter()` function. Back-insert iterators only work on containers that support a `.push_back()` member function (such as `std::vector<>`, `std::deque<>`, and `std::list<>`).

```
std::vector<int32_t> vec = {1, 2, 3};
std::back_insert_iterator<std::vector<int32_t>> it = std::back_inserter(vec);
*it = 4;  // inserts 4 at the back of the container, final vec = {1, 2, 3, 4}
```

Back-insert iterators can be used in conjunction with STL algorithms to insert values into a sequence rather than overwriting them. For example, the following code passes in a back-insert iterator as the output iterator of the `std::copy()` function. By doing this, everything that the algorithm copies is inserted to the back of the container:

```
std::vector<int32_t> orig = {5, 6, 7, 8};
std::vector<int32_t> dest = {1, 2, 3, 4};
// copy elements in orig and write to back_inserter(dest)
std::copy(orig.begin(), orig.end(), std::back_inserter(dest));
// final contents of dest now {1, 2, 3, 4, 5, 6, 7, 8}
```

The **front-insert iterator** is an insert iterator that allows algorithms to insert new elements at the front of the underlying container (before the first element). When a front-insert iterator is dereferenced and assigned to, a `.push_front()` operation is called on the container. To construct a front-insert iterator for a container, pass the container into the `std::front_inserter()` function. Front-insert iterators only work on containers that support a `.push_front()` member function (such as `std::deque<>` and `std::list<>`).

```
std::deque<int32_t> deq = {2, 3, 4, 5};
std::front_insert_iterator<std::deque<int32_t>> it = std::front_inserter(deq);
*it = 1;  // inserts 1 at the front of the container, final deq = {1, 2, 3, 4, 5}
```

When a front-insert iterator is used to insert a range of elements to the front of a container, the elements in the range are pushed to the front of the container one by one. As a result, using the front-insert iterator to insert a range will reverse the order of the inserted elements (i.e., if you use a front-insert iterator to insert `{1, 2, 3, 4}`, `1` would be pushed to the front first, then `2`, then `3`, then `4`; at the end, `4` ends up being at the very front since it was pushed last).

```
std::deque<int32_t> orig = {5, 6, 7, 8};
std::deque<int32_t> dest = {1, 2, 3, 4};
// copy elements in orig and write to front_inserter(dest)
std::copy(orig.begin(), orig.end(), std::front_inserter(dest));
// contents of dest now {8, 7, 6, 5, 1, 2, 3, 4}
```

Lastly, the generic **insert iterator** can be used to insert elements anywhere in a container. To construct a generic insert iterator, pass the underlying container and an iterator to the position of insertion into the `std::inserter()` function. Generic insert iterators only work on containers that support an `.insert()` member function (this function is supported for most of the containers you will use in this class; the only exceptions are container adaptors, arrays, and forward lists). Incrementing an insert iterator does nothing; if you want to insert at a new position, you would have to redeclare it with the new desired position.

```
std::vector<int32_t> vec = {1, 2, 3, 6, 7, 8};
// vec.begin() + 3 points to 6
std::insert_iterator<std::vector<int32_t>> it = std::inserter(vec, vec.begin() + 3);
*it = 4;  // inserts 4 before 6, vec now {1, 2, 3, 4, 6, 7, 8}
*it = 5;  // inserts 5 before 6, vec now {1, 2, 3, 4, 5, 6, 7, 8}
```

As mentioned earlier, you can use `auto` instead of typing out the full name of the iterator type (which can be long and complicated). This is a situation where using `auto` can make your code a lot cleaner without losing valuable information.

11.6 Auxiliary Iterator Functions (⁎)

※ 11.6.1 *std::advance* (⁎)

The STL <iterator> library provides several useful operations that can make iterator usage cleaner. As mentioned earlier, iterators can be grouped into five categories, where the random access iterator has the most functionality. Unlike the other iterator types, a random access iterator can support pointer arithmetic. This makes it easy to move a random access iterator around relative to its current position. For example, if we had a random access iterator `it`, and we wanted to increment the iterator by 10 positions, we could just do some simple math:

```
it += 10;
```

However, this is not something we can do with a non-random access iterator. In cases where we cannot increment an iterator by a large distance using math operations, we can instead use the `std::advance()` function:[1]

```
void std::advance(InputIterator &it, int n);
```

The `advance()` function takes the iterator `it` and increments it by `n` positions. If `n` is negative, the iterator is moved backward instead of forward. However, `n` can only be negative if a random access or bidirectional iterator is passed in (since these are the only iterator types that support backward movement).

For example, if we are given a non-random access iterator `it`, and we wanted to increment it by 10 positions, we could run the following code (which is equivalent to doing `it += 10` for a random access iterator):

```
std::advance(it, 10);
```

※ 11.6.2 *std::distance* (⁎)

It is also tougher to find the distance between two iterators that do not support random access. If two iterators `it_first` and `it_last` support random access, we can find the distance between them by doing subtraction:

```
int dist = it_last - it_first;
```

However, subtraction is not allowed for non-random access iterators. To make distance calculations cleaner for these iterators, we can use the `std::distance()` function:

```
int std::distance(InputIterator first, InputIterator last);
```

This function calculates the number of elements between `first` and `last`. For example, if we wanted to find the distance between two non-random access iterators `it_first` and `it_last`, we can run the following code:

```
int dist = std::distance(it_first, it_last);
```

The iterator passed in as the first argument should come before the iterator passed in as the second argument.

※ 11.6.3 *std::next and std::prev* (⁎)

Lastly, if we wanted to instantiate a *separate* iterator that points to the element at index `n` of a container `c`, we could do something like this if `c` supports random access:[2]

```
RandomAccessIterator it = std::begin(c) + n;
```

If `c` does not provide random access, we can instead use the `std::next()` and `std::prev()` functions:

```
ForwardIterator std::next(ForwardIterator it, int n);
BidirectionalIterator std::prev(BidirectionalIterator it, int n);
```

The `std::next()` function returns an iterator pointing to the element that `it` would be pointing to if it advanced `n` positions from its current position. Similarly, the `std::prev()` function returns an iterator to the element that `it` would be pointing to if it were decremented `n` positions. Unlike `std::advance()`, the `std::next()` and `std::prev()` functions do *not* move the iterators that are passed in. An example of iterator function usage is shown below:

```
1  std::list<int32_t> lst = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2  auto it = std::begin(lst);  // 'it' points to 0
3  std::advance(it, 3);  // 'it' points to 3
4  std::advance(it, 4);  // 'it' points to 7
5  std::advance(it, -2);  // 'it' points to 5
6  auto foo = std::next(it, 4);  // 'foo' points to 9, 'it' points to 5
7  auto bar = std::prev(it, 4);  // 'bar' points to 1, 'it' points to 5
8  std::cout << std::distance(bar, foo) << '\n';  // prints 8, the distance between 9 and 1
```

[1] The type of `n` is not explicitly defined as an `int`, but rather something known as a `difference_type`. A `difference_type` is basically any numerical type that is able to represent iterator distance. This is mainly here for portability sake across different compilers, and you can just treat it like an `int` (which is what we do here). This type also applies for the other iterator functions in this section. If you are confused about what this information even means, don't worry about it — it's not something you really need to know.

[2] `RandomAccessIterator` isn't an actual type, but rather a placeholder for any random access iterator, such as a `std::vector<int>::iterator`.

11.7 Predicates in the STL

A **function object**, or functor, is an object that can be used with `operator()` to behave like a function. A **predicate** is a special type of function object that returns a `bool` and is stateless (i.e., it produces the same result for the same input value). A *unary predicate* is a predicate that takes in one argument and checks a property of that argument. A *binary predicate* is a predicate that takes in two arguments and compares a property of these two arguments.

Function objects can also be used to compare two objects. A **comparator** is a binary predicate that takes in two objects and returns whether one object goes before or after the other. The C++ standard library provides several predefined comparators, the most common of which are `std::less<>` and `std::greater<>`.

The comparator `std::less<>` is a binary predicate that returns whether its first argument is less than the second (as returned by `operator<`). An example is shown below:

```
1   std::less<int32_t> comp;
2   std::cout << comp(4, 5) << '\n';   // returns 1 (true) since 4 is less than 5
3   std::cout << comp(5, 4) << '\n';   // returns 0 (false) since 5 is not less than 4
4   std::cout << comp(5, 5) << '\n';   // returns 0 (false) since 5 is not less than 5
```

The comparator `std::greater<>` is a binary predicate that returns whether its first argument is greater than the second (as returned by `operator>`). An example is shown below:

```
1   std::greater<int32_t> comp;
2   std::cout << comp(4, 5) << '\n';   // returns 0 (false) since 4 is not greater than 5
3   std::cout << comp(5, 4) << '\n';   // returns 1 (true) since 5 is not greater than 4
4   std::cout << comp(5, 5) << '\n';   // returns 0 (false) since 5 is not greater than 5
```

The type of the arguments (i.e., the `T` in `std::less<T>` and `std::greater<T>`) must support `operator<` for `std::less<T>` to work, and `operator>` for `std::greater<T>` to work. Thus, if you want to use `std::less<T>` on a custom `struct` or `class`, the object definition must implement `operator<` (or `operator>` for `std::greater<T>`).

The `std::less<>` and `std::greater<>` comparators aren't the only two function objects that are provided by the STL. Other objects that are predefined by the standard library include:

- `std::equal_to<>`, which checks for equality of two arguments
- `std::not_equal_to<>`, which checks for inequality of two arguments
- `std::less_equal<>`, which checks whether one argument is less than or equal to the other
- `std::greater_equal<>`, which checks whether one argument is greater than or equal to the other

This list is not comprehensive. However, `std::less<>` and `std::greater<>` are the two STL function objects that you are most likely to see and use in this course (and thus are the only two you need to know well). These comparators play an important role in STL algorithms and container declarations, such as with the `std::priority_queue<>`.

11.8 Standard Math Functions and Numeric Limits

※ 11.8.1 *The Math Library*

In this section, we will explore several functions in the standard library that work with numbers and mathematics. First, we will look at the `<cmath>` library, which contains several common mathematical operations, some of which are reproduced below (this list is not comprehensive, but it contains operations you are most likely to encounter):

Function	Behavior
`double std::exp(double x);`	Returns e^x
`double std::log(double x);`	Returns $\ln(x)$
`double std::log10(double x);`	Returns $\log_{10}(x)$
`double std::log2(double x);`	Returns $\log_2(x)$
`double std::pow(double base, double exponent);`	Returns base$^{(\text{exponent})}$
`double std::sqrt(double x);`	Returns $\sqrt{x}$
`double std::cbrt(double x);`	Returns $\sqrt[3]{x}$
`double std::ceil(double x);`	Rounds x upwards and returns smallest integer value that is not less than x
`double std::floor(double x);`	Rounds x downwards and returns largest integer value that is not greater than x
`double std::round(double x);`	Returns the integer value that is nearest to x, with halfway cases rounded away from zero
`double std::abs(double x);`	Returns the absolute value of x (may be unsafe for floating point numbers)[3]
`double std::fabs(double x);`	Returns the absolute value of x, safer for `double` and `float` (see footnote on the next page)

These functions are general-purpose operations that are implemented to work for many different cases. As a result, their versatility makes them slower for simpler operations. For example, `pow(3, 20)` is really efficient at calculating 3^{20}. However, if you want to calculate 3^2, it would be much faster to run `3*3` instead of `pow(3, 2)`, since a function that is implemented to handle huge powers is unnecessarily complex for something as simple as squaring a number.

※ 11.8.2 *Setting Precision for Floating Point Output*

Another useful operation is the `std::setprecision()` function, which can be used to set decimal precision for floating point output. This method is provided in the `<iomanip>` library:

Function	Behavior
`std::setprecision(int n);`	Sets the decimal precision of a floating point number to *n* digits before it is displayed on output

The following examples use `std::setprecision()` to format decimal output:

```
double pi = 3.14159265358979;
std::cout << std::setprecision(1) << pi << '\n';    // prints 3
std::cout << std::setprecision(2) << pi << '\n';    // prints 3.1
std::cout << std::setprecision(3) << pi << '\n';    // prints 3.14
std::cout << std::setprecision(4) << pi << '\n';    // prints 3.142
std::cout << std::setprecision(5) << pi << '\n';    // prints 3.1416
std::cout << std::setprecision(6) << pi << '\n';    // prints 3.14159
std::cout << std::setprecision(7) << pi << '\n';    // prints 3.141593
std::cout << std::setprecision(8) << pi << '\n';    // prints 3.1415927
std::cout << std::setprecision(9) << pi << '\n';    // prints 3.14159265
std::cout << std::setprecision(10) << pi << '\n';   // prints 3.141592654
std::cout << std::setprecision(11) << pi << '\n';   // prints 3.1415926536
std::cout << std::setprecision(12) << pi << '\n';   // prints 3.14159265359
std::cout << std::setprecision(13) << pi << '\n';   // prints 3.14159265359
std::cout << std::setprecision(14) << pi << '\n';   // prints 3.1415926535898
std::cout << std::setprecision(15) << pi << '\n';   // prints 3.14159265358979
```

※ 11.8.3 *Numeric Limits*

Another useful component of the standard library is the `std::numeric_limits<>` class template, which provides a generalized method for retrieving properties of arithmetic types, such as integers and floating point numbers. Examples of arithmetic types include `bool`, `char`, `int`, `float`, `double`, and variations such as `short int`, `long double`, `unsigned int`, etc. This template is defined in the `<limits>` header, so you must `#include <limits>` to use these methods.

The `std::numeric_limits<>` class provides several member functions, four of which are `min()`, `lowest()`, `max()`, and `infinity()`. Details about these functions are provided below:

Function	Behavior
`std::numeric_limits<T>::min();`	Returns the minimum finite value representable by the numeric type `T`. For non-floating point types, this is the lowest possible value that the type can hold (i.e., the value that has no values less than it). For floating point types with denormalization, such as `float` and `double`, `numeric_limits<T>::min()` returns the smallest *positive* finite value that the type can hold.
`std::numeric_limits<T>::lowest();`	Returns the lowest finite value representable by the numeric type `T`. Nothing of type `T` can have a value lower than it.
`std::numeric_limits<T>::max();`	Returns the maximum finite value representable by the numeric type `T`.
`std::numeric_limits<T>::infinity();`	Returns a special "infinity" value if the type `T` supports infinity. This is only useful for floating point numbers such as `float` and `double`. Otherwise, this returns 0.

A few numeric limits values are shown below. You do not need to memorize the rules above!

