Physics 2660

Lecture 1: Introduction

Today:
• Inside the computer
• The shell
• First steps towards programmming

Welcome again to Physics 2660!

Remember that you can find much information about the class at the web page:

http://faculty.virginia.edu/comp-phys/phys2660

There you'll find:
• The list of textbooks
• Homework, pre-lab and lab assignments
• Reading assignments
• Grading policies
• Lots of programming documentation.

If you've recently joined the class, please let me know and I'll e-mail you some introductory material and get you an account on our computing cluster, Galileo. You'll need this to do the pre-lab assignment that's due before Thursday's lab.

Part 1: Where We're Going

Our goal in this class is to jump-start you from your current level of computing skill to a level where you're able to do some fairly sophisticated computational physics. By the time this semester is over you'll be able to create powerful programs that:

- Chew up data and produce useful results,
- Accurately model complex physical systems,
- Visualize data sets clearly and easily,

and lots of other things.

This is more than just a general programming course. We want to teach you the skills you need to do real science.

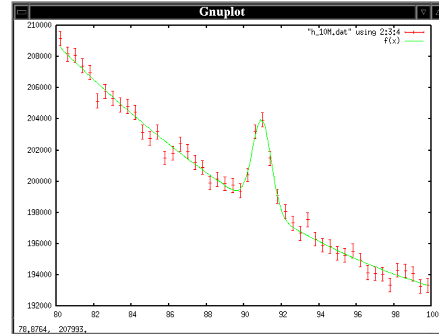
A Reminder About Our Goals:

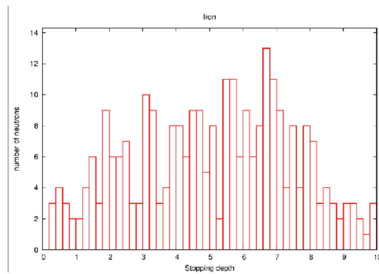
- Introduction to programming in C/C++
- Introduction to Linux
- Converting problems to code
- Good programming style, best practices
- Optimization and Debugging
- Basic data handling, with some
  - Statistical Analysis,
  - Fitting,
  - Visualization tools

Some Examples:

Temperature distribution in a solid, after heat flow:

Parameter Extraction/Fitting models to data, significance of results:



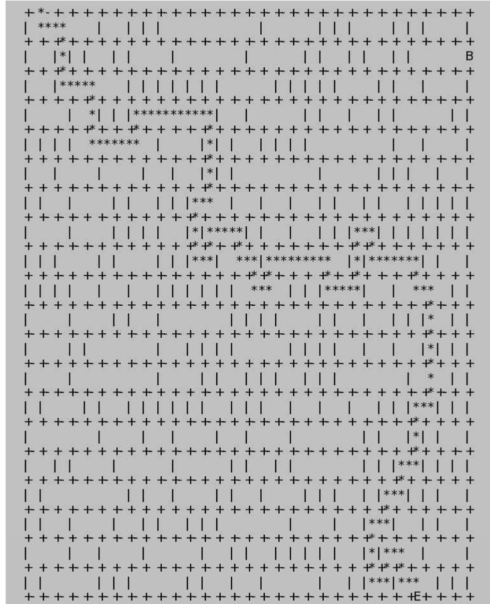


Monte Carlo Simulation of particle shielding: Distribution of stopping depths in material based on random scattering and energy loss in collisions in shield wall.

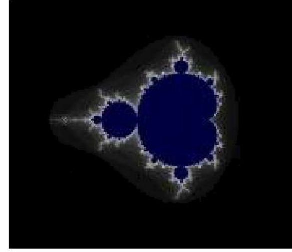
Here are some of the types of problems you'll learn to solve during this course.

More Examples:

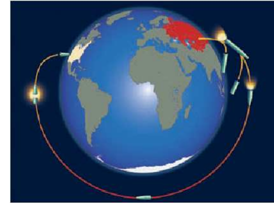
Maze Solving:



Fractals:



Trajectories, intercept simulations:



...and many more.

Part 2: Computer Hardware

IBM PC 5150
(1981)

