Introduction to Computational Physics

Meeting 4: Files and Whiles

Today:
- Reading and writing files
- Using “while” 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 adding a couple of new C functions: “fprintf”, which is like “printf”, but writes its output into a file, and “fscanf”, which is like “scanf”, but reads its input from a file. We'll also look at a new kind of loop, the “while” loop.

Let's get started!

Last week's zombie-simulation exercise used features of the C language that you'd seen before, but it introduced some new ways to use them.

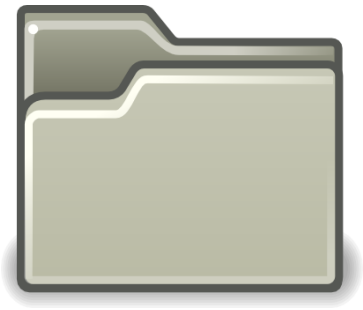
In particular, most of these examples made use of the fact that we can simulate the progression of a system if we know the probabilities that the system's components will behave in certain ways. (For example, the probability that a non-zombified person will contract zombism when he/she encounters a zombie.)

It's often difficult or impossible to write down equations that describe the time-evolution of a complex system (the weather, for example). But, if we know a few simple rules about how the system's components behave and interact with each other locally, we can often write a program that approximates the system's future behavior.

Files and Directories:

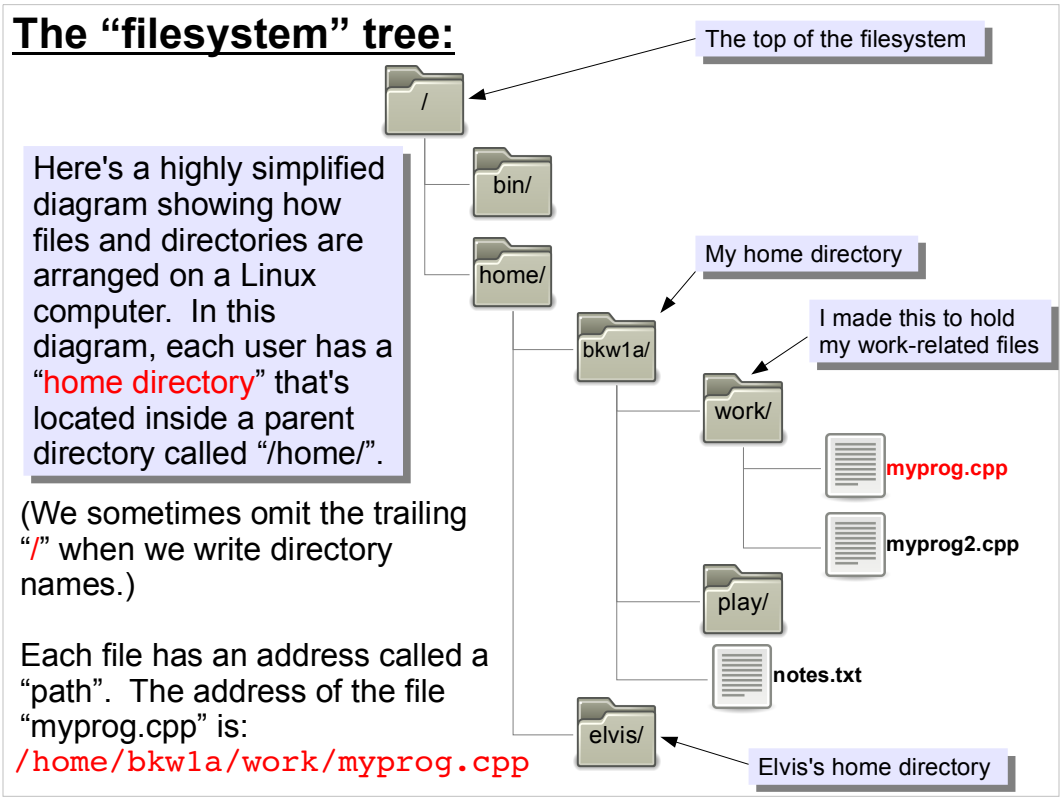
"Document" or "File":

"Folder" or "Directory":



We'll spend a while today talking about files and directories, and then go on to talk about how to use those in C programs.

