Computing IV Book

James Daly

December 19, 2024

Contents

Part I

C++ Language

Chapter 1

Namespaces

1.1 Motivation

1.2 Creating Namespaces

1.3 Qualifying

1.4 `using` Directives

Chapter 2

Inheritance

Chapter 3

Friends

Chapter 4

Operator Overloading

Chapter 5

Templates

Part II

Data Structures

Chapter 6

Iterators

Chapter 7

String

Chapter 8

`vector` (Array List)

Chapter 9

`list` (Linked List)

Chapter 10

`stack`

Chapter 11

queue

Chapter 12

deque

Chapter 13

`set` (Tree Set)

Chapter 14

`map` (Tree Map)

Chapter 15

`unordered_set` (Hash Set)

Chapter 16

`unordered_map` (Hash Map)

Chapter 17

`bitset`

Part III

Utilities

Chapter 18

Unit Testing

18.1 Motivation

When you write a program, it can be difficult to tell if it does what you want it to. There can be any number of errors in the program, some of which may be hard to identify. Furthermore, once you fix one problem, you have to ensure that you didn't inadvertently introduce a new bug in the process. Running your program and testing everything by hand is a very time-consuming process. It is also easy to miss something, especially when you are rechecking a section for the umpteenth time.

Automated unit testing has us writing test programs that will run portions of our main program to ensure that they behave as expected. These test programs are made up of a suite of small "unit tests" that each run a small portion of the overall code and verifies that it performs as expected. If the developer accidentally makes a change to the behavior of the system, these tests will identify the change in behavior. By running these tests frequently, these mistakes can be caught quickly and fixed before too many other changes have also been made. This makes debugging much easier.

There are a number of different unit testing frameworks, both for C++ and for other languages. The Boost Unit Test Framework is one such framework.

18.2 Simple Test

Below is a simple test for the `std::string` class.

```
1 #define BOOST_TEST_DYN_LINK
2 #define BOOST_TEST_MODULE StringTest
3 #include <string>
4 #include <boost/test/unit_test.hpp>
5
6 BOOST_AUTO_TEST_CASE(TestSize) {
7     std::string str = "hello, world!";
8     BOOST_REQUIRE(str.size() == 13);
```

```
9 }
```

Here, the `<string>` library is the component that we are testing and `<boost/test/unit_test.cpp` is the header for the test library. The `BOOST_TEST_DYN_LINK` macro is an option that tells the library that it should create a `main` function that will run your tests. Without it, you would have to write your own `main` function and call the tests yourself. The `BOOST_TEST_MODULE` defines a name for this suite of tests (`StringTest` in this case). Both of these macros must be defined *before* including the test library.

`BOOST_AUTO_TEST_CASE` is a macro that will create a test function (here named `TestSize`), including a lot of code to add it to the test suite that you don't have to worry about. You can have as many `BOOST_AUTO_TEST_CASE`s as you want; usually you will have a lot of them that each test specific things. Note that since they create functions, you need to obey all of the usual C++ naming rules including about having multiple functions with identical names.

Inside the test, we use the `BOOST_REQUIRE` macro to create an assertion that the string has the correct size. If the condition inside the macro is `false`, then the test ends in failure. If it is `true`, then the test continues. If the test ends normally, then it passes.

There is also a `BOOST_CHECK` macro. Like `BOOST_REQUIRE`, `BOOST_CHECK` will cause the test to fail if the condition is `false`. Unlike `BOOST_REQUIRE`, the test continues even on a failure. This means that a single test could have multiple failures. You use `BOOST_CHECK` when testing multiple independent assertions such as functions without side-effects and `BOOST_REQUIRE` when an earlier failure implies that later assertions will also fail (such as when modifying an object).

18.3 Comparisons

The assertion in Chapter 18.2 has a limitation; if the assertion fails, we don't know *why* it failed (we don't know what the size is). Instead, we can use the `BOOST_REQUIRE_EQUAL`. If this fails, it will report what the two values are using the `<<` operator. This gives us more information about what went wrong.

There are a number of different assertion macros that we can use for different types of comparisons. Each comparison has both a `REQUIRE` and a `CHECK` version, depending on whether we want to stop or continue on a failure. A list of these macros can be seen in Table 18.1

Since floating-point numbers (like `float` and `double`) can have a small amount of error, there also exists a `BOOST_REQUIRE_CLOSE` macro that can determine if two floating-point numbers are within a specified threshold of each other.