- `std::numeric_limits<int>::min() == -2147483648` (lowest possible `int`)
- `std::numeric_limits<double>::min() == 2.22507E-308` (note that this is positive, but small)
- `std::numeric_limits<int>::lowest() == -2147483648` (lowest possible `int`)
- `std::numeric_limits<double>::lowest() == -1.79769E308` (lowest possible `double`)
- `std::numeric_limits<int>::max() == 2147483647` (largest possible `int`)
- `std::numeric_limits<double>::max() == 1.79769E308` (largest possible `double`)

[3] WARNING: Even though this function works for floating point numbers, you must use `std::abs()` and `#include <cmath>` in your program for this to work properly. The default version of `abs()` is the C version that only takes in integers! If you want to find the absolute value of a floating point number, it is safer to use `fabs()` to find the absolute value since it clarifies your intent and prevents undetectable bugs.

If you want to use infinity in your program, you should use `std::numeric_limits<>` with a `double` rather than an `int`. This is because infinity is *not* defined for integers. The value of infinity is considered larger than `1.79769E308`, the largest value a `double` can take on.

Infinity in C++: `std::numeric_limits<double>::infinity();`

If you want to use the largest or smallest possible `int`, you can either use numeric limits, or you can use the constants `INT_MIN` and `INT_MAX` in the `<climits>` library. `INT_MIN` has the same value as `numeric_limits<int>::min()`, and `INT_MAX` has the same value as `numeric_limits<int>::max()`. In C++, however, the numeric limits style is preferred.

※ 11.8.4 *std::iota*

Another useful function that you might encounter is the `std::iota()` function defined in the `<numeric>` library. This function takes in an iterator range defined by `[first, last)` (where `last` is exclusive) and a value `n`, and it fills the range with the values `n`, `n + 1`, `n + 2`, …, all the way to the end of the iterator range. The type of `T` must be incrementable.

```
template <typename ForwardIterator, typename T>
void std::iota(ForwardIterator first, ForwardIterator last, T val);
```
Fills the iterator range `[first, last)` with sequentially increasing values, starting with `n`.

An example of this function's usage is shown below:

```
// initialize vector of size 10
std::vector<int32_t> vec(10);
// fill up vector with values 1, 2, 3, 4, ...
std::iota(vec.begin(), vec.end(), 1);
// contents of vec: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// fill up vector with values 11, 12, 13, 14 in the range
// [vec.begin() + 3, vec.begin() + 7)
std::iota(vec.begin() + 3, vec.begin() + 7, 11);
// contents of vec: {1, 2, 3, 11, 12, 13, 14, 8, 9, 10}
```

The `std::iota()` function can be quite useful in certain situations, as we will see when we discuss the union-find algorithm in chapter 13.

※ 11.8.5 *std::accumulate* (✻)

Another useful function in the `<numeric>` library is the `std::accumulate()` function, which sums up all the values in an iterator range:

```
template <typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last, T init);
```
Sums up all the elements in the range `[first, last)` and adds the result to the value `init`.

An example of the `std::accumulate()` function is shown below:

```
std::vector<int32_t> vec = {1, 2, 3, 4, 5};
std::cout << std::accumulate(vec.begin(), vec.end(), 0) << '\n';  // prints 15
std::cout << std::accumulate(vec.begin(), vec.end(), 5) << '\n';  // prints 20
```

Both calls to `std::accumulate()` sum up all the values in the vector `vec`, but the second call starts with an initial value of `5` (which is why it prints `20` instead of `15`).

The `std::accumulate()` function can do more than summing up elements. Even though summation is the default behavior of the function, you can also pass in a separate binary operation if you want it to do something different.

```
template <typename InputIterator, typename T, typename BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation op);
```
Applies `op` to accumulate all the values in the range `[first, last)` to `init`.

For instance, the following code multiplies all the values in the vector `vec` and prints the result:

```
int32_t prod(int32_t accumulator, int32_t element) {
  return accumulator * element;
} // prod()

int main() {
  std::vector<int32_t> vec = {1, 2, 3, 4, 5};
  std::cout << std::accumulate(vec.begin(), vec.end(), 1, prod) << '\n';  // 120
} // main()
```

The binary operation that you pass in should take in two arguments: an accumulator and an element. The accumulator is a value that starts at `init` and gets accumulated over every `element` in the iterator range. The `std::accumulate()` function iterates through the iterator range and applies the binary operation to each element using the accumulator. Thus, the following:

```
std::cout << std::accumulate(vec.begin(), vec.end(), 1, prod) << '\n';
```

behaves identically to this:

```
int32_t init = 1;  // accumulator value initialized to 3rd argument of function call
for (int32_t elt : vec) {
  init = prod(init, elt);
} // for elt
std::cout << init << '\n';
```

The binary operation makes `std::accumulate()` much more versatile. For instance, the following code can be used to accumulate a container of custom objects, even though each object does not define `operator+` on its own:

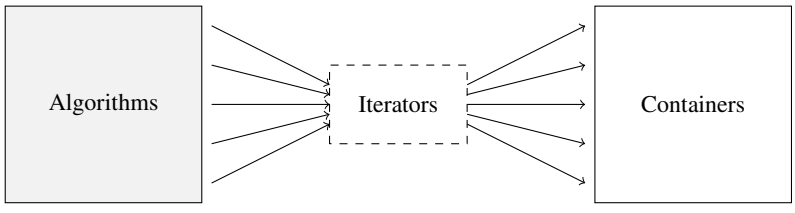
```
struct Student {
  std::string name;
  double wealth;
};

double add_wealth(double accumulator, const Student& element) {
  return accumulator + element.wealth;
} // add_wealth()

int main() {
  std::vector<Student> students = {{"Alice", 1.00}, {"Bob",   1.50}, {"Cathy", 2.00},
                                   {"Drew",  2.50}, {"Emily", 3.00}, {"Frank", 3.50}};
  std::cout << std::accumulate(students.begin(), students.end(), 0.00, add_wealth) << '\n';
} // main()
```

Here, the `std::accumulate()` function sums up the wealth of each of the students in the vector and prints the total sum (in this case, $13.50). Note that the accumulator is initialized to a value of `0.00`, as shown in the third argument of the function call. The decimal was included so that `std::accumulate()` would return an object of type `double` rather than an object of type `int` (which would have happened if `0` were passed in without decimals). Remember that the return type of `std::accumulate()` is the same type as the third argument, `init`!

11.9 The Algorithm Library

※ 11.9.1 *Introduction to the Algorithm Library*

In this section, we will look at several of the algorithms that C++ provides in the `<algorithm>` library. As mentioned earlier, algorithms in the algorithm library utilize iterators to perform operations on a container's data:



Iterators provide an interface that allow STL algorithms to access the data of an underlying container. However, iterators only provide algorithms with access to the data; they do *not* identify the specific container type the data is stored in (if there even is one). That is, if an algorithm receives an iterator range, it can only see the data values within that range, *not* how the data is organized in memory. As a result, algorithms in the STL `<algorithm>` library only work on the data values they are given and *cannot modify the internal structure of the container that holds the data*.

Important: STL algorithms can modify the data *values* of a container, but they cannot modify the *structure* of the container that the data is stored in. This is because iterators provide an interface that gives algorithms access to data values, but not the container type itself. This is especially important to remember for STL algorithms that are designed to mutate the container, such as `std::remove()` and `std::remove_if()`. These algorithms simply modify the ordering of data to make removing easier, but the actual removal needs to be done separately (by calling a member function of the container itself, such as `.erase()`).

Algorithms in the STL `<algorithm>` library typically accept two iterators that denote an iterator range: a `begin` iterator that denotes the start of the range, and an `end` iterator that denotes the element *one past* the last element in the range. By doing this, the algorithm has access to all data values in the range `[begin, end)` (i.e., the element pointed to by `begin` is included in the range, but the element pointed to by `end` is not). Iterator ranges can be used to capture any section of a container to invoke an algorithm on.

The <algorithm> library is quite extensive, and this section will not be able to cover every function that is available. Instead, we will focus on a few algorithms that you will most likely encounter at some point in your programming career. These algorithms are listed below:

- minimum and maximum algorithms
- sorting algorithms
- find algorithms
- removal algorithms
- replacement algorithms
- transformation algorithms
- copy algorithms
- std::reverse()
- std::nth_element()
- lower and upper bound algorithms

Other algorithms in this library will be introduced in later chapters as well, in sections that are more related to their use cases (for example, we will cover set functions in the chapter about sets).

※ 11.9.2 *Minimum and Maximum Algorithms*

C++ provides several different ways to find the minimum or maximum value from a collection of data. Of these functions, the simplest ones are std::min() and std::max(). These two functions can be used to find the smaller or larger of two values.

```
template <typename T, typename Compare>
T std::min(const T& a, const T& b, Compare comp);
```
Returns the smaller of a and b. The comparator is optional, and operator< is used if it is not specified.

```
template <typename T, typename Compare>
T std::max(const T& a, const T& b, Compare comp);
```
Returns the larger of a and b. The comparator is optional, and operator< is used if it is not specified.

The std::minmax() function returns both the smaller and larger elements as a pair. Given two values of a and b, the smaller element is returned as the first element of the pair, while the larger element is returned as the second element. If both a and b are equivalent, the function returns std::make_pair(a, b).

```
template <typename T, typename Compare>
std::pair<const T&, const T&> std::minmax(const T& a, const T& b, Compare comp);
```
Returns a pair with the smaller of a and b as the first element, and the larger as the second. If both are equal, returns std::make_pair(a, b). The comparator is optional, and operator< is used if it is not specified.

An example is shown below:

```
1  std::cout << std::min(14, 12) << '\n';                          // prints 12
2  std::cout << std::max(14, 12) << '\n';                          // prints 14
3  std::pair<int32_t, int32_t> p = std::minmax(14, 12);
4  std::cout << p.first << '\n';                                   // prints 12
5  std::cout << p.second << '\n';                                  // prints 14
```

These functions cannot take in multiple arguments beyond the two you want to compare. If you want to find the minimum or maximum of more than two values, you would have to pass the values in as an *initializer list* (a comma-separated sequence of values enclosed by curly braces). An example of this is shown below:

```
1  int32_t a = 12, b = 15, c = 11, d = 19, e = 10, f = 14;
2  std::cout << std::min({a, b, c, d, e, f}) << '\n';              // prints 10
3  std::cout << std::max({a, b, c, d, e, f}) << '\n';              // prints 19
4  std::pair<int32_t, int32_t> p = std::minmax({a, b, c, d, e, f});
5  std::cout << p.first << '\n';                                   // prints 10
6  std::cout << p.second << '\n';                                  // prints 19
```

If you want to find the min or max value of an entire container, you should use the STL's std::min_element(), std::max_element(), or std::minmax_element() functions. These functions accept iterators to the sequence you want to identify the minimum or maximum for. Their behaviors are similar to the previous three functions, with the exception that the data is passed in using an iterator range as input.

```
template <typename ForwardIterator, typename Compare>
ForwardIterator std::min_element(ForwardIterator first, ForwardIterator last, Compare comp);
```
Returns an iterator pointing to the smallest element in the range [first, last). The comparator is optional, and comparisons are done using operator< if it is not specified.

```
template <typename ForwardIterator, typename Compare>
ForwardIterator std::max_element(ForwardIterator first, ForwardIterator last, Compare comp);
```
Returns an iterator pointing to the largest element in the range [first, last). The comparator is optional, and comparisons are done using operator< if it is not specified.

```
template <typename ForwardIterator, typename Compare>
std::pair<ForwardIterator, ForwardIterator>
std::minmax_element(ForwardIterator first, ForwardIterator last, Compare comp);
```
Returns a pair with an iterator pointing the smallest value in the range [first, last) as the first element, and an iterator pointer to the largest as the second. The comparator is optional, with comparisons done using operator< if not specified.

Examples using these functions are shown below:

```
std::vector<int32_t> vec = {6, 2, 4, 9, 8, 3, 7, 1, 5};
std::cout << *std::min_element(vec.begin(), vec.end()) << '\n';     // prints 1
std::cout << *std::max_element(vec.begin(), vec.end()) << '\n';     // prints 9
auto it_pair = std::minmax_element(vec.begin(), vec.end());
std::cout << *it_pair.first << '\n';                                // prints 1
std::cout << *it_pair.second << '\n';                               // prints 9
```

The time complexity of all these operations is linear on the number of elements that are compared (with the exception of `std::min()` and `std::max()` with only two arguments, which both take constant time).

※ 11.9.3 *Sorting Algorithms*

The algorithm library provides a `std::sort()` function, which can be used to sort elements within an iterator range `[first, last)`. An optional comparator may be passed in as a third argument; without the comparator specified, elements in the range are sorted in ascending order. Equivalent elements are not guaranteed to maintain their relative ordering with this function.

```
template <typename RandomAccessIterator, typename Compare>
void std::sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```
Sorts the elements in the range `[first, last)` in ascending order, or using the comparator `comp` if specified.

Without the comparator, elements are sorted in ascending order using `operator<` as the default. If a comparator is passed in, the order of elements after sorting is determined using the following rule: given two elements that are passed into the comparator, a return value of `true` indicates that the first element comes before the second in sorted order; otherwise, it comes after.

For example, if you wanted to sort the elements of a container in descending order, you would want to pass in a comparator that returns `true` only if the first element passed in is larger than the second (i.e., `comp(a, b)` returns `true` only if `a > b`, since that would indicate `a` comes before `b` in sorted order). This can be done using `std::greater<>` as the comparator, as shown below:

```
std::vector<int32_t> vec = {6, 2, 4, 9, 8, 3, 7, 1, 5};

// sort in ascending order -> no comparator needed (since std::less<> is default)
std::sort(vec.begin(), vec.end());
// ordering of elements after sort: 1 2 3 4 5 6 7 8 9

// sort in descending order -> pass in std::greater<> comparator
std::sort(vec.begin(), vec.end(), std::greater<int32_t>());
// ordering of elements after sort: 9 8 7 6 5 4 3 2 1
```

Since `std::sort()` takes in an iterator range, you can use it to partially sort a range of values within a container. In the following example, only the elements in the provided iterator range (4, 9, 8, and 3) end up being sorted relative to each other (since the sorting algorithm does not know about the existence of elements outside the iterator range):

```
std::vector<int32_t> vec = {6, 2, 4, 9, 8, 3, 7, 1, 5};

// only sort elements in range [vec.begin() + 2, vec.begin() + 6)
std::sort(vec.begin() + 2, vec.begin() + 6);
// ordering of elements after sort: 6 2 3 4 8 9 7 1 5
```

It is important to note that the algorithm library's sorting function requires random access iterators to work. This is because the underlying sorting implementation uses iterator arithmetic. As a result, if a container does not support random access iterators, the algorithm library's `std::sort()` function cannot be used to sort it.

Sorting also works with reverse iterators, as long as they support random access. When you pass a range of data into `std::sort()` using reverse iterators, the algorithm sees the data in reverse order. Thus, the final result you obtain will be the reverse of what the algorithm would have done with normal non-reverse iterators. For example, if you try to sort a *reverse* iterator range in ascending order, you will end up with the elements in *descending* order!