First of all, a word on nomenclature: In the Linux world, we usually talk about "files" and "directories". If you're coming from the Windows or Mac world, you may be more familiar with the terms "documents" and "folders". Don't let the terms confuse you. A "file" is just a "document" by another name, and a "directory" is just a "folder".



Although in most situations you can omit the trailing “/” in directory names, it's always good to put it in since that can help avoid some common errors.

Listing Files in a Directory:

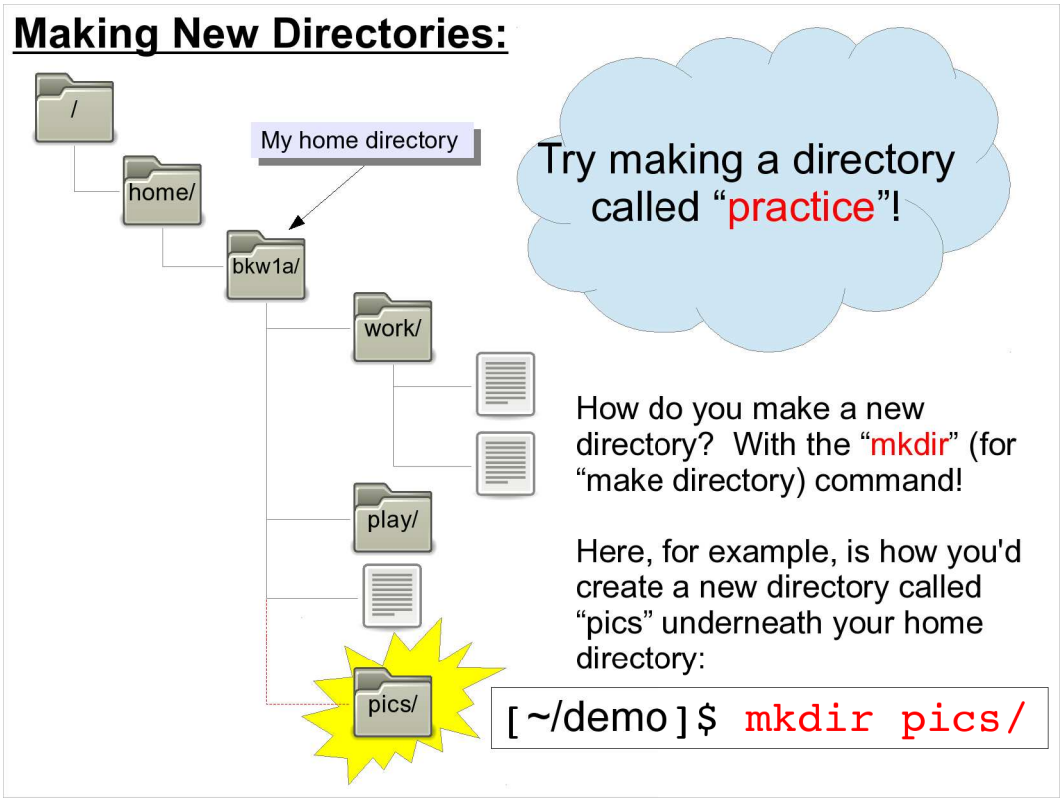
As we've seen before, you can use the "ls" command to list the files in a directory. If you want even more information about the files, you can type "ls -lp":


```
[~/demo]$  ls -lp
total 72
lrwxrwxrwx 1 bkw1a bkw1a    11 Mar 26  2010 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18  2009 cluster.pdf
-rw-r----- 1 bkw1a bkw1a   983 Jan 18  2009 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18  2009 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18  2009 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18  2009 phase1/
drwxr-x--- 2 bkw1a bkw1a  4096 Jul 16  2009 phase2/
drwxrwxr-x 2 bkw1a bkw1a  4096 Jan 26  2012 phase3/
-rw-r----- 1 bkw1a bkw1a    72 Jan 18  2009 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18  2009 ReadMe.txt
drwxrwsr-x 2 bkw1a bkw1a  4096 Jul 16  2009 shared/
drwxr-xr-x 2 bkw1a bkw1a  4096 Apr 14  2009 tmp/
```

