```
201.555.2368 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0000
```

**Exercises Section 17.3.4**

**Exercise 17.24:** Write your own version of the program to reformat phone numbers.

**Exercise 17.25:** Rewrite your phone program so that it writes only the first phone number for each person.

**Exercise 17.26:** Rewrite your phone program so that it writes only the second and subsequent phone numbers for people with more than one phone number.

**Exercise 17.27:** Write a program that reformats a nine-digit zip code as `ddddd-dddd`.

## 17.4. Random Numbers

Programs often need a source of random numbers. Prior to the new standard, both C and C++ relied on a simple C library function named `rand`. That function produces pseudorandom integers that are uniformly distributed in the range from 0 to a system-dependent maximum value that is at least 32767.

The `rand` function has several problems: Many, if not most, programs need random numbers in a different range from the one produced by `rand`. Some applications require random floating-point numbers. Some programs need numbers that reflect a nonuniform distribution. Programmers often introduce nonrandomness when they try to transform the range, type, or distribution of the numbers generated by `rand`.

The random-number library, defined in the `random` header, solves these problems through a set of cooperating classes: **random-number engines** and **random-number distribution classes**. These clases are described in Table 17.14. An engine generates a sequence of `unsigned` random numbers. A distribution uses an engine to generate random numbers of a specified type, in a given range, distributed according to a particular probability distribution.

**Table 17.14. Random Number Library Components**

| | |
|---|---|
| Engine | Types that generate a sequence of random `unsigned` integers |
| Distribution | Types that use an engine to return numbers according to a particular probability distribution |

**Best Practices**

C++ programs should not use the library `rand` function. Instead, they should use the `default_random_engine` along with an appropriate distribution object.

### 17.4.1. Random-Number Engines and Distribution

The random-number engines are function-object classes (§ 14.8, p. 571) that define a call operator that takes no arguments and returns a random `unsigned` number. We can generate raw random numbers by calling an object of a random-number engine type:

Click here to view code image

```
default_random_engine e;   // generates random unsigned integers
for (size_t i = 0; i < 10; ++i)
     // e() "calls" the object to produce the next random number
     cout << e() << " ";
```

On our system, this program generates:

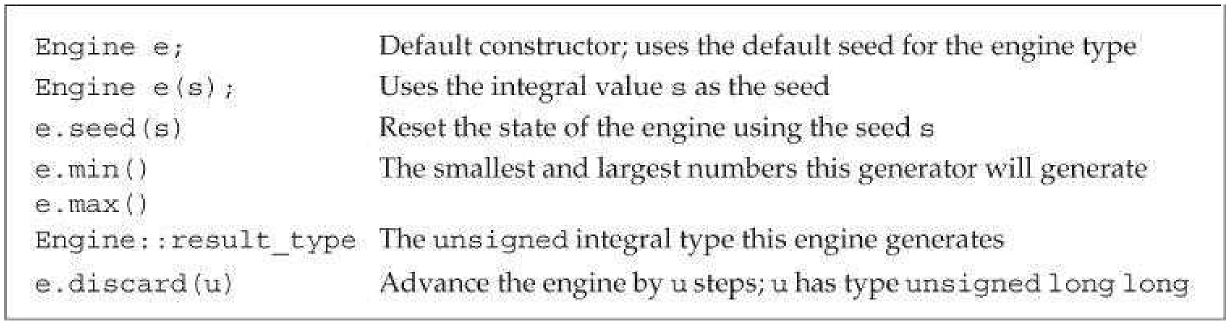
Click here to view code image

```
16807 282475249 1622650073 984943658 1144108930 470211272 ...
```

Here, we defined an object named `e` that has type **default_random_engine**. Inside the `for`, we call the object `e` to obtain the next random number.

The library defines several random-number engines that differ in terms of their performance and quality of randomness. Each compiler designates one of these engines as the `default_random_engine` type. This type is intended to be the engine with the most generally useful properties. Table 17.15 lists the engine operations and the engine types defined by the standard are listed in § A.3.2 (p. 884).