* Price competitive with Apple's home computers

* Largely non-proprietary architecture could be cloned or re-engineered easily

* Ecosystem of clones created market for software vendors

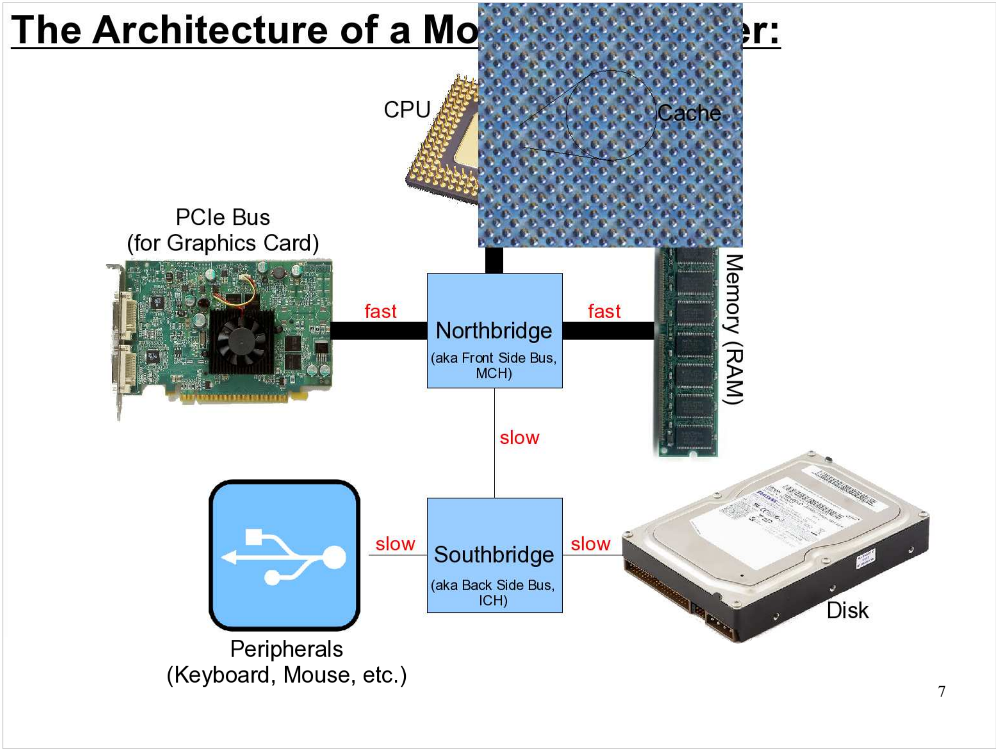
In order to understand computer programming well, you need to first understand a little about computer hardware.

Before 1981 people had “home computers”, but there was no such thing as a “PC”. Then, in 1981, IBM decided that there was money to be made in the burgeoning home computer market. They released their first home computer, called the IBM Personal Computer 5150.

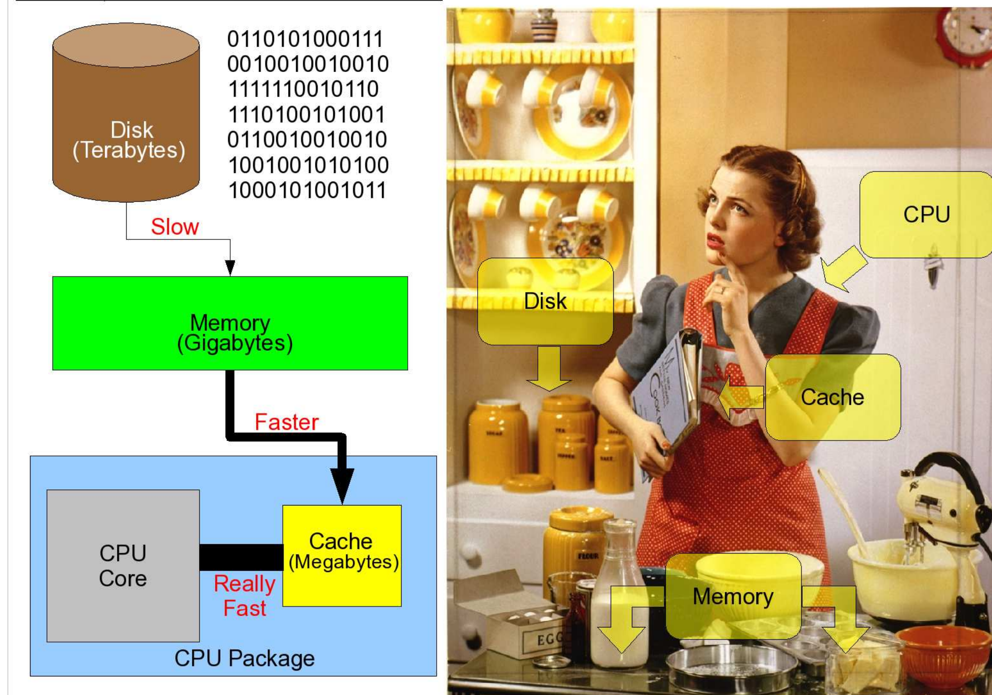
Its price was competitive with the popular Apple home computers, and it was based largely on non-proprietary technology, which made it possible for other companies to clone the PC and sell PC-compatible computers. This created an enormous new market of compatible computers for hardware and software vendors.

Competition drove prices down, and consumers valued the variety of products available for the PC, and valued the IBM name. PCs came to dominate the low-priced computer market.

Today, almost any computer you encounter, from powerful servers to low-end netbooks, will have a similar architecture, inherited from the original IBM PC.



Today's PCs are still architecturally very similar to the original IBM PC. Some components to pay particular attention to are the CPU, which does the computer's thinking, and the Northbridge and Southbridge, which handle fast and slow I/O, respectively. Northbridge and Southbridge are often referred to collectively as the “chipset”. I've shown relative speeds for some of the connections for comparison. Note that disk access is much slower than memory access. We'll come back to this later.