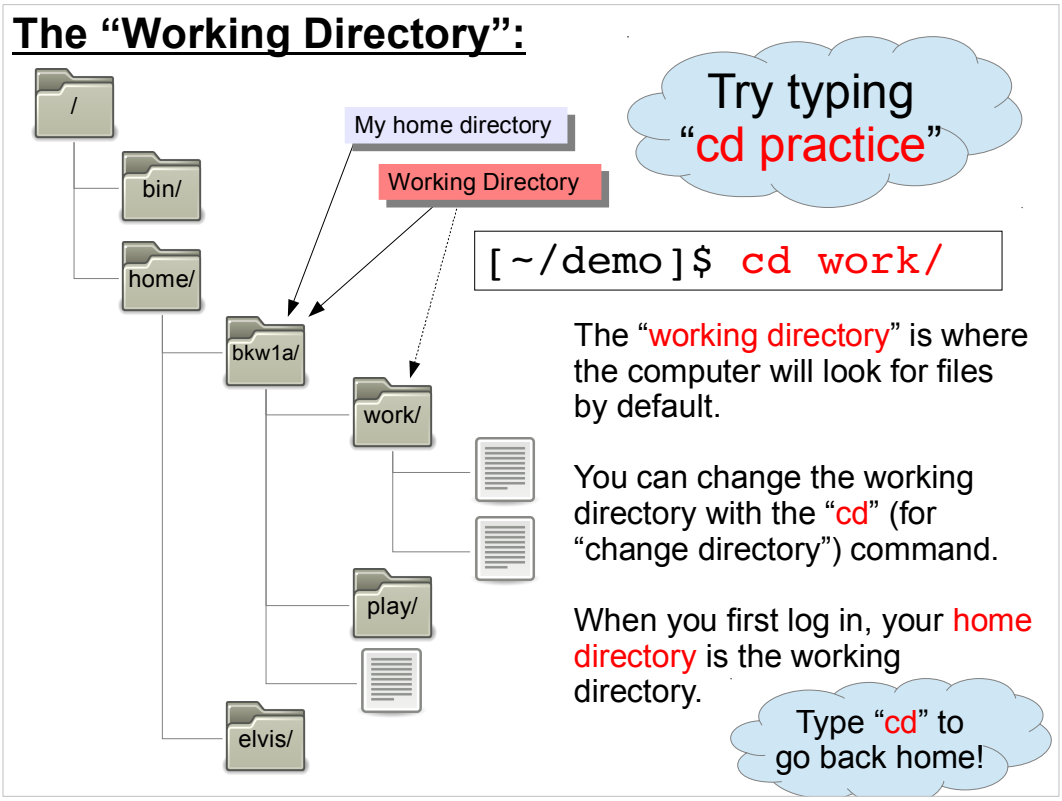
This tells you who owns the file, how big it is and who's allowed to read or write the file, among other things. Notice that this directory contains several sub-directories. When you type "ls -lp" these will show up with a "/" at the end of their names, although "ls" by itself will omit this.

The output of "ls -lp" also shows you the time and date at which the file was last modified.

The first letter of the first column tells you what kind of thing this is: "d" for directory, "-" for file, and some other letters for more esoteric things.



You can use this to create sub-directories underneath your home directory, into which you can sort your files.



When you type a command like "nano hello.cpp" the computer looks for (or creates) the file "hello.cpp" in the current working directory.

The command "pwd" will show you what the current working directory is.

Typing "cd" by itself will just take you back to your home directory.

More on the “Working Directory”:

You can see what directory you're currently working in by using the “pwd” command:

```
[~/demo]$ pwd
/home/bryan/work
```

Note that the path to a file or directory is given as a list of parent directories, separated by slashes, starting with the root directory (“/”). In this case, the current working directory is “/home/bryan/demo”.

You can change your current directory by using the “cd” command, like:

```
[~/demo]$ cd work/
```

Or, equivalently:

```
[~/demo]$ cd /home/bryan/work/
```

In the first case, we specify the name of a directory relative to the current directory, and in the second case we explicitly give the full path name (the complete name of the directory we're interested in.)

