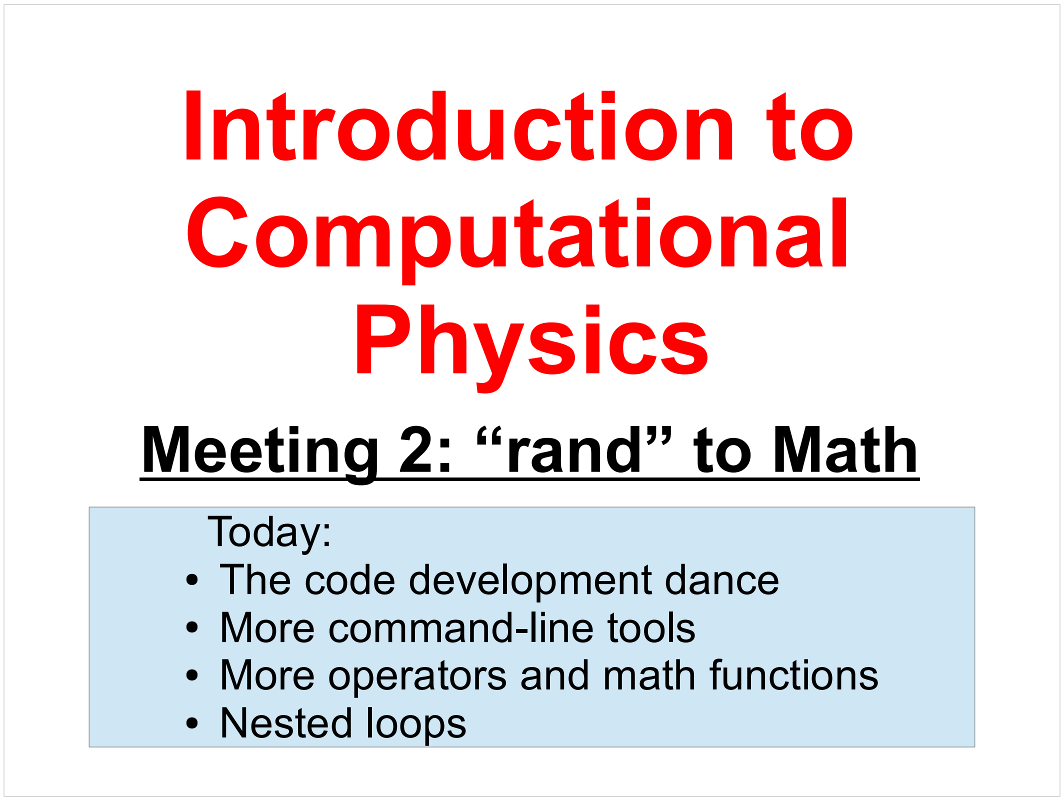


Introduction to Computational Physics

Meeting 2: “rand” to Math

Today:
- The code development dance
- More command-line tools
- More operators and math functions
- Nested loops

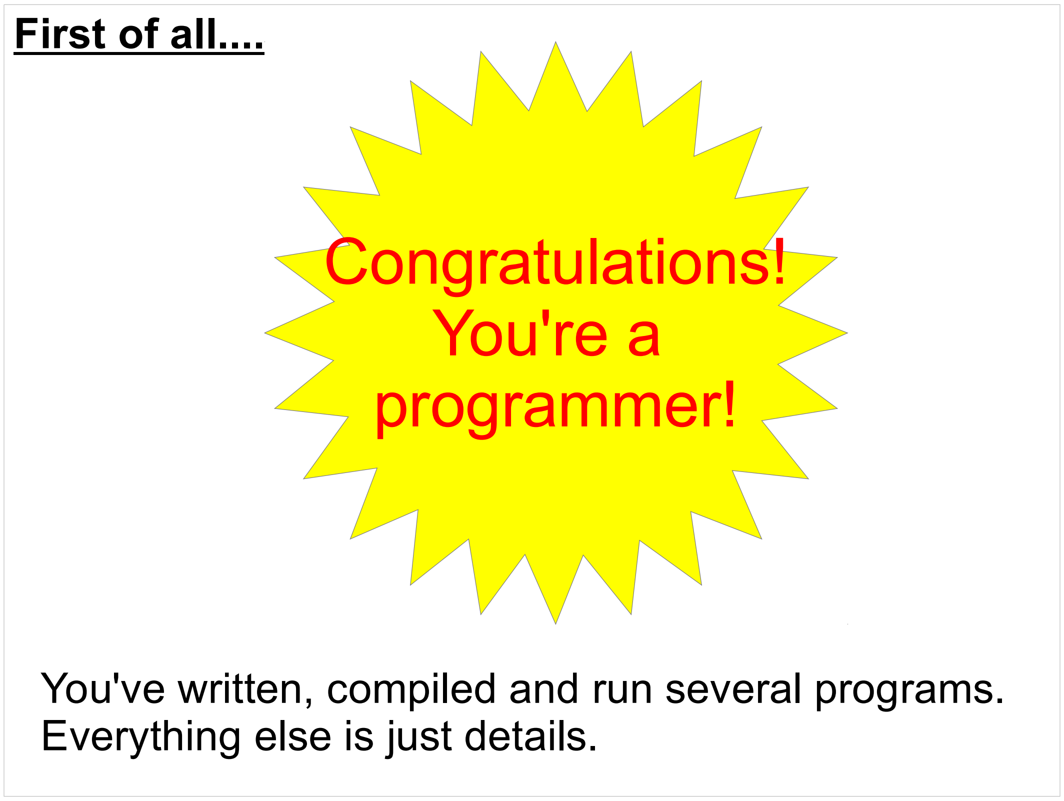
Welcome back!

From now on, remember that you can try out these programs using your account on Galileo. For instructions on how to use Galileo, see:

http://galileo.phys.virginia.edu/compfac/courses/comp_intro/connecting.html

Today we'll be following up on the random-number practice problem from last week. We'll use random numbers to write a program that simulates a physics problem.

Let's get started!



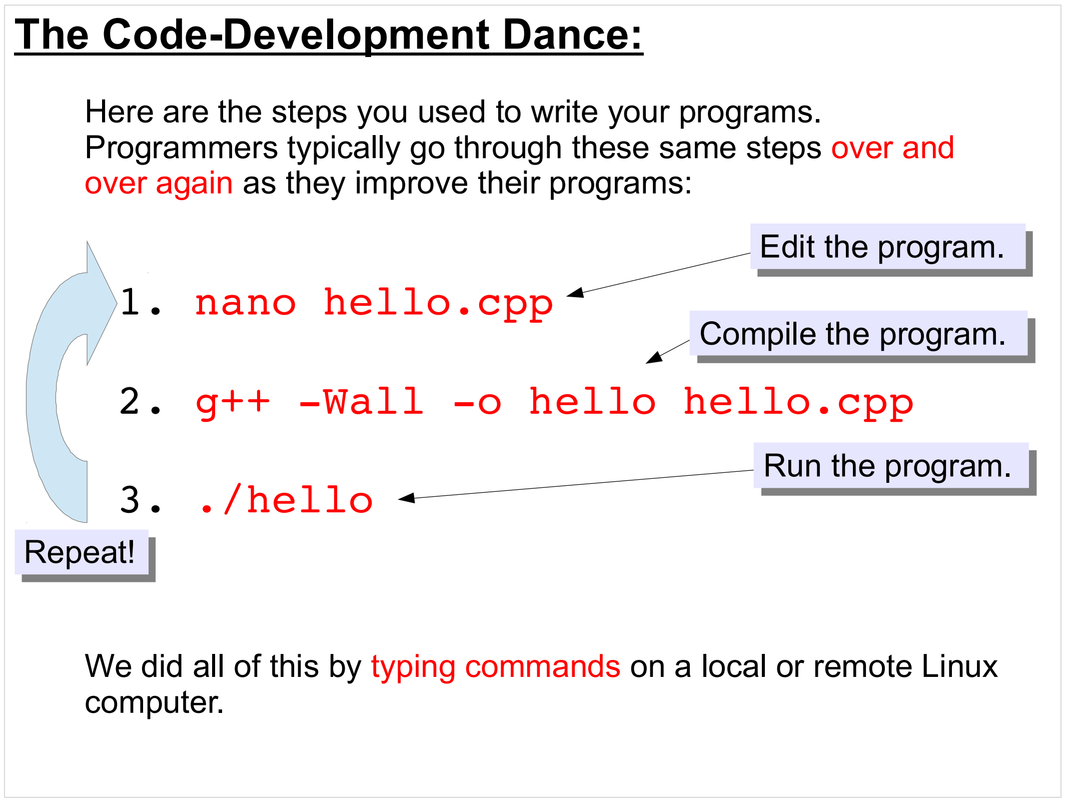
First of all....