Non-programmers often confuse the terms “disk space”, “memory” (or “RAM”) and “cache”. I like to use a cooking analogy to explain the differences, and describe how computers compensate for the relatively low speed at which data can be read off of a disk.

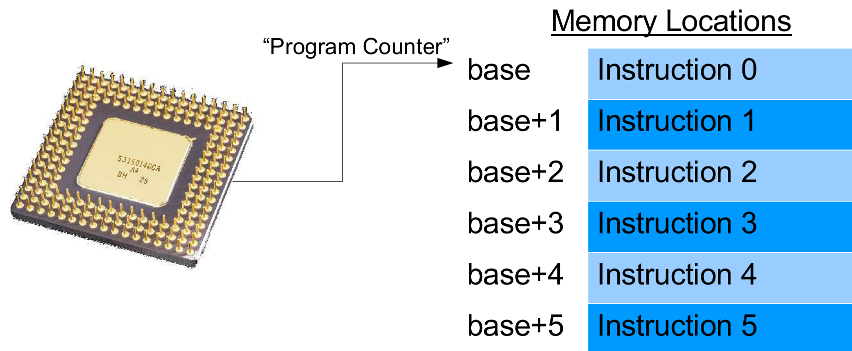
Since disk access is slow, computers try to predict which disk data you'll need next. Then, while the computer is digesting a bit of data you've just read from the disk, in the background it reads the data it thinks you might need next, and stores it in memory. The next time you request data from the disk, the computer first checks to see if the data is already stored in memory, and fetches it from there (much faster) if it is.

There's even a third level of storage, called “cache memory”, that's actually inside the CPU package itself, and can be accessed faster than RAM. Just like reading from the disk, the computer tries to predict which data from RAM you'll need next, and store it in the cache.

Disk: large, slow, non-volatile
RAM: smaller, faster, volatile
Cache: tiny, really fast, volatile
TIP: minimize your i/o to maximize your performance

How a Program Runs:

We usually write programs in human-readable languages like C, but CPUs don't understand C. The programs we write must be translated into a series of low-level instructions that the CPU can understand. This translation is done by a program called a compiler.



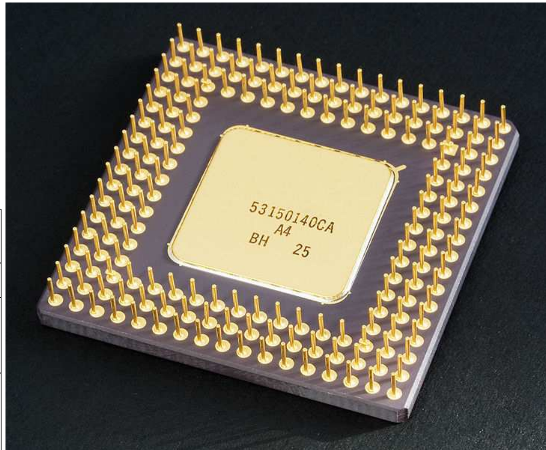
When running a program, the CPU fetches instructions from the computer's memory. A storage location inside the CPU (a “register”) points to the memory location for the next CPU instruction. This register is called the “program counter” (PC). After fetching each instruction, the CPU increments the PC and executes the instruction

Of course, different CPUs use different instruction sets.

Central Processing Units (CPUs):

Most modern computers use processors that are descendants of the Intel processor used in the original IBM PC in 1981, but there are others:

CPU	Company	Used in...
Alpha	DEC	DEC servers and workstations
ARM	Acorn	Mobile devices
m68k	Motorola	Early Macintosh, Amiga, Atari, Sega
MIPS	MIPS	SGI Workstations, Sony Playstation2
PowerPC	Apple/IBM/ Motorola	Apple Macintosh
s390	IBM	IBM mainframes
Sparc	Sun	Sun workstations and servers
IA-64	Intel	...not much.



CPU in a PGA (Pin Grid Array) package

10

It's important to note that, in general, different processors will have different “instruction sets”. A program compiled for one type of CPU may not be able to run on a computer with a CPU of a different type.

The solution is often to re-compile the program, using a compiler capable of producing the right instructions for the new CPU.

This is one of the many advantages of writing code in a high-level language, rather than trying to write machine code directly. High-level languages let you specify what you want to do, without worrying about the details of the underlying CPU.

Part 3: Using Linux

Okay, so now we know everything about computer hardware. Now let's talk a little bit about the Operating System: the framework of software within which we'll be writing programs.

Command-line Shells:

Command-line shells accept typed commands, parse them and execute them. They also:

- Expand wild-card expressions.
- Usually store a history of previously-typed commands, and provide a way of recalling these.
- Provide a set of built-in functions that supplement (or sometimes replace) the commands provided by the operating system.
- Provide the user with the ability to define abbreviations for commands (aliases).
- Maintain a set of user-defined variables that can be used in command lines (environment variables and shell variables).

And many other things.