The “Home Directory”:

Each user has a “home directory”. This directory will be your current directory right after you log in.

```
[~/demo]$ echo $HOME
/home/bryan
```

You can use the $HOME environment variable in commands, to refer to your home directory.

```
[~/demo]$ ls $HOME/demo
```

You can also refer to your home directory as “~”, in most shells.

```
[~/demo]$ ls ~/demo
```

$HOME is a shell variable, or “environment variable”. These are similar to the variables we'll use in C programs. In fact, you can write programs in the shell language, too. These are usually called “scripts” or “shell scripts”. They can be used to automate shell tasks you do often.

Sending a Progam's Output to a File:

Let's look back at yoiur “Hello World” program. When you ran it, it wrote some output on your screen:

```
[~/demo]$   ./hello
Hello World!
```

Try this with one of your programs. Use “nano” to look at the file you create!

When you run your program, you can tell the computer to send the output into a file instead of to the screen. To do so, use the “>” symbol like this:

```
[~/demo]$   ./hello > hello.txt
```

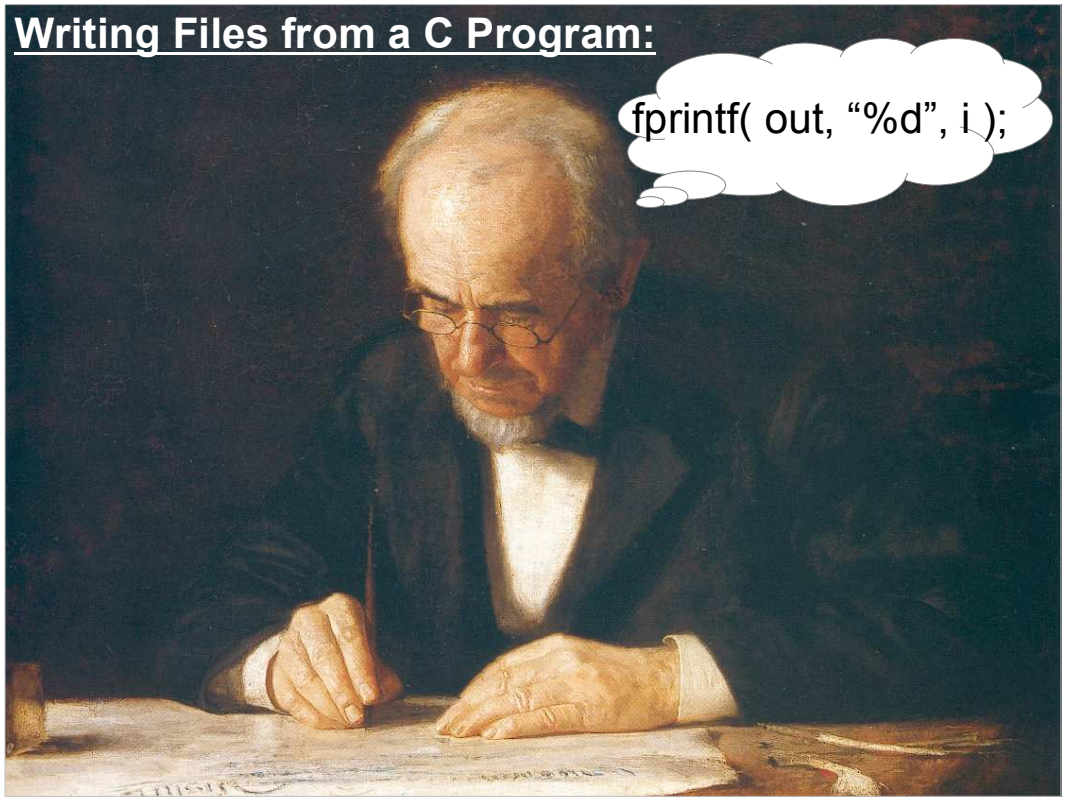
In the example above, if hello.txt already existed it would be overwritten, so be careful!

You can also ask to append things onto the end of an existing file, like this:

```
[~/demo]$   ./hello >> hello.txt
```

Yay! We can write things to a file.

So, we're all done with that, right?.....