Here is a revised test case for the example in Section 18.2.

```
1 BOOST_AUTO_TEST_CASE(TestSize) {
2     std::string str = "hello, world!";
3     BOOST_REQUIRE_EQUAL(str.size(), 13);
4 }
```

Operator	Require Macro	Check Macro
==	BOOST_REQUIRE_EQUAL	BOOST_CHECK_EQUAL
!=	BOOST_REQUIRE_NE	BOOST_CHECK_NE
<	BOOST_REQUIRE_LT	BOOST_CHECK_LT
<=	BOOST_REQUIRE_LE	BOOST_CHECK_LE
>	BOOST_REQUIRE_GT	BOOST_CHECK_GT
>=	BOOST_REQUIRE_GE	BOOST_CHECK_GE

Table 18.1: Comparison Macros

18.4 Exceptions

It is important to test both the success cases and failure cases. For example, we should check that the program behaves properly when given values outside of the allowable range. Often times, the correct behavior is to throw an exception. We can use the `BOOST_REQUIRE_THROW` or `BOOST_CHECK_THROW` macros to verify that a piece of code throws a particular kind of exception. The second argument is the type of exception that is expected, so if it throws the wrong kind of exception it will fail the test. Note that if a subclass of the exception is thrown it will still pass the test. Conversely, if you want to check that it *doesn't* throw an exception, you can use `BOOST_REQUIRE_NO_THROW` or `BOOST_CHECK_NO_THROW`.

```
1 BOOST_AUTO_TEST_CASE(TestAt) {
2     std::vector<int> v = {1};
3     BOOST_REQUIRE(v.empty());
4     BOOST_REQUIRE_NO_THROW(v.at(0));
5     BOOST_REQUIRE_THROW(v.at(1), std::out_of_range);
6 }
```

18.5 Test-Driven Development

When we write a new test and it passes it raises a question; did it pass because the code it is testing is good, or because the test is not good enough to detect problems? For example, we could write a trivial test that does nothing and always passes. This test would be useless for detecting problems.

The key to answering the question is to write the test first, *before* the code which it tests. Since the feature being tested doesn't exist yet, the test will fail. Once we implement the feature, the test should pass. In this way, the test becomes a representation of what it means to have implemented the feature.

When we write a new feature, we follow these steps.

1. Write some tests that define the expected behavior for the the new feature. These tests may not compile if they call new functions.

2. Write a stub for these new functions.

3. Run the tests. The new tests should fail since the feature hasn't been implemented yet.
4. Write the new feature.
5. Rerun the tests. If the feature was implemented properly, all of the tests will pass. Fix any problems until this is the case.
6. Refactor the code to make it more readable. Rerun the tests to make sure you didn't accidentally break anything.

Chapter 19

Smart Pointers

19.1 Problem with Ownership

Whenever you allocate memory with `new`, it must be deallocated at some later point in time with `delete`. It is important that this happens exactly once; if you forget to `delete` it when you are done you end up with a memory leak and if you call `delete` on it multiple times you get a double free error. Even if you call `delete` on the pointer exactly once you can still end up with problems; if you copied the pointer then you have a wild pointer. Using this pointer will cause undefined behavior. What this means is we need some way to tell who is responsible for deleting the pointer and letting others know not to use any copies they may have.

This may seem like it should be a simple process. Each pointer has an *owner* who is responsible for deleting the pointer and ownership can be transferred. However, it isn't always clear if calling a function transfers ownership or not.

```
1 vector<Item*> items;
```

Consider a `vector` of pointers to `Item`s. It is possible that the `vector` claims ownership of any pointers and that whoever manages the `vector` is responsible for deleting them. It is also possible that the `vector` is a view into a larger structure or a composition of several smaller collections. For example, consider a multiplayer game where each player has a `vector` of their own pieces and a separate combined `vector` also exists for things like physics or drawing. In this case, it may not be clear whether the players or the model are responsible for deleting the units.

19.2 Reference counting

It is possible to share ownership of an object. This requires several owners to coordinate when they claim and relinquish ownership of an object. The last

one to relinquish ownership of the object then destroys it. This is often done through *reference counting*.

When an object is created through `new`, a *reference counter* is also created and initialized to 1. Whenever a new pointer to the object is created the counter is incremented. Whenever a pointer is cleared, the counter is decremented. If the counter ever reaches 0, the object must be also be deleted.

19.3 `shared_ptr`

The `shared_ptr` is a smart pointer that uses the reference counting strategy mentioned in Section 19.2. When the first `shared_ptr` is created, it sets a reference count to 1; each other `shared_ptr` created with either the copy constructor or copy assignment operator increments the counter and the destructor decrements the counter, destroying the owned object if the count reaches 0. Note that you should not create a second `shared_ptr` from the original raw pointer. Doing so will create a second reference count and will cause problems. New smart pointers should only be created from existing smart pointers.

There is a factory function `make_shared` that both constructs an object and creates a `shared_ptr` to it. This factory takes the parameters that will be passed to the new object's constructor, which it will call with `new`. It then constructs and returns a `shared_ptr` with that object. The factory has some advantages over the constructor. First, it slightly simplifies things by combining the memory allocation and `shared_ptr` creation into a single step. Second, the raw pointer never exists outside of the factory and so there is no chance of accidentally creating a second reference count or a well-meaning fool from accidentally deleting the raw pointer. Third, the process is completed in a single step, so there is no chance of the object being allocated and then afterwards an exception being thrown before the `shared_ptr` is created (which would leak the object).