```
std::vector<int32_t> u = {4, 2, 7, 8, 3, 6};
std::vector<int32_t> v = {4, 2, 7, 8, 3, 6};

std::sort(u.begin(), u.end());                  // order of elements is 2 3 4 6 7 8
std::sort(v.rbegin(), v.rend());                // order of elements is 8 7 6 4 3 2

std::vector<int32_t> a = {4, 2, 7, 8, 3, 6};
std::vector<int32_t> b = {4, 2, 7, 8, 3, 6};

std::sort(a.begin() + 2, a.begin() + 5);        // order of elements is 4 2 3 7 8 6
std::sort(b.rbegin() + 1, b.rbegin() + 4);      // order of elements is 4 2 8 7 3 6
```

The time complexity of `std::sort()` given n elements is $\Theta(n\log(n))$ (the reason why will be covered in chapter 14).

Another sorting algorithm that is provided by the algorithm library is the `std::partial_sort()` function:

```
template <typename RandomAccessIterator, typename Compare>
void std::partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                       RandomAccessIterator last, Compare comp);
```
Modifies the order of elements in the range `[first, last)` so that all elements in the range `[first, middle)` are correct had the entire range from `[first, last)` been sorted. Without the optional comparator `comp`, elements in the range are sorted using `operator<` (i.e., ascending order).

The `std::partial_sort()` operation can be helpful if the information you need does not require an entire container to be sorted (e.g., if you wanted to identify the three smallest values in a vector of a million elements).

```
1  std::vector<int32_t> vec = {4, 2, 7, 5, 3, 6};
2
3  // ensures smallest three elements are in the correct position
4  std::partial_sort(vec.begin(), vec.begin() + 3, vec.end());
5  // ordering of elements: 2 3 4 7 5 6 (only 2 3 4 are guaranteed to be in correct
6  // position, other elements can be in an arbitrary order)
```

An additional sorting algorithm is the `std::stable_sort()` algorithm, which ensures that the relative order of equivalent elements is preserved. We will discuss stable sorts in more detail in chapter 14.

```
template <typename RandomAccessIterator, typename Compare>
void std::stable_sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```
Sorts the elements in the range `[first, last)` in ascending order, or using the comparator `comp` if specified. If two elements are equivalent, their relative ordering is preserved after the sort completes (i.e., if `a==b`, and `a` came before `b` prior to sorting, `a` will still come before `b` after sorting).

※ 11.9.4 *Find Algorithms*

The algorithm library also provides several useful functions that can be used to find a specific element within an iterator range. The first of these is the `std::find()` function, which returns an iterator to the first element in the range `[first, last)` that has a value equal to `val`. If no such value is found, the function returns the iterator `last`. This function uses `operator==` to compare elements in the range with `val`:

```
template <typename InputIterator, typename T>
InputIterator std::find(InputIterator first, InputIterator last, const T& val);
```
Returns an iterator to the first element in the range `[first, last)` equal to `val`. If no match is found, `last` is returned.

Because this algorithm compares all elements in the given range for a match, the complexity of the `std::find()` operation is linear in the distance between `first` and `last`.

In addition to `std::find()`, you can use the `std::find_if()` function to find an element in an iterator range that satisfies some predefined condition. The `std::find_if()` function takes in two iterators and a unary predicate, and it returns an iterator to the first element in the range `[first, last)` for which the unary predicate returns `true`. If no such element is found, the function returns the iterator `last`.

```
template <typename InputIterator, typename UnaryPredicate>
InputIterator std::find_if(InputIterator first, InputIterator last, UnaryPredicate pred);
```
Returns an iterator to the first element in the range `[first, last)` that satisfies `pred`. If no such element is found, the function returns the iterator `last`.

Examples using `find()` algorithms are shown below:

```
1  struct IsOddPred {
2    bool operator() (int32_t i) {
3      return i % 2 == 1;
4    } // operator()()
5  };
6
7  int main() {
8    std::vector<int32_t> vec = {24, 66, 12, 74, 26, 43, 92, 95, 92, 71};
9    auto it1 = std::find(vec.begin(), vec.end(), 92);
10   auto it2 = std::find(vec.begin(), vec.end(), 93);
11   auto it3 = std::find_if(vec.begin(), vec.end(), IsOddPred());
12 } // main()
```

In this case, `it1` points to the first `92` in the vector (the one between `43` and `95`), `it2` points to `vec.end()` since `93` cannot be found in the vector, and `it3` points to `43` because `43` is the first element for which `IsOddPredicate` returns `true` (since it is the first odd number).

※ 11.9.5 *Removal Algorithms (⁎)*

The `<algorithm>` library provides functions that can be used to remove elements in an iterator range that fit a certain condition. The `std::remove()` function receives an iterator range `[first, last)` and a target value `val` to remove, and it "removes" all elements in the range that compare equal to `val`.

```
template <typename ForwardIterator, typename T>
ForwardIterator std::remove(ForwardIterator first, ForwardIterator last, const T& val);
```
Transforms the range `[first, last)` into a range with all the elements that compare equal to `val` removed, and returns an iterator to the new end of the range.

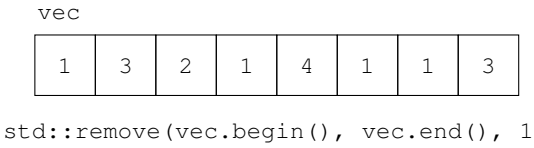
However, there is a catch: despite the function's name, the `std::remove()` function does not actually physically remove the elements from the range. As mentioned in the beginning of the section, STL algorithms only receive iterators to data, and they have no power to modify the properties of the underlying container itself. As a result, `std::remove()` works around this by moving all the elements that should be removed to the back of the iterator range. After doing this, it returns an iterator to the position *one past* the last element that should not be removed. *It is your responsibility to physically remove these elements from the container — this can be done by erasing all contents from the iterator that is returned to the end of the iterator range.* This is known as the **erase-remove idiom**.

Thus, every time you call `std::remove()` (or any variant of this operation), you must also call `.erase()` to erase all elements starting from the iterator that `std::remove()` returns to the end of the container:

```
auto it = std::remove(vec.begin(), vec.end(), 1);
vec.erase(it, vec.end());
```

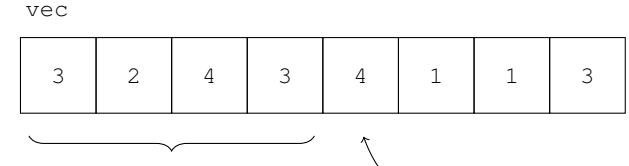
This can be condensed to one line of code without creating an additional variable for the iterator:

```
vec.erase(std::remove(vec.begin(), vec.end(), 1), vec.end());
```



`std::remove(vec.begin(), vec.end(), 1);`

Algorithms in the algorithm library cannot modify the structure of the vector, so the 1's in the container are not physically removed. Instead, all positions that follow the last unremoved element contain junk data; it is the programmer's responsibility to clean up this part of the container since the `std::remove()` function cannot.



This is what the container should look like after the 1's are removed.

`std::remove()` cannot modify the size of the vector on its own, so it returns an iterator to this element; it is your responsibility to erase all elements in the range from here to the end!

Another variant of `std::remove()` is `std::remove_if()`, which removes elements that satisfy a given unary predicate.

```
template <typename ForwardIterator, typename UnaryPredicate>
ForwardIterator std::remove_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred);
```
Transforms the range `[first, last)` into a range with all the elements for which `pred` returns `true` removed, and returns an iterator to the new end of the range.

Like with `std::remove()`, `std::remove_if()` does not change the size of the underlying container that stores the iterator range. To physically eliminate the values that should be removed, `std::remove_if()` should be coupled with `.erase()`. The following code uses the `IsOddPred()` predicate defined earlier.

```
auto it = std::remove_if(vec.begin(), vec.end(), IsOddPred());
vec.erase(it, vec.end());
```

This can be condensed to one line of code without creating an additional variable for the iterator:

```
vec.erase(std::remove_if(vec.begin(), vec.end(), IsOddPred()), vec.end());
```

Examples of these two functions in use are shown below:

```
1  std::vector<int32_t> v1 = {1, 3, 2, 1, 4, 1, 1, 3};
2  // remove all 1's from the above vector
3  v1.erase(std::remove(v1.begin(), v1.end(), 1), v1.end());
4  // contents of v1 are {3, 2, 4, 3}
5
6  std::vector<int32_t> v2 = {1, 3, 2, 1, 4, 1, 1, 3};
7  // remove all odd numbers from the vector using IsOddPred functor
8  v2.erase(std::remove_if(v2.begin(), v2.end(), IsOddPred()), v2.end());
9  // contents of v2 are {2, 4}
```

If you instead wanted to make a copy of the iterator range with specific values removed (instead of modifying the original container), you could use the `std::remove_copy()` and `std::remove_copy_if()` functions to write non-removed values to a new location.

```
template <typename ForwardIterator, typename OutputIterator, typename T>
OutputIterator std::remove_copy(ForwardIterator first, ForwardIterator last,
                                OutputIterator dest, const T& val);
```
Copies the elements in the range `[first, last)` to the range beginning at `dest`, except elements that compare equal to `val`.

```
template <typename ForwardIterator, typename OutputIterator, typename UnaryPredicate>
OutputIterator std::remove_copy_if(ForwardIterator first, ForwardIterator last,
                                   OutputIterator dest, UnaryPredicate pred);
```
Copies the elements in the range `[first, last)` to the range beginning at `dest`, except elements for which `pred` returns `true`.

The `std::remove_copy()` operation behaves similarly to `std::remove()`, except that the result of `std::remove()` is written to `dest` (leaving the original container unchanged). The same applies with `std::remove_copy_if()`. Examples are shown below:

```
std::vector<int32_t> v1 = {1, 3, 2, 1, 4, 1, 1, 3};
// init v2 to the same size as v1
std::vector<int32_t> v2(v1.size());
// remove all 1's from v1 and store the result in v2; v1 unchanged
std::remove_copy(v1.begin(), v1.end(), v2.begin(), 1);
// v2 now stores {3, 2, 4, 3, 0, 0, 0, 0}
// remove excess zeros from v2 using the standard remove function
auto it = std::remove(v2.begin(), v2.end(), 0);
v2.erase(it, v2.end());
// v2 now stores {3, 2, 4, 3}
```

All of these removal algorithms take linear time in the distance between `first` and `last`, since each element in the given range is compared with the value to remove.

Remark: Beginning in C++20, certain containers gain access to `std::erase_if()`, which physically erases elements from the container. For instance, the following removes all odd numbers from a vector, similar to the example covered earlier with `std::remove_if()`:

```
struct IsOddPred {
  bool operator() (int32_t i) {
    return i % 2 == 1;
  } // operator()()
};

int main() {
  std::vector<int32_t> v = {1, 3, 2, 1, 4, 1, 1, 3};
  auto num_values_erased = std::erase_if(v, IsOddPred());  // contents of v are {2, 4}
  std::cout << num_values_erased << '\n';                  // prints 6
} // main()
```

This is not something you will need to know, however.

※ 11.9.6 *Replacement Algorithms* (✻)

Along with these removal algorithms, the STL `<algorithm>` also provides algorithms that can be used to replace elements instead of removing them. The functions are shown below:

```
template <typename ForwardIterator, typename T>
void std::replace(ForwardIterator first, ForwardIterator last, const T& old_val, const T& new_val);
```
Replaces all instances of `old_val` in the range `[first, last)` with `new_val`.

```
template <typename ForwardIterator, typename UnaryPredicate, typename T>
void std::replace_if(ForwardIterator first, ForwardIterator last, UnaryPredicate pred, const T& new_val);
```
Replaces all elements for which `pred` returns `true` in the range `[first, last)` with `new_val`.

```
template <typename InputIterator, typename OutputIterator, typename T>
OutputIterator std::replace_copy(InputIterator first, InputIterator last, OutputIterator dest,
                                 const T& old_val, const T& new_val);
```
Copies elements in the range `[first, last)` to the range beginning at `dest`, replacing all instances of `old_val` with `new_val`.

```
template <typename InputIterator, typename OutputIterator, typename UnaryPredicate, typename T>
OutputIterator std::replace_copy_if(InputIterator first, InputIterator last, OutputIterator dest,
                                    UnaryPredicate pred, const T& new_val);
```
Copies the elements in the range `[first, last)` to the range beginning at `dest`, replacing all values for which `pred` returns `true` with `new_val`.

Like with `std::remove()`, these replacement algorithms take linear time in the distance between `first` and `last`. Examples using these functions are shown below:

```
std::vector<int32_t> vec = {1, 3, 2, 1, 1, 7, 3, 6, 4, 5};

// replace all 1's with 0's
std::replace(vec.begin(), vec.end(), 1, 0);
// contents of vec now {0, 3, 2, 0, 0, 7, 3, 6, 4, 5}

// replace all odd numbers with 8
std::replace_if(vec.begin(), vec.end(), IsOddPred(), 8);
// contents of vec now {0, 8, 2, 0, 0, 8, 8, 6, 4, 8}

// replace all 8's with 9's and store result in foo
std::vector<int32_t> foo(vec.size());
std::replace_copy(vec.begin(), vec.end(), foo.begin(), 8, 9);
// contents of vec now {0, 8, 2, 0, 0, 8, 8, 6, 4, 8}
// contents of foo now {0, 9, 2, 0, 0, 9, 9, 6, 4, 9}

// replace all odd numbers with 4 and store result in bar
std::vector<int32_t> bar(foo.size());
std::replace_copy_if(foo.begin(), foo.end(), bar.begin(), IsOddPred(), 4);
// contents of vec now {0, 8, 2, 0, 0, 8, 8, 6, 4, 8}
// contents of foo now {0, 9, 2, 0, 0, 9, 9, 6, 4, 9}
// contents of bar now {0, 4, 2, 0, 0, 4, 4, 6, 4, 4}
```

※ 11.9.7 *Transformation Algorithms* (⁎)

The `std::transform()` function in the `<algorithm>` library can be used to apply an operation to every element in an iterator range. The function can be invoked in two forms, which are shown below:

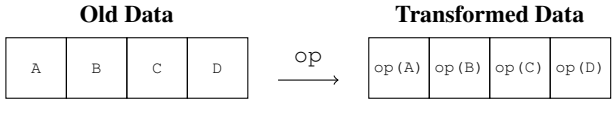
```
template <typename InputIterator, typename OutputIterator, typename UnaryOperation>
OutputIterator std::transform(InputIterator first1, InputIterator last1,
                              OutputIterator result, UnaryOperation op);
```

Calls the operation `op` on each element in the range `[first1, last1)` and stores the value returned by each operation in the range beginning at `result`. An iterator pointing one past the last element written is returned.

```
template <typename InputIterator1, typename InputIterator2, typename OutputIterator, typename BinaryOperation>
OutputIterator std::transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
                              OutputIterator result, BinaryOperation binary_op);
```

Calls `binary_op` with each element in the range `[first1, last1)` as the first argument, and the corresponding element in the range beginning at `first2` as the second argument. The value returned by each call is stored at the range beginning at `result`. An iterator pointing one past the last element written is returned.