Congratulations! You're a programmer!

You've written, compiled and run several programs. Everything else is just details.

OK, so you're a novice programmer. There are still lots of things to learn. But no matter what field you go into, that will always be true.

You've already shown that you have the skills to write programs. All you need is practice.



The Code-Development Dance:

Here are the steps you used to write your programs. Programmers typically go through these same steps over and over again as they improve their programs:

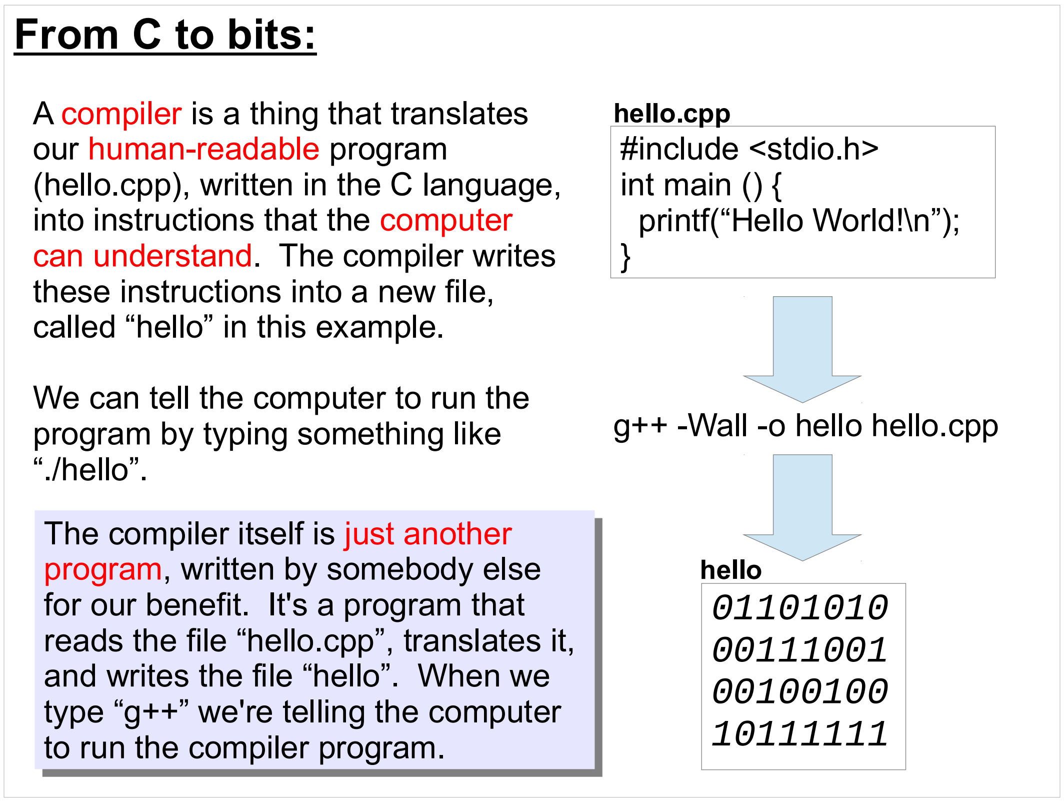
```
1. nano hello.cpp
2. g++ -Wall -o hello hello.cpp
3. ./hello
```

Edit the program.
Compile the program.
Run the program.
Repeat!

We did all of this by typing commands on a local or remote Linux computer.

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

Don't worry if your program doesn't work the first time. (Mine seldom do!) You just work on it, try it again, and keep repeating until it does what you want.



From C to bits:

A compiler is a thing that translates our human-readable program (hello.cpp), written in the C language, into instructions that the computer can understand. The compiler writes these instructions into a new file, called “hello” in this example.

We can tell the computer to run the program by typing something like “./hello”.

The compiler itself is just another program, written by somebody else for our benefit. It's a program that reads the file “hello.cpp”, translates it, and writes the file “hello”. When we type “g++” we're telling the computer to run the compiler program.

hello.cpp

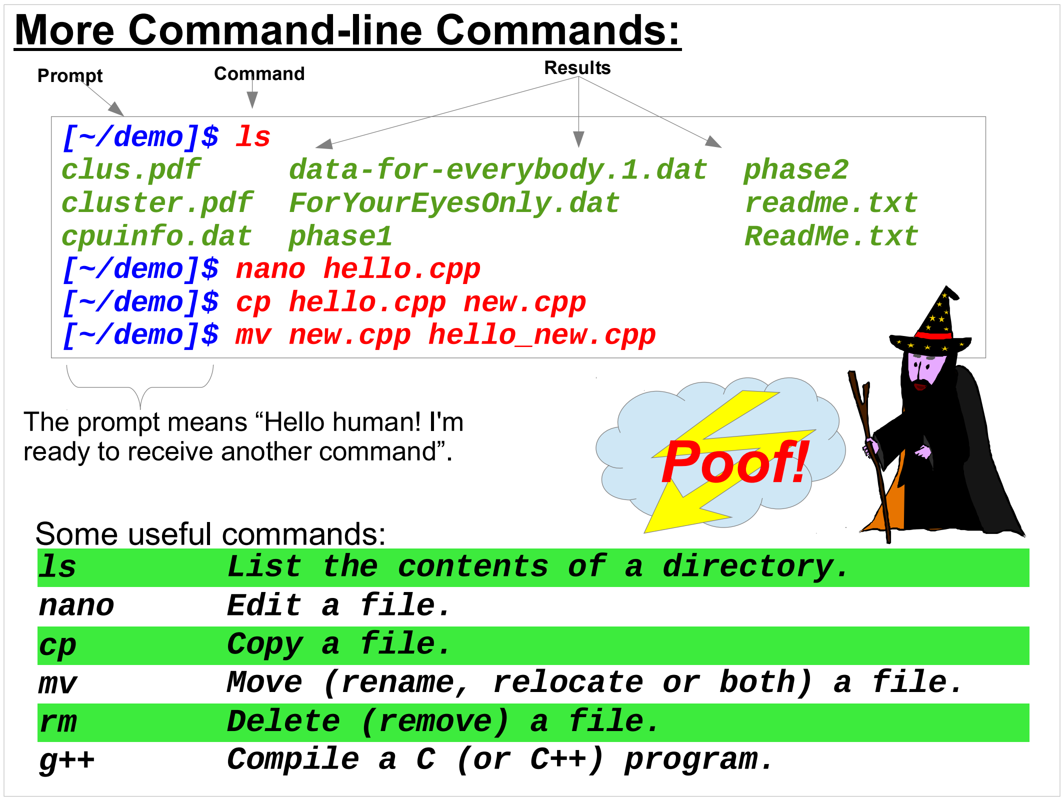
```
#include <stdio.h>
int main () {
  printf(“Hello World!\n”);
}
```

```
g++ -Wall -o hello hello.cpp
```

hello

```
01101010
00111001
00100100
10111111
```

Remember that “compiling” just means “translating our human-readable C program into a binary format that the computer can understand”.


Prompt
Command
Results
Poof!

More Command-line Commands:

```
[~/demo]$ ls
clus.pdf      data-for-everybody.1.dat  phase2
cluster.pdf   ForYourEyesOnly.dat       readme.txt
cpuinfo.dat   phase1                    ReadMe.txt
[~/demo]$ nano hello.cpp
[~/demo]$ cp hello.cpp new.cpp
[~/demo]$ mv new.cpp hello_new.cpp
```

The prompt means “Hello human! I'm ready to receive another command”.