**Table 17.15. Random Number Engine Operations**



| | |
|---|---|
| `Engine e;` | Default constructor; uses the default seed for the engine type |
| `Engine e(s);` | Uses the integral value `s` as the seed |
| `e.seed(s)` | Reset the state of the engine using the seed `s` |
| `e.min()`<br>`e.max()` | The smallest and largest numbers this generator will generate |
| `Engine::result_type` | The `unsigned` integral type this engine generates |
| `e.discard(u)` | Advance the engine by `u` steps; `u` has type `unsigned long long` |

For most purposes, the output of an engine is not directly usable, which is why we described them earlier as raw random numbers. The problem is that the numbers usually span a range that differs from the one we need. *Correctly* transforming the range of a random number is surprisingly hard.

### Distribution Types and Engines

To get a number in a specified range, we use an object of a distribution type:

**Click here to view code image**

```
// uniformly distributed from 0 to 9 inclusive
uniform_int_distribution<unsigned> u(0,9);
default_random_engine e;  // generates unsigned random integers
for (size_t i = 0; i < 10; ++i)
    // u uses e as a source of numbers
    // each call returns a uniformly distributed value in the specified range
    cout << u(e) << " ";
```

This code produces output such as

```
0 1 7 4 5 2 0 6 6 9
```

Here we define u as a `uniform_int_distribution<unsigned>`. That type generates uniformly distributed `unsigned` values. When we define an object of this type, we can supply the minimum and maximum values we want. In this program, `u(0,9)` says that we want numbers to be in the range `0` to `9` *inclusive*. The random number distributions use inclusive ranges so that we can obtain every possible value of the given integral type.

Like the engine types, the distribution types are also function-object classes. The distribution types define a call operator that takes a random-number engine as its argument. The distribution object uses its engine argument to produce random numbers that the distribution object maps to the specified distribution.

Note that we pass the engine object itself, `u(e)`. Had we written the call as `u(e())`, we would have tried to pass the next value generated by e to u, which would be a compile-time error. We pass the engine, not the next result of the engine, because some distributions may need to call the engine more than once.

**Note**

When we refer to a **random-number generator**, we mean the combination of a distribution object with an engine.

### Comparing Random Engines and the `rand` Function

For readers familiar with the C library `rand` function, it is worth noting that the output of calling a `default_random_engine` object is similar to the output of `rand`. Engines deliver `unsigned` integers in a system-defined range. The range for `rand` is 0 to `RAND_MAX`. The range for an engine type is returned by calling the `min` and `max` members on an object of that type:

**Click here to view code image**

```
cout << "min: " << e.min() << " max: " << e.max() << endl;
```

On our system this program produces the following output:

```
min: 1 max: 2147483646
```

### Engines Generate a Sequence of Numbers

Random number generators have one property that often confuses new users: Even though the numbers that are generated appear to be random, a given generator returns the same sequence of numbers each time it is run. The fact that the sequence is unchanging is very helpful during testing. On the other hand, programs that use random-number generators have to take this fact into account.

As one example, assume we need a function that will generate a `vector` of 100 random integers uniformly distributed in the range from 0 to 9. We might think we'd write this function as follows:

**Click here to view code image**

```
// almost surely the wrong way to generate a vector of random integers
// output from this function will be the same 100 numbers on every call!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

However, this function will return the same `vector` every time it is called:

**Click here to view code image**

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// will print equal
cout << ((v1 == v2) ? "equal" : "not equal") << endl;
```

This code will print `equal` because the `vector`s `v1` and `v2` have the same values.

The right way to write our function is to make the engine and associated distribution objects `static` (§ 6.1.1, p. 205):

**Click here to view code image**

```
// returns a vector of 100 uniformly distributed random numbers
vector<unsigned> good_randVec()
{
    // because engines and distributions retain state, they usually should be
    // defined as static so that new numbers are generated on each call
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0,9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

Because `e` and `u` are `static`, they will hold their state across calls to the function. The first call will use the first 100 random numbers from the sequence `u(e)` generates, the second call will get the next 100, and so on.

**Warning**

A given random-number generator always produces the same sequence of numbers. A function with a local random-number generator should make that generator (both the engine and distribution objects) `static`. Otherwise, the function will generate the identical sequence on each call.

#### Seeding a Generator

The fact that a generator returns the same sequence of numbers is helpful during debugging. However, once our program is tested, we often want to cause each run of the program to generate different random results. We do so by providing a **seed**. A seed is a value that an engine can use to cause it to start generating numbers at a new point in its sequence.

We can seed an engine in one of two ways: We can provide the seed when we create an engine object, or we can call the engine's `seed` member:

**Click here to view code image**