`shared_ptr` has both a `*` and `->` operator so the syntax for using it is like a normal raw pointer. It also has a `bool` cast operator which is `true` if the pointer is non-empty and `false` if it is empty like a raw pointer so you can use it as part of conditionals. The `reset` method clears the pointer (decrementing the reference count) and makes it a `nullptr`, whereas the `swap` method swaps the pointers owned by two `shared_ptr` s (preserving their reference counts). You can use the `get` method if you need the raw pointer (such as when interfacing with a function that takes in raw pointers. Note that the `==`, `!=`, and `<<` operators work with the *pointer* and not the owned object.

In the following example, we create a `shared_ptr` to a `string`. When we copy the pointer, we can make changes to the `string` which are reflected with the original pointer since they refer to the same object.

```
1 std::shared_ptr<string> p1 = std::make_shared<string>("hello");
2 std::shared_ptr<string> p2 = p1;
3 *p2 += "world";
4 cout << *p1 << endl; // prints "helloworld"
```

19.4 weak_ptr

Smart pointers help prevent memory leaks, but certain problems can arise. If an object owns a copy of its own smart pointer, either directly or indirectly through something it owns, then that object will never be deleted since its reference count will never hit 0. Consider, for example, a doubly linked list where each node holds `shared_ptr`s to both the next and previous nodes. When the container is destroyed, it discards its pointers to the node chain. However, if there are two or more nodes in the list, then the next and previous nodes keep each other alive through their own `shared_ptr`s. If the nodes only had `shared_ptr`s to their next nodes, then the destruction of the container would cause the head node to be destroyed, which would then cause each subsequent node to be destroyed in the chain.

We then replace the previous nodes with `weak_ptr`s. A `weak_ptr` is a kind of smart pointer that follows, but doesn't own, the pointer it references (creating a `weak_ptr` doesn't affect the reference count). This means that if all of the `shared_ptr`s to an object are cleared, the object is `delete`d, even if there are still `weak_ptr`s to the object.

`weak_ptr`s can be copy constructed from either `shared_ptr`s or other `weak_ptr`s, but cannot be instantiated from a raw pointer (since the reference count would be 0). Also unlike `shared_ptr`s, `weak_ptr`s *don't* have the usual pointer operators `*` and `->` and cannot directly access the stored pointer. Instead, you must first use the `lock` method, which creates a `shared_ptr` that you can use. This `shared_ptr` shares the same reference counter as the `weak_ptr`'s progenitor and will keep the object alive until it is discarded. Note that if the reference counter reaches 0 *before* you `lock` the `weak_ptr`, the object will have already been `delete`dand you will receive a `null` `shared_ptr` instead. You can use the `expired` method to check if the `weak_ptr` is still valid, but in a multi-threaded environment there is the chance that the object could be destroyed between the calls to `expired` and `lock`.

19.5 unique_ptr

Sometimes there should be only a single access point to an object. A `unique_ptr` is a smart pointer for sole ownership over the object. Unlike a `shared_ptr`, a `unique_ptr` does not use reference counting. Instead, the `unique_ptr` promises to be the only owner of the object.

Similar to the `shared_ptr`, there is a `make_unique` factory that will construct a new object and the owning `unique_ptr` simultaneously with the same benefits as mentioned in Chapter 19.3 for `make_shared`. Unlike `shared_ptr`, `unique_ptr` does not have a copy constructor or copy assignment operator. Since `unique_ptr` s are *unique* they cannot be cloned. Instead, you can use the move constructor or move assignment operator to *transfer* ownership of the pointer to a different `unique_ptr`. When you use this, the object owned by the recipient `unique_ptr` is destroyed and then it claims ownership of the source

object owned by the source `unique_ptr`, which is then set to `nullptr`. You can also move to a `shared_ptr` using move constructor or move operator overloads. Note that this is a one-way street; you cannot move from a `shared_ptr` back to a `unique_ptr` since additional copies of the `shared_ptr` could have already been made. You can also `swap` two `unique_ptr` s (but not between a `unique_ptr` and a `shared_ptr`).

`unique_ptr` has the same access and equality operators as a `shared_ptr` as well as the `reset` method. Additionally, it has a `release` method that relinquishes control over the object and returns its pointer. This is different from `get` in that after calling `release` the object is no longer managed by the `unique_ptr` and must be manually `delete`d.

Chapter 20

Function Pointers

20.1 Motivation

There are times where we want to have variations on a piece of code where we inject a single behavioral change. For example, consider the following code that insertion sorts a vector of strings.

```
1  void insertionSort(std::vector<std::string>& v) {
2      for (size_t i = 1; i < v.size(); i++) {
3          std::string s = v[i];
4          size_t j;
5          for (j = i; j > 0 && v[j - 1] < s; j--) {
6              v[j] = v[j - 1];
7          }
8          v[j - 1] = s;
9      }
10 }
```