12

Why should you do things from the command line?:

* In Linux, graphical tools provide a front-end to help you do tasks, but you can do more from the command line.

* There are several sets of graphical tools available for Linux, so if you learn one of them you may find that it's not available on the next computer you use.

* There's no guarantee that a given computer will have graphical tools installed, or even a monitor.

* Text commands are easily reproduced. It's easy to document what you've done, or to tell someone else how to do it, or to automate what you've done.

A Few Useful Linux Commands:

ls	List the contents of a directory.
pwd	Show the name of the current directory.
cd	Change the current directory.
less or more	Show the contents of a file, one page at a time.
cp	Copy a file.
mv	Move (rename, relocate or both) a file.
rm	Delete (remove) a file.
mkdir	Make a new directory.
rmdir	Delete (remove) a directory.
man	Show docs (manual pages) for a command.
ln	Make a link to a file.
cat	Spit out the concatenated contents of one or more files, without paging.
touch	Change the timestamp on a file, or create an empty file.
which	Find a command in the search path.

13

This is just a list to get you started.

As you can see, the commands are typically terse.

In prelab 1 you'll have some practice using Linux shell commands before our first lab.

Command Syntax:

```
~/demo> ls -l
total 60
lrwxrwxrwx 1 bkw1a bkw1a    11 Jan 18 11:39 clus.pdf -> cluster.pdf
-rw-r----- 1 bkw1a bkw1a 20601 Jan 18 10:51 cluster.pdf
-rw-r----- 1 bkw1a demo    983 Jan 18 10:53 cpuinfo.dat
-rw-r--r-- 1 bkw1a bkw1a    29 Jan 18 10:59 data-for-everybody.1.dat
-rw------- 1 bkw1a bkw1a    41 Jan 18 10:56 ForYourEyesOnly.dat
drwxr-x--- 3 bkw1a bkw1a  4096 Jan 18 11:35 phase1
drwxr-x--- 2 bkw1a bkw1a  4096 Jan 18 10:55 phase2
-rw-r----- 1 bkw1a demo     72 Jan 18 10:52 readme.txt
-rw-r----- 1 bkw1a bkw1a  9552 Jan 18 10:52 ReadMe.txt
```

Linux commands are often modified by the addition of switches or qualifiers like the “-l”, for “long”, switch used in the ls command above. These modifiers will often take one of these forms:

* A dash followed by a letter or number, optionally followed by an argument
* Two dashes followed by a word, optionally followed by an argument.

For ls, some useful switches are:

-l Gives more information about the files.
-T Combined with -l, sorts the files in reverse time order.
-S Combined with -l, sorts the files in order of descending size.
-a Lists all files, including hidden files.

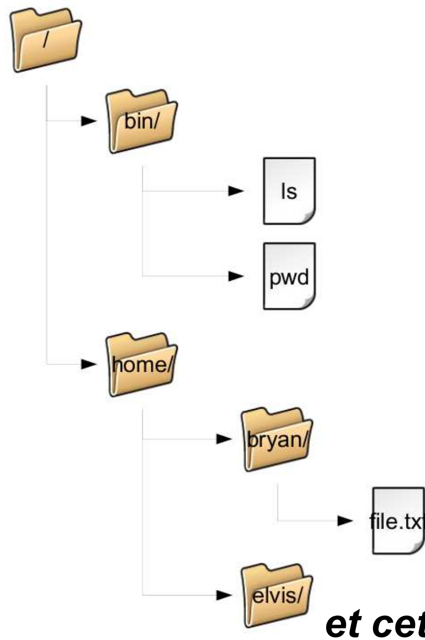
Multiple single-letter switches can often be combined, like “ls -lT” instead of “ls -l -T”

In this case, we can change the behavior of the “ls” command by adding the “-l” switch.

Note, though, that Linux commands were all developed independently, and they have a long history. Syntax conventions have evolved over time, and different developers have used different conventions.

We'll see examples of some odd command syntax with commands like “tar” and “ps”.

The Linux Filesystem:



Here's a graphical representation of a highly simplified Linux directory tree. One of the basic principles of Linux (and other varieties of Unix) is that there's only one directory tree. Everything lives somewhere under the "/" (root) directory.

This is unlike Windows, for example, where each device has a separate directory tree. In Windows we have a directory tree on drive C:, a different one on drive D:, and so on. Under Linux all files on all devices show up somewhere in the same directory tree, with "/" at its top.

This file is "/home/bryan/file.txt"

et cetera...

Note that, whereas Windows uses "\" as the directory separator, Linux uses "/".

The “Current Directory”:

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

```
~/demo> pwd
/home/bryan/demo
```

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 phase1
```

Or, equivalently:

```
~/demo> cd /home/bryan/demo/phase1
```

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.

The PATH Environment Variable:

Most of the commands you type will really just be the names of programs. When you type the command, the shell looks around for an executable file with the same name, and then runs it.

```
~/demo> echo $PATH
.:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
```

The PATH variable defines a search path for the shell to use when looking for a program. It's composed of a list of directory names, separated by colons. When looking for a program, the shell starts at the left of the list and looks in each directory until it finds a program with the matching name (or fails).

