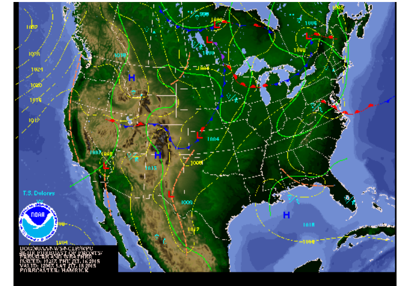
2. *Random Numbers and Simulations*

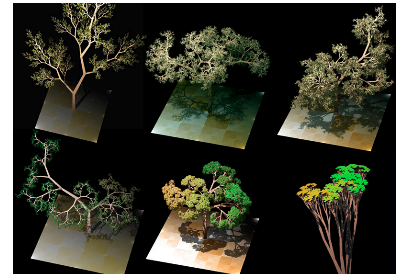
2.1. Introduction

Some of the world's most powerful computers and most sophisticated software exist for the purpose of telling you whether you need to carry an umbrella tomorrow. Weather predictions demand extreme computing power. These predictions are made by simulating the earth's atmosphere. They begin with current weather conditions (temperature, pressure, humidity, wind speed) at many locations around the world and at different heights within the atmosphere. Then they approximate the atmosphere by pretending it's made of millions of discrete "cells", and the behavior of each of these cells is simulated as it changes over time. Simulations like this allow us to find approximate answers to problems that would be difficult or impossible to solve exactly.



A weather forecast.
Source: NOAA

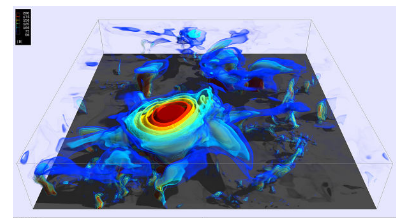
Computer simulations often make use of random numbers. If you've ever played a video game (or watched a movie with computer-generated special effects) you've seen images made with the help of random numbers. The trees in a video game forest probably aren't drawn by hand. They're generated from a recipe that uses random numbers to decide where to put the branches and leaves, how tall the tree is, and its location in the forest.



Computer-generated trees.
Source: Wikimedia Commons

Simulations can let us take random numbers, combine them with a few simple rules that describe how neighboring components interact with each other, and turn that into a prediction about the complex behavior of a large system.

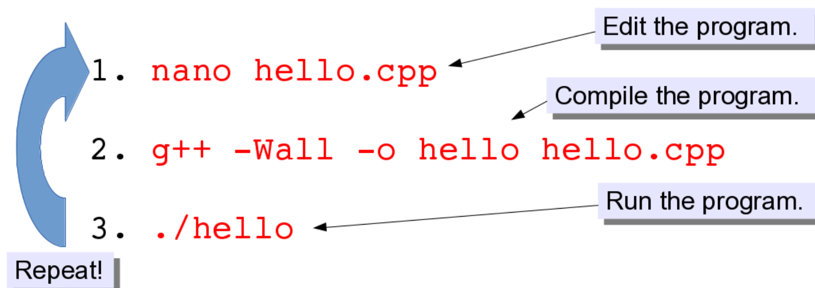
In this chapter we'll learn how to create programs that use random numbers to simulate processes in the real world.



A computer simulation of twisted magnetic fields in the Sun's atmosphere.
Source: Tim Sandstrom, NASA/Ames

2.2. The Code Development Dance

In the last chapter we saw how to create programs using an *editor* and a *compiler*. The process of creating a program is usually a loop, like the loops we created inside our programs. We start out by writing some statements in the C language and saving them into a file, then we compile the file and run the resulting binary version of the program. If the program doesn't do what we want it to do, we go back and edit some more, then try again until we have a working program. I call this process "The Code-Development Dance" (see Figure 2.1).



Dance in the Moonbeam by Theodor Kittelsen.
Source: Wikimedia Commons

Figure 2.1: The Code-Development Dance

No matter how far you go in programming, you'll still follow this same process while developing programs.

Programmers often refer to the instructions in a computer program as "code". The C language statements you've written are called "source code" and the binary files created by the compiler are called "binary code"

In the exercises that follow, we'll be working on two new programs. In each case, we'll start out with a simple version of the program, then make improvements. Each time we change something, we'll go through the process of editing our program, compiling it, and running it. Refer back to Figure 2.1 if you need help.

2.3. Using the `rand` Function

Take a look at Program 2.1, named `rand.cpp`. This program is similar to the loop programs we've written previously, but it introduces two new things. First, at the top of the program there's an extra `#include` statement. Second, the program makes use of a new function, called `rand`.

Program 2.1: rand.cpp (Version 1)

```
#include <stdio.h>
#include <stdlib.h>
int main () {
        int i;
        for ( i=0; i<10; i++ ) {
                printf ( "%d\n", rand() );
        }
}
```

Notice that `rand` is a function, like `printf`, but it's a function that takes no arguments. It just generates random numbers out of nothing.