```
default_random_engine e1;              // uses the default seed
default_random_engine e2(2147483646); // use the given seed value
// e3 and e4 will generate the same sequence because they use the same seed
default_random_engine e3;        // uses the default seed value
e3.seed(32767);                  // call seed to set a new seed value
default_random_engine e4(32767); // set the seed value to 32767
```

```
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
           cout << "unseeded match at iteration: " << i <<
endl;
    if (e3() != e4())
           cout << "seeded differs at iteration: " << i <<
endl;
}
```

Here we define four engines. The first two, `e1` and `e2`, have different seeds and *should* generate different sequences. The second two, `e3` and `e4`, have the same seed value. These two objects *will* generate the same sequence.

Picking a good seed, like most things about generating good random numbers, is surprisingly hard. Perhaps the most common approach is to call the system `time` function. This function, defined in the `ctime` header, returns the number of seconds since a given epoch. The `time` function takes a single parameter that is a pointer to a structure into which to write the time. If that pointer is null, the function just returns the time:

**Click here to view code image**

```
default_random_engine e1(time(0));  // a somewhat random seed
```

Because `time` returns time as the number of seconds, this seed is useful only for applications that generate the seed at second-level, or longer, intervals.

**Warning**

Using `time` as a seed usually doesn't work if the program is run repeatedly as part of an automated process; it might wind up with the same seed several times.

**Exercises Section 17.4.1**

**Exercise 17.28:** Write a function that generates and returns a uniformly distributed random `unsigned int` each time it is called.

**Exercise 17.29:** Allow the user to supply a seed as an optional argument to the function you wrote in the previous exercise.

**Exercise 17.30:** Revise your function again this time to take a minimum and maximum value for the numbers that the function should return.

### 17.4.2. Other Kinds of Distributions

The engines produce `unsigned` numbers, and each number in the engine's range has the same likelihood of being generated. Applications often need numbers of different types or distributions. The library handles both these needs by defining different distributions that, when used with an engine, produce the desired results. Table 17.16 (overleaf) lists the operations supported by the distribution types.

**Table 17.16. Distribution Operations**

| | |
|---|---|
| *Dist* `d;` | Default constructor; makes `d` ready to use. Other constructors depend on the type of *Dist*; see § A.3 (p. 882). The distribution constructors are `explicit` (§ 7.5.4, p. 296). |
| `d(e)` | Successive calls with the same `e` produce a sequence of random numbers according to the distribution type of `d`; `e` is a random-number engine object. |
| `d.min()` `d.max()` | Return the smallest and largest numbers `d(e)` will generate. |
| `d.reset()` | Reestablish the state of `d` so that subsequent uses of `d` don't depend on values `d` has already generated. |

**Generating Random Real Numbers**

Programs often need a source of random floating-point values. In particular, programs frequently need random numbers between zero and one.

The most common, *but incorrect*, way to obtain a random floating-point from `rand` is to divide the result of `rand()` by `RAND_MAX`, which is a system-defined upper limit that is the largest random number that `rand` can return. This technique is incorrect because random integers usually have less precision than floating-point numbers, in which case there are some floating-point values that will never be produced as output.

With the new library facilities, we can easily obtain a floating-point random number. We define an object of type `uniform_real_distribution` and let the library handle mapping random integers to random floating-point numbers. As we did for `uniform_int_distribution`, we specify the minimum and maximum values when we define the object:

**Click here to view code image**

```
default_random_engine e; // generates unsigned random integers
// uniformly distributed from 0 to 1 inclusive
uniform_real_distribution<double> u(0,1);
for (size_t i = 0; i < 10; ++i)
    cout << u(e) << " ";
```

This code is nearly identical to the previous program that generated `unsigned` values. However, because we used a different distribution type, this version generates different results:

**Click here to view code image**

```
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
```

### Using the Distribution's Default Result Type

With one exception, which we'll cover in § 17.4.2 (p. 752), the distribution types are templates that have a single template type parameter that represents the type of the numbers that the distribution generates. These types always generate either a floating-point type or an integral type.

Each distribution template has a default template argument (§ 16.1.3, p. 670). The distribution types that generate floating-point values generate `double` by default. Distributions that generate integral results use `int` as their default. Because the distribution types have only one template parameter, when we want to use the default we must remember to follow the template's name with empty angle brackets to signify that we want the default (§ 16.1.3, p. 671):

**Click here to view code image**