```
~/demo> which ls
/bin/ls
```

The "which" command looks through the search path, just as the shell would, and tells you where the shell would find a given program. When you type "ls" at the command line, the shell finds the program /bin/ls and runs it. You (or a system administrator) can add new commands just by putting appropriately-named programs into directories along your search path.

18

Note that:

1. ".", the current directory, is included. This is not generally the case for users with administrative privileges, for security reasons.

2. By putting an alternative program with the same name in /usr/local/bin, a local administrator can provide a modified version of a program that overrides any version that might already exist in /bin or /usr/bin.

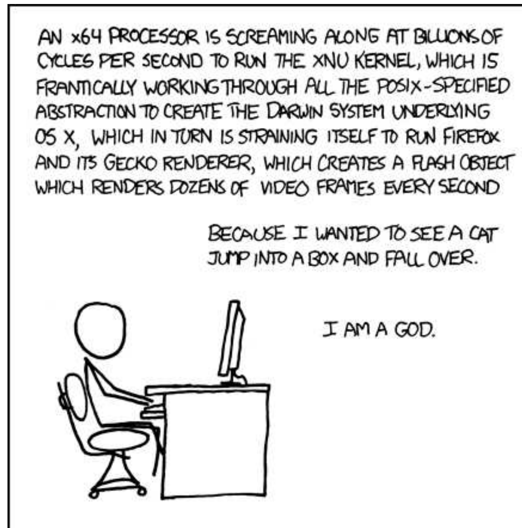
Command-Line History:

```
~/demo> history
   349  16:14   wget http://download.adobe.com/pub/adobe/magic/svgviewer....
   350  16:15   tar tzvf adobesvg-3.01x88-linux-i386.tar.gz
   351  16:15   tar xzvf adobesvg-3.01x88-linux-i386.tar.gz
   352  16:15   cd adobesvg-3.01
   353  16:15   dir
   354  16:15   cd ..
   355  16:15   rm adobesvg-3.01*
   356  16:15   rm -rf adobesvg-3.01*
   357  16:19   git clone git://people.freedesktop.org/~cworth/svg2pdf
   358  16:19   cd svg2pdf
   359  16:19   dif
   360  16:19   dir
   361  16:19   git pull
   362  16:19   make
   363  16:19   dir
   364  16:20   ./svg2pdf ../drawing.svg
   365  16:20   ./svg2pdf ../drawing.svg junk.pdf
   366  16:20   acroread junk.pdf
```

The “history” command shows you commands you've recently entered.

You can use the up and down arrow keys to recall previously-typed commands and re-use them. If you know the beginning of a previously-entered command, you can re-run it by entering a “!” followed by the beginning of the command.

Jargon Overload?



Don't worry, you'll pick up what you need to know naturally as you start working in the labs and on the homework problems.

And there's a lot of documentation available....

Documentation: Command-line help:

Many commands will tell you about themselves if you give them a "-h" or "--help" switch on the command line. For example:

```
~/demo> ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  do not ignore entries starting with .
  -A, --almost-all           do not list implied . and ..
      --author               with -l, print the author of each file
  -b, --escape               print octal escapes for nongraphic characters
      --block-size=SIZE      use SIZE-byte blocks
  -B, --ignore-backups       do not list implied entries ending with ~
  -c                         with -lt: sort by, and show, ctime (time of last
                               modification of file status information)
                               with -l: show ctime and sort by name
                               otherwise: sort by ctime
  -C                         list entries by columns
      --color[=WHEN]         control whether color is used to distinguish file
                               types.  WHEN may be `never', `always', or `auto'
  -d, --directory            list directory entries instead of contents,
                               and do not dereference symbolic links
  -D, --dired                generate output designed for Emacs' dired mode
  -f                         do not sort, enable -aU, disable -lst
  -F, --classify             append indicator (one of */=>@|) to entries
.....
```


Note that this is just a convention, and not all commands will honor it. As we noted before, these commands have a long history, and were written by many authors.

Documentation: Man Pages:

"Man Pages" (online documents in a standard format) are available for most common commands. The "man" command will show these to you, one page at a time. To exit from man, type "q" (for "quit"). To go to the next page, press the spacebar. To go back up, press "b".

```
~/demo> man ls
LS(1)                            User Commands                           LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort.

       Mandatory arguments to long options are  mandatory  for  short  options
       too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

q=quit,b=back,space=forward,h=help
```


For information about using the man command, don't hesitate type type "man man".

Man pages are the most common type of online documentation for Unix-like operating systems.

Documentation: Info Pages:

"GNU Info Pages" are another standard format for online documentation. Fewer commands have info pages, but when present this documentation may be more extensive than the command's man page. Info pages are arranged in a tree, with links between documents, much like a primitive version of the World Wide Web.

```
~/demo> info ls
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directo\
ry listing

10.1 `ls': List directory contents
==================================

The `ls' program lists information about files (of any type, including
directories).  Options and file arguments can be intermixed
arbitrarily, as usual.