The first variant takes in an unary operation, or an operation that takes in a single argument. The `std::transform()` function applies this unary operation to every element in the given range and writes the result to the range beginning at `result`. The caller of the function must make sure that the range beginning at `result` is large enough to hold the transformed elements (by either resizing the output range or using insert iterators). To physically overwrite the elements in the original range with the transformed data, `std::transform()` can be called with `first1` and `result` as the exact same iterator.



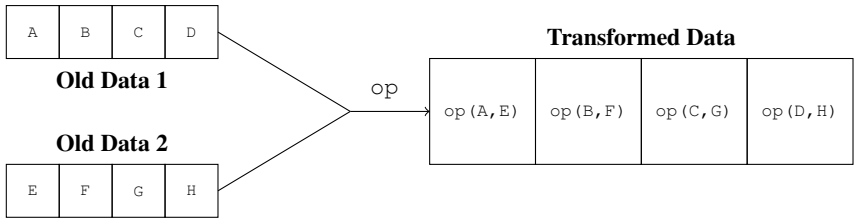

```
std::transform(old_data.begin(), old_data.end(), new_data.begin(), op);
```

As an example, consider the `::toupper()` function, which can be used to turn a lowercase letter into an uppercase one. However, this function only operates on one character at a time. If you want to convert an entire string from lowercase to uppercase, you can use `std::transform()` to apply the `::toupper()` function to every letter in the string.[4]

```
std::string s = "eecs281";
std::transform(s.begin(), s.end(), s.begin(), ::toupper);
std::cout << s << '\n';  // prints "EECS281"
```

[4] If you are `using namespace std`, you need the two colons before `toupper` to tell the program *not* to use `std::toupper()`, which is a separate function defined in the `<locale>` library. Instead, you want to use the non-`std` version provided in the C library.

The second variant of `std::transform()` takes in a binary operation, which takes two arguments rather than one. This binary operation is applied to the contents of *two* iterator ranges. The `std::transform()` goes through both iterator ranges and passes in each element into the first range as the first argument of the binary operation, and each element into the second range as the second argument of the binary operation.



For example, the following code applies the function `binary_op()` to both iterator ranges, where elements in `v1` are passed in as `a` and elements in `v2` are passed in as `b`. The final contents of the vector `v3` (the destination vector) are `{26, 38, 52, 68, 86}`, since $5^2 + 1 = 26$, $6^2 + 2 = 38$, $7^2 + 3 = 52$, $8^2 + 4 = 68$, and $9^2 + 5 = 86$.

```
1  int32_t binary_op(int32_t a, int32_t b) {
2    return a * a + b;
3  } // binary_op()
4
5  int main() {
6    std::vector<int32_t> v1 = {5, 6, 7, 8, 9};
7    std::vector<int32_t> v2 = {1, 2, 3, 4, 5};
8    std::vector<int32_t> v3; // stores v1 * v1 + v2
9    std::transform(v1.begin(), v1.end(), v2.begin(), std::back_inserter(v3), binary_op);
10 } // main()
```

※ 11.9.8 *Copy Algorithms* (✻)

The `<algorithm>` library provides several methods that can be used to copy data between containers. The `std::copy()` function can be used to copy elements in an iterator range to another location. Given three iterators, `first`, `last`, and `dest`, the function copies the elements in the range `[first, last)` into the range that begins at the position of `dest`. An iterator to the end of the destination range, or the element that follows the last element copied, is returned by the `std::copy()` function.

```
template <typename InputIterator, typename OutputIterator>
OutputIterator std::copy(InputIterator first, InputIterator last, OutputIterator dest);
```
Copies the elements in the range `[first, last)` into the range beginning at `dest`, returns an iterator one past the last element copied.

The iterator `dest` should not point to an element in the range `[first, last)`, since this causes overlap problems. Because the `std::copy()` operation performs an assignment for each element in the range, this function is linear in the distance between `first` and `last`.

The following code uses `std::copy()` to copy a range of data from one container to another. Since the `std::copy()` algorithm accepts iterators and operates independently of the underlying container that the data is stored in, it can be used to copy data even if the input and output containers are different.

```
1  std::vector<int32_t> vec = {0, 1, 2, 3, 4, 5, 6, 7};
2  std::deque<int32_t> deq(4); // deque of size 4
3  std::copy(vec.begin() + 2, vec.begin() + 6, deq.begin());
4  // elements 2, 3, 4, and 5 are copied to the deque
5  // contents of deque now {2, 3, 4, 5}
```

If you only want to copy elements that fit a certain criteria, you can use `std::copy_if()`, which takes in a unary predicate. This function only copies over elements for which `pred` returns `true`:

```
template <typename InputIterator, typename OutputIterator, typename UnaryPredicate>
OutputIterator std::copy_if(InputIterator first, InputIterator last,
                            OutputIterator dest, UnaryPredicate pred);
```
Copies the elements in the range `[first, last)` for which `pred` returns `true` into the range beginning at `dest` and returns an iterator one past the last element copied.

The `std::copy_if()` function returns an iterator pointing to the position that follows the last element written to `dest`. This is useful since it allows you to shrink a container to the correct size after a call to the function, since you may not know how many elements will be copied over. Since the iterator returned is an output iterator, it also allows you to continue writing data from where you left off.

```
1  std::vector<int32_t> vec = {4, 2, 7, 5, 3, 6};
2  // copy over only odd numbers, init to size of vec to ensure space
3  std::vector<int32_t> odds(vec.size());
4  auto end = std::copy_if(vec.begin(), vec.end(), odds.begin(), IsOddPred());
5  // contents of odds are {7, 5, 3, 0, 0, 0}, resize to correct size
6  odds.resize(std::distance(odds.begin(), end));
7  // contents of odds are {7, 5, 3}
```

※ 11.9.9 *Reversal Algorithms (✻)*

The `std::reverse()` function takes in a range `[first, last)` and reverses the order of the elements in this range. Similarly, the `std::reverse_copy()` function does the same thing, but it copies the result to a new location beginning at `dest` instead of modifying the data in the original container.

```
template <typename BidirectionalIterator>
void std::reverse(BidirectionalIterator first, BidirectionalIterator last);
```
Reverses the order of elements in the range `[first, last)`.

```
template <typename BidirectionalIterator, typename OutputIterator>
OutputIterator std::reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
                                 OutputIterator dest);
```
Writes the elements in the range `[first, last)` to the range beginning at `dest` in reverse order and returns an iterator one past the last element copied.

Example code that uses these reverse functions is shown below:

```
1  std::vector<int32_t> vec = {1, 2, 6, 3, 8, 9, 4, 5, 7};
2  std::reverse(vec.begin(), vec.end());
3  // contents of vec now {7, 5, 4, 9, 8, 3, 6, 2, 1}
4  std::reverse(vec.begin() + 3, vec.begin() + 6);
5  // contents of vec now {7, 5, 4, 3, 8, 9, 6, 2, 1}
```

※ 11.9.10 *n^{th} Element (✻)*

Suppose you wanted to find the 281st smallest integer in a container of 1,000,000 integers. One way to find this number is to sort the container and retrieve the 281st element. However, `std::sort()` takes $\Theta(n \log(n))$ time, and a lot of time is wasted on sorting the remaining 999,999 elements, which you may not even care about. Is there a way to find this element without sorting the entire container?

The `std::nth_element()` can help you with this. If you have a container that supports random access iterators, you can use `std::nth_element()` to find the element that should go in the n^{th} position if the container were sorted.

```
template <typename RandomAccessIterator, typename Compare>
void std::nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                      RandomAccessIterator last, Compare comp);
```
Rearranges the elements in the range `[first, last)` so that the element pointed to by `nth` is the element that should be at that position if the range had been completely sorted. The comparator is optional; if you do not pass in a comparator, elements in the container are compared using `operator<` by default.

The benefit of using `std::nth_element()` instead of `std::sort()` is that a call to `std::nth_element()` takes $\Theta(n)$ time, where n is the distance between `first` and `last`. This function has many practical uses. For instance, you can use `std::nth_element()` to find the median of a collection of data values without having to sort the entire collection:

```
1  std::vector<int32_t> vec = {14, 33, 74, 11, 25, 75, 24, 47, 85, 17, 59};
2  size_t middle_idx = vec.size() / 2;
3  std::nth_element(vec.begin(), vec.begin() + middle_idx, vec.end());
4  // now the median is guaranteed to be in the correct position of
5  // the vector if the vector had been fully sorted
6  std::cout << vec[middle_idx] << '\n';  // prints 33
```

The remaining elements in the vector are left in an arbitrary order, and they are not guaranteed to be sorted. The only guarantee that you can make after a call to `std::nth_element()` for the n^{th} element is that none of the elements preceding the position of this element are greater, and that none of the elements following this position are smaller. In the above example, `33` is guaranteed to be in the correct position, since `33` would have been at position `vec.begin() + middle_idx` if the vector were fully sorted. However, the other elements can follow any order, as long as elements to the left of `vec.begin() + middle_idx` are less than or equal to `33`, and elements to the right of `vec.begin() + middle_idx` are greater than or equal to `33`.

※ 11.9.11 *Lower and Upper Bound Algorithms*

The STL also provides several algorithms that can be used if the data in a container is in *sorted* order. The following algorithms can be used to identify if a value exists in a sorted range of data, and if it doesn't, it identifies where it should go to keep the data sorted. The `std::lower_bound()` function takes in an iterator range `[first, last)` and returns an iterator to the first element in the range that does not compare less than `val`. The `std::upper_bound()` function takes in an iterator range and returns an iterator to the first element in the range that compares greater than `val`. If no such element fits these criteria, both functions return `last`.

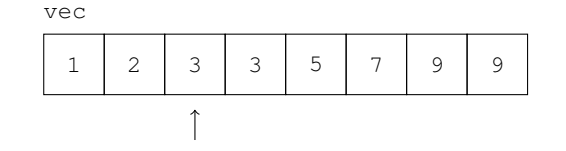
```
template <typename ForwardIterator, typename T, typename Compare>
ForwardIterator std::lower_bound(ForwardIterator first, ForwardIterator last,
                                 const T& val, Compare comp);
```
Returns an iterator pointing to the first element in the sorted range `[first, last)` that does not compare less than `val`. The comparator is optional; without it, elements are compared using `operator<`. The value pointed to by the return iterator may also be equivalent to `val`, and not just greater.

```
template <typename ForwardIterator, typename T, typename Compare>
ForwardIterator std::upper_bound(ForwardIterator first, ForwardIterator last,
                                 const T& val, Compare comp);
```

Returns an iterator pointing to the first element in the sorted range `[first, last)` that compares greater than `val`. The comparator is optional; without it, elements are compared using `operator<`. The value pointed to by the return iterator cannot be equivalent to `val`, only greater.

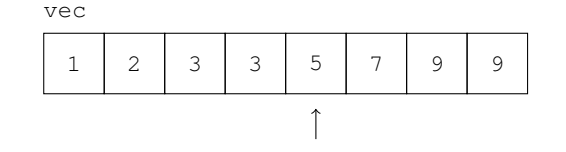
The data passed into `std::lower_bound()` and `std::upper_bound()` must be sorted for these functions to work. If the data is not sorted, then the behaviors of these two functions are undefined. Also, notice that, if `val` did exist in a container, then `[lower_bound, upper_bound)` would represent an iterator range whose elements are all equal to `val` (assuming no custom comparator is used).

As an example, consider the vector below. Using `std::lower_bound()` on the value `3` returns an iterator to the first `3` in the container. Using `std::upper_bound()` on the value `3` returns an iterator to the position *one past* the last `3` in the container. This effectively constructs an iterator range that provides a bound for where the element `3` can be found.



`std::lower_bound(vec.begin(), vec.end(), 3);`

(returns first element greater than or equal to 3)



`std::upper_bound(vec.begin(), vec.end(), 3);`

(returns first element strictly greater than 3)

The `std::equal_range()` function combines the behavior of `std::lower_bound()` and `std::upper_bound()` to return an iterator range whose values are all equal to `val`. If `val` is not in the iterator range, `std::equal_range()` returns two iterators that point to the closest value greater than `val`, or `last` if `val` compares greater to all elements in the range.

```
template <typename ForwardIterator, typename T, typename Compare>
std::pair<ForwardIterator, ForwardIterator>
  std::equal_range(ForwardIterator first, ForwardIterator last, const T& val, Compare comp);
```

Returns the bounds of the subrange that includes all the elements of the sorted range `[first, last)` with values equivalent to `val`. The comparator is optional, and values are compared with `operator<` if it is not specified. If `val` is not equivalent to any value in the range, the subrange returned has a length of zero.

When `std::equal_range()` is called on a value `val`, it returns a pair of iterators, where `first` represents the lower bound on `val` and `second` represents the upper bound on `val`. Examples using these functions are shown below:

```
1  std::vector<int32_t> vec = {1, 2, 5, 5, 6, 9};
2
3  // it1 points to first 5
4  auto it1 = std::lower_bound(vec.begin(), vec.end(), 5);
5
6  // it2 points to 6
7  auto it2 = std::upper_bound(vec.begin(), vec.end(), 5);
8
9  // it3 points to 9
10 auto it3 = std::lower_bound(vec.begin(), vec.end(), 7);
11
12 // it4 points to 9
13 auto it4 = std::upper_bound(vec.begin(), vec.end(), 7);
14
15 // it5 points to vec.end()
16 auto it5 = std::lower_bound(vec.begin(), vec.end(), 10);
17
18 // it6 points to vec.end()
19 auto it6 = std::upper_bound(vec.begin(), vec.end(), 10);
20
21 // it_pair1.first points to first 5
22 // it_pair1.second points to 6
23 auto it_pair1 = std::equal_range(vec.begin(), vec.end(), 5);
24
25 // it_pair2.first points to 1
26 // it_pair2.second points to 1
27 auto it_pair2 = std::equal_range(vec.begin(), vec.end(), 0);
```

※ 11.9.12 *Other Algorithms in the STL*

There are a few important algorithms that were not covered in this section, but will be covered in a later chapter. These include the set algorithms of `set_union()`, `set_intersection()`, `set_difference()`, and `set_symmetric_difference()`, which will be covered in chapter 13, as well as the sequence operations of `equal()` and `unique()`, which will be covered in chapter 16. To learn more about these methods, you should consult their corresponding sections in the chapters listed above.