What if we wanted to compare the strings in a case-insensitive manner? We would write a very similar function.

```
1  bool ltIgnoreCase(const std::string& s1, const std::string& s2) {
2      for (size_t i = 0; i < s1.size() && i < s2.size(); i++) {
3          if (std::tolower(s1[i]) != std::tolower(s2[i])) {
4              return std::tolower(s1[i]) < std::tolower(s2[i]);
5          }
6      }
7      return s1.size() < s2.size();
8  }
9  void insertionSortIgnoreCase(std::vector<std::string>& v) {
10     for (size_t i = 1; i < v.size(); i++) {
11         std::string s = v[i];
12         size_t j;
13         for (j = i; j > 0 && ltIgnoreCase(v[j - 1], s); j--) {
14             v[j] = v[j - 1];
15         }
16         v[j - 1] = s;
17     }
18 }
```

In this example, the `insertionSortIgnoreCase` function is the same as the `insertionSort` function except that it uses the `ltIgnoreCase` function in place of the `<` operator. We can imagine a number of variants to this that use a variety of alternate comparisons here (e.g. using a greater than comparison to reverse sort), especially if we templated the function so that it could work for any kind of `vector`.

20.2 Function Pointers

The code for a function exists in memory, which means that it has an address. A pointer stores an address, which means that a pointer variable can store the address of a function. For a comparison function like the `ltIgnoreCase` function above, we could write something like:

```
1 bool (*comp)(const std::string&, const std::string&) = &ltIgnoreCase;
```

We read this as `comp` is a pointer to a function that takes two `string`s and returns a `bool`, and is bound to `ltIgnoreCase`. Note that there are no parenthesis on `ltIgnoreCase`; we are getting the address of the function itself, not for whatever invoking the function would return. Also, the `&` is optional, so this would also have been acceptable:

```
1 bool (*comp)(const std::string&, const std::string&) = &ltIgnoreCase;
```

At this point, we can treat `comp` just like a function and invoke it as though it were the function it points to:

```
1 bool result = comp("hello", "world"); //  evaluates to true
```

Putting it all together (with some templates to make it more useful), we get the following function:

```
1  template <typename T>
2  void insertionSort(std::vector<T>& v, bool (*comp)(const T&, const T&)) {
3      for (size_t i = 1; i < v.size(); i++) {
4          T s = v[i];
5          size_t j;
6          for (j = i; j > 0 && comp(v[j - 1], s); j--) {
7              v[j] = v[j - 1];
8          }
9          v[j - 1] = s;
10     }
11 }
```

20.3 `Function`

The `<functional>` library defines a lot of tools for working with functions. One of these tools is a class `function` that wraps a function pointer or function-like

Arithmetic	Relational	Logical
`plus<T>`	`equal_to<T>`	`logical_and<T>`
`minus<T>`	`not_equal_to<T>`	`logical_or<T>`
`multiplies<T>`	`greater<T>`	`logical_not<T>`
`divides<T>`	`greater_equal<T>`	
`modulus<T>`	`less<T>`	
`negate<T>`	`less_equal<T>`	

Table 20.1: Function objects representing operators

object (one that defines an `operator()`). This allows us to create variables that can represent not just functions, but also to things that act like functions. Additionally, the notation is a little easier to understand; the previous notation may not immediately be obvious (especially to newer users) that it represents a function.

```
1 std::function<bool(const std::string&, const std::string&)> comp = &ltIgnoreCase
```

The `<functional>` library also defines a number of objects that represent function calls, such as `plus` (representing the `+` operator) and `equal_to` (representing the `==`) operator). A list of these function objects can be seen in Table 20.1.

Putting this all together, we create the final version of our `insertionSort` function that uses the `less` function object to sort items according to their natural ordering if no comparison function is provided.

```
1  template <typename T>
2  void insertionSort(
3          std::vector<T>& v,
4          std::function<bool(const T&, const T&)> comp = std::less<T>()) {
5      for (size_t i = 1; i < v.size(); i++) {
6          T s = v[i];
7          size_t j;
8          for (j = i; j > 0 && comp(v[j - 1], s); j--) {
9              v[j] = v[j - 1];
10         }
11         v[j - 1] = s;
12     }
13 }
```

20.4 Lambda Expression

The version of our `insertionSort` function makes sorting in non-standard orders significantly more convenient. As long as we have an appropriate comparison function, we can order the items according to that function. This does mean that for each different ordering we might want to use will require a separate one of these helper functions. Many of these functions will be used in only one place

and our namespace may become littered with a bunch of these little functions that are used in only one place.

Since C++11, we can alleviate this by using *lambda expressions*. Lambda expressions are statements that create *closures*, which bind some local variables into an anonymous function. Generally, a lambda expression in C++ will take the form of:

```
1 [capture] (params) -> type { body }
```

Here, `params` is the normal parameter list of the function, `type` is the return type of the function, and `body` is the usual function body. Note that the return type comes *after* the parameter list rather than before like with regular functions, and can usually be intuited and is optional in most cases. Likewise, the `auto` keyword is often used for the parameter variable types when their types can be intuited from the context.