   For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the file name.  If no non-option
argument is specified, `ls' operates on the current directory, acting
as if it had been invoked with a single argument of `.'.

   By default, the output is sorted alphabetically, according to the
locale settings in effect.(1) If standard output is a terminal, the
output is in columns (sorted vertically) and control characters are
output as question marks; otherwise, the output is listed one per line
and control characters are output as-is.
--zz-Info: (coreutils.info.gz)ls invocation, 54 lines --Top---------------------
Welcome to Info version 4.8. Type ? for help, m for menu item.
```


Some commands have only info pages. These commands will typically have a minimal man page that only refers you to the info page.

For information about navigating around inside info, try typing "info info" at the command line.

Part 4: Programming Languages

ARTICLE XXIX.

Sketch of the Analytical Engine invented by Charles Babbage *Esq. By* L. F. MENABREA, *of Turin, Officer of the Military Engineers.*

[From the *Bibliothèque Universelle de Génève*, No. 82. October 1842.]

[BEFORE submitting to our readers the translation of M. Menabrea's memoir 'On the Mathematical Principles of the ANALYTICAL ENGINE' invented by Mr. Babbage, we shall present to them a list of the printed papers connected with the subject, and also of those relating to the Difference Engine by which it was preceded.

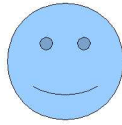
For information on Mr. Babbage's "*Difference* Engine," which is but slightly alluded to by M. Menabrea, we refer the reader to the following sources:—

1. Letter to Sir Humphry Davy, Bart., P.R.S., on the Application of Machinery to Calculate and Print Mathematical Tables. By Charles Babbage, Esq., F.R.S. London, July 1822. Reprinted, with a Report of the Council of the Royal Society, by order of the House of Commons, May 1823.
2. On the Application of Machinery to the Calculation of Astronomical and Mathematical Tables. By Charles Babbage, Esq.—Memoirs of the Astronomical Society, vol. i. part 2. London, 1822.
3. Address to the Astronomical Society by Henry Thomas Colebrooke, Esq., F.R.S., President, on presenting the first Gold Medal of the Society to Charles Babbage, Esq., for the invention of the Calculating Engine.—Memoirs of the Astronomical Society. London, 1822.
4. On the Determination of the General Term of a New Class of Infinite Series. By Charles Babbage, Esq.—Transactions of the Cambridge Philosophical Society.
5. On Mr. Babbage's New Machine for Calculating and Printing Mathematical Tables.—Letter from Francis Baily, Esq., F.R.S., to M. Schumacher. No. 46, Astronomische Nachrichten.

This is Ada Lovelace, the only legitimate daughter of Lord Byron. She was a talented mathematician, and is considered to be the first computer programmer. In the publication above, she described an algorithm (or "recipe") for use with Charles Babbage's "Analytical Engine" to calculate Bernoulli numbers. The programming language "ada" is named after her.

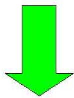
A recipe doesn't specify the type of measuring cup you should use, and it doesn't tell you what brand of flour to use (well, it may, but you know you can ignore that). It gives you a general set of instructions for getting something done. A programming algorithm is the same: it doesn't tell you what language to use, or what to name your variables, but tells you a useful technique for doing something.

Compilers:



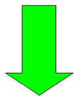
printf (“Hello world!”);

A programming language is like a human language. It has a vocabulary (reserved words with special meanings) and a syntax or fixed rules for using the words.

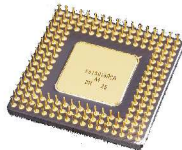


457f 464c 0101 0001
0000 0000 0002 0003
...

The programming language is a high-level tool to define the steps your computer must take to solve a problem.



In the end a program in any language you choose must be reduced (compiled / assembled / interpreted) to a set instructions the CPU can understand in order for it to function (machine code).



In general, these instructions will differ from one type of CPU to another, but most computers today use CPUs that understand a common set of instructions called “IA-32”.

Timeline of Programming Languages:

Hundreds of programming languages have been developed since the first digital computers appeared in the 1940s. Many of them are still in use today. The list below shows the years in which several widely-used languages were invented.

The C language, in which we'll do most of our work, was developed by Dennis Ritchie in 1972. (The C++ language appeared a little over a decade later.)

In developing C, Ritchie did two things:

- Defined the C language, by describing its vocabulary and syntax,
- Wrote a compiler that could translate that language into machine code for a particular CPU.

1957	FORTRAN
1959	LISP
1960	COBOL
1964	BASIC
1970	Forth
1972	C →
1983	C++
1987	Perl
1991	Python
1995	Java

Dennis Ritchie

One of the attractive qualities of C was its simplicity. This made it easy to write a C compiler for each new type of CPU that came along. During the early years of computing, this was even more important than it is now, since back then each manufacturer had a different CPU with a different instruction set.

Many other people wrote C compilers that could understand Ritchie's C language.

The original purpose of C was the development of the Unix operating system. The name “C” was chosen as a successor to an earlier language called “B”, which was the successor to the even earlier “BCPL” (“Basic Combined Programming Language”).

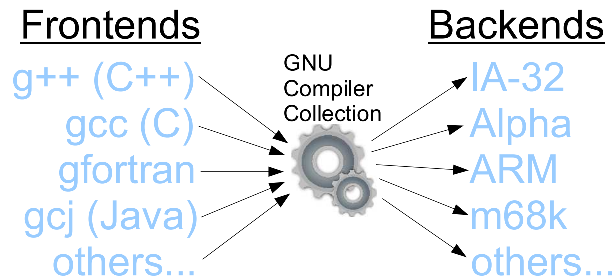
A Simple C Program:

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

  int a = 2;
  int b = 2;
  int c;
  double d;

  printf ("Hello, world!\n");

  c = a*b;
  printf ("The value of c is %d\n", c);

  d = sqrt(a);
  printf("The square root of %d is %f\n",a,d);

  return(0);
}
```