Additionally, there are a few algorithms that will not be covered in these notes at all. This does not mean that they are not important, however. You are encouraged to familiarize yourself with everything that the `<algorithm>` library provides (as well as other libraries) — there are a lot of resources for this, such as cppreference.com, that provide the most up-to-date features of the language.

11.10 Lambdas (⁎)

※ 11.10.1 *Anatomy of a Lambda* (⁎)

Recall the `IsOddPred()` function object we created to find the first odd number in a container:

```
1  struct IsOddPred {
2    bool operator() (int32_t i) {
3      return i % 2 == 1;
4    } // operator()()
5  };
6
7  int main() {
8    std::vector<int32_t> vec = {24, 66, 12, 74, 26, 43, 92, 95, 92, 71};
9    auto it = std::find_if(vec.begin(), vec.end(), IsOddPred());
10 } // main()
```

Here, we defined a function object and gave it the name `IsOddPred`. However, if this function object is only used once, it is a bit excessive to define a separate `struct` and implement a special-case function object with its own name just to use it once and never touch it again. It would be much better if we could implement the function object "on the fly" when it is actually used, within the `std::find_if()` function.

The **lambda** is a construct introduced in C++11 to address this issue. Lambdas can be used to construct a short-lived, anonymous function object *in place*, directly in the code where it is called. For example, the code below reimplements the code above, but defines the `IsOddPred()` predicate in place using a lambda, directly where we want to use it:

```
1  int main() {
2    std::vector<int32_t> vec = {24, 66, 12, 74, 26, 43, 92, 95, 92, 71};
3    auto it = std::find_if(vec.begin(), vec.end(), [](int32_t i) {
4      return i % 2 == 1;
5    });
6  } // main()
```

How do you declare a lambda? A lambda expression consists of the following:

```
[capture clause] (parameter list) {function body}
```

A lambda always begins with a pair of square brackets, known as the *capture clause*. Following the square brackets is a pair of parentheses that holds the *parameter list*. This component stores the arguments that would normally be passed into the function.

For instance, the following function checks if a number is odd. Here, the arguments of the function (i.e., `int n`) would go in the parameter list of the lambda.

```
1  bool is_odd(int32_t i) {
2    return i % 2 == 1;
3  } // is_odd()
```

Finally, the behavior of the function object should be defined in the function body, between the curly braces. Putting it all together, we can convert the above function into a lambda by replacing the function name and return type with square brackets:

```
1  [](int32_t i) {
2    return i % 2 == 1;
3  }
```

We can now pass this lambda as a function object into STL algorithms. This is exactly what we did with `std::find_if()` above; instead of defining a separate function object elsewhere, we just implemented the function we wanted "on demand" when we called `std::find_if()`. This makes using STL algorithms a lot easier.

Remark: Starting in C++20, lambdas can be templated, with the template parameter coming between the capture clause and parameter list.

```
[capture clause] <template  parameter> (parameter list) {function body}
```

As an example, the following templated function adds two values of the same type.

```
template <typename T>
T add(T first, T second) {
  return first + second;
} // add()
```

Converting this to a templated lambda gives us the following:

```
[]<typename T>(T first, T second) { return first + second; }
```

Because a lambda behaves similarly to a function object, you can pass it an argument as if the expression

```
[](int n){ return i % 2 == 1; }
```

were the name of the function object itself:

```
std::cout << [](int32_t i){ return i % 2 == 1; }(5) << '\n'; // prints 1 (true), 5 is odd
std::cout << [](int32_t i){ return i % 2 == 1; }(6) << '\n'; // prints 0 (false), 6 is even
```

You can also store the lambda as a variable and use it as a function object:

```
auto is_odd_pred = [](int32_t i){ return i % 2 == 1; };
std::cout << is_odd_pred(5) << '\n';  // prints 1 (true), 5 is odd
std::cout << is_odd_pred(6) << '\n';  // prints 0 (false), 6 is even
```

Remark: What is the type of a lambda? It turns out that the type of a lambda is unspecified, and each instance of a lambda introduces its own unique type. This type is unnamed and cannot be explicitly written out, so you should use `auto` to deduce the type of a lambda for you. If you want to pass a lambda into a function, you can either use a templated function or convert the lambda to a `std::function<>` object (which we will discuss later). The templated function, however, is preferred, since `std::function<>` is a wrapper that adds a level of indirection and has a bit more overhead, and its type will also need to be changed if the return value or arguments of a lambda are changed.

Much like a normal function, a lambda does not need to accept any parameters. Consider the following function:

```
void print_hello_world() {
  std::cout << "hello world" << '\n';
} // print_hello_world()
```

By removing the function name and adding square brackets, we can convert this into a lambda as follows. This lambda emulates a function object that prints out `"hello world"` whenever it is invoked.

```
[]() {
  std::cout << "hello world" << '\n';
}
```

Lambdas can automatically deduce the return type of its function body in most cases, so there is no need to define the return type of a lambda definition in most cases. However, if you want to specify a return type that is different from the type that would normally be returned (e.g., using a conversion), you can use the arrow operator as follows:

```
[capture clause] (parameter list) -> return_type {function body}
```

For example, the following lambda returns a `double`:

```
[](double x, double y) {
  return x + y;
}
```

If you want this lambda to return an integer instead, you would have to specify it as the return type:

```
[](double x, double y) -> int32_t {
  return x + y;
}
```

If we called our original lambda (without the return type) with `x = 1.23` and `y = 2.46`, we would get a return value of `3.69`:

```
std::cout << [](double x, double y){ return x + y; }(1.23, 2.46) << '\n';
```

On the other hand, if we specify the return type as an integer, the return value would be `3`, since `3.69` would get truncated during conversion:

```
std::cout << [](double x, double y) -> int32_t { return x + y; }(1.23, 2.46) << '\n';
```

Another feature of lambdas is *variable capture*. This feature gives lambdas the ability to capture and use variables that are in the surrounding scope at the time of the lambda's creation. To capture a variable, place it within the square brackets (the *capture clause*), as shown in the example below. A variable that is captured by the lambda may be used within its function body.

```
int main() {
  std::vector<int32_t> vec = {24, 66, 12, 74, 26, 43, 92, 95, 92, 71};
  int32_t factor;
  std::cin >> factor;
  std::cout << "The first number that is a multiple of " << factor << " is "
            << *std::find_if(vec.begin(), vec.end(), [factor](int32_t i) { return i % factor == 0; });
} // main()
```

In this code, the lambda captures and stores a read-only copy of the `factor` variable.[5] Even though the `factor` variable was not declared in the body of the lambda, we are still allowed to use it if we capture the variable in our capture clause.

It is important to note that the lambda grabs and saves a *copy* of a variable when it is captured. If the original variable is modified outside the lambda, the copy stored inside the lambda does not change. In the following example, the lambda grabs a copy of `var` when `var` equals `281`. Because the lambda saved an internal copy, even if `var` is modified outside the lambda, the value of `var` within the lambda would still be `281`.

```
int32_t var = 281;
auto func = [var]() { std::cout << var << '\n'; };  // function object that prints var

// the lambda grabbed var and made a copy; changing var now does NOT
// change the value of var for the lambda
++var;   // var is now 282
func();  // this prints out 281 since the lambda's copy of var is still equal to 281
```

If you want the lambda to capture a *reference* to a variable rather than a copy, you must specify it in the capture clause by using an ampersand:[6]

```
int32_t var = 281;
auto func = [&var]() { std::cout << var << '\n'; };  // function object that prints var

// the lambda grabbed var as a reference, so changing var outside the
// lambda also changes the value of var inside the lambda
++var;  // var is now 282
func(); // this prints out 282 since the lambda's stores a reference to var
```

This also works the other way around: changes made to a reference in the lambda are reflected in the outside code:

```
int32_t var = 281;
auto func = [&var]() { var += 10; };  // adds 10 to var
func();                               // lambda runs, var becomes 291
std::cout << var << '\n';             // prints 291
```

There are several ways to capture variables in a capture clause. Several of these options are shown below:

- `[]` captures no variables
- `[=]` captures all local variables used in the lambda by value (i.e., a copy is made)
- `[&]` captures all local variables used in the lambda by reference
- `[foo]` captures only the variable `foo` by making a copy
- `[&foo]` captures only the variable `foo`, but by reference
- `[foo, bar]` captures only the variables `foo` and `bar`, both by value
- `[=, &foo]` captures all local variables used in the lambda by value, except `foo`, which is captured by reference
- `[&, foo]` captures all local variables used in the lambda by reference, except `foo`, which is captured by value
- `[this]` captures a reference to the current object — this is useful if you are trying to declare a lambda within an object's member function, and the lambda needs access to that object's member variables

Capturing everything by value or reference (i.e., `[=]` and `[&]`) is generally bad practice and should be avoided, unless you actually want to capture everything in the surrounding scope. It is better style to explicitly select the variables that you want to capture when defining a lambda.

[5] To modify variables captured by value, add the `mutable` keyword directly after the parameter list (i.e., `[] () mutable {}`).

[6] If you capture variables by reference, you must make sure that the variables you capture do not go out of scope before you use the lambda. Otherwise, you may get undefined behavior!

※ 11.10.2 *Lambdas and the STL* (✻)

Lambdas can be quite powerful when combined with the STL. One particular algorithm that works well with lambdas is the `std::for_each()` function in the `<algorithm>` library, which applies a function to every element in an iterator range. Unlike a range-for loop, `for_each()` makes it easy to apply an operation over a range of data rather than an entire container.

```
template <typename InputIterator, typename UnaryFunctor>
UnaryFunctor std::for_each(InputIterator first, InputIterator last, UnaryFunctor fn);
```
Applies the function `fn` to each element in the range `[first, last)` and returns `fn` after completion.

For example, the following code adds 281 to every element in an iterator range:

```
1  std::vector<int32_t> vec = {1, 2, 3, 4, 5, 6, 7, 8};
2  std::for_each(vec.begin(), vec.end(), [](int32_t& i) { i += 281; });
3  // vec now {282, 283, 284, 285, 286, 287, 288, 289}
```

When you run `std::for_each()`, every element in the provided iterator range is passed in as an argument to the lambda. Thus, if you want to modify the elements in the range, make sure to pass the variable in the parameter list by reference:

```
[](int32_t& i) { i += 281; }
```

A few more examples are shown below. The following code sorts a vector of students by age and prints out their names in sorted order. Using lambdas, we can define the comparator in the `std::sort()` function call instead of in a separate comparator object.

```
1  struct Student {
2    std::string name;
3    int32_t age;
4  };
5
6  std::vector<Student> vec = { {"Alice", 21}, {"Bob",  20}, {"Cathy", 16},
7                               {"Drew",  19}, {"Erin", 22}, {"Frank", 17} };
8
9  std::sort(vec.begin(), vec.end(), [](const Student& lhs, const Student& rhs) {
10   return lhs.age < rhs.age;
11 });
12
13 std::for_each(vec.begin(), vec.end(), [](const Student& s) {
14   std::cout << s.name << ' ';
15 });
```

The output for the above code is:

```
Cathy Frank Drew Bob Alice Erin
```

The following code function takes in a vector `nums` and an integer `n` and returns a separate container with all the numbers in `nums` that are multiples of `n`. Because we passed a back-insert iterator for `result` (which pushes elements to the back of the container), we do not have to explicitly allocate space for the `result` vector on line 2.

```
1  std::vector<int32_t> multiples_of_n(std::vector<int32_t>& nums, int32_t n) {
2    std::vector<int32_t> result;
3    std::copy_if(nums.begin(), nums.end(), std::back_inserter(result), [n](int32_t i) {
4      return i % n == 0;
5    });
6    return result;
7  } // multiples_of_n()
8
9  int main() {
10   std::vector<int32_t> nums = {6489, 2374, 3822, 1238, 7684, 5999, 6548, 2947};
11   std::vector<int32_t> mult_7 = multiples_of_n(nums, 7);
12   std::for_each(mult_7.begin(), mult_7.end(), [](int32_t i) {
13     std::cout << i << " = 7 * " << (i / 7) << '\n';
14   });
15 } // main()
```

The output for the above code is:

```
6489 = 7 * 927
3822 = 7 * 546
5999 = 7 * 857
2947 = 7 * 421
```

Lastly, we can rewrite the following example using lambdas. This example was first introduced in section 11.9.7:

```
int32_t binary_op(int32_t a, int32_t b) {
  return a * a + b;
} // binary_op()

int main() {
  std::vector<int32_t> v1 = {5, 6, 7, 8, 9};
  std::vector<int32_t> v2 = {1, 2, 3, 4, 5};
  std::vector<int32_t> v3; // stores v1 * v1 + v2
  std::transform(v1.begin(), v1.end(), v2.begin(), std::back_inserter(v3), binary_op);
} // main()
```

Redefining the `binary_op()` function as a lambda within the `std::transform()` function gives us the following:

```
main() {
  std::vector<int32_t> v1 = {5, 6, 7, 8, 9};
  std::vector<int32_t> v2 = {1, 2, 3, 4, 5};
  std::vector<int32_t> v3; // stores v1 * v1 + v2
  std::transform(v1.begin(), v1.end(), v2.begin(), std::back_inserter(v3),
                 [](int a, int b) { return a * a + b; });
} // main()
```

This code does the exact same thing as before.