The capture list is new. This is a comma separated list or any local variables that are needed in the closure. The closure has access to any variables that are captured by the lambda expression. These variables can either by captured by value using `=`, in which case their value is copied into the closure, or captured by reference using `&`, in which case references to the variable are given to the closure. Similar to regular functions, any changes made to a variable captured by value will not be reflected in the surrounding context, by changes to ones captured by reference will be. If you need to capture multiple variables, either you can list each one or you can place either just an `=` or an `&` to capture all variables.

The capture list is required, even if no variables are captured. This will mean that the closure will not have access to any variables from the surrounding environment. For example, here is a call to our `insertionSort` function with a lambda expression to sort in reverse order.

```
1 insertionSort(words, [](auto a, auto b) { return a > b; });
```

In this case, we only need to compare the two parameters `a` and `b`, so we don't need any variables from the local scope. Likewise, the types of the variables can be intuited from context and so the `auto` keyword can be used. Note that in this case, we could have used the `std::greater` function object to achieve the same effect.

Chapter 21

<algorithm>

21.1 Motivation

21.2 accumulate

The std::accumulate represents a fold where each element in the sequence is repeatedly "'folded"' into a result using a provided combining function and initial result. Unlike the other functions included in this chapter, accumulate is located in the <numeric> library. The default version uses addition as the combining function to calculate the sum of the elements. For example, the following code segment computes the sum of a vector.

```
1 vector<int> v = {1, 2, 3, 4, 5};
2 int sum = accumulate(v.begin(), v.end(), 0); // 15
```

There is also an overload that takes the combining function as an argument. The following code segment uses the multiplies class from the <function> library to compute the product of the elements. Note that 1 is used for the initial result since that is the multiplicative identity.

```
1 vector<int> v = {1, 2, 3, 4, 5};
2 int product = accumulate(v.begin(), v.end(), 1, multiplies<int>()); // 120
```

Since the accumulate function is generic, you can use it with any type that has a + operator. This code segment will concatenate several strings together into a single string.

```
1 vector<string> v = {"hello", "world"}
2 string concat = accumulate(v.begin(), v.end(), string()); // "helloworld"
```

21.3 find

find, findif, and search

21.4 any_of

anyof, allof, and noneof

21.5 count

count and countif

21.6 min and max

21.7 equal

21.8 copy

copy, copyif

21.9 transform

21.10 move

21.11 fill and generate

21.12 replace

21.13 sort

sort, stablesort, partialsort binarysearch, includes, lowerbound, upperbound

Chapter 22

Exceptions

Chapter 23

<chrono>

Time is an important part of our lives. We care about when things happen and how long they take. The `<chrono>` library gives us tools for monitoring and measuring time.

Chapter 24

<random>

Many things in life happen seemingly at random. When we roll dice, any of the six faces (for standard cube dice) could come up with equal probability (for ideal "fair" dice). For a well-shuffled deck of cards, we could draw any of the cards, but that card is not available for future draws.

Yet, computers are by and large not random. If you put in the same input, you always get the same output. These systems are *deterministic*. While stochastic measures that measure real phenomena to generate random numbers exist, computer usually simulate randomness by using a special *seed* value and transform it using some sort of algorithm to generate a series of pseudo-random numbers. These numbers are not actually random, but they have a number of statistical properties that are close enough for most purposes.

There are several algorithms for generating pseudo-random numbers, each with their own pros and cons. The C++ `<random>` library includes three: a linear congruential generator (LCG), a Mersenne twister, and subtract-with-carry (also known as subtract-with-borrow). Additionally, there is a `random_device` that uses stochastic processes to generate true random numbers.

24.1 Linear Congruential Generator (LCG)

An LCG is a pseudo-random number generator defined by a recurrence relation

$$x_{i+1} = (a \cdot x_i + c)\%m$$

where x_i is the previous random number (with x_0 being the seed), x_{i+1} is the next random number, and a, c, and m are constants which are particular to the chosen generator. Not all choices for a, c, and m give useful generators. In particular, m is tied to the period (how long the sequence goes before repeating) and so should be relatively large.

The `minstd_rand` and `minstd_rand0` classes are both LCG implementations. Overall, the `minstd_rand` class has better randomness with identical

overhead and so should be used in preference to `minstd_rand0`. The chosen constants for `minstd_rand` are $a = 48271$ (a prime number), $c = 0$, and $m = 2^{31} - 1$ (a Mersenne prime).

Overall, LCGs are very fast and memory efficient, requiring only a single numeric field for the seed (as the constants are often hardcoded) and a single multiplication, addition, and division operation. This makes them good for situations where lots of random numbers must be generated quickly, when multiple independent sequences of random numbers are required, or when memory is tight. The randomness ranges from passable (for good choices of constants) to awful (for bad choices) and so they are not suitable for cases where high-quality randomness is required (such as for cryptography). Finally, the period length is relatively short (being at most m) and so is not suitable for long sequences.

24.2 Mersenne Twister

A Mersenne twister is a pseudo-random number generators that maintains an array of n w-bit numbers and uses a matrix recurrence relation to "twist" the numbers. Note that the matrix is chosen such that the result can be easily computed with a series of bit shifts and bit masks rather than doing a full matrix multiplication.