Exercise 7: Random Numbers

Write and compile Program 2.1, using *nano* and *g++*, then run it to see what it does.

You should find that the program generates a list of seemingly random numbers. That's the whole purpose of the `rand` function. Each time your program uses `rand`, it gives you a different number.

Try running your program several times. Do you notice anything surprising?

Here's a useful tip: If you want to run your program again without having to type "./rand", you can use the up arrow key on the keyboard to bring back commands you've used before. Just keep pressing the up arrow until you see the command that you want to re-do, then press enter to repeat that command. You can also use the left and right arrow keys to move back and forth in what you've typed and make changes before you press enter.

Before we can use `rand`, we need to add the extra `#include` statement at the top of the program. This statement tells the C compiler some necessary information about the `rand` function. The first `#include` statement, which we've used in our earlier programs, provides the compiler with information it needs in order to use the `printf` function. We'll learn more about these `#include` statements in later chapters.

2.4. Making it Better

If you run Program 2.1 several times, you should find that, although the numbers look random, you get the same set of numbers each time you run the program. That doesn't seem very random, does it? Let's try to do better. Take a look at Program 2.2.

In Program 2.2 we've added two more lines. Before the "`for`" loop there's now a cryptic-looking statement involving two new functions, `srand` and `time`. Then, at the top, we've added yet another `#include` statement.

Program 2.2: rand.cpp (Version 2)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        srand(time(NULL));
        for ( i=0; i<10; i++ ) {
                printf ( "%d\n", rand() );
        }
}
```

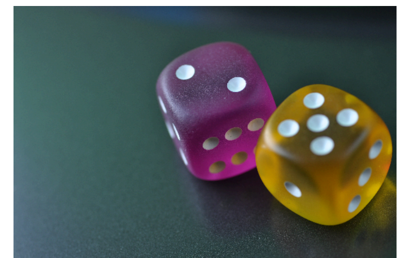
Exercise 8: More Random!

What do these changes do? Let's try it. Remember that you can modify your program by typing "`nano rand.cpp`", then make your changes, and press Ctrl-X to save your changes and exit *nano*.

Edit your `rand.cpp` program, compile it again and then try running it several times. (Wait at least one second between tries.) You should now see that you get a different set of numbers each time you run the program. That's great, but how did it happen?

2.5. Pseudo-Random Numbers

Let's think about what we mean by "random". If we roll a fair die, it should be impossible to predict which number will come up. Even if we roll the die many times, the outcome of the next roll should be unpredictable and independent of all the previous rolls. If the numbers are really random, it should be impossible to predict what the next number will be.



Rolling a fair six-sided die will give you a truly random number between 1 and 6, inclusive.

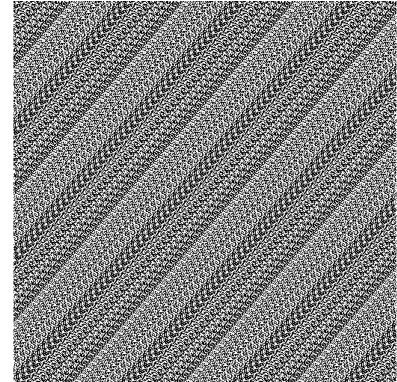
Source: Wikimedia Commons

It's not possible to generate truly random numbers using only a computer program. A function like `rand` can ultimately only do math, and we can expect that the same set of mathematical operations will always give the same answer. The `rand` function starts with an initial number (called a "seed") and then just does some very roundabout calculations that give us another number that has no *obvious* relation to the preceding number. Thereafter, each time we use `rand` in our program it builds on the number it had before.

Our first program gave us a chain of seemingly random numbers, but because the seed gets set to the same value each time we start the program, the list of numbers was always the same. The second version of the program sets the seed to a different value each time we run the program. It does this by using the computer's clock. Whenever we run Program 2.2 the seed is set to the current time, expressed as the number of seconds that have elapsed since January 1, 1970.[1] That's what "`srand(time(NULL))`" does. The `srand` function sets the seed used by `rand`. The expression "`time(NULL)`" gives us the time. The extra `#include` statement tells the compiler what it needs to know in order to use the `time` function.

[1] This is why I told you to wait at least one second between tries. Otherwise you might run the program twice with the same seed, and get the same set of numbers.

Even with this change, it's important to know that if your program generates millions or billions of numbers, `rand` will eventually start repeating itself. (See Figure 2.2.)



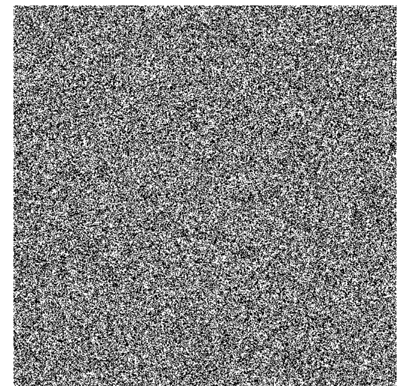


Figure 2.2: These two images show the output of a bad random number generator (top) and a better generator (bottom). The lines in the top image indicate that the generator soon starts repeating the same set of numbers. The generator used for the bottom image goes much longer without repeating.

Functions like `rand` are called "pseudo-random number generators" (PRNGs). The numbers they generate aren't really random, but they're good enough for many purposes. Some computers now include a device called a "true random number generator" (TRNG). These devices generate random numbers by observing real physical processes, such as thermal noise. They effectively roll real miniature dice to generate their random numbers. TRNGs are becoming more important because good random numbers are essential to cryptography.

2.6. Random Numbers Between Zero and One

You've probably noticed that the numbers generated by `rand` are large integers. That's fine for some things, but programmers often want to generate random real numbers that fall in the range between zero and one (for reasons that will soon become apparent). How can we do this using `rand`? Take a look at Program 2.3.

The `rand` function generates integers between zero and a large number called `RAND_MAX`. `RAND_MAX` is one of the things defined when we say `#include < stdlib.h >`.[2] If you're curious, you could print out the value of `RAND_MAX` with a statement like:

[2] The numerical value of RAND_MAX may vary, depending on what version of the C compiler you use.