※ 11.10.3 *Generic Lambdas* (✻)

C++14 introduced **generic lambdas**, which can take in arguments of arbitrary types. Generic lambdas can be specified by using `auto` within the parameter list, as shown by the example below:

```
auto add_two_values = [](auto x, auto y) { return x + y; };
```

Behind the scenes, the compiler would essentially create an anonymous templated function object with the following behavior:

```
struct /* _anonymous_lambda_type_ */ {
  /* _anonymous_lambda_type_ constructor only callable by compiler */
  template <typename T, typename U>
  auto operator()(T& x, U& y) const { return x + y; }
};
```

Some examples using the `add_two_values` lambda are shown below:

```
auto add_two_values = [](auto x, auto y) { return x + y; };

int32_t v1 = 1;
int32_t v2 = 2;
double v3 = 1.2;
double v4 = 2.5;
std::string v5 = "eecs";
std::string v6 = "281";

std::cout << add_two_values(v1, v2) << '\n';    // prints 3
std::cout << add_two_values(v1, v4) << '\n';    // prints 3.5
std::cout << add_two_values(v3, v4) << '\n';    // prints 3.7
std::cout << add_two_values(v5, v6) << '\n';    // prints "eecs281"
```

Not all the parameters in a generic lambda need to be generic: it is perfectly fine to define a generic lambda with certain parameters defined with `auto` and other parameters defined with a fixed type (such as `int`, `double`, etc.).

11.11 The Functional Library (✻)

※ 11.11.1 *std::function* (✻)

In C++11, the `<functional>` library introduced the `std::function<>` class, which serves as a general-purpose polymorphic function wrapper that can be used to store any callable object, ranging from explicitly defined functions and function objects to anonymous lambdas. The declaration for a `std::function<>` is defined as follows:

```
template <class R, class... Args>
class function<R(Args...)>
```

Here `R` represents the type of the return value, while `Args` represents the types of the given parameters. In other words, the declaration syntax for a `std::function<>` is `std::function<return_type(parameter_types)>` for the given return and function parameter types. For example, consider the following function definition, which takes in a constant string reference and an integer and returns a Boolean:

```
bool func(const std::string& str, int32_t val);
```

The corresponding `std::function<>` type would therefore be `std::function<bool(const std::string&, int32_t)>`.

```
std::function<bool(const std::string&, int32_t)> my_func_object = func;
```

As long as a function object can be called with the specified return type and parameter types, it can be assigned to a `std::function<>` that is defined using those types. For instance, we can also assign a lambda that returns a Boolean and takes in a constant string reference and an integer into the `std::function<>` type we introduced above:

```
std::function<bool(const std::string&, int32_t)> my_func_object =
  [] (const std::string& str, int32_t val) { return str == "eecs" && val == 281; };
```

Technically, we can also assign this lambda to a `std::function<>` that accepts its first argument as `std::string` instead of `const std::string&`. The following assignment also compiles.

```
std::function<bool(std::string, int32_t)> my_func_object =
  [] (const std::string& str, int32_t val) { return str == "eecs" && val == 281; };
```

This is okay because the lambda can still be called with a `std::string` as its first parameter. If we instead tried to pass a function object that accepts a `std::string&` into a `std::function<>` that is declared with a `const std::string&` as its first parameter, then the assignment would no longer compile:

```
// DOES NOT COMPILE
std::function<bool(const std::string&, int32_t)> my_func_object =
  [] (std::string& str, int32_t val) { return str == "eecs" && val == 281; };
```

This is because the lambda expects a `std::string&`, but you cannot pass in a `const std::string&` into a `std::string&` parameter.

Because `std::function<>` can encapsulate any callable that matches its return and parameter types, it gives you flexibility to work with different callable objects even if they have different types. An example is shown in the code below (this prints out "`EECS 281 is fun!`"):

```
1  // regular function
2  void foo() {
3    std::cout << "EECS ";
4  } // foo()
5
6  // regular function that will be bound (we will cover this later)
7  void bar(int32_t best_class) {
8    std::cout << best_class << " ";
9  } // bar()
10
11 // functor
12 struct Baz {
13   /* can store internal state */
14   void operator()() {
15     std::cout << "is ";
16   } // operator()()
17 };
18
19 int main() {
20   // put all callable objects into a container of std::function<void()>
21   std::vector<std::function<void()>> callables;
22   callables.push_back(foo);
23   callables.push_back(std::bind(bar, 281));  // binds 'best_class' to a value of 281
24   callables.push_back(Baz());
25   callables.push_back([]() { std::cout << "fun!\n"; });  // pushes back a lambda
26
27   // iterate over all the std::function<void()> objects and invoke each of them
28   for (auto& func : callables) {
29     func();
30   } // for func
31 } // main()
```

`std::function<>` can also be used to store a callable object as internal state within a custom object. On line 9, an instance of `Foo` is constructed using a lambda is passed into its constructor. This lambda is stored internally in the class as the member variable `func`, and it is invoked via the function call to `Foo::invoke()` on line 10.

```
1  class Foo {
2    std::function<int32_t(int32_t, int32_t)> func;
3  public:
4    explicit Foo(const std::function<int32_t(int32_t, int32_t)>& func_in) : func{func_in} {}
5    int32_t invoke(int32_t x, int32_t y) { return func(x, y); }
6  };
7
8  int32_t subtract(int32_t x, int32_t y) {
9    return x - y;
10 } // subtract()
11
12 int main() {
13   Foo f1{[](int32_t x, int32_t y) { return x + y; }};
14   std::cout << f1.invoke(280, 1) << '\n';  // prints 281
15   Foo f2{subtract};
16   std::cout << f2.invoke(280, 1) << '\n';  // prints 279
17 } // main()
```

With all that being said, `std::function<>` should only be used in cases where you need to use it (such as to encapsulate multiple callable objects that may each have different types on their own, such as in the examples above). This is because the inner workings of a `std::function<>` is quite complicated, involving inheritance, virtual functions, and dynamic memory allocation. Thus, if there is a simpler way to represent a callable object, and you do not need the functionality of `std:function<>`, it is likely a more efficient choice.

※ 11.11.2 *Binding Function Arguments* (✻)

The `std::bind()` method in the `<functional>` library can be used to "bind" parameters of a function to certain argument values. This method creates a function object that stores the original callable object with its bound arguments and provides a function call operator (i.e., overloaded `operator()`) that takes in any remaining arguments (if there are any that have not been bound) and calls the original callable with these additional values, along with the values of the other parameters that have been previously bound.

```
std::bind(callable object, arguments to bind);
```

For instance, consider the following function, which prints out a given number.

```
void print(int32_t num) {
  std::cout << num << '\n';
} // print()
```

We can use `std::bind()` to construct another function object that behaves just like `print()` but with the value of `num` bound to 281:

```
void print(int32_t num) {
  std::cout << num << '\n';
} // print()

int main() {
  // func is a function object that runs print() with its first argument bound to a value of 281
  auto func = std::bind(print, 281);
  func();  // prints 281
} // main()
```

`std::bind()` makes a copy of the arguments that are passed in, so if you want the function object returned by `std::bind()` to track changes to a bound parameter, you would need to pass that parameter within a standard reference wrapper using `std::ref()`. By doing so, we allow the function object returned by `std::bind()` to internally store a reference to the value that is bound.

```
void print(int32_t num) {
  std::cout << num << '\n';
} // print()

int main() {
  int32_t val = 280;
  auto func1 = std::bind(print, val);
  auto func2 = std::bind(print, std::ref(val));  // reference wrapper
  func1();  // prints 280
  func2();  // prints 280

  ++val;    // val becomes 281
  func1();  // still prints 280, since func1 stored a copy of val
  func2();  // prints 281, since func2 took in val as a reference wrapper
} // main()
```

You can also bind certain arguments in a function while also allowing other arguments to be provided by the user. This can be done using special placeholders defined in `std::placeholders`. These placeholders have the special symbols `_1`, `_2`, `_3`, and so on. The number of the placeholder identifies which argument of a bound function should be filled with each value in a function call. An example is shown below:

```
void print(const std::string& department, int32_t num) {
  std::cout << department << " " << num << '\n';
} // print()

int main() {
  std::string department = "EECS";
  // bind print()'s department argument to the value "EECS"
  auto func = std::bind(print, department, std::placeholders::_1);
  // first argument to function call (281) is passed in place of std::placeholders::_1
  func(281);  // prints "EECS 281"
} // main()
```

These placeholders also allow you to do other things like reorder the parameters of a function. The following creates a function object that behaves like the original `print()`, but takes in the first parameter as if it were the second, and the second parameter as if it were the first.

```
void print(const std::string& department, int32_t num) {
  std::cout << department << " " << num << '\n';
} // print()

int main() {
  // swap the order of the arguments, so func takes in the num first, then the department string
  auto func = std::bind(print, std::placeholders::_2, std::placeholders::_1);
  func(281, "EECS");  // prints "EECS 281"
} // main()
```

Because `std::bind()` returns a function object, it can also be passed into standard library algorithms that accept them. The following prints out all the classes in the vector, with the first parameter of the `print()` function bound to a value of `"EECS"`:

```
1  void print(const std::string& department, int32_t num) {
2    std::cout << department << " " << num << '\n';
3  } // print()
4
5  int main() {
6    std::vector<int32_t> classes = {183, 203, 280, 281, 370, 376};
7    std::for_each(classes.begin(), classes.end(), std::bind(print, "EECS", std::placeholders::_1));
8  } // main()
```

The output of this code is:

```
EECS 183
EECS 203
EECS 280
EECS 281
EECS 370
EECS 376
```

The `std::bind()` method is a useful tool for partial function application: you can use it to fix arguments to a function to have certain values. However, in C++14 and beyond, there are essentially no use cases where `std::bind()` should be the method of choice — this is thanks to the power of lambdas. Instead of using `std::bind()` to create a function object with fixed arguments and/or placeholders, we can use lambdas to do the same thing in a cleaner, more expressive manner. While `std::bind()` was able to do things that lambdas could not in versions before C++14, these deficiencies have been addressed in later C++ versions.

Consider our `std::bind()` example that fixed the first argument of the `print()` method to have a value of `"EECS"`. We can use a lambda to accomplish the same thing:

```
1  void print(const std::string& department, int32_t num) {
2    std::cout << department << " " << num << '\n';
3  } // print()
4
5  int main() {
6    std::string department = "EECS";
7    auto func = [&department](int32_t num) { print(department, num); };
8    func(281);  // prints "EECS 281"
9  } // main()
```

The same goes with the example where we swapped the order of parameters using placeholders:

```
1  void print(const std::string& department, int32_t num) {
2    std::cout << department << " " << num << '\n';
3  } // print()
4
5  int main() {
6    auto func = [](int32_t num, const std::string& department) { print(department, num); };
7    func(281, "EECS");  // prints "EECS 281"
8  } // main()
```

So, why do we need to talk about `std::bind()` at all? Even though its functionality can fully emulated with lambdas in newer versions of C++, it is still something that you will likely see if you have to work with a large, older codebase. However, if you ever find yourself in the position where you need to bind arguments in a function, you should consider using a lambda as your go-to method instead of `std::bind()`.

※ 11.11.3 *Callback Functions* (✻)

A **callback function** is a function that is passed into another function as an argument, with the intention that it will be executed either immediately or after the other function has completed a task. Callback functions are typically used in event-driven systems, where program execution and behavior is determined by events (such as a user clicking a button, data being received from a service, etc.). In C++11 and beyond, callbacks can be implemented and passed around using `std::function<>` and lambdas.

To demonstrate, we will consider a high-level example that uses a callback function. Suppose you are implementing a component that processes data updates that are streamed from a service. When you get a message back from the service, you want to immediately execute a function that processes and publishes this data. To do this, you can pass the function as a callback to the client that is subscribed to the service (in this case, the `DataRetrievalClient`), and then execute the function as soon as an update is received. This is shown in the example code below:

```
1  // this class is in charge of processing the data; it stores a DataRetrievalClient
2  // that directly subscribes to the service that the data is streamed from
3  class DataProcessor {
4  private:
5    DataRetrievalClient* client;
6    Logger* logger;
7    /* ... other member variables ... */
8  public:
9    // this method should be executed whenever the client receives a data update
10   void process_and_publish_update(const DataUpdate& update) {
11     // does work to process the data update message
12   } // process_and_publish_update()
13
14   // this method makes a subscription request to the DataRetrievalClient
15   SubscriptionToken subscribe(const DataRequest& request) {
16     auto token = client->subscribe_for_updates(request, [this](const DataUpdate& update) {
17       logger->log_message("Received data update to process and publish.", update);
18       this->process_and_publish_update(update);
19     });
20     return token;
21   } // subscribe()
22 };
23
24 // this class directly subscribes to the service and gets its data
25 class DataRetrievalClient {
26 private:
27   TcpCommunicator* comm;  // sends and receives data via the network
28   /* ... other member variables ... */
29 public:
30   SubscriptionToken subscribe_for_updates(const DataRequest& request,
31                                           const std::function<void(const DataUpdate&)>& callback) {
32     /* ... do work ... */
33     // subscribes to another service and gets data that needs to be processed
34     DataUpdate update = get_and_parse_update(...);
35     // invoke callback, this executes the process_data() method from the DataProcessor
36     callback(update);
37     /* ... do work ... */
38   } // subscribe_for_updates()
39 };
```

In this code, we have two classes, a `DataProcessor` and a `DataRetrievalClient`. The `DataRetrievalClient` is responsible for making a network connection to a service that streams back data, and the `DataProcessor` is responsible for processing the data whenever an update is received from the server. To ensure that the `process_and_publish_update()` method on line 10 is executed immediately whenever a message is received back from the server, it is passed in a `std::function<>` callback to the `DataRetrievalClient`. Then, whenever the `DataRetrievalClient` successfully gets a data update (line 34), it immediately invokes the callback on line 36 — this would execute the `DataProcessor::process_and_publish_update()` method on the `DataUpdate` response that was received.

11.12 Random Number Generators (⁎)

※ 11.12.1 *rand* (⁎)

The C++ standard library provides an assortment of tools that can be used to generate random values. Randomization is a fantastic approach for generating test cases to test your code (in fact, many of the autograder test cases for projects were generated using these resources).

You may have heard about the `rand()` operation, which can be used to generate a random integer. If you want to limit random numbers to a specific range (e.g., if you only wanted numbers between 0 and 99), you can use the modulo operation to mod the random number with the size of the range. If you want a non-zero initial value to the range, simply add it to the result of the modulus. Examples are shown below:

```
1  int32_t r1 = rand() % 100;       // r1 stores random number between 0-99
2  int32_t r2 = rand() % 100 + 1    // r2 stores random number between 1-100
3  int32_t r3 = rand() % 50 + 281;  // r3 stores random number between 281-330
```

However, if you were to run `rand()` over and over again, you will always get the same sequence of numbers:

```
1804289383, 846930886, 1681692777, 1714636915, 1957747793, 424238335, ...
```

It turns out that, as long as `rand()` is fed with the same seed, the sequence of numbers produced will always be the same. The **seed** is a number used to initialize a random number generator. If you do not explicitly set the seed of the random number generator, the seed is default set to 1.