- Commonly-used statements can be stored in external “header” files and re-used in other programs.
- Each part of your program is a function. At the highest level is a special function named “main”.
- Every statement must end with a semicolon.

GNU C/C++

In this class, we'll be using g++, a compiler originally written by Richard Stallman as part of his GNU Compiler Collection project:



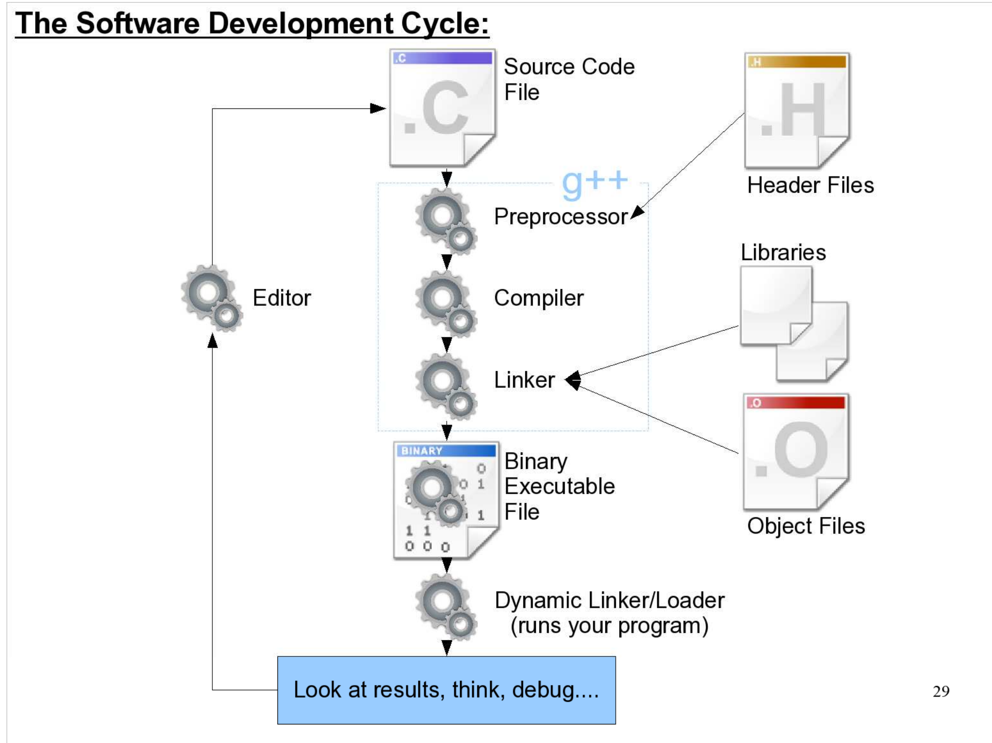
g++ is just a program, and you invoke it from the command line like this:

```
~demo> g++ -o myprogram myprogram.cpp
```

This would cause g++ to read the file “myprogram.cpp” and produce the output file “myprogram”, which is an executable binary file. You could then run your program by typing “myprogram” at the command line.

The GNU Compiler Collection contains a core that provides the common functionality needed by most compilers, and modular “frontends” and “backends” that allow the collection to understand several different languages and generate machine code for many different types of CPU.

Although specialized compilers may be able to generate faster or smaller code, the flexibility of the GNU Compiler Collection has made it very popular and it is widely used.



Note that, by default, g++ does more than just compile. It also runs a preprocessor before compiling and a linker afterward. We'll talk more about these steps soon.

The C Vocabulary:

C is a simple language in at least one respect: it has a small vocabulary. Shown below is a list of all of the words in the C language. There are only 32 of them!

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 should note two things about these words:

• Remember that C is case-sensitive, so it won't recognize the words "`IF`" or "`If`" for example.