If we can use “>” to redirect a program's output into a file, why would we want to make our C programs write files in any other way? There are at least a couple of reasons:

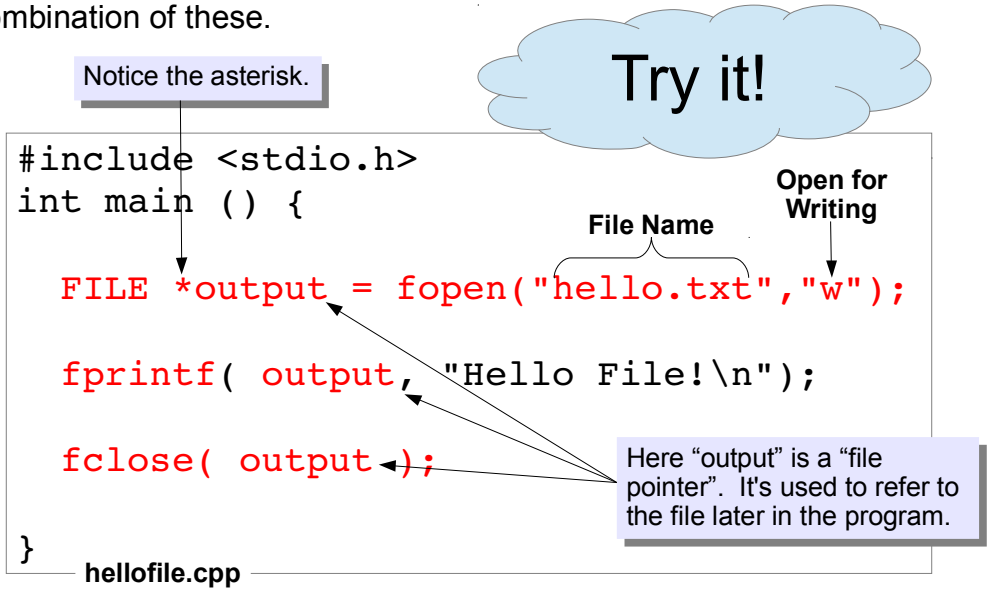
* Sometimes we want to send some output to the screen and some to a file. Think about a program that asks the user for some input, and then writes out some data. Text that says “Please enter your age” should go to the screen, but we might want the rest of what the program writes to go into a file.

* Sometimes a program needs to write more than one file. Think about a program that sorts data into several categories, and writes each category to a different file.

Writing Files Directly from Your Program:

C gives us the functions “fopen” and “fclose” for opening and closing files, and the function “fprintf” (like “printf”) for writing into files.

Files can be opened for reading, writing, appending, or some combination of these.



The variable “output” can be called a “file pointer”, a “file descriptor” or a “file handle”. You may see it referred to by any of these terms.

Notice the “*” in front of “output” in the fopen statement. We'll talk about what this means when we look at pointers.

You don't have to use “output” as the file pointer variable. You can call it anything you want to.

In the fopen statement, we can say that we want to read (“r”), write (“w”) or append (“a”) to the file. There are other options, too.

Another Writing Example:

Here's a slightly less trivial example:

writer.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main () {
  int i;

  FILE *output = fopen("writerdata.txt","w");

  for ( i=0; i<1000; i++ ) {
    fprintf ( output, "%d ", i );
    fprintf ( output, "%d ", i*i );
    fprintf ( output, "%lf ", sin( 2.0 * M_PI * i/1000.0 ) );
    fprintf ( output, "%lf ", cos( 2.0 * M_PI * i/1000.0 ) );
    fprintf ( output, "%lf ", 250000.0 - (i-500)*(i-500) );
    fprintf ( output, "\n" );
  }

  fclose( output );

}
```

This program writes 1,000 lines of the form:

x, y1, y2, y3, y4

where the “y” values are various functions of x.

End the line here.

Try it!

What happens when you run this program? You won't see anything if the program is working correctly! That's because everything the program writes is going into the file “writerdata.txt”.

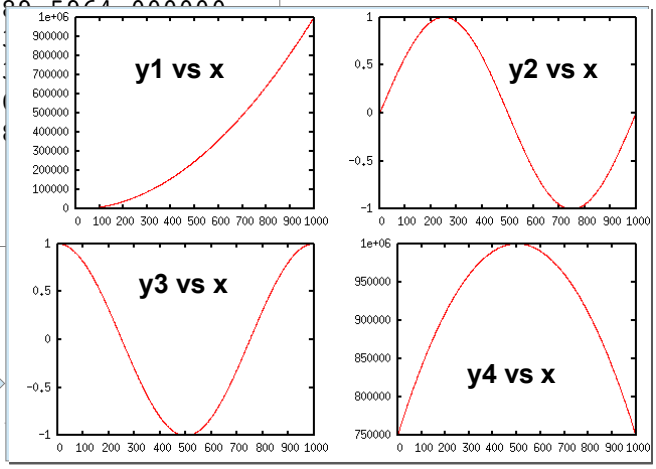
You can type “nano writerdata.txt” to look at the output file.

The Output File:

writerdata.txt