The `mt19934` and `mt19934_64` classes for generating 32-bit (`uint32`) and 64-bit (`uint64`) numbers, respectfully.

Mersenne twisters have incredibly long periods ($2^{19934} - 1$ for the most common version), making them especially good when a long sequence of numbers is required. Unfortunately, they have a relatively long warm-up period where similar seed numbers will give correlated sequences before they diverge. This makes them unsuitable for repeated experiments such as Monte Carlo simulations. They also require a large amount of memory ($\sim 2.5\,\mathrm{KiB}$) making them unsuitable for multiple parallel streams of random numbers, and are relatively slow making them unsuitable for situations where numbers must be generated quickly.

24.3 `random_device`

The LCG and Mersenne twister are both pseudo-random number generators; they use a formula to generate new numbers from previous numbers rather than using a random process. In contrast, the `random_device` class uses a stochastic process (like measuring the least-significant bit of the system clock on a key press or measuring stellar radiation with specialized hardware) to generate random numbers in a manner not unlike flipping coins. Note that the process used is system dependent and may not be available on all systems.

There are pros and cons to this. Since PRNGs are reproducible, they are useful for simulations or video games where the results need to be repeatable (such as for saving replays or synchronizing between players). Certain encryp-

tion techniques also essentially boil down to synchronizing PRNGs between the sender and receiver and using the generated numbers to modify the cypher. On the flip side, many other security situations *don't* want this property because an adversary may be able to pick a seed that reduces the amount of protection afforded. Note that there are separate cryptographically secure pseudo-random number generators that should be used for security rather than either the LCG or Mersenne twister.

Hardware RNGs require making some sort of stochastic measurement, a process which is comparatively slow. Generally, the operating system will measure entropy over time from some noise source and store the results in a file (`/dev/random` on Linux). When randomness is required, bits are consumed from this file which will be slowly refilled over time. This means that if a large number of random numbers are requested in a short period of time the request may block until further measurements can be taken.

24.4 Seeding the Generator

PRNGs like the LCG and Mersenne twister are initialized with a *seed* value. Two copies of the same PRNG seeded with the same value will generate the same sequence. This is useful for being able to rerun a simulation (allowing the exact same experiment to be rerun) or for sharing procedurally-generated content across multiple players (since each player can generate the same content from the same seed).

In most situations, you want this seed to be picked essentially at random. One common strategy is to use the system clock since the least significant bits of a single timestamp are essentially random. Care must be taken to only do this once though; subsequent calls will yield numbers that are highly correlated with the first (and may be identical if your system clock resolution is poor enough).

```
1 unsigned int seed = std::chrono::system_clock::now().time_since_epoch().count();
2 std::minstd_rand gen(seed);
```

Alternately, you can use a `random_device` to generate a single random number which is used to seed your PRNG. While `random_device` is slow and can block if excessive numbers are requested, generating a single number should be fine.

```
1 std::random_device rd;
2 std::mt19937 gen(rd());
```

24.5 Distributions

The `()` operator for each of the random number generator classes creates a single integer between the `min()` and `max()` defined for that specific generator. Generally speaking, this will not be the desired range. The `<random>`

library defines several distribution classes which can be used to generate numbers in different ranges and with particular distributions. The simplest of these are `uniform_int_distribution` and `uniform_read_distribution` which generate numbers uniformly at random in the specified range. Both are templated so that they can generate different kinds of numbers and take two constructor parameters a and b for the lower and upper bounds of their ranges. Note that the `uniform_int_distribution` produces the closed range $[a, b]$ and `uniform_real_distribution` produces the semi-open range $[a, b)$.

Other distribution objects will produce different distributions. For example, the `normal_distribution` will produce a normal distribution with a mean μ and a standard deviation of σ.

```
unsigned int seed = std::chrono::system_clock::now().time_since_epoch().count();
std::minstd_rand gen(seed);
std::uniform_int_distribution<int> d6(1, 6); // Standard die
int roll = d6(gen) + d6(gen); // Sum of two dice
```

24.6 Dice Example

Below we present a `Dice` class that can be used for simulating dice for a bunch of games. There are two constructors. The default constructor initializes the random-number generator using the system clock so that each instantiation will generate a different sequence of numbers. The second constructor takes a seed that is used to initialize the random-number generator. This allows the same sequence to be replayed.

This class uses the `minstd_rand` generator, but the `mt19934` generator could have been used instead. The generator is a member variable so that it generates a single sequence of numbers rather than restarting the sequence after each roll.