```
printf( "%d\n", RAND_MAX ).
```

Program 2.3 introduces a new variable, `x`. We'll want `x` to be a random number between zero and one, so this variable can't be an integer. Instead, we'll make it a `double`.[3] We calculate the value of `x` by

[3] See Chapter 1.

Program 2.3: rand.cpp (Version 3)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        double x;
        srand(time(NULL));
        for ( i=0; i<10; i++ ) {
                x = rand()/(1.0 + RAND_MAX);
                printf ( "%lf\n", x );
        }
}
```

getting a random integer from `rand` and dividing that number by `1.0 + RAND_MAX`. Since the numbers generated by `rand` are always between zero and `RAND_MAX`, `x` should always be between zero and something slightly less than one[4].

[4] Why don't we want to go all the way to one? We'll see the benefits of that in a later chapter. For now, don't worry too much about it. Since `RAND_MAX` is a very large number, the biggest numbers we generate will be very close to one (less than a billionth smaller).

Note that it's important to say `1.0 + RAND_MAX` here instead of `1 + RAND_MAX`. To understand why, we have to think about the way C does arithmetic with integers. `RAND_MAX` and the numbers generated by the `rand` function are integers.

When C divides one integer by another, it assumes that you want the result to be an integer, too. If the result were equal to 0.7, the computer would drop everything after the decimal point and just leave zero. Since `RAND_MAX` is an integer, C would see the expression `1 + RAND_MAX` as an integer, and `rand()/(1 + RAND_MAX)` would always be zero. By just saying `1.0` instead of `1`, we give C a clue that we want to keep decimal places in our results.

Exercise 9: Making Real Numbers

Try modifying your program so that it looks like Program 2.3. Compile it, run it, and look at the results. You should now see a list of numbers that are all between 0 and 1.

2.7. Random Integers Between Some Limits

Sometimes we want to generate a random integer between some minimum and maximum values. For example, maybe we want to simulate rolling a six-sided die, so we want to generate numbers between one and six.

We can do this by starting with a random real number between zero and one, as described in the preceding section. For example, we might have a `double` variable named `x` in our program, and a line that says:

```
x = rand()/(1.0 + RAND_MAX);
```

That would give `x` a random value between 0 and 0.9999999...[5]. We could multiply this by six to get a number between 0 and 5.9999999.... Let's create a new `double` variable named `y` that does that:

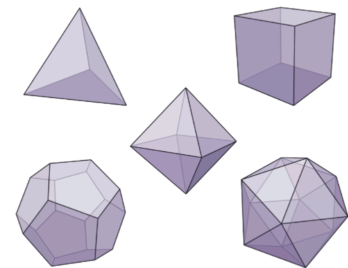
[5] It never quite gets to 1.0 because the maximum value returned by `rand` is `RAND_MAX` and we're dividing by `1.0 + RAND_MAX`.

```
y = 6 * x;
```

C provides us with a way of chopping the decimal part off of a number. All we need to do is put `(int)` in front of the value. Let's modify our program so that we have an integer variable named `i` instead of the `double` variable named `y`:

```
i = (int)( 6 * x );
```

Notice that we've put parentheses around `6 * x` so that `(int)` applies to the whole thing. Otherwise, it would just apply to `6`. Before the `(int)` is applied, we have a random number between 0 and 5.9999999.... The `(int)` chops off the decimals and leaves us with a number between 0 and 5.