Some useful commands:

ls	*List the contents of a directory.*
nano	*Edit a file.*
cp	*Copy a file.*
mv	*Move (rename, relocate or both) a file.*
rm	*Delete (remove) a file.*
g++	*Compile a C (or C++) program.*

The “cp” command is particularly useful. Instead of typing a whole new program, you can copy an old program that's similar, and then edit the copy.

These commands may seem odd, but think of them like magic spells. You type the magic words, and the computer does something for you.

Also remember that you can re-use commands you've typed before, by using the up and down arrow keys on your keyboard. Press up repeatedly to recall commands you've typed before.

Last Week's Practice Problem #1:

We had a “for” loop that counted from 0 to 9. How can we change the program so that it counts from 0 to 900 without changing the “for” line? Here's one way:

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

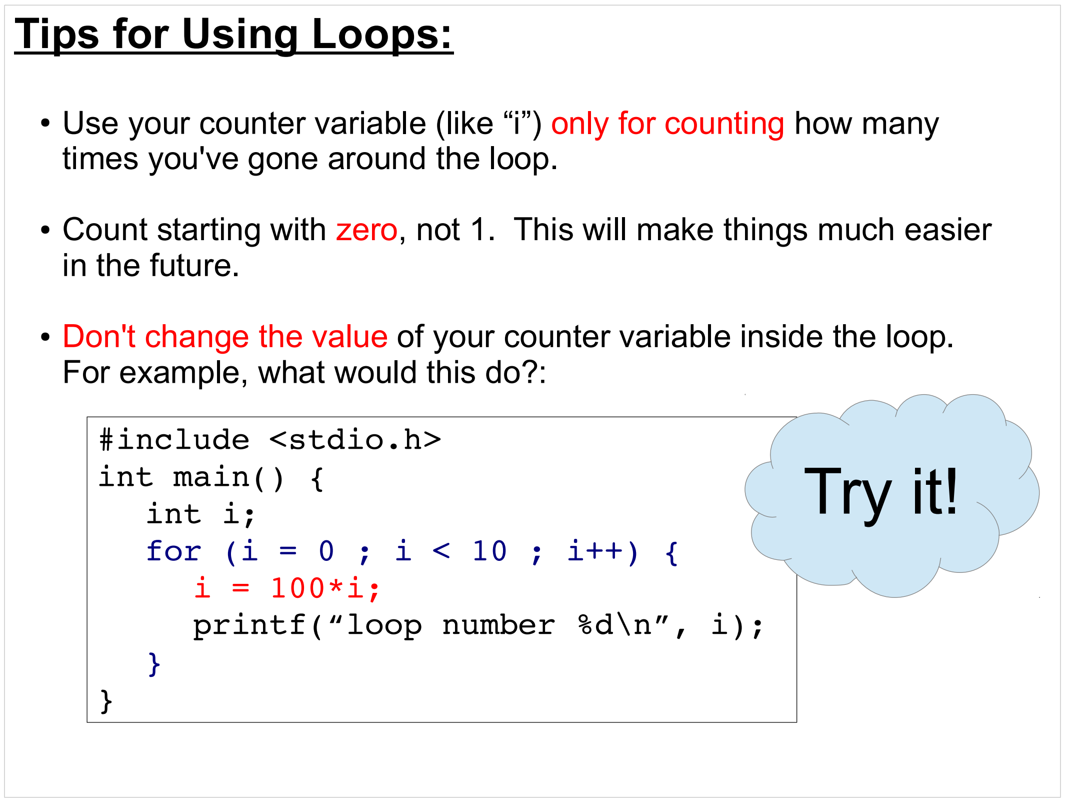
and here's another:

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

In the first case, we just print out 100*i instead of i by itself.

In the second case, we create a new variable called “value”, and set it to 100*i.

Either way is fine.



Tips for Using Loops:

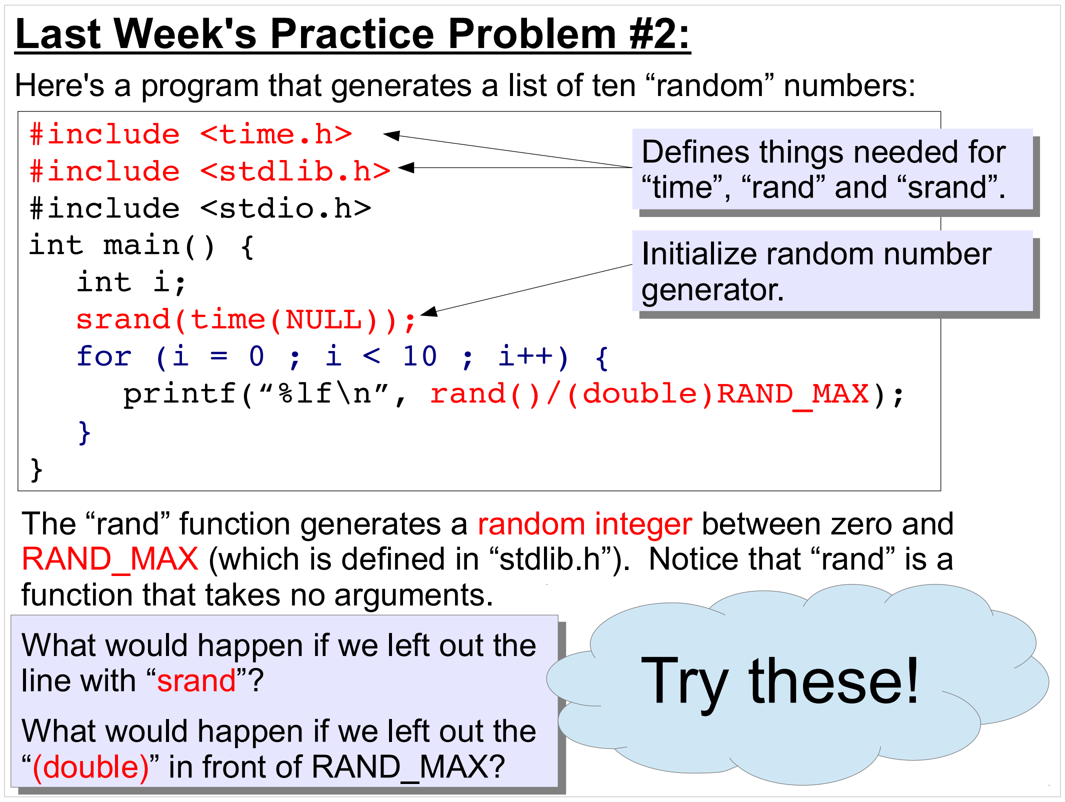
- Use your counter variable (like “i”) only for counting how many times you've gone around the loop.
- Count starting with zero, not 1. This will make things much easier in the future.
- Don't change the value of your counter variable inside the loop. For example, what would this do?:

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

If you tried the example program above, you'd see that it 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 second time around the loop we set i to a value of 100. 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.



Last Week's Practice Problem #2:

Here's a program that generates a list of ten “random” numbers:

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

Defines things needed for “time”, “rand” and “srand”.

Initialize random number generator.

The “rand” function generates a random integer between zero and RAND_MAX (which is defined in “stdlib.h”). Notice that “rand” is a function that takes no arguments.

What would happen if we left out the line with “srand”?

What would happen if we left out the “(double)” in front of RAND_MAX?