The `roll()` method takes parameters for the number and size of the dice, but have default values for the most common option of a single six-sided die. It creates a `uniform_int_distribution` which it uses to generate numbers in the desired range before discarding it after the sum has been tallied.

```
#include <random>

class Dice {
 public:
    Dice();
    explicit Dice(unsigned int seed);

    int roll(int num = 1, int size = 6);
 private:
    std::minstd_rand gen;
};
```

```
#include "dice.hpp"

#include <chrono>

```

```
5  Dice::Dice():
6      Dice(std::chrono::system_clock::now().time_since_epoch().count()) {}
7  Dice::Dice(unsigned int seed): gen(seed) {}
8
9  int Dice::roll(int num = 1, int size = 6) {
10     int sum = 0;
11     std::uniform_int_distribution<int> dist(1, size);
12     for (int i = 0; i < num; i++) {
13         sum += dist(gen);
14     }
15     return sum;
16 }
```

Chapter 25

<regex>

Chapter 26

Time

Chapter 27

Threads

Part IV

Design Patterns

Chapter 28

Factory Method

Chapter 29

Observer

Chapter 30

Adapter

Chapter 31

Decorator

Chapter 32

Model-View-Controller

Chapter 33

Singleton

Part V

Miscellaneous

Chapter 34

Makefiles

34.1 Motivation

Consider compiling a small project with a `Complex` number class in `Complex.hpp` and `Complex.cpp` files and a `main` function in a `main.cpp` file. You might compile it like this:

```
1 g++ -Wall -Werror -pedantic -g -c Complex.cpp
2 g++ -Wall -Werror -pedantic -g -c main.cpp
3 g++ -Wall -Werror -pedantic -g -o Program main.o Complex.o
```

Each time you edit any of the files you need to rerun some or all of the above commands. Remembering to run the correct ones in the correct order is important; if you don't your compiled `Program` won't reflect the changes you made. With all of the options, each of the commands are kind of lengthy and annoying to type and it is easy to make an error; a problem that will only get worse if you add other options or need to link to a library. Additionally, as you add more files, you need to run more commands. You might be tempted to run something like

```
1 g++ -Wall -Werror -pedantic -g -o Program *.cpp
```

but this doesn't work if you have multiple main files or otherwise want to exclude certain files.

Clearly, this isn't sustainable for larger projects and so we need some way to automate the process of running these commands. We will create special `Makefile`s that can be interpreted by the program `make` to execute the commands to compile our program.

34.2 Rules

We'll create a simple `Makefile` that will compile the same program as in Section 34.1.

```
Program: main.o Complex.o
    g++ -Wall -Werror -pedantic -g -o Program main.o Complex.o

main.o: main.cpp Complex.hpp
    g++ -Wall -Werror -pedantic -g -c main.cpp

Complex.o: Complex.cpp Complex.hpp
    g++ -Wall -Werror -pedantic -g -c Complex.cpp
```

This `Makefile` contains three *rules*. We see that each rule contains three parts: a target, some dependencies, and a recipe. The *target* appears on the left side of the colon and gives the name of the program that successfully running this rule will create. The *prerequisites* or *components* appear on the right side of the color and gives the names of other files that must already exist for the target to be built. In the case of `Program`, these are other compiled files which `make` will build using the other rules in the `Makefile`. Once all of the dependencies are built, `make` will follow the *recipe*, which is a series of *commands* listed under the rule. The recipe must be indented with a hard tab; spaces are not allowed. Each command in the recipe is executed sequentially and stops if any command has an error.

When you run `make [target]`, `make` will use the corresponding rule to build the file, defaulting to using the first rule if no target is specified. This is why `Program` is listed first. Running `make` will cause the following commands to be run:

```
g++ -Wall -Werror -pedantic -g -c main.cpp
g++ -Wall -Werror -pedantic -g -c Complex.cpp
g++ -Wall -Werror -pedantic -g -o Program main.o Complex.o
```

`make` will attempt to build `Program`, but since `Program` depends on both `main.o` and `Complex.o`, those must be built first. Those files both depend on the code files that we wrote and so it then executes their corresponding commands to compile them. Afterward, it then links them together into a single `Program`.

Running `make` again has different results:

```
make: Nothing to be done for 'all'.
```

Make is smart enough to realize that it doesn't need to recompile files if none of its dependencies have changed, so it doesn't do anything. It does this by comparing the timestamps of the file to those of its dependencies and if it is newer than all of the dependencies it determines the file is up to date and doesn't have to be rebuilt. This can occasionally cause problems, like if the timestamps are messed up because some of the files were updated from a remote device and the timestamps are out of sync. You can add the `-B` flag to force `make` to rebuild.

Changing only some of your files, such as the `Complex.cpp` file, will cause only some of the files to be rebuilt. In this case, running `make` again would produce these results:

```
g++ -Wall -Werror -pedantic -g -c Complex.cpp
```

```
2 g++ -Wall -Werror -pedantic -g -o Program main.o Complex.o
```

34.3 Variables

You may have noticed that a lot of stuff is repeated throughout our different rules. We can reduce a lot of repetition by using variables. For example, all of the flags to g++ are repeated in every command. If we want to change some of the flags, perhaps to use a different version of C++ than the default, then we would have to add this to every command. Instead, it is common to have a variable called `FLAGS` that contains these. Then we only have to change it in one place. Note that there is nothing special about this name and you will sometimes see similar names like `CFLAGS` instead. Also, you will use `$` to evaluate an expression, but not when you want to set a variable.