Dice come in many shapes. Often they're shaped like one of the five platonic solids. These are the only regular convex polyhedra that are possible in three dimensions. In four dimensions there are six such shapes, but in five and higher dimensions, there are only three. See this excellent video by Carlo Sequin for some fun with higher-dimensional "polytopes": https://www.youtube.com/watch?v=2s4TqVAbfz4.
Source: *Wikimedia Commons*

If our goal is to generate a number between 1 and 6, we just need to do one more thing: add 1 to the value of `i`.

```
i = 1 + (int)( 6 * x );
```

What if we wanted numbers between 2 and 7 instead of 1 and 6? Then we'd just need to change one thing:

```
i = 2 + (int)( 6 * x );
```

Notice that the multiplier, 6, didn't change. This is because the our new range still includes six possible values. Now they're 2, 3, 4, 5, 6, and 7. In general, if we want integers between n_{min} and n_{max}, the number of values will be $n_{max} - n_{min} + 1$.

So, if we want to get a random integer between `min` and `max` we can do it like this:

```
nvals = max - min + 1;
i = min + (int)( nvals * x );
```

Program 2.4 uses this strategy to generate a random number between 1 and 6.

Program 2.4: diceroll.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  double x;
  int i;
  int min = 1;
  int max = 6;
  int nvals;

  nvals = max - min + 1;

  srand(time(NULL));

  x = rand()/(1.0 + RAND_MAX);
  i = min + (int)(nvals*x );

  printf ( "%d\n", i );
}
```

This program could be modified to generate a random integer in any range you want, just by changing the values of `min` and `max`.

Exercise 10: Gonna Roll The Bones

Write a program based on Program 2.4 that rolls *two* six-sided dice and prints (1) the number on each die and (2) the sum of their two numbers. For example, if both dice roll six, the sum would be twelve. Run the program several times to see if you can roll a twelve!

Claus Meyer, 1886, *Die Würfelspieler*.

Source: Wikimedia Commons

2.8. Writing a Simulation Program

Imagine a rock in a gutter. In this place it rains once per day, and every time it rains the rock slides some random distance, Δx, down the gutter. Assume Δx is always between zero and 100 cm. Let's try to simulate this physical system with a computer program, and see how the rock behaves.

For more on stones in gutters, see the excellent short story "Fall of Pebble-Stones" by R.A. Lafferty.

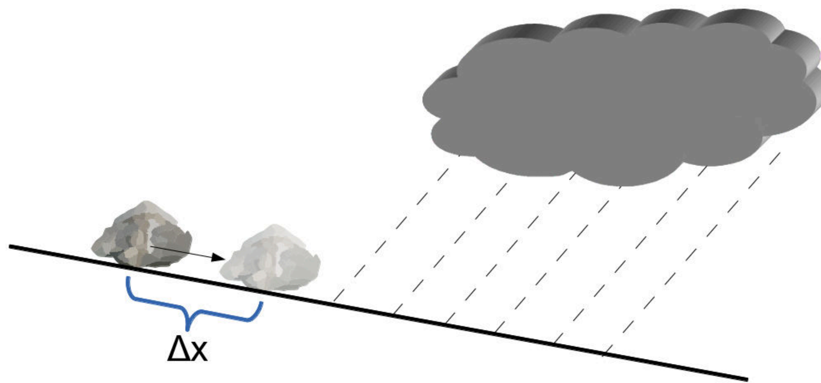


Figure 2.3: A rock, sliding along a gutter.

Program 2.5: gutter.cpp (Version 1)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        double delta_x;
        double x;
        srand(time(NULL));
        x = 0.0;
        for ( i=0; i<10; i++ ) {
                delta_x = 100.0 * rand()/(1.0 + RAND_MAX);
                x = x + delta_x;
                printf ( "%lf\n", x );
        }
}
```

You'll notice that Program 2.5 is very similar to Program 2.3. The main differences are that (1) we set the variable x equal to 0.0 before starting our loop, and (2) each time around the loop we add a random amount to x. Also, instead of x as our random number, we've renamed this variable delta_x.

As you saw in Chapter 1, when you see an expression like `x = x+d` in a C program it means "Set the new value of x equal to the old value plus d". Remember that this is a little different from what you may be used to in algebra. It might help if you keep in mind that, in C, the statement `x = 1` means "assign the value 1 to the variable x". In algebra, on the other hand, the same statement would mean "I promise you that x is equal to 1".

The variable x stores the rock's current position, in centimeters. It starts out at `x = 0.0`. Each time around the loop represents one rainstorm, which washes the rock a random distance, Δx, down the gutter. We want Δx to be a number between zero and 100 centimeters, so we calculate it by taking a random number between zero and one (as we did in Program 2.3) and multiplying that by 100. The new value of x after the rainstorm is $x + \Delta x$. At the bottom of the loop we print out the new value of x. The program simulates the movement of a rock as it slides down the gutter over the course of ten days in this very rainy location.