```
// empty <> signify we want to use the default result type
uniform_real_distribution<> u(0,1); // generates double by default
```

### Generating Numbers That Are Not Uniformly Distributed

In addition to correctly generating numbers in a specified range, another advantage of the new library is that we can obtain numbers that are nonuniformly distributed. Indeed, the library defines 20 distribution types! These types are listed in § A.3 (p. 882).

As an example, we'll generate a series of normally distributed values and plot the resulting distribution. Because `normal_distribution` generates floating-point numbers, our program will use the `lround` function from the `cmath` header to round each result to its nearest integer. We'll generate 200 numbers centered around a mean of 4 with a standard deviation of 1.5. Because we're using a normal distribution, we can expect all but about 1 percent of the generated numbers to be in the range from 0 to 8, inclusive. Our program will count how many values appear that map to the integers in this range:

**Click here to view code image**

```
default_random_engine e;          // generates random integers
normal_distribution<> n(4,1.5); // mean 4, standard deviation 1.5
vector<unsigned> vals(9);         // nine elements each 0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e));   // round to the nearest integer
    if (v < vals.size())          // if this result is in range
        ++vals[v];                // count how often each number appears
}
```

```
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ": " << string(vals[j], '*') << endl;
```

We start by defining our random generator objects and a `vector` named `vals`. We'll use `vals` to count how often each number in the range 0 . . . 9 occurs. Unlike most of our programs that use `vector`, we allocate `vals` at its desired size. By doing so, we start out with each element initialized to 0.

Inside the `for` loop, we call `lround(n(e))` to round the value returned by `n(e)` to the nearest integer. Having obtained the integer that corresponds to our floating-point random number, we use that number to index our `vector` of counters. Because `n(e)` can produce a number outside the range 0 to 9, we check that the number we got is in range before using it to index `vals`. If the number is in range, we increment the associated counter.

When the loop completes, we print the contents of `vals`, which will generate output such as

**Click here to view code image**

```
0: ***
1: ********
2: ********************
3: **************************************
4: **********************************************************
5: ******************************************
6: ***********************
7: *******
8: *
```

Here we print a `string` with as many asterisks as the count of the times the current value was returned by our random-number generator. Note that this figure is not perfectly symmetrical. If it were, that symmetry should give us reason to suspect the quality of our random-number generator.

#### The bernoulli_distribution Class

We noted that there was one distribution that does not take a template parameter. That distribution is the `bernoulli_distribution`, which is an ordinary class, not a template. This distribution always returns a `bool` value. It returns `true` with a given probability. By default that probability is .5.

As an example of this kind of distribution, we might have a program that plays a game with a user. To play the game, one of the players—either the user or the program—has to go first. We could use a `uniform_int_distribution` object with a range of 0 to 1 to select the first player. Alternatively, we can use a Bernoulli distribution to make this choice. Assuming that we have a function named `play` that plays the game, we might have a loop such as the following to interact with the user:

**Click here to view code image**

```
string resp;
default_random_engine e;   // e has state, so it must be outside the loop!
bernoulli_distribution b;  // 50/50 odds by default
do {
    bool first = b(e);     // if true, the program will go first
    cout << (first ? "We go first"
                   : "You get to go first") << endl;
    // play the game passing the indicator of who goes first
    cout << ((play(first)) ? "sorry, you lost"
                           : "congrats, you won") << endl;
    cout << "play again? Enter 'yes' or 'no'" << endl;
} while (cin >> resp && resp[0] == 'y');
```

We use a `do while` (§ 5.4.4, p. 189) to repeatedly prompt the user to play.

**Warning**

Because engines return the same sequence of numbers (§ 17.4.1, p. 747), it is essential that we declare engines outside of loops. Otherwise, we'd create a new engine on each iteration and generate the same values on each iteration. Similarly, distributions may retain state and should also be defined outside loops.

One reason to use a `bernoulli_distribution` in this program is that doing so lets us give the program a better chance of going first:

**Click here to view code image**