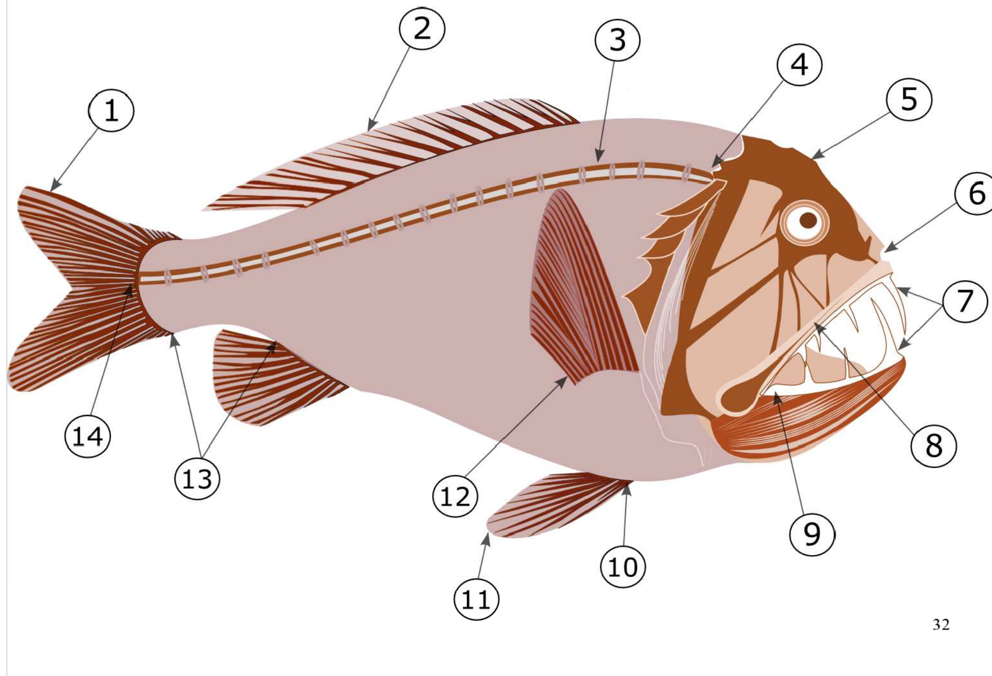
• None of these words is allowed to be used as the name of a variable. Because of C's syntax, it can't tell the difference between a variable name and an instruction. Be sure to avoid these words when naming your variables.

The C++ Vocabulary:

The C++ language starts with the vocabulary of C, but it adds many new words:

asm	dynamic_cast	namespace	reinterpret_cast	try	bool
explicit	new	static_cast	typeid	catch	false
operator	template	typename	class	friend	private
this	using	const_cast	inline	public	throw
virtual	delete	mutable	protected	true	wchar_t
ALSO					
and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

As with C, you need to avoid using these words as variable names when programming in C++.



Now let's look at the structure of a typical C program.

The Structure of a C Program:

A C or C++ program is built up out of units called "functions". Each program must have at least one function, called "main". The large-scale structure of a C program consists of a statement like the one below, defining what this "main" function should do.

In the example below, the "main" function doesn't have any arguments, but we'll see later that it can do so.

```
int main() {
    The body of the
    program goes here.

    return(0);
}
```

The body of the function is delimited by braces.

The function returns a value. In the case of "main", the operating system uses this value to determine whether the program completed successfully. By convention, non-zero means an error occurred.

C uses several types of parenthesis-like delimiters. Here's a list, along with the names people commonly use to refer to them:

( ) Parentheses
[ ] Square brackets
{ } Curly brackets or braces
< > Angle brackets

Variables, Assignment and Types:

C and C++ are strongly typed languages. This means that every variable you use must be defined to hold a specific type of data.

Some frequently used numeric types are shown below.

These lines define the types of these variables.

```
int a;
int b = 25;
float c;
float d;
double e;
```

These lines assign values to some variables.

```
a = 50;
c = 3.14;
d = c;
```

int	An integer
float	A real (floating-point) number
double	A “double precision” floating-point number

- Variables can optionally be assigned a value when they're defined.

- Assigning the value of “c” to the variable “d”.

Arithmetic Operators:

C and C++ include the following arithmetic operators:

+	a+b	Addition
-	a-b	Subtraction
*	a*b	Multiplication
/	a/b	Division
%	a%b	Remainder (modulo)

That's all! Other mathematical operations (like square roots and trigonometric functions) are available as functions that live in the standard math library, libm. We'll use these extensively, soon.

```
a = 50;
b = 2*a - 10;
c = 3*b/(a-1);
```

Note that parentheses can be used in the expected way.

This illustrates one of C's design philosophies. The language provides a minimal core functionality that can be extended by writing functions. Some of these functions (like printf and sqrt) are so widely useful that they've been included in the standard libraries, libc and libm, that are usually distributed along with the C compiler.

Adding Comments to Your Code:

C++ allows you to add comments to your code in two different ways. Comments are very important. They help you:

- Keep track of what you're doing while you write, and they
- Remind you of what you did when you or someone else looks at your code later.

```
/* Traditional C defines comments by using
opening and closing comment markers as shown
in this example.  These comments may span
multiple lines.*/

// However, it is often better to write
// multiple line comments