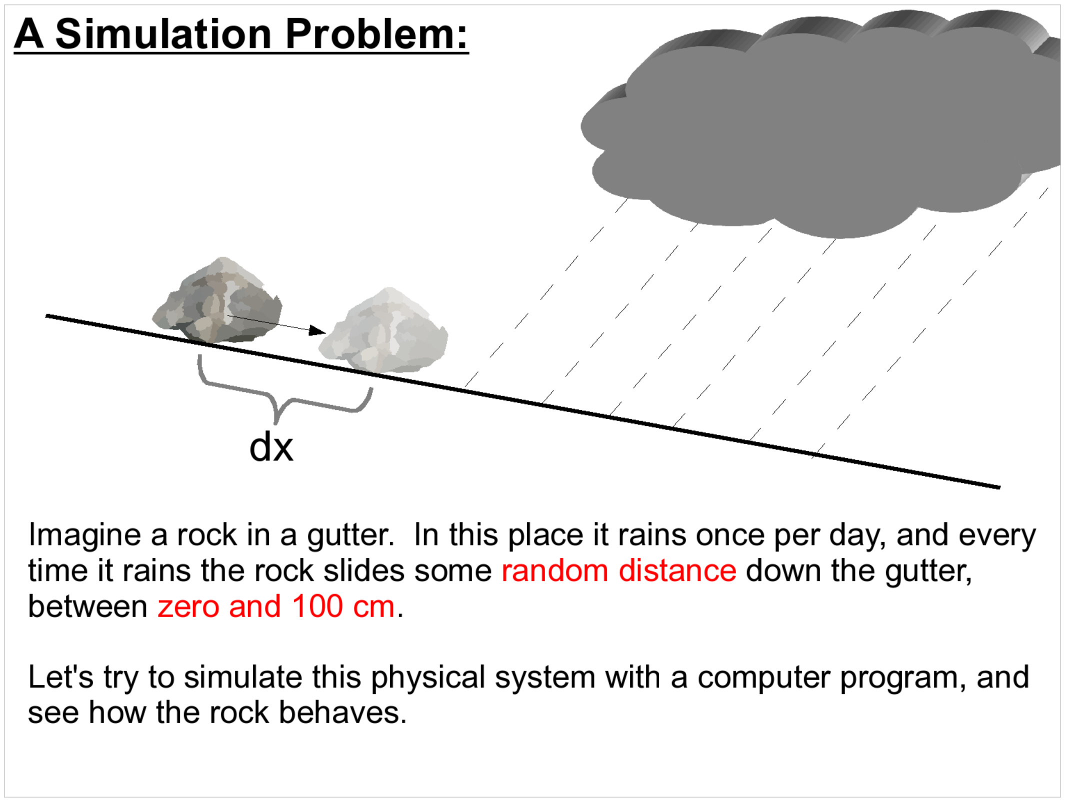
Try these!

An easy way to temporarily remove a line from your program is to convert it into a comment. Just put “//” in front of the “srand” line above, and g++ will ignore it just like any other comments. Then, when you want to put the line back, remove the “//”. Programmers often “comment out” lines while they're experimenting with a program.

Without “srand” you'll find that the program always generates the same set of numbers. This is because computers can usually only generate “pseudo-random” numbers. Computers use math, and math is deterministic. The numbers “rand” generates look random to us, but they're really determined by a starting value that rand uses to begin generating values. If we don't tell the program otherwise, this starting number (called a “seed”) is always set to 1, so we always get the same set of numbers.

“srand” can be used to choose a different seed. In this case, we're telling rand to use the current time (in seconds since Jan 1, 1970) as the seed. Thus, every time we run the program we'll get a different set of numbers (as long as we wait at least 1 second between tries).

Finally, if you temporarily take the “(double)” out, you'll find that the program just generates a bunch of zeros. This is because the value returned by rand() is an integer, and RAND_MAX is an integer, so the compiler assumes that we want an integer answer if one by the other. The result gets truncated to zero.



A Simulation Problem:

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 down the gutter, between zero and 100 cm.

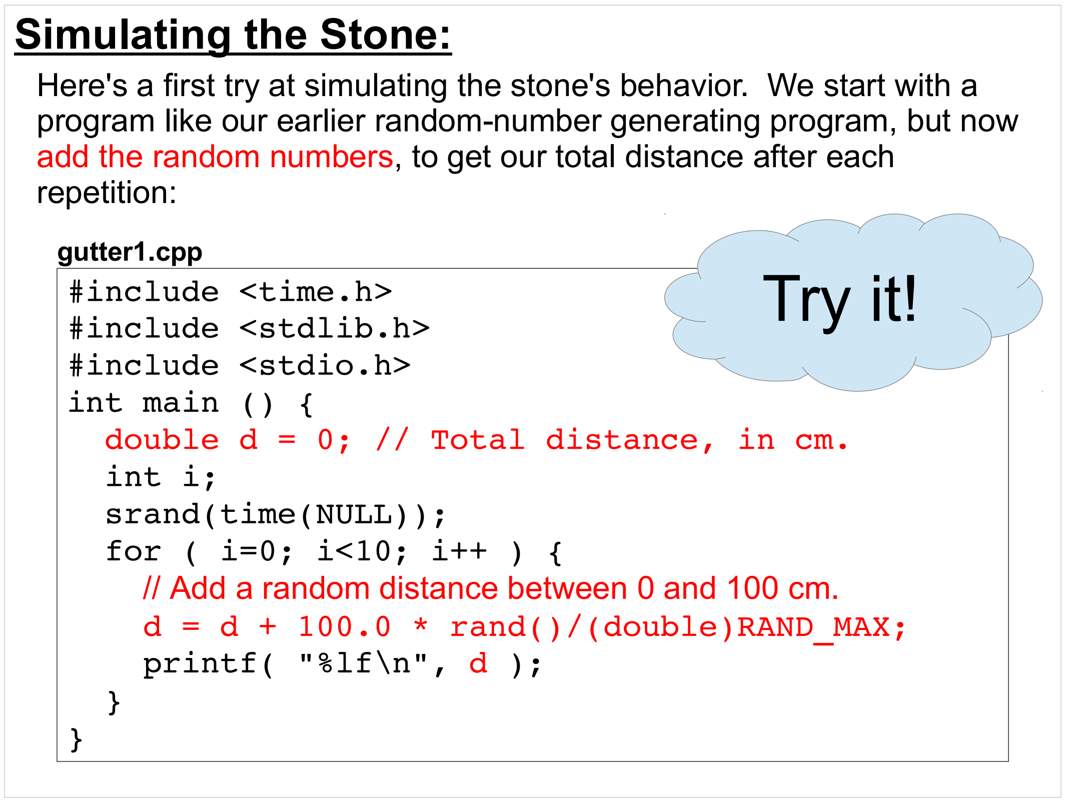
Let's try to simulate this physical system with a computer program, and see how the rock behaves.

Computers are very important for simulating complex physical systems that can't be understood analytically.

Think about how NOAA predicts the weather. It would be impossible to “calculate” tomorrow's weather, because of all of the complex interactions between air, water, sunlight and a million other things. Instead, NOAA can write programs to simulate the behavior of small volumes of air, and simulate the relatively simple interactions between these small volumes.

Let's try to simulate the very simple physical process described in the slide above.

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



Simulating the Stone:

Here's a first try at simulating the stone's behavior. We start with a program like our earlier random-number generating program, but now add the random numbers, to get our total distance after each repetition:

Try it!