This updates our Makefile to

```
1  FLAGS = -Wall -Werror -pedantic -g
2
3  Program: main.o Complex.o
4      g++ $(FLAGS) -o Program main.o Complex.o
5
6  main.o: main.cpp Complex.hpp
7      g++ $(FLAGS) -c main.cpp
8
9  Complex.o: Complex.cpp Complex.hpp
10     g++ $(FLAGS) -c Complex.cpp
```

We can do a few other improvements to improve maintainability. We'll add a `CC` variable in case we want to switch out the compiler for another one and a `LIBS` variable for any libraries we might use in the future (although we currently aren't using any). Finally, we'll also add a `DEPS` variable that holds common components where any change is going to require recompiling most of our code, such as for many of our header files.

After this, our updated Makefile is

```
1  CC = g++
2  FLAGS = -Wall -Werror -pedantic -g
3  LIBS =
4  DEPS = Complex.hpp
5
6  Program: main.o Complex.o
7      $(CC) $(FLAGS) -o Program main.o Complex.o
8
9  main.o: main.cpp $(DEPS)
10     $(CC) $(FLAGS) -c main.cpp
11
12 Complex.o: Complex.cpp $(DEPS)
13     $(CC) $(FLAGS) -c Complex.cpp
```

34.4 Automatic Variables

In Section 34.3, we managed to remove repetition between rules by using variables, but each rule still has some repetition. In the first rule, the component files are repeated as part of the command. We can replace them with the automatic variable `$^` which represents all of the components.

For the other two rules, only one of the components is duplicated. We don't want to try to compile the header files, only the single source file. We can instead use the automatic variable `$<` to insert only the first component.

Finally, the program name also appears in the command for the first rule. We can use the the automatic variable `$@` to insert the target name.

All combined, we now have an updated `Makefile`:

```
1  CC = g++
2  FLAGS = -Wall -Werror -pedantic -g
3  LIBS =
4  DEPS = Complex.hpp
5
6  Program: main.o Complex.o
7      $(CC) $(FLAGS) -o $@ $^
8
9  main.o: main.cpp $(DEPS)
10     $(CC) $(FLAGS) -c $<
11
12 Complex.o: Complex.cpp $(DEPS)
13     $(CC) $(FLAGS) -c $<
```

At this point, we notice that the commands for `main.o` and `Complex.o` are identical. We can use the wildcard symbol `%` to represent the name of the file. If we make a rule with target `%.o` then that rule will be used when building *any* `.o` file. We can then use it in the prequisites to match the corresponding `.cpp` file. This allows us to make a single rule that replaces the individual rules for each `.o` file. This makes adding new files significantly easier.

Our final `Makefile` is now

```
1  CC = g++
2  FLAGS = -Wall -Werror -pedantic -g
3  LIBS =
4  DEPS = Complex.hpp
5
6  Program: main.o Complex.o
7      $(CC) $(FLAGS) -o $@ $^
8
9  %.o: %.cpp $(DEPS)
10     $(CC) $(FLAGS) -c $<
```

34.5 Standard Targets

There are several targets that appear in the `Makefile`s of most projects. These names do not have an particular meaning to `make`, but are a convention expected by others.

By convention, `all` is used to build everything. This is normally the first rule in the `Makefile` so that it will be the default rule.

```
1 all: Program
```

If we added other helper programs, we can add them to `all` to have all of them be built.

`install` will compile the program and copy all of the necessary files to where they should reside for use. This means that any libraries created would be usable by other programs. This will likely require creating new folders. A separate `uninstall` target will remove the installed files.

`clean` removes generated files created by this `Makefile`, but not configuration information. `distclean` is stronger and also removes configuration information and `mostlyclean` is weaker and refrains from removing files that people don't want to recompile.

None of these standard targets correspond to actual files. This can cause problems if files by that name were ever created; for example if there is a file named `clean` then `make clean` would not actually remove anything because `make` would think that it is up to date. To combat this, we create a special target called `.PHONY`. Any prerequisite of `.PHONY` is always rebuilt regardless of whether a file by that name exists.

We end with our final `Makefile`

```
1  CC = g++
2  FLAGS = -Wall -Werror -pedantic -g
3  LIBS =
4  DEPS = Complex.hpp
5
6  .PHONY: all clean mostlyclean
7
8  all: Program
9
10 Program: main.o Complex.o
11     $(CC) $(FLAGS) -o $@ $^
12
13 %.o: %.cpp $(DEPS)
14     $(CC) $(FLAGS) -c $<
15
16 clean: mostlyclean
17     rm -f Program
18
19 mostlyclean:
20     rm -f *.o
```

Here, `make mostlyclean` leaves our `Program` intact, but removes the `.o` files that aren't needed after `Program` is built and `make clean` removes everything.

Chapter 35

Valgrind

Chapter 36

GDB

Chapter 37

Big-Oh Notation

Chapter 38

Top-Down Design

Chapter 39

Dynamic Programming

Chapter 40

Defensive Programming