Behind the scenes, `rand()` takes a starting number (the seed) and performs mathematical operations on it to create another number that appears unrelated to the starting number. For the next number in the sequence, it takes the previously generated number and performs the same mathematical operations on it to get a new number. This process continues until the seed is reset.

The seed can be set using `srand()`. For instance, `srand(21)` would set the seed to `21`, and all subsequent calls to `rand()` would produce the sequence of numbers associated with a starting seed value of 21. An example is shown below:

```
// no seed, so it is default set to 1
for (int32_t i = 0; i < 10; ++i) {
  std::cout << rand() << " ";       // prints sequence 1804289383 846930886 1714636915 ...
} // for i

// set seed of rand to 281
srand(281);
for (int32_t j = 0; j < 10; ++j) {
  std::cout << rand() << " ";       // prints new sequence: 1462582710 1177062221 1109108190 ...
} // for j

// reset seed back to 1
srand(1);
for (int32_t k = 0; k < 10; ++k) {
  std::cout << rand() << " ";       // prints original sequence 1804289383 846930886 1714636915 ...
} // for k
```

If you want a quick and simple random number generator, using `rand()` and modulo is sufficient. However, `rand()` does not produce the best results in terms of statistical randomness, and using modulo to set a range for random output actually biases the output toward certain values. This is because remainders are not random! Some remainders will appear more than others; for instance, if you restrict a `rand()` random number generator to the range 0-1999, the number 500 may be more likely to appear than the number 1500 due to the nature of modular arithmetic. For better random-number generation, the C++ `<random>` library is preferred over `rand()`.

※ 11.12.2 *The Random Library* (✻)

To use the C++ random number library, you will need to `#include <random>` at the top of your code. Random number generation using the `<random>` library relies on two types of objects:

- **generators:** function objects that generate uniformly distributed numbers
- **distributions:** objects that transform sequences of numbers from a generator into sequences that follow a specific random variable distribution (e.g., Uniform, Normal, Binomial, Poisson, etc.)

If you use a generator, it should be paired with a distribution. The C++ standard library provides several different generators that you can use. In this section, we will focus on the `std::mt19937` generator as our primary random number generator. The `std::mt19937` generator uses the Mersenne Twister algorithm to generate random numbers, and it is by far one of the most widely used general-purpose pseudorandom number generators.[7] This generator is much better than `rand()` at producing values that are statistically random. To construct a `std::mt19937` generator, you can use the following constructor:

```
std::mt19937 gen(seed);
```

where `gen` is the name of the generator (you can give it any variable name you want), and `seed` is the seed value that the generator is initialized to. For the best randomness, the value of the seed should be determined at runtime. One example is to use the current time to initialize the seed of the generator, as shown below. This method ensures that the seed is different every time the program is run. For example, when the program was run on July 5, 2019 at 11:47 AM EST (when this section was first written), the value of seed was `1562341651852838259`, which produced the random number sequence `3563885013`, `1545485723`, `4272454046`, `3127047453`, …. When the program was run ten minutes later, the value of seed became `1562342251099698619`, which produced an entirely different sequence of random numbers.

```
#include <iostream>
#include <chrono>    // used to retrieve system clock
#include <random>    // std::mt19937 generator

int main() {
  // the seed depends on the system clock
  auto seed = std::chrono::high_resolution_clock::now().time_since_epoch().count();
  // construct generator using seed
  std::mt19937 gen(seed);
  // generate and print out a random number using generator
  std::cout << gen() << '\n';
} // main()
```

[7] A pseudorandom number generator (PRNG) generates random numbers using arithmetic methods, and they often require a starting seed. This is different from a true random number generator (TRNG), which uses unpredictable physical means to generate random numbers (e.g., atmospheric noise, physical state of hardware device, etc.).

Another way to seed the `std::mt19937` generator is to feed it a `std::random_device`, which is also provided in the `<random>` library. A `std::random_device` can be used to generate "true" random numbers based on stochastic processes that may involve physical hardware devices on your machine. Because generating random numbers using a `std::random_device` can be expensive, it is often only used to seed the generator. An example is shown below:

```
int main() {
  // create a random_device to generate random number for seed
  std::random_device rd;
  // sample the random_device and use result to seed the mt19937 generator
  std::mt19937 gen(rd());
  // print out a random number using generator
  std::cout << gen() << '\n';
} // main()
```

As shown by the examples above, the generator can be used to generate a giant range of numbers. How can we limit the random numbers generated into a specific range? For instance, suppose we only wanted random numbers between 0 and 1,000. We cannot use modulus if we want a good distribution, since remainders may not be evenly distributed. Instead, we can use a *distribution*. The C++ standard library provides around 20 different distributions that can be used to specify the frequency of random numbers within a given range.

For random number generation, two important distributions in the standard library are the `std::uniform_int_distribution<>` and the `std::uniform_real_distribution<>` distributions. Both distributions take in two numbers, `a` and `b`, that set the lower and upper bounds of the output range. When a distribution is applied to a generator, it ensures that the generator produces a random value in the range `[a, b]`, and that all possible values within the range have an equal likelihood of being produced. This is known as a **uniform distribution**. The `std::uniform_int_distribution<>` should be used for integer types (such as `int`, `long`, `unsigned`), while `std::uniform_real_distribution<>` should be used for floating points (such as `double` and `float`).

The following code creates a `std::mt19937` generator and a uniform distribution. The distribution is then applied to the generator to specify the range and frequency of values that are generated. To apply a distribution to a generator, simply pass in the generator object into the distribution using `operator()`.

```
int main() {
  // create a mt19937 generator
  std::random_device rd;
  std::mt19937 gen(rd());
  // limit the generator so that it only generates numbers between
  // 1 and 100, inclusive; this can be done by creating a distribution
  std::uniform_int_distribution<int32_t> dist(1, 100);
  // apply the distribution to the generator every time a random
  // number is generated (10 random numbers are generated below)
  for (int32_t i = 0; i < 10; ++i) {
    std::cout << dist(gen) << " ";
  } // for i
} // main()
```

Running the above code five times in a row yields the following output (this will be different on different machines because the `std::mt19937` generator was seeded with a `std::random_device`):

```
Run #1: 60 93 40 96 91 22 35 90 60 21
Run #2: 8 45 57 51 48 8 81 26 3 36
Run #3: 96 16 22 67 100 63 57 25 19 55
Run #4: 65 80 40 70 83 93 86 8 85 16
Run #5: 87 31 60 28 46 78 68 32 62 39
```

The following code can be used to generate random *floating point* values between 1 and 100:

```
int main() {
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_real_distribution<double> dist(1, 100);
  for (int32_t i = 0; i < 10; ++i) {
    std::cout << dist(gen) << " ";
  } // for i
} // main()
```

Running this code yields the following results (your results may differ since this is random):

```
Run #1: 14.6965 39.4479 81.9015 37.1515 27.7026 82.0444 31.7411 47.3614 7.16891 98.6985
Run #2: 33.4811 49.7041 93.0337 40.8993 54.661 8.63654 79.6556 11.052 69.1065 75.305
Run #3: 86.6864 12.0326 72.6361 93.8472 53.819 78.9021 64.1423 88.8143 7.22277 63.2723
Run #4: 68.8516 78.4234 56.8878 86.7274 64.0702 89.8117 9.73159 68.9019 4.01285 52.5718
Run #5: 2.55291 65.394 2.9479 88.4693 1.73457 65.3447 42.9447 80.437 76.5555 21.75
```

Random number generators can be used in more ways than just to return a random number. For instance, generators can be used to shuffle the contents of a container using `std::shuffle()` in the `<algorithm>` library:

```
template <typename RandomAccessIterator, typename URNG>
void std::shuffle(RandomAccessIterator first, RandomAccessIterator last, URNG&& g);
```

Rearranges the elements in the range `[first, last)` randomly, using `g` as a uniform random number generator. This function swaps each element with another random element picked using `g()`.

An example is shown below:

```
int main() {
  std::vector<int32_t> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
  // randomly shuffle contents of v
  std::random_device rd;
  std::mt19937 gen(rd());
  std::shuffle(v.begin(), v.end(), gen);
  std::cout << "Shuffled elements:";
  for (int32_t i : v) {
    std::cout << " " << i;
  } // for i
} // main()
```

One possible output of this program is (each run produces a different shuffled order):

```
Shuffled elements: 3 7 10 9 12 2 14 8 13 1 15 5 11 4 6
```

When generating test cases for projects, you may need to generate objects of non-numeric types. For instance, you may want to randomly generate words for projects that require strings as input. If you have a dictionary file that contains the words you want to select from, you can simply add them all to a vector and use a generator to randomly select the indices that the words are chosen from.

```
std::ifstream fin;
fin.open("dictionary.txt");
std::vector<std::string> words;  // vector of strings that store the words
std::string word;
while (fin >> word) {
  words.push_back(word);
} // while
fin.close();

// create a generator for random number generation
std::random_device rd;
std::mt19937 gen(rd());

// create a distribution to select a word at random from the vector
std::uniform_int_distribution<int32_t> random_word_dist(0, words.size() - 1);

// apply the distribution to the generator to select a random word
std::string random_word = words[random_word_dist(gen)];
std::cout << random_word << '\n';
// random_word_dist(gen) generates a random index between 0 and words.size() - 1
```

The above program can be modified to fit the specifications of any project that requires random word generation. Note that you can use input redirection and `std::cin` instead of `std::ifstream` to read in the contents of a file.

The uniform distribution is not the only distribution you can use to generate random numbers. You can also generate random numbers to fit other distributions, such as Binomial, Geometric, Poisson, Exponential, and Normal. However, you likely will not be using these distributions in this class (unless you want to have fun simulating exam scores for EECS 281). The contents of the `<random>` library are a lot more comprehensive than what was covered here, with many more generators and distributions available. However, the concepts detailed in this section should be more than enough for this class.

Remark: The `rand()` random number generator and distributions in the `<random>` library are not guaranteed to be portable. This means that the random numbers generated may not be consistent across platforms and compilers. Someone who runs `rand()` or a uniform distribution on CAEN may not obtain the same numbers as someone who runs the code locally on Visual Studio, even if the seed and everything else is the same! Thus, you will need to be careful before sending out files that rely on randomization, as the person you are sending the code to may not get the same results!

11.13 Additional Features in C++17 (*)

Even though the language has existed for decades, C++ is still being actively maintained, with continuous updates being made every few years. The arrival of C++11 in 2011 made drastic improvements to the existing C++03, with features that have become core components of the language (such as `auto` type deduction, `nullptr`, move semantics, lambdas, and so much more). Three years later, C++14 made some minor improvements to C++11, such as function return type deduction and generic lambdas.

In this section, we will go over a few important features that were introduced in C++17. While these features may not be required material, they are still valuable tools that are useful to know, especially as you gain experience developing in C++. At the least, these features can help make your code more expressive and readable, if not more efficient.

※ 11.13.1 *Structured Bindings* (*)

Structured bindings are a new feature introduced in C++17 that allows you to easily bind names to the subobjects of another object. The anatomy of a structured binding is as follows:

```
<type> [<name>, <name>, ...] = <tuple-like object>
```

where the `<type>` is typically `auto` (either by value, reference, or const reference), `<name>` is a variable name to be assigned to the unpacked values, and `<tuple-like object>` is an object like a tuple, struct, or array that can be unpacked.

We introduced an example previously involving pairs: before C++17, to declare variables that represent the first and second values of a pair, we would have to define them explicitly using the member variables `.first` and `.second`.

```
1  std::pair<std::string, int> p = {"EECS", 281};        // p stores {"EECS", 281}
2  auto str = p.first;
3  auto num = p.second;
4  std::cout << "The value of str is: " << str << '\n';  // "The value of str is: EECS"
5  std::cout << "The value of num is: " << num << '\n';  // "The value of num is: 281"
```

With a structured binding, we can accomplish the same thing using the syntax on line 2:

```
1  std::pair<std::string, int> p = {"EECS", 281};        // p stores {"EECS", 281}
2  auto [str, num] = p;                                  // structured binding
3  std::cout << "The value of str is: " << str << '\n';  // "The value of str is: EECS"
4  std::cout << "The value of num is: " << num << '\n';  // "The value of num is: 281"
```

The identifiers in a structured binding can also be `const` qualified and defined as a reference:

```
1  std::pair<std::string, int> p = {"EECS", 281};           // p stores {"EECS", 281}
2  const auto& [str, num] = p;                              // structured binding (const ref)
3  std::cout << "The value of str is: " << str << '\n';     // "The value of str is: EECS"
4  std::cout << "The value of num is: " << num << '\n';     // "The value of num is: 281"
```

This process is similar for a tuple, which is essentially a pair that is generalized to multiple elements:

```
1  std::tuple<std::string, int, char> t = {"EECS", 281, 'A'};
2  auto [str, num, grade] = t;
3  std::cout << "The value of str is: " << str << '\n';     // "The value of str is: EECS"
4  std::cout << "The value of num is: " << num << '\n';     // "The value of num is: 281"
5  std::cout << "The value of grade is: " << grade << '\n'; // "The value of grade is: A"
```

Structured bindings can also be used to unpack values in an array or compound object. The following example uses a structured binding to bind names to the contents of an array (note that we cannot do this with a vector, since the structure of a vector is not known at compile time):

```
1  int32_t arr[3] = {203, 280, 281};  // also works with std::array<int32_t, 3> arr = {203, 280, 281};
2  auto [x, y, z] = arr;
3  std::cout << x << '\n';            // 203
4  std::cout << y << '\n';            // 280
5  std::cout << z << '\n';            // 281
```

The following uses a structured binding to identify values in a compound object:

```
1  struct ClassName {
2    std::string department = "EECS";
3    int32_t number = 281;
4  };
5
6  int main() {
7    ClassName cn;
8    auto [a, b] = cn;
9    std::cout << a << '\n';  // "EECS"
10   std::cout << b << '\n';  // 281
11 } // main()
```

Structured bindings provide a elegant mechanism for emulating a function with multiple return values. Prior to the introduction of structured bindings, there were three primary ways to have a function "return" multiple values:

1. Pass in pointers or references to the return values as parameters.

```
void get_student_data(std::string& name, int32_t& id, double& gpa) {
  name = get_student_name();
  id = get_student_id();
  gpa = get_student_gpa();
} // get_student_data()
```

2. Return a `struct` that houses all of the values.

```
struct StudentData {
  std::string name;
  int32_t id;
  double gpa;
};

StudentData get_student_data() {
  StudentData sd;
  sd.name = get_student_name();
  sd.id = get_student_id();
  sd.gpa = get_student_gpa();
  return sd;
} // get_student_data()
```

3. Return a `std::tuple<>` that houses all of the values.

```
std::tuple<std::string, int32_t, double> get_student_data() {
  return std::make_tuple(get_student_name(), get_student_id(), get_student_gpa());
} // get_student_data()
```

With structured bindings, we have a cleaner way to unpack all the values for the latter two methods. An example is shown below.

```
std::tuple<std::string, int32_t, double> get_student_data() {
  return std::make_tuple(get_student_name(), get_student_id(), get_student_gpa());
} // get_student_data()

int main() {
  auto [name, id, gpa] = get_student_data();
  // ... do work ...
} // main()
```

Although we will not discuss it here, structured bindings can be quite useful when working with map containers in the STL, since these containers support insertion methods that return both an iterator to the element with a given key and a Boolean indicating whether the insertion was successful. We will go over these containers in more detail when we get to hash tables and trees in chapters 17 and 18.