Exercise 11: First Gutter Program

Try writing Program 2.5, compiling it, and running it. Do the values it prints out make sense? Run it several times (waiting at least one second between tries). You should get different, but still reasonable, results each time.

Each time you run it, the last number printed by the program is the stone's position at the end of day number ten. Do these numbers seem reasonable? Keep in mind that if the stone traveled exactly 50 cm each day (halfway between zero and 100 cm), it would end up 500 cm from the origin at the end of day ten.

But what about...?

In Program 2.5 we named one of the variables `delta_x`. What kinds of names are allowed for variables in the C language?

Allowed Characters:

Variable names can only contain letters (upper- or lower-case), numbers and the underscore character, "_". Names must begin with a letter or an underscore (not a number).

It's good practice to always use a letter as the first character in variable names. Leading or trailing underscores are sometimes used internally by the compiler. If you get into the habit of using an underscore at the beginning of variable names, you may run into confusion later in your programming career.

Remember that C is case-sensitive, so that a variable named `Velocity`, with an upper-case "`V`", is completeley different from

a variable named `velocity`. Also, note in particular that spaces aren't allowed in variable names.

Maximum Length:

Different versions of the C compiler have different limits on the maximum length of variable names. The compiler we're using, *g++*, has no limit. In principle, you could give a variable a name that was thousands of letters long, although this would obviously be awkward to type! Some C compilers limit variable names to 2,048 characters, and others require that at least the first 31 characters of each name be different from any other name in your program. With all of that in mind, it would be a good idea to limit yourself to variable names that are 31 characters or fewer.

It's good practice to give your variables clear, concise names like `velocity`, `width`, `temperature`, *et cetera*. This helps you remember what they're for, and makes it easier for other people to understand your program.

Reserved Words:

Some names are simply not allowed. For one thing, you can't give your variable a name that's the same as any of the words that make up the C language. You couldn't, for example, name a variable `int`, `double` or `for`. There are 32 words of this type. For the record, they are:

> **auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while**

You also can't give your variable the same name as any function your program knows about. It wouldn't be allowable to name a variable `printf`, for example, in any of the programs we've written so far.

(Note that I'm being careful to say "any function your program knows about". You'll understand what I mean later, when we talk about libraries of functions.)

2.9. Some New Arithmetic Operators

The C compiler understands many arithmetic operators. Besides $+$, $-$, $*$, and / there are several "combination" operators that provide shortcuts for doing common operations. Figure 2.4 shows some of these.

If we say, for example, `d += 100`, we mean "increment the value of `d` by 100". It's exactly equivalent to writing `d = d + 100`, but a little easier to type. I find that it also helps prevent typing errors, especially with long variable names. Consider the following for example:

```
somelongname = somelongnome + 10;
```

The += operator is similar to the ++ operator we've been using in "`for`" loops. The difference is that ++ increments the value by 1, but += can increment by any amount.

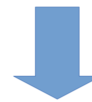
Did you catch the typo? If I'd written `somelongname += 10` instead, I'd have one less opportunity to misspell the variable name.

Arithmetic Operators:

C has many arithmetic operators. Here are some of them:

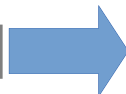
+	a+b	Addition
-	a-b	Subtraction
*	a*b	Multiplication
/	a/b	Division

Some operators let you do arithmetic while assigning a value to a variable.



Operator	Usage	Equivalent to
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b

++ and -- do this too:



```
decrement   a++   → a = a+1
decrement   a--   → a = a-1
```

Figure 2.4: Some of C's arithmetic operators.

2.10. Focusing on the Important Results

What if we're only interested in the total distance a stone has travelled at the end of ten days? We can modify our program, as shown in Program 2.6, so that instead of printing each new position, it only prints out the final position. As you can see, this just requires us to move the `printf` statement outside of the "`for`" loop.

Note that Program 2.6 also takes advantage of the += operator to make one of the statements a little shorter. Remember that `x += delta_x` does exactly the same thing as `x = x + delta_x`.

Program 2.6: gutter.cpp (Version 2)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        double delta_x;
        double x;
        srand(time(NULL));
        x = 0.0;
        for ( i=0; i<10; i++ ) {
                delta_x = 100.0 * rand()/(1.0 + RAND_MAX);
                x += delta_x;
        }
        printf ( "%lf\n", x );
}
```

Exercise 12: Let's Race!

Modify your `gutter` program so that it looks like Program 2.6. Compile it, and then run it a few times. Each time you run it, you should see a single number, and you should get a different number each time (assuming you wait at least one second between tries, as before). Try racing your stone with your neighbors!

2.11. Tips for Using Loops

Almost all of the programs we write will use loops. Here are a few tips that will help keep you out of trouble when using them:

- **Count starting with zero, not one.** You could write a "`for`" loop like this to count from 1 to ten:

  ```
  for ( i=1; i<11; i++ )
  ```

  but you'll find later that it's more natural in C to number items starting with zero instead of one. So, in the programs we've been writing we loop ten times by writing a "`for`" statement like this, instead:

  ```
  for ( i=0; i<10; i++ )
  ```

  Doing it this way will make things much easier for you in the future.

- **Don't change the value of your counter variable inside the loop.** For example, what would this do?:

  ```
  #include <stdio.h>
  int main() {
          int i;
          for (i = 0 ; i < 10 ; i++) {
                  i = 100*i;
                  printf("loop number %d\n", i);
          }
  }
  ```

  (Note the line that reads `i = 100*i`.)

  If you tried it, you'd see that the program only prints out two numbers, instead of the ten numbers you might have expected. Why is this? It's because you've changed the value of `i` inside the loop.

  The first time around the loop, the program prints "0", and the second time around the loop it prints "100". So far, so good. But then the program stops.

  This happens because the value of `i` is now 100, so when we get back to the top of the loop, the "`for`" statement sees that "`i<10`" is no longer true, and the loop stops.[6]

[6] See the discussion about how "`for`" loops work in Chapter 1.

- **Finally, don't assume that your counter variable has a useful value any place outside its loop.** After the loop is finished, does "`i`" contain the number of times around the loop, or something more or less? (Or even something completely different?) The answer can get complicated. It's better to assume that you can only trust the value of the counter variable when you're inside its loop.

2.12. Nested Loops

Let's get back to our gutter program now. Imagine that we draw a starting line and arrange a bunch of our rocks behind it, ready to race each other down the gutter like racehorses in their starting gates. After many rainstorms, the rocks would all be at different locations somewhere lower down the gutter.

They'd be at different locations because each rock slides a different random amount during each rainstorm. A few rocks will get lucky and travel a long way. A few will travel unusually short distances. Most of the rocks will end up somewhere between these extremes, mounded up around some average distance.

Does the output of our program match this prediction? If we wanted something really boring to do, we could run Program 2.6 once for each rock, write down the results, and then graph them. Computers can save us that effort, though, and they're less likely to make the mistakes we might make while doing the work ourselves.

The inner loop of Program 2.7 is nested inside the outer loop, like these Russian Matryoshka dolls.

Source: Wikimedia Commons

We can modify our program so that it effectively runs the simulation many times. To do this, we'll need to add another loop. Take a look at Program 2.7.

Program 2.7: gutter.cpp (Version 3)

Changes from Program 2.6 are shown in bold.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
        int i;
        int j;
        double delta_x;
        double x;
        srand(time(NULL));
        for ( j=0; j<10000; j++ ) {
          x = 0.0;
           for ( i=0; i<10; i++ ) {
                delta_x = 100.0 * rand()/(1.0 + RAND_MAX);
                x += delta_x;
           }
          printf("%lf %d\n", x, j);
        }
}
```

Nested Loops

The new loop wraps around the loop that was already there. (We say that the old loop is "nested" inside the new loop.) Each time we go around the new loop we'll simulate another stone washing down the gutter for ten days. The variable `i` counts the number of rainstorms and `j` counts the number of stones. The program simulates 10,000 stones! That would be a lot of work by hand, but it's trivial for a modern computer.

The program prints out two numbers[7] for each stone: The total distance the stone travels, and the number of the stone's "starting gate". We number these gates from zero to 9,999, and use these numbers to keep track of which stone is which. We use the new variable `j` to represent the starting gate number, and this is the counter variable for the newly-introduced loop.

[7] Notice that our `printf` statement here has two placeholders, "`%lf %d`", one for the stone's final position, which is a number containing decimal places, and one for the stone's starting gate, which is an integer.

Each stone will start at the same place, so every time the program starts a new stone, it resets `x` (which represents the stone's position) to zero. When a stone has been through ten rainstorms, its final position and starting gate number are printed out, and then the program starts working on another stone.

Exercise 13: Scattering Stones

Modify your "gutter" program so that it looks like Program 2.7. Compile it, but don't run it like you've run the preceding programs. Instead, use the trick we saw in Chapter 1 that lets you send the program's output into a file, like this:

```
./gutter > gutter.dat
```

Now plot your results using *gnuplot*. Type *gnuplot*, then enter the following commands (can you guess what the *xrange* command does?):

```
set xrange [0:]
plot "gutter.dat"
```

You should see something like Figure 2.5. The horizontal axis shows how far each stone traveled. The vertical axis shows which gate the stone started from. As you can see, a "typical" stone travels about 500 cm, but some stones only make it to about 200 cm, and some go over 800 cm.