```
bernoulli_distribution b(.55); // give the house a slight edge
```

If we use this definition for `b`, then the program has 55/45 odds of going first.

## 17.5. The IO Library Revisited

In Chapter 8 we introduced the basic architecture and most commonly used parts of the IO library. In this section we'll look at three of the more specialized features that the IO library supports: format control, unformatted IO, and random access.

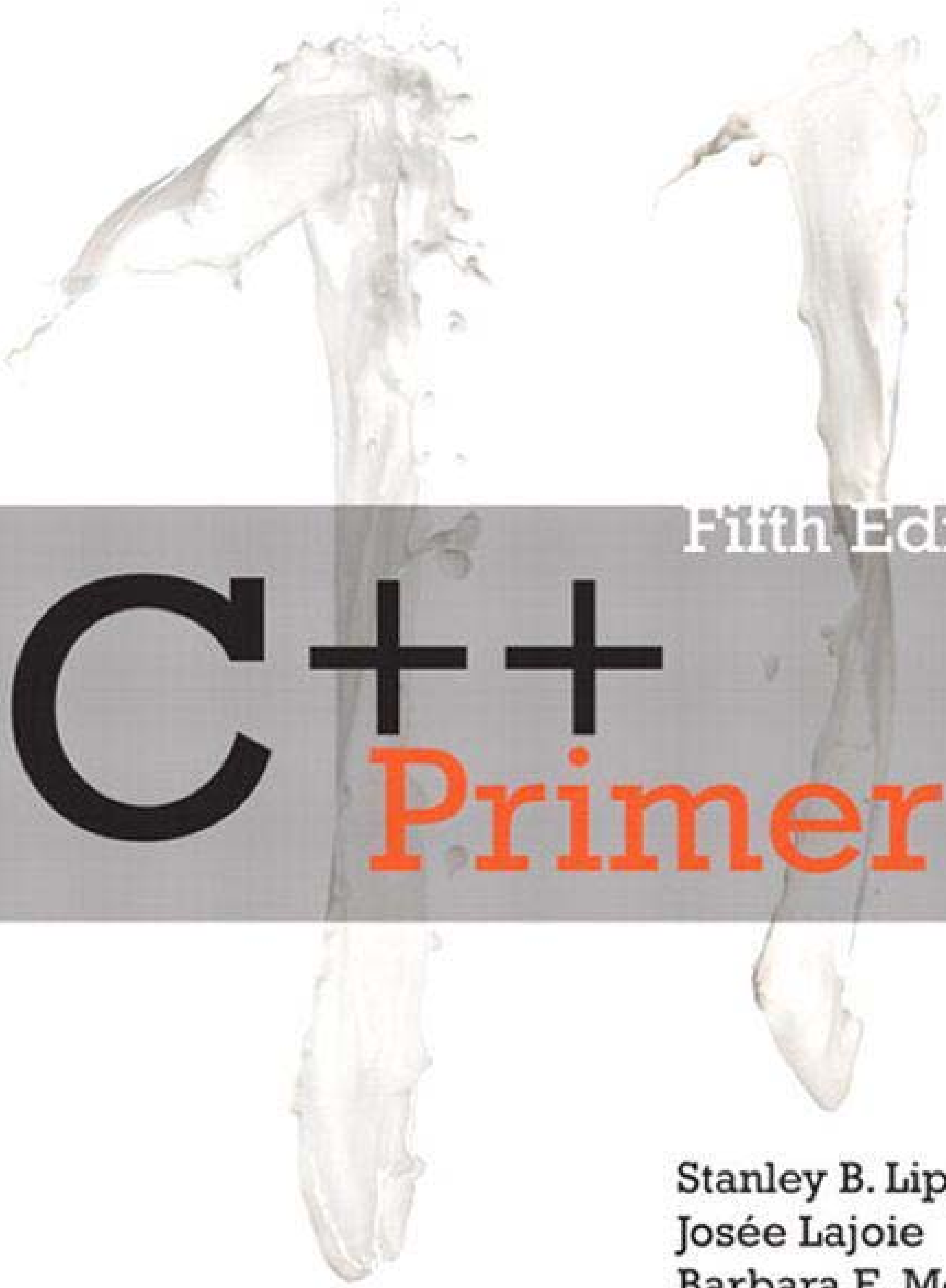
**Exercises Section 17.4.2**

**Exercise 17.31:** What would happen if we defined `b` and `e` inside the `do` loop of the game-playing program from this section?

**Exercise 17.32:** What would happen if we defined `resp` inside the loop?

Bestseller Since 1986

**Completely Rewritten for the New C++11 Standard**



Fifth Edition

# C++ Primer

Stanley B. Lippman
Josée Lajoie
Barbara E. Moo