※ 11.13.2 *std::optional* (✻)

Prior to C++17, one notable omission of the language was that there was no standardized way to represent the absence of a meaningful value, despite the fact that there are situations where being able to return "no solution" or represent something as having "no value" is desired behavior. Consider the following function definition, which is intended to return the value of the first element in the vector that is a multiple of `n`:

```
int32_t find_first_multiple(const std::vector<int32_t>& vec, int32_t n);
```

What would you return if there are no elements that are a multiple of `n`? Prior to C++17, a common workaround is to return a dummy value, such as `-1`, that can be used to identify the absence of a value. However, it is clear there are pitfalls to this approach. What if `-1` is a valid return value? How can we differentiate between a dummy return value of `-1` and an actual solution of `-1`? We could avoid this issue by returning a pointer to the integer we want and return `nullptr` if there is no solution, or by passing in the desired integer as a parameter reference and having the function return a Boolean, but these approaches are messy and not as expressive.

This was fixed with the introduction of `std::optional<>`, which allows you to represent an object that may or may not have a value. We can rewrite our initial example to use the `std::optional<>` construct, as shown:

```
std::optional<int32_t> find_first_multiple(const std::vector<int32_t>& vec, int32_t n);
```

Here, our function can return the first multiple if it exists, or `std::nullopt` if it does not (this value indicates an optional has no value). A list of supported operations for an `std::optional<>` are summarized below. To use an optional, you should `#include <optional>`.

`template <typename T>`
`std::optional<T>();`
Constructs a `std::optional<>` that does not contain a value.

`template <typename T>`
`std::optional<T>(const std::optional<T>& other);`
Constructs a `std::optional<>` that does not contain a value if `other` has no value, otherwise initializes the `std::optional<>` with the value of `other`.

`template <typename T>`
`std::optional<T>(T&& value);`
Constructs a `std::optional<>` with the value of `value`.

```
template <typename T>
std::optional<T>::make_optional(T&& value);
```
Constructs a `std::optional<>` with the value of `value`.

```
template <typename T, typename... Args>
std::optional<T>::make_optional(Args&&... args);
```
Constructs a `std::optional<>` that is constructed in-place using the arguments of `T`'s constructor.

```
std::optional<int32_t> opt1;                 // has no value
std::optional<int32_t> opt2{42};             // has a value of 42
std::optional<int32_t> opt3 = 281;           // has a value of 281
std::optional<int32_t> opt4 = std::nullopt;  // has no value

// has a value of 370
std::optional<int32_t> opt5 = std::make_optional<int32_t>(370);

// has a value of {9, 9, 9, 9, 9}
std::optional<std::vector<int32_t>> opt6 = std::make_optional<std::vector<int32_t>>(5, 9);

// has a value of {"EECS", 281}
std::optional<std::pair<std::string, int32_t>> opt7 =
  std::make_optional<std::pair<std::string, int32_t>>("EECS", 281);
```

The following methods can be used to access the value of an `std::optional<>`:

```
template <typename T>
bool std::optional<T>::has_value();
```
Returns whether the `std::optional<>` has a value.

```
template <typename T>
T& std::optional<T>::value();
```
Returns the value of the `std::optional<>` if it exists, otherwise throws a `std::bad_optional_access` exception.

```
template <typename T, typename U>
T std::optional<T>::value_or(U&& default_value) const&;
```
Returns the value of the `std::optional<>` if it exists, otherwise returns `default_value`.

In addition to `.value()`, you can also access an `std::optional<>`'s value using `operator*` and `operator->`. The only difference is that these methods do *not* check if a value exists beforehand, which causes undefined behavior if used on an `std::optional<>` with no value. Before you use `operator*` or `operator->` on an `std::optional<>`, make sure to verify it has a value first using `.has_value()`.

```
std::optional<int32_t> opt1;
std::cout << opt1.has_value() << '\n';       // prints 0 for false
std::cout << opt1.value_or(281) << '\n';     // prints 281

std::optional<int32_t> opt2{280};
std::cout << opt2.has_value() << '\n';       // prints 1 for true
std::cout << opt2.value_or(281) << '\n';     // prints 280
if (opt2.has_value()) {                      // always check .has_value() before using * or ->
  std::cout << *opt2 << '\n';                // prints 280
} // if

std::optional<std::vector<int32_t>> opt3 = std::make_optional<std::vector<int32_t>>(5, 9);
if (opt3.has_value()) {
  std::cout << opt3->size() << '\n';         // prints vector size, or 5
} // if
```

Remark: A call to `operator*()` does not check if an `std::optional<>` contains a value, so it may be more efficient than calling `std::optional<T>::value()` if `std::optional<T>::has_value()` has already been checked before.

※ 11.13.3 *std::variant* (✻)

A `std::variant<>`, defined in the header `<variant>`, makes it possible to store multiple types of data in a single variable. As an example, the following is a variant that can either store an `int32_t` or `std::string`:

```
std::variant<int32_t, std::string> var;
var = 281;                                   // variant holds the integer 281
var = "EECS";                                // variant holds the string "EECS"
```

A variant makes it easier to defer knowledge of a data's type to when you actually need to use that data. For instance, suppose you wanted to parse a file or command line arguments, but you are not sure what types are you are going to get. With a variant, you can simply list out all the possible types using a single variable and then assign your data to that variable.

One way to get the value of a variant is to use `std::get<>` with the correct type as its template, as shown below:

```
std::variant<int32_t, std::string> var;
var = 281;                                      // variant holds the integer 281
std::cout << std::get<int32_t>(var) << '\n';    // prints 281
var = "EECS";                                   // variant holds the string "EECS"
std::cout << std::get<std::string>(var) << '\n';  // prints "EECS"
```

However, if you attempt use `std::get<>` with the wrong type, you will get a `std::bad_variant_access` exception:

```
std::variant<int32_t, std::string> var;
var = "EECS";                                   // variant holds the string "EECS"
std::cout << std::get<int32_t>(var) << '\n';    // exception thrown, var is not an integer
```

To check the type of a variant before getting its data, you can use the `std::get_if<>` method. This method takes in the memory address of a `std::variant<>` and returns either a pointer to its data if the given type is correct, or `nullptr` if the given type is not correct.

```
std::variant<int32_t, std::string> var;
var = "EECS";
auto* str_value = std::get_if<std::string>(&var);  // pointer to string value "EECS"
auto* int_value = std::get_if<int32_t>(&var);      // nullptr since var is not an integer
```

Because `std::get_if<>` returns a `nullptr` if the given type is not correct, we can use it handle variant logic within `if` statements (since `if (nullptr)` evaluates to `false`). This is shown below: the first `if` statement runs if the variant's data is a string, and the second `if` runs if the variant's data is an integer:

```
std::variant<int32_t, std::string> var;
var = get_variant_data();
// runs if var's data is a std::string
if (auto* value_ptr = std::get_if<std::string>(&var)) {
  std::string& value = *value_ptr;
  std::cout << "var is a string with value " << value << '\n';
} // if
// runs if var's data is an int32_t
if (auto* value_ptr = std::get_if<int32_t>(&var)) {
  int32_t value = *value_ptr;
  std::cout << "var is an integer with value " << value << '\n';
} // if
```

One particularly notable restriction of variants is that they are not allowed to initialize additional dynamic memory beyond the data they are designed to store. This feature can be quite useful, since dynamic memory allocation is typically more expensive than memory allocation in automatic storage. Note that the objects in a variant can allocate dynamic memory on their own (e.g., a variant that holds a `std::vector<>`); it's just that the `std::variant<>` itself cannot allocate any additional dynamic memory to store its data.

※ 11.13.4 *std::visit and the Visitor Pattern* (✻)

Another useful feature provided by C++17 is the `std::visit()` method, which can be used to apply a callable method (known as a *visitor function*) depending on the type that is held in a variant. When `std::visit()` is called on a variant, it dispatches the appropriate visitor function based on the type that is currently held in that variant. The visitor object passed into `std::visit()` must be callable with all possible types that are held in the variant. An example is shown below:

```
struct Visitor {
  void operator()(bool value) {
    std::cout << "Variant is a bool!\n";
  } // operator()(bool)
  void operator()(const std::string& value) {
    std::cout << "Variant is a string!\n";
  } // operator()(string)
};

int main() {
  std::variant<bool, std::string> var1;
  var1 = std::string{"EECS 281"};
  std::visit(Visitor{}, var1);  // prints "Variant is a string!"

  std::variant<bool, std::string> var2;
  var2 = false;
  std::visit(Visitor{}, var2);  // prints "Variant is a bool!"
} // main()
```

The `std::visit()` method makes it cleaner to implement objects using the *visitor pattern*, which is a design strategy of separating an algorithm from the object it operates on. This pattern allows us to add new virtual methods to a collection of objects without having to modify all the objects themselves. As an example, consider the following collection of shape objects that inherit from a base `Shape` class:

```
struct Shape {
};

struct Circle : public Shape {
  double radius;
  Circle(double radius_in) : radius{radius_in} {}
};

struct Rectangle : public Shape {
  double length;
  double width;
  Rectangle(double length_in, double width_in) : length{length_in}, width{width_in} {}
};

struct Triangle : public Shape {
  double base;
  double height;
  Triangle(double base_in, double height_in) : base{base_in}, height{height_in} {}
};
```

Suppose we wanted to add a new method `calculate_area()` that can be used to calculate the area of any `Shape`. How do we go about doing this? One strategy is to use plain old inheritance and polymorphism, as shown.

```
struct Shape {
  virtual double calculate_area() const = 0;
};

struct Circle : public Shape {
  double radius;
  Circle(double radius_in) : radius{radius_in} {}
  double calculate_area() const override {
    return std::numbers::pi * radius * radius;
  } // print_area()
};

struct Rectangle : public Shape {
  double length;
  double width;
  Rectangle(double length_in, double width_in) : length{length_in}, width{width_in} {}
  double calculate_area() const override {
    return length * width;
  } // print_area()
};

struct Triangle : public Shape {
  double base;
  double height;
  Triangle(double base_in, double height_in) : base{base_in}, height{height_in} {}
  double calculate_area() const override {
    return 0.5 * base * height;
  } // print_area()
};

int main() {
  Circle c = Circle{5};
  Rectangle r = Rectangle{3, 7};
  Triangle t = Triangle{4, 6};
  std::vector<Shape*> shapes = { &c, &r, &t };
  std::vector<double> areas;
  for (const auto& shape : shapes) {
    double area = shape->calculate_area();
    areas.push_back(area);
  } // for shape
} // main()
```

Another option is to implement a separate class that implements the behavior of this new method. This allows each of the shape objects to be simpler and more flexible, as they would only need to define their own characteristics (and not any actions that may be performed on them). This can be easily done using variants and `std::visit()`, as shown. The following code exhibits the same behavior as the code above:

```
struct Shape {
};

struct Circle : public Shape {
  double radius;
  Circle(double radius_in) : radius{radius_in} {}
};

struct Rectangle : public Shape {
  double length;
  double width;
  Rectangle(double length_in, double width_in) : length{length_in}, width{width_in} {}
};

struct Triangle : public Shape {
  double base;
  double height;
  Triangle(double base_in, double height_in) : base{base_in}, height{height_in} {}
};

struct ShapesAreaCalculator {
  double operator()(const Circle& c) {
    return std::numbers::pi * c.radius * c.radius;
  } // operator()(Circle)
  double operator()(const Rectangle& r) {
    return r.length * r.width;
  } // operator()(Rectangle)
  double operator()(const Triangle& t) {
    return 0.5 * t.base * t.height;
  } // operator()(Triangle)
};
/* ...continued from previous page... */

int main() {
  Circle c = Circle{5};
  Rectangle r = Rectangle{3, 7};
  Triangle t = Triangle{4, 6};
  std::vector<std::variant<Circle, Rectangle, Triangle>> shapes = { c, r, t };
  std::vector<double> areas;
  for (const auto& shape : shapes) {
    double area = std::visit(ShapesAreaCalculator{}, shape);
    areas.push_back(area);
  } // for shape
} // main()
```

Remark: The `std::monostate` type can be used to represent an empty state in a `std::variant<>`. If you want a variant where having no value is a valid outcome, you can specify `std::monostate` as one of its types. An example of this usage is shown below:

```
std::variant<std::monostate, int32_t, std::string> response;
response = "EECS";            // variant stores string
response = 281;               // variant stores integer
response = std::monostate{};  // variant stores "empty" state
```

※ 11.13.5 *std::any* (✻)

C++17 also introduced `std::any` in the `<any>` header, which can be used to store *any* copy-constructible type:

```
std::any a = 281;
std::any b = nullptr;
std::any c = "potato";
std::any d = std::vector<int32_t>{1, 2, 3};
```

That being said, if you want the ability to store multiple data types in the same variable, `std::variant<>` is almost always preferable to `std::any`. To get the data stored in an `std::any`, you would still need to know its underlying type, and the absence of type restrictions makes it easier to run into bugs at runtime (for example, `"potato"` in the above example is a `const char*` and not a `std::string`, so trying to convert to a `std::string` using `std::any_cast<>` would cause an exception to be thrown). Furthermore, `std::any` does not have the same dynamic memory restrictions that a `std::variant<>` has, so it may be less efficient for storing larger objects.

A `std::any` still has its use cases, however, and it was introduced to provide a safer means for storing data of any type. Prior to C++17, `void*` was often used as an alternative, since a `void` pointer was allowed to hold the address of any data type. An object of type `std::any` is significantly safer than a `void*`, as `std::any` requires you to know the type of its underlying data before you can access it, while `void*` does not provide this safety guarantee.