Does Figure 2.5 look the way we'd expect it to? Let's think about it. During each rainstorm, a stone travels a random distance between

zero and 100 centimeters. We'd expect the average distance to be 50 centimeters. So, after ten rainstorms, we'd expect a typical stone would travel $50 \times 10 = 500$ centimeters. This is the position of the densest part of Figure 2.5. A maximally sticky stone wouldn't move at all (travelling zero centimeters), and a maximally slippery stone would zip through a distance of $100 \times 10 = 1,000$ centimeters. We'd expect our graph to range from zero to 1,000 centimeters, with a peak at around 500 centimeters, and that's indeed what it shows.

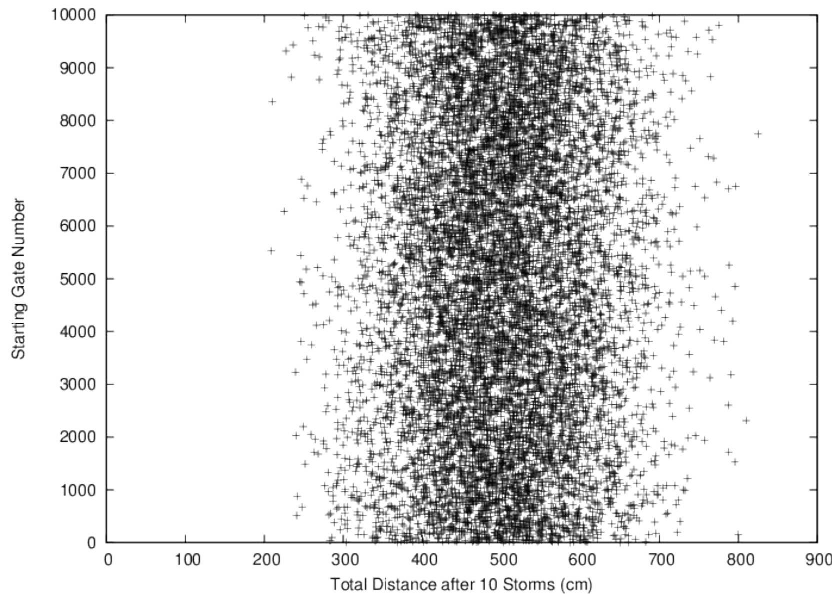


Figure 2.5: A plot of the results from our latest version of the "gutter" program.

But what about...?

Sometimes, you'll make a mistake that causes your program to keep looping forever. What can you do to stop this?

You can tell the program to stop running by pressing **Ctrl-C** (hold down the Ctrl key while pressing the "C" key).

Stopping a runaway program.

Source: Wikimedia Commons

2.13. Conclusion

Imagine that we continued to extend and improve our "gutter" program. We could add the effects of friction, rainstorms of random duration and strength, the slope of the gutter, and so forth. Eventually, we might have a program that could realistically simulate erosion, an avalanche or a mudslide.

Erosion near Bern, Switzerland
Source: Wikimedia Commons

For example, we could modify our program so that the range of random distances was determined by the duration of the rainstorm, instead of always being zero to 100 cm. Then we'd generate rainstorms of random durations and see what happens. By adding more and more refinements, we can make our simulation's results similar enough to reality to meet our needs.

Simulation programs like this allow us to handle large, complex problems by breaking them up into simple, understandable pieces. They represent an important computing technique that you can apply to many problems.

Practice Problems

1. As described in Section 2.6, write a program that prints out the value of `RAND_MAX`. Call your program `printrand.cpp`.

2. Write a program named `epoch.cpp` that prints the following:

```
Seconds since 1970: ...
Years since 1970: ...
```

   Where the `...` is replaced by the current number of seconds and years since 1970, based on the value returned by the `time` function, as described in Section 2.5. Check your program by running it several times to make sure that the number of seconds changes as time passes.
   Hint 1: The statement "`t = time(NULL);`" will store the number of seconds in the variable `t`.
   Hint 2: Assume that the number of seconds in a year is `60*60*24*365.25`.
   Hint 3: You'll probably want to use `%lf` as the placeholder when printing the year. Otherwise, *g++* might give you warnings or errors.

"Time keeps on slippin, slippin, slippin, into the future...." Steve Miller in 1977.
Source: Wikimedia Commons

3. Modify Program 2.4 so that it generates either a zero or a one. Then modify your new program so that it uses a loop to do this ten times. The resulting program does the equivalent of ten coin flips, with zero or one representing heads or tails. Call your new program `coinflip.cpp`.

4. Modify Program 2.4 so that it prints out two random digits between zero and nine. Make the program write the digits side-by-side, like `67` or `03`. Call your new program `percentile.cpp`. If you've ever played a roll-playing game like Dungeons and Dragons you've used ten- or twenty-sided dice to generate pairs of digits like this. In these games such a pair of dice are called *percentile dice*. The two digits they give you are interpreted as a percentage between 00% and 99%.