```
x  y1    y2          y3          y4
0  0  0.000000  1.000000  0.000000
1  1  0.006283  0.999980  999.000000
2  4  0.012566  0.999921  1996.000000
3  9  0.018848  0.999822  2991.000000
4  16  0.025130  0.999684  3984.000000
5  25  0.031411  0.999507  4975.000000
6  36  0.037690  0.999
7  49  0.043968  0.9990
8  64  0.050244  0.9987
9  81  0.056519  0.9984
10  100  0.062791  0.99
.
.
.
```

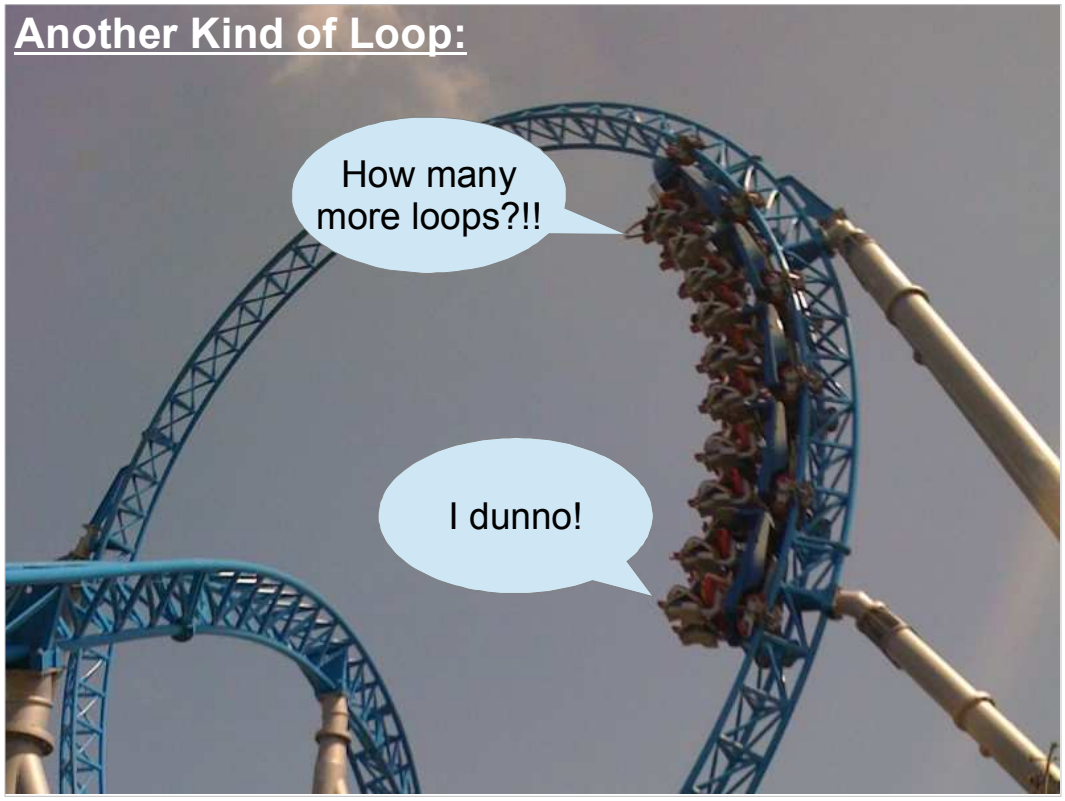
Here's what the output would look like if we graphed it:



On the left is what you should see if you type “nano writerdata.txt”.

As you can see, we've written a file with five columns of numbers.

Another Kind of Loop:



To write a “for” loop, we need to know how many times we're going to go around. We can say “do ten loops” or “do 100 loops”.

But sometimes we'd like to say “loop until you're done”, because we don't know how many times we'll need to do something.

Think about a program that reads a file line by line, for example. We generally won't know how many lines are in the file. We just want to keep reading until we get to the end.

Fortunately, C provides an easy way to do this kind of thing.

Another Kind of Loop:

With a “for” loop, you need to know the number of loops in advance. C also gives us a “while” loop, which just keeps going until some condition is met. Here's an example:

addem.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main () {
  int sum = 0;

  srand(time(NULL));

  while ( sum < 21 ) {
    sum += 13.0*rand()/(double)RAND_MAX;
    printf ("Sum is now %d\n", sum );
  }

}
```

Why do we bother?