gutter1.cpp

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double d = 0; // Total distance, in cm.
  int i;
  srand(time(NULL));
  for ( i=0; i<10; i++ ) {
    // Add a random distance between 0 and 100 cm.
    d = d + 100.0 * rand()/(double)RAND_MAX;
    printf( "%lf\n", d );
  }
}
```

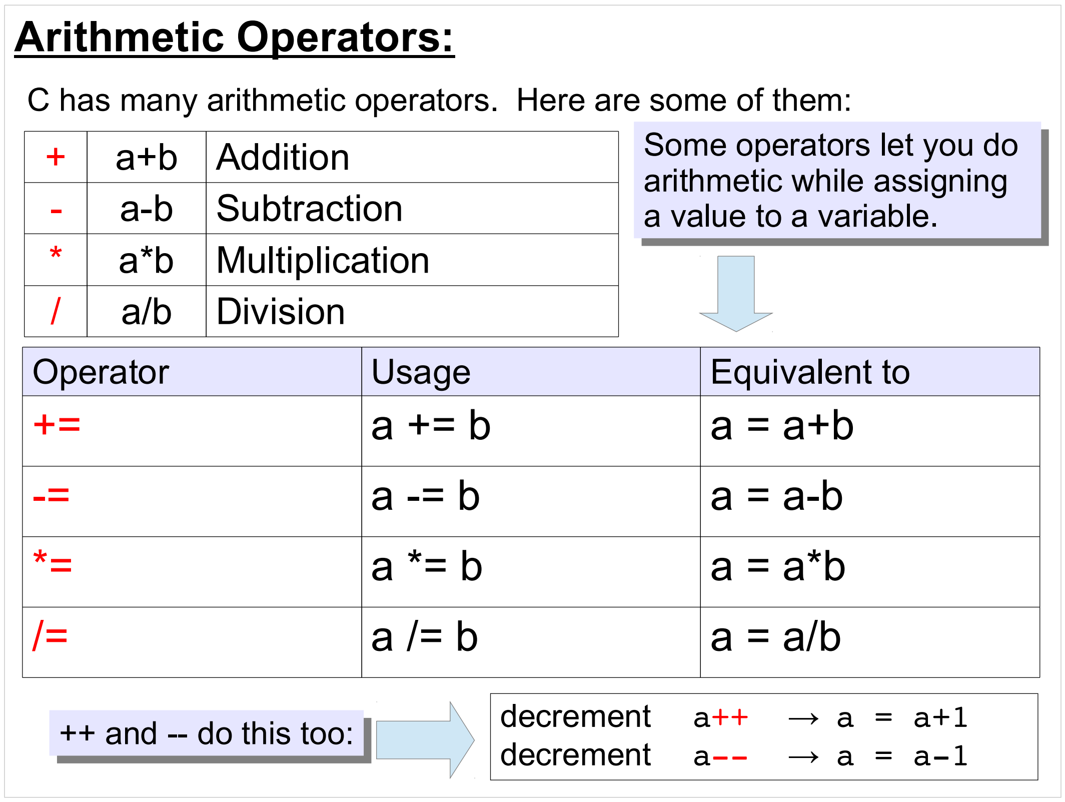
This is very similar to the second Practice Problem from last week. I've highlighted the differences in red. You might think about copying your earlier program, and using that as a starting point for writing this new program.

The program defines a variable, d, to hold the total distance the stone has travelled so far. Then the program loops through ten days, moving the stone by a random distance between 0 and 100 cm each day. Each time the stone moves, the program prints out its new position. When you run the program, you should see a list of increasing numbers as the stone moves along the gutter.

For now, treat this long expression:

rand()/(double)RAND_MAX

as magic, and don't worry too much about how it works. Just remember that it generates a number between zero and one.



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:

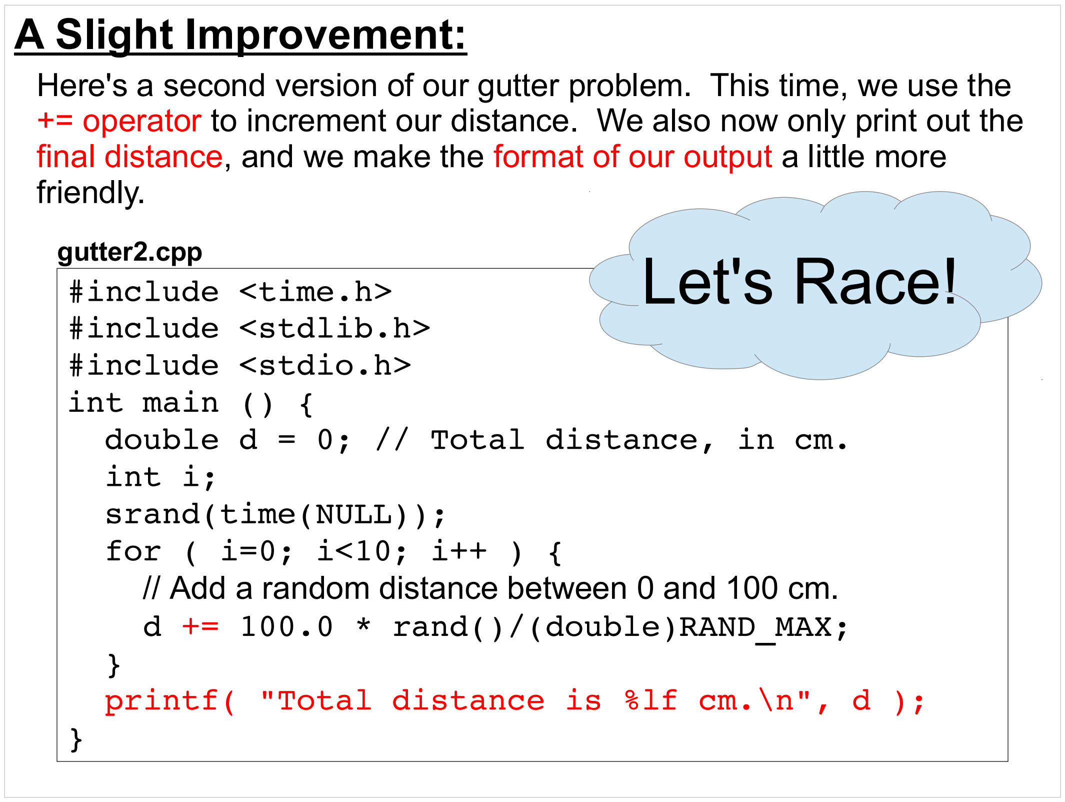
```
decrement   a++   → a = a+1
decrement   a--   → a = a-1
```

If we say:

d += 100;

we mean "increment the value of d by 100". This is similar to the "++" operator we've used before, in "for" loops. The difference is that "++" increments the value by 1.

We can use "+=" to make our previous program a little nicer.



A Slight Improvement:

Here's a second version of our gutter problem. This time, we use the += operator to increment our distance. We also now only print out the final distance, and we make the format of our output a little more friendly.

Let's Race!

gutter2.cpp