A 20-sided die shaped like an icosahedron. Two dice like this were originally used in Dungeons and Dragons for rolling percentiles. Later, they were replaced by two ten-sided dice. In this author's opinion, ten-sided dice are an abomination, since they aren't one of the five platonic solids!

5. Each line printed by Program 2.2 shows a single random integer. Using that program as an example, write a program that prints out *two* random integers on each line, separated by a space. Make the program print 10,000 pairs of integers. Let's call this program `tworand.cpp`. Use the trick you learned in Chapter 1 to send the program's output into a file named `tworand.dat`:

```
./tworand > tworand.dat
```

   Check the program's output by using *gnuplot* to plot the data in this file. Start *gnuplot* and give it the command:

```
plot "tworand.dat"
```

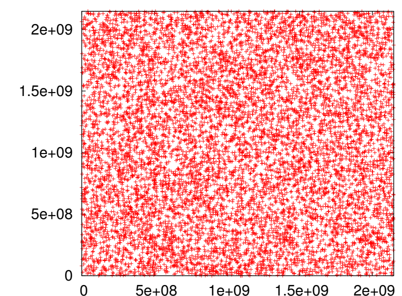


Figure 2.6: The output of the `tworand` program, plotted by *gnuplot*.

This causes *gnuplot* to use the two numbers on each line as the x and y coordinates of a point. You should see a graph that looks like Figure 2.6.

The `tworand` program generates a set of random points in the x, y plane. As we'll see later (in Chapter 10) this can be very useful.

6. Our gutter programs have a lot of numbers written into them: 10 days, 100 cm, 10,000 trials. If we want to change to, say, 1,000 trials, we need to find all of the places in the program where we currently assume a value of 10,000, and change them.

   It would be better if these numbers were more easily changed. Can you rewrite Program 2.7 so that the number of days, the maximum "slide" in cm, and the number of trials are given by variables defined near the top of the program?

   For example:

```
int ndays = 10;
double maxslide = 100.0; // in cm.
int ntrials = 1000;
```

7. Using a nested pair of loops, as described in Section 2.12, write a program named `grid.cpp` that prints out the grid shown below:

```
[0,0] [0,1] [0,2] [0,3] [0,4]
[1,0] [1,1] [1,2] [1,3] [1,4]
[2,0] [2,1] [2,2] [2,3] [2,4]
[3,0] [3,1] [3,2] [3,3] [3,4]
[4,0] [4,1] [4,2] [4,3] [4,4]
```

   Hint 1: Remember that you can leave off `\n` if you want `printf` to keep printing things on the same line.
   Hint 2: It's perfectly OK to use `printf` to print nothing but a newline, like this: `printf ("\n");`

A grid like the one you produce in Problem 7 might be used to identify the squares on a Bingo card.
Source: *publicdomainpictures.net*

8. Make a new program named `bingo.cpp` that is a modified version of Program 2.4. The new program should be different from Program 2.4 in two ways: (1) The numbers it prints should be between 1 and 75, inclusive, and (2) instead of printing just one random number, it should use a pair of nested loops[8] to print a grid of random numbers, like a Bingo card. You could use this program to generate Bingo cards! See the two hints in Problem 7 for advice about how to print a nice-looking grid. Also, don't try to make a "Free Space" in the middle: Just put a number there, like all the other squares.

[8] See Program 2.7 for an example of nested loops.

9. Imagine you have twelve 6-sided dice. Now roll all the dice at once and add up the numbers they show. This should give you a sum

between 12 and 72. Write a program named 12dice.cpp that rolls twelve dice and prints their sum. Have the program repeat this 10,000 times. Run the program like this to send its output into a file named 12dice.dat:

```
./12dice > 12dice.dat
```

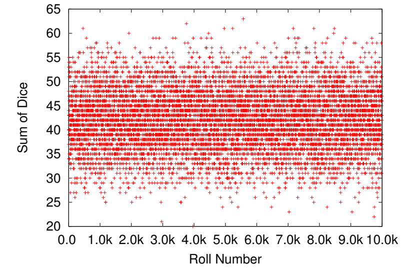


Figure 2.7: The sum of twelve dice, repeated 10,000 times. They cluster toward the center due to something mathematicians call the Central Limit Theorem. We'll talk more about this in Chapter 7.

Now start *gnuplot* and give it the command `plot "12dice.dat"`. You should see a graph like Figure 2.7. Notice that the numbers tend to cluster around a value of 42. You might expect this, since the average value for rolling a single die is $(6+1)/2 = 3.5$ and $12 \times 3.5 = 42$.

Hint: Use a variable named `sum` to hold the sum of the 12 dice. Each time you start rolling the dice, remember to set `sum` to zero at the beginning. Then just add the value of each die to `sum` until you've added all twelve numbers. Print `sum`, then go on to the next roll.