Loop until sum is no longer less than 21.

Try it!

This program does something like the game “blackjack”, which deals out cards to players who try to get a sum as near to 21 as possible without exceeding it.

A “while” loop is what's called a “conditional loop”. It keeps going until some condition is met. In contrast, a “for” loop is called a “counted loop”, since we tell it in advance how many times it should go around.

An Infinite Loop!

What if there might be several different conditions for stopping the loop? we can use C's "break" statement to stop the loop at any time. In fact, we can tell the program to just keep looping forever, until we decide to stop it with a "break":

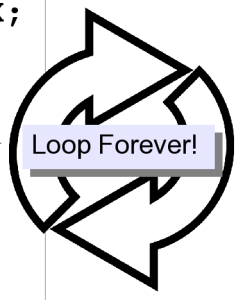
addem2.cpp

```
while ( 1 ) {
  sum += 13.0*rand()/(double)RAND_MAX;
  printf ("Sum is now %d\n", sum );

  if ( sum == 21 ) {
    printf ("You WIN!\n");
    break;

  } else if ( sum > 21 ) {
    printf ("You lose!\n");
    break;

  }
}
```



Be very sure that there's always SOME condition that will make your program stop!

Remember: If you find that your program just isn't stopping because there's a mistake somewhere, you can kill it by typing CTRL-C.

The condition in the parentheses after "while" is just a logical expression that evaluates to either zero (false) or some non-zero value (true), just like a condition associated with an "if" statement.

We can use any of the conditions we used with "if" statements with "while" statements too.

Letting the User Make a Choice:

One more version of this. Here' we let the user decide whether to keep the current score, or try for a better one (possibly overshooting and losing):

addem3.cpp

```
 while ( 1 ) {
  sum += 13.0*rand()/(double)RAND_MAX;
  printf ("Sum is now %d\n", sum );

  if ( sum == 21 ) {
    printf ("You WIN!\n");
    break;

  } else if ( sum > 21 ) {
    printf ("You lose!\n");
    break;

  } else {
    printf ("Enter 1 to continue or 0 to quit while you're ahead: ");
    scanf("%d", &ans);
    if ( ans != 1 ) {
      printf("Your final score was %d\n",sum);
      break;
    }

  }
}
```

This is more like the card game “blackjack”, where players try to get as close to 21 as possible without going over.

Ask the user if he/she wants to go on.

This is just an improved versionof the previous program. The only addition is the “else” clause.

OK, so we know how to write files now. How about reading them?

Just as C has an “fprintf” that's analogous to “printf”, there's also an “fscanf” that's like “scanf”, but for files.

Like “fprintf”, “fscanf” takes a file pointer as its first argument.

Reading a File:

readfile.cpp

This program reads the data file we wrote earlier, in the format:
x, y1, y2, y3, y4

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main () {
  int status;
  double x, y1, y2, y3, y4;

  FILE *input = fopen("writerdata.txt","r");
  while ( 1 ) {
    status = fscanf( input,
                     "%lf %lf %lf %lf %lf",
                     &x,&y1,&y2,&y3,&y4 );
    if ( status == EOF ) {
      break;
    }
    printf ("%lf\n",y4);
  }
  fclose( input );
}
```

Open for reading.

fscanf is like scanf, but from a file.

fscanf returns a value that can be used to see if we've reached the end of the file.

Here's a simple program that reads the data file we wrote with one of the previous programs and prints out one of the columns.

Notice that fscanf tells us about its status by returning a value that we can capture in a variable (“status”, in this example). One of the things fscanf might tell is is “Hey! I've reached the end of the file!”. It does this by returning the value “EOF” (for “End Of File”). This is really just a number, but the symbol EOF is defined in the file “stdio.h”, and it's easier to remember than a number. Also, there's no guarantee that different C compilers will return the same number, but they'll all have a stdio.h that defines EOF appropriately for that particular compiler.

Finding the Maximum:

Now let's do something with the data we read. This program finds the maximum value of y4:

readfile2.cpp