```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double d = 0; // Total distance, in cm.
  int i;
  srand(time(NULL));
  for ( i=0; i<10; i++ ) {
    // Add a random distance between 0 and 100 cm.
    d += 100.0 * rand()/(double)RAND_MAX;
  }
  printf( "Total distance is %lf cm.\n", d );
}
```

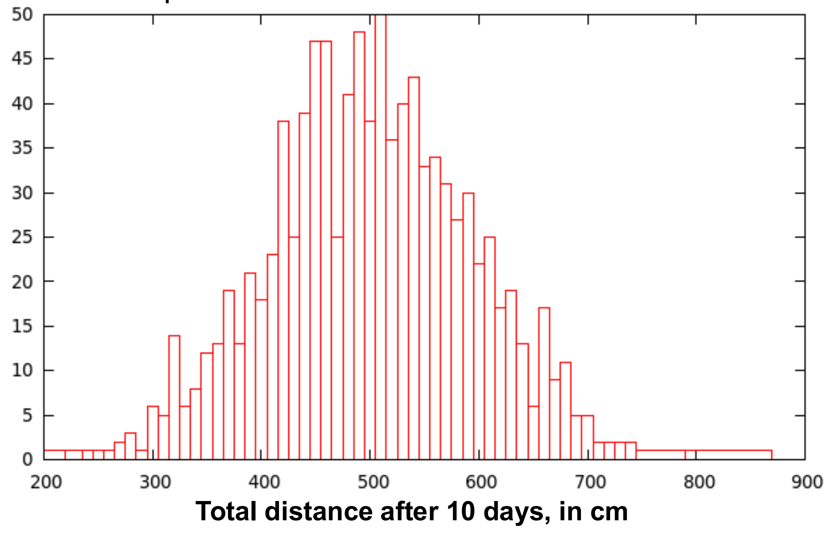
Notice that I've moved the printf statement outside the “for” loop. The new program only prints out the final value of “d”.

The printf statement also give the user a little more information, by telling him/her what this number means.

Finally, notice that I've used the “+=” operator here to increment the value of d. This can help catch typos in a large program, since we're only typing the name of the variable once. In a large program, where we might have a variable named “d” and another variable named (say) “e”, we could run into trouble if we accidentally typed “d = e+1” instead of “d=d+1”.

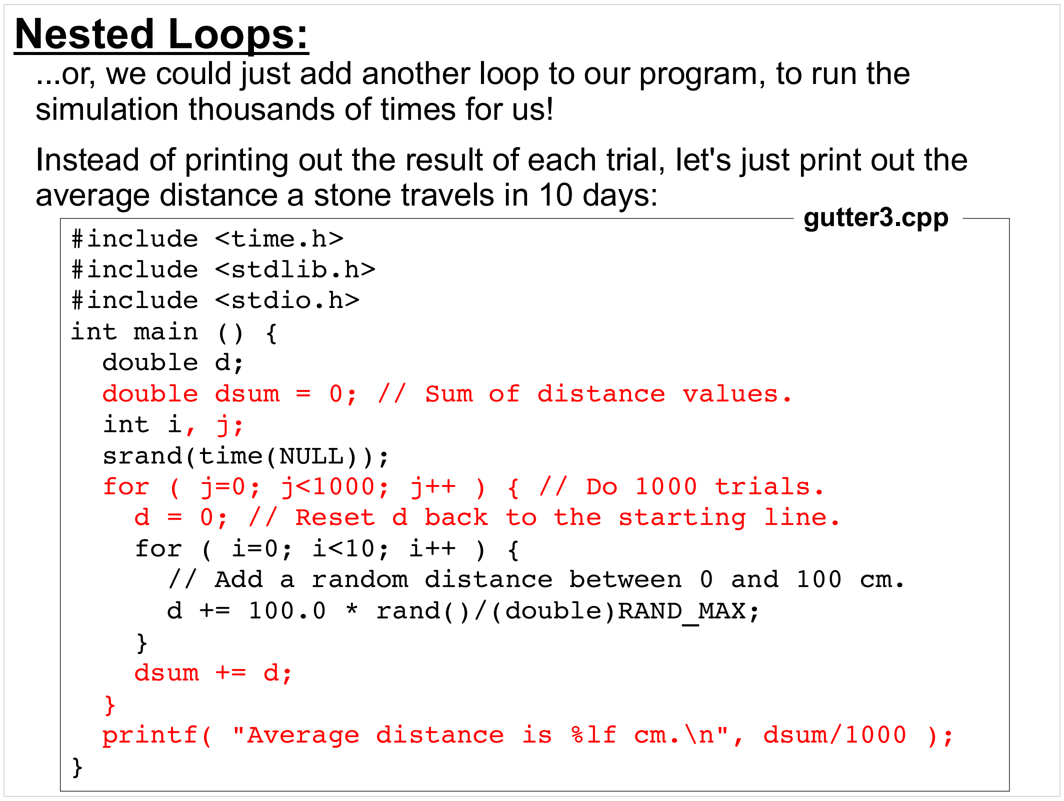
Lots of Rocks:

If we ran our previous program lots of times, we'd see that the results were spread out in a distribution like this:



This might represent the way 1,000 rocks would be distributed if we let them wash down our gutter. We could run our program 1,000 of times to simulate this.

You can imagine that it might actually be useful to know how these rocks would be distributed. Can you see that we're starting to get some rather complicated, useful information out of our simple simulation?



Nested Loops:

...or, we could just add another loop to our program, to run the simulation thousands of times for us!

Instead of printing out the result of each trial, let's just print out the average distance a stone travels in 10 days:

gutter3.cpp

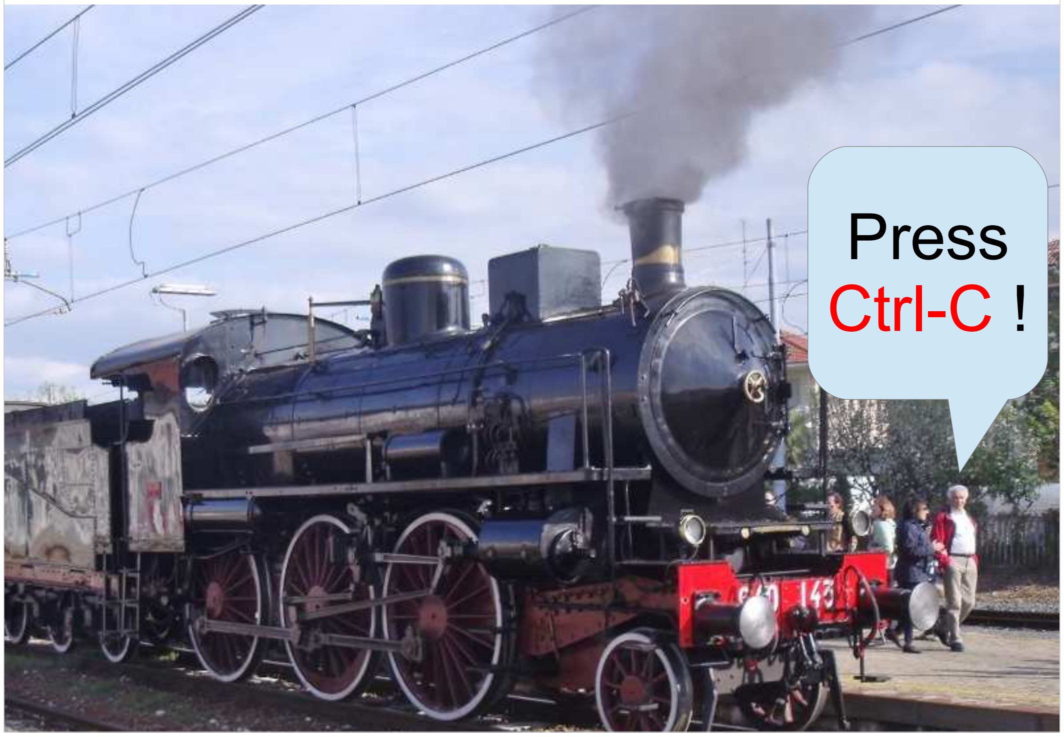
```
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double d;
  double dsum = 0; // Sum of distance values.
  int i, j;
  srand(time(NULL));
  for ( j=0; j<1000; j++ ) { // Do 1000 trials.
    d = 0; // Reset d back to the starting line.
    for ( i=0; i<10; i++ ) {
      // Add a random distance between 0 and 100 cm.
      d += 100.0 * rand()/(double)RAND_MAX;
    }
    dsum += d;
  }
  printf( "Average distance is %lf cm.\n", dsum/1000 );
}
```

Now we have a loop inside another loop. The inner loop is the same one we had in the previous program. The outer loop is new.

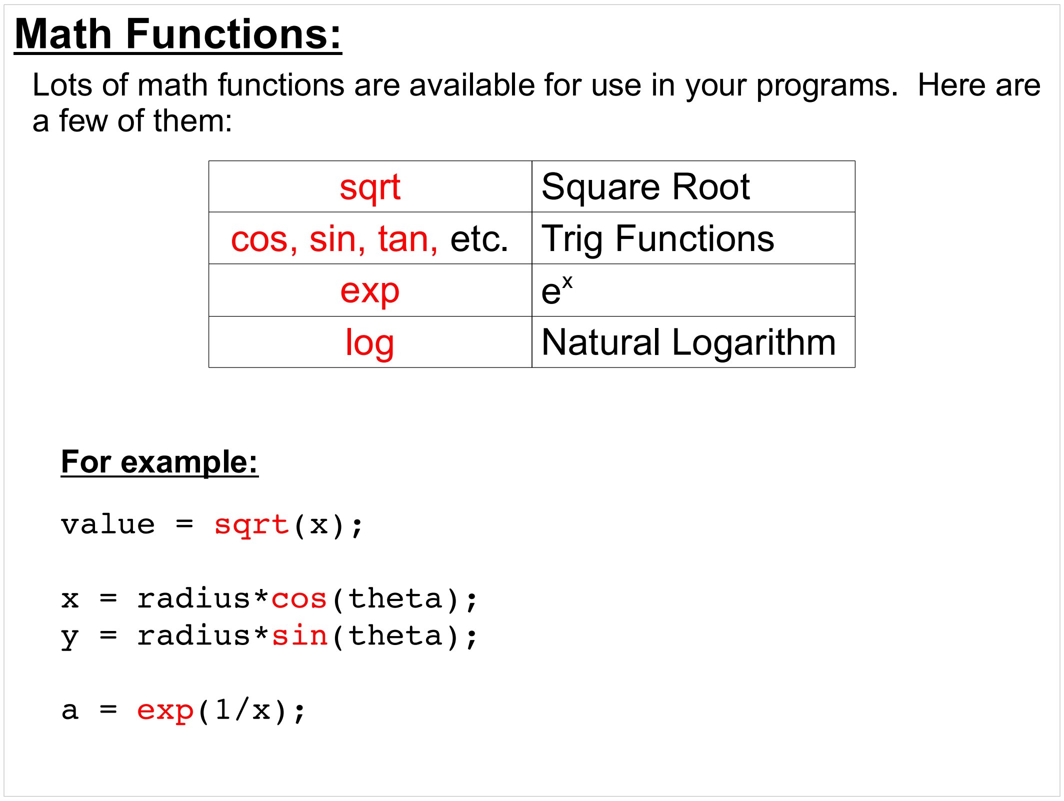
The outer loop runs our rock simulation 1000 times, and sums up the final distances for each of the trials. Once all of the trials are done, the program prints out the average final distance (by just dividing the total by 1000).

If you run the program several times, you'll see that you get slightly different values for the average, varying by maybe +/- 10. If you increase the number of trials to 1,000,000 (remembering to also divide by 1,000,000 in the printf statement), you'll find that the average is much more stable.

Stopping Runaway Loops:



When you have loops inside loops, it's very easy to make a mistake that can cause your program to run away, printing out lots of stuff. If you don't want to wait for it to stop (and maybe it never will!), you can tell the program to stop running by pressing Ctrl-C.



Math Functions:

Lots of math functions are available for use in your programs. Here are a few of them:

sqrt	Square Root
cos, sin, tan, etc.	Trig Functions
exp	e^x
log	Natural Logarithm

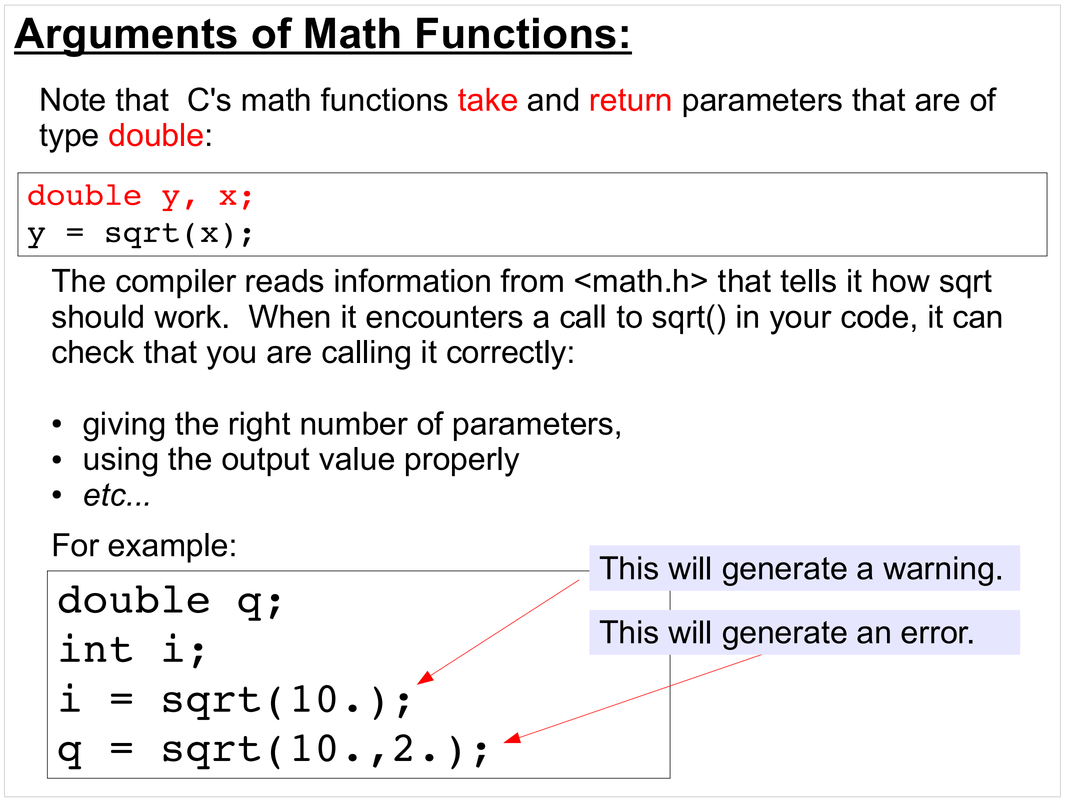
For example:

```
value = sqrt(x);

x = radius*cos(theta);
y = radius*sin(theta);

a = exp(1/x);
```

We can make use of some of these math functions in our simuation program.



Arguments of Math Functions:

Note that C's math functions take and return parameters that are of type double:

```
double y, x;
y = sqrt(x);
```

The compiler reads information from <math.h> that tells it how sqrt should work. When it encounters a call to sqrt() in your code, it can check that you are calling it correctly:

- giving the right number of parameters,
- using the output value properly
- *etc...*

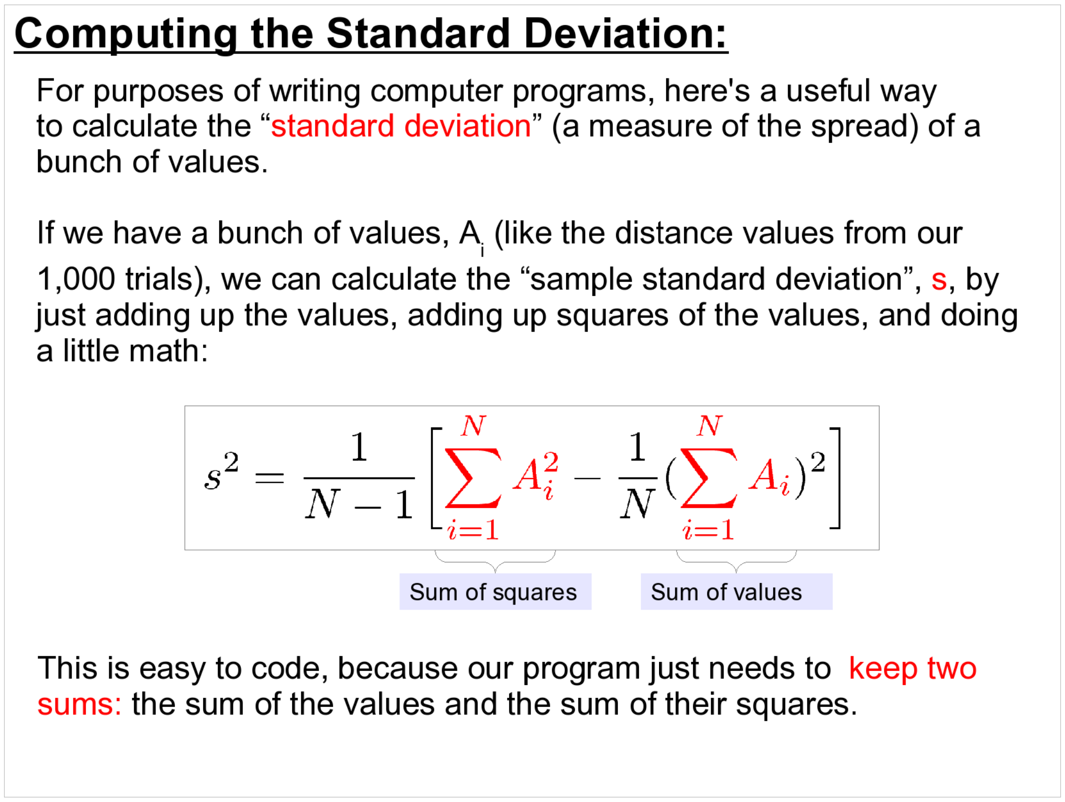
For example:

```
double q;
int i;
i = sqrt(10.);
q = sqrt(10.,2.);
```

This will generate a warning.

This will generate an error.

This is one reason we usually use “double” for floating-point numbers in this class.



Computing the Standard Deviation:

For purposes of writing computer programs, here's a useful way to calculate the "standard deviation" (a measure of the spread) of a bunch of values.

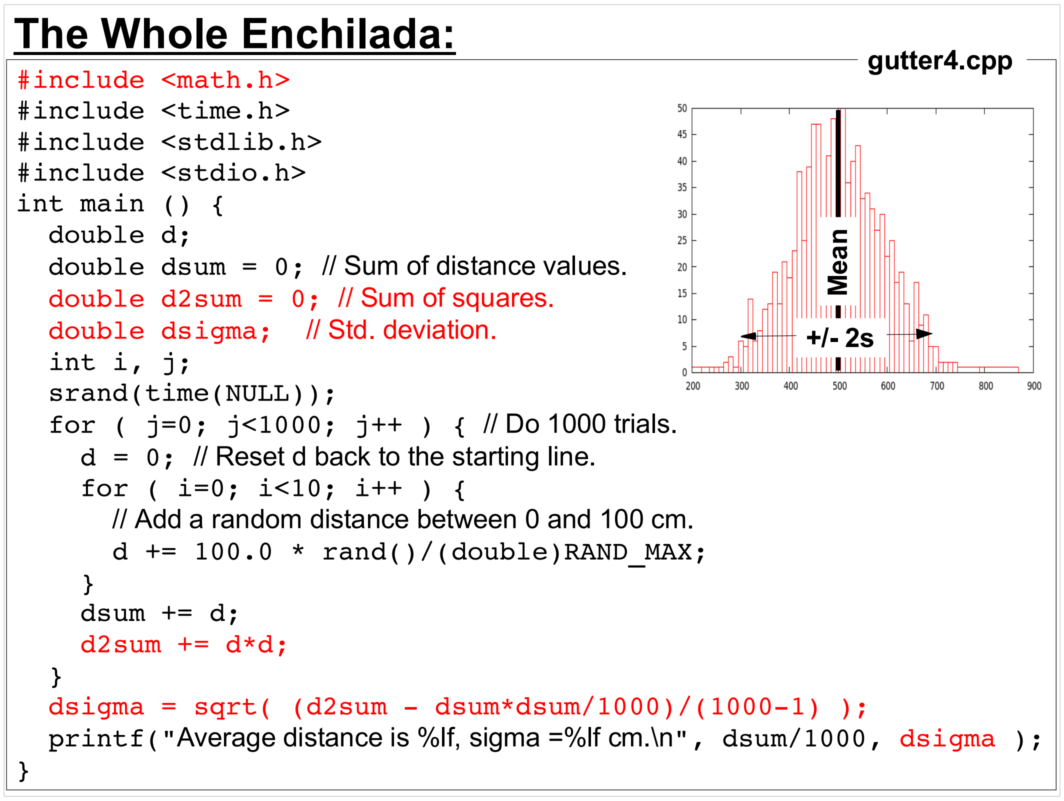
If we have a bunch of values, A_i (like the distance values from our 1,000 trials), we can calculate the "sample standard deviation", s, by just adding up the values, adding up squares of the values, and doing a little math:

$$s^2 = \frac{1}{N-1}\left[\underbrace{\sum_{i=1}^{N} A_i^2}_{\text{Sum of squares}} - \frac{1}{N}(\underbrace{\sum_{i=1}^{N} A_i}_{\text{Sum of values}})^2\right]$$

This is easy to code, because our program just needs to keep two sums: the sum of the values and the sum of their squares.

Our previous program already keeps track of the sum of the "d" values (since we use it to calculate the average). If we start summing up the squares of the "d" values too, we can combine these two sums as shown above to tell us something about the width of our distribution of values.

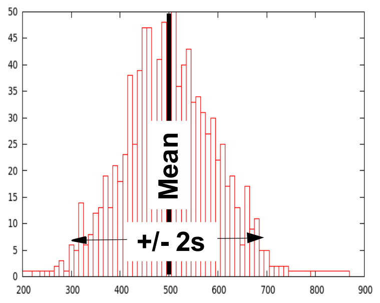
This technique is very useful, and it's something you'll use over and over again if you go on to write more complicated programs.



The Whole Enchilada:

gutter4.cpp

```
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
int main () {
  double d;
  double dsum = 0; // Sum of distance values.
  double d2sum = 0; // Sum of squares.
  double dsigma;   // Std. deviation.
  int i, j;
  srand(time(NULL));
  for ( j=0; j<1000; j++ ) { // Do 1000 trials.
    d = 0; // Reset d back to the starting line.
    for ( i=0; i<10; i++ ) {
      // Add a random distance between 0 and 100 cm.
      d += 100.0 * rand()/(double)RAND_MAX;
    }
    dsum += d;
    d2sum += d*d;
  }
  dsigma = sqrt( (d2sum - dsum*dsum/1000)/(1000-1) );
  printf("Average distance is %lf, sigma =%lf cm.\n", dsum/1000, dsigma );
}
```



So here's the final version of our simulation program. It calculates the average distance travelled in 10 days by 1000 stones, and tells us about how far they'd spread out.

If the numbers were distributed in a “bell curve” (aka a “Gaussian distribution”), statistics tells us that we should expect 95% of the stones to lie within 2*s centimeters of the average. If you run the program above, you'll find that s is about 90 cm.

Practice Problem:

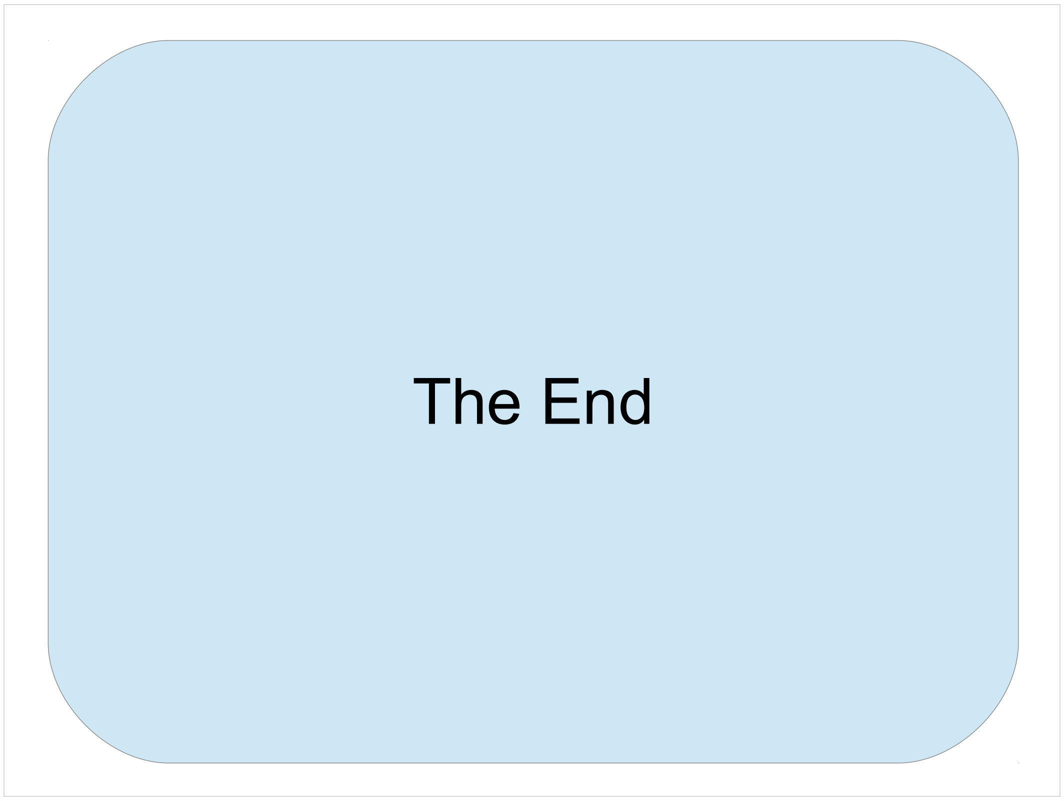
Our gutter programs have a lot of numbers written into them: 10 days, 100 cm, 1000 trials. If we want to change to, say, 10,000 trials, we need to find all of the places in the program where we currently assume a value of 1000, and change them.

It would be better if these numbers were more easily changed. Can you rewrite the program 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;
```

Here's something else to work on. We'll look at a solution next time.



The End

Thanks!