```
double max = 0;

FILE *input = fopen("writerdata.txt","r");

while ( 1 ) {
  status = fscanf( input,
                   "%lf %lf %lf %lf %lf",
                   &x,&y1,&y2,&y3,&y4 );
  if ( status == EOF ) {
    break;
  }
  if ( y4 > max ) {
    max = y4;
  }
}
printf ("Maximum value of y4 is %lf\n", max);
fclose( input );
```

Is y4 greater than the previous largest value? If so, set max to y4.

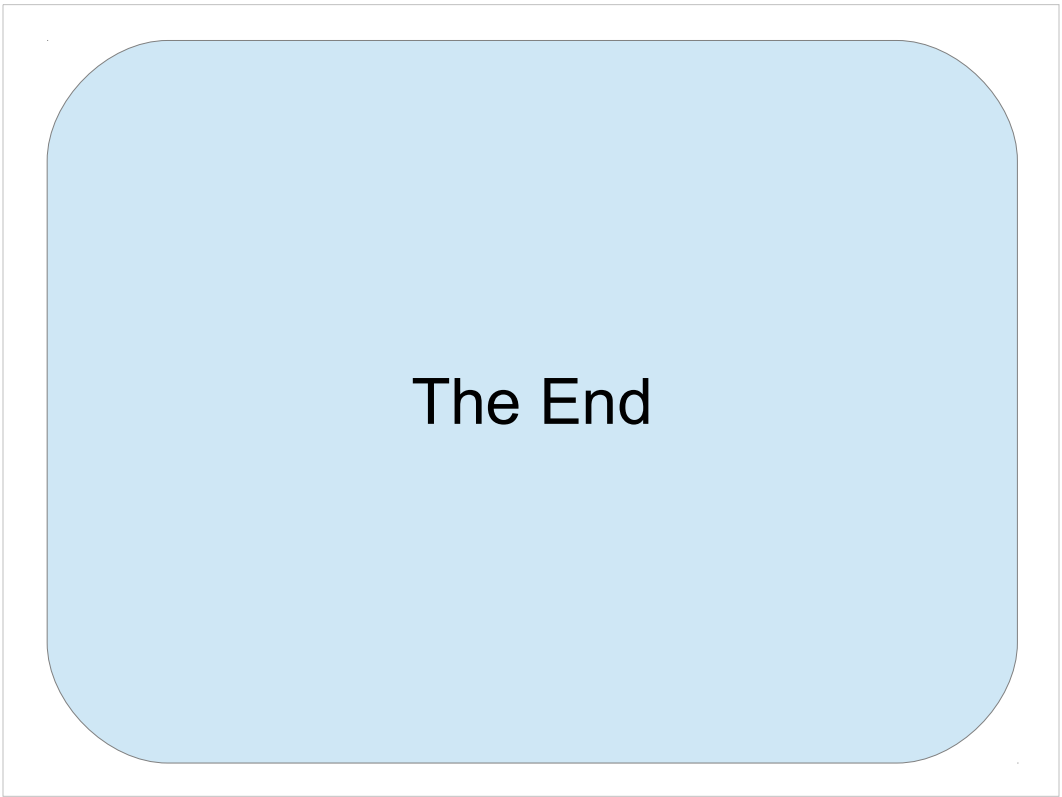
Here's a program that does something more useful. It scans through a file, and prints out the maximum value from one of the columns.

Can you see how it finds the maximum? Each time around the loop, the program compares the current value of y4 with the biggest value it's seen before ("max"). If y4 is bigger than max, it sets max to the new value of y4.

Practice Problem:

How would you modify the “readfile.cpp” program so that it reads data from “writerdata.txt” and then writes two columns of data into a new file? Let's say the two columns are x and ($y2^2 + y3^2$).

Here's something else to work on. The previous examples either wrote a file or read a file. Now we're thinking about doing both reading and writing in the same program. We'll look at solutions next time.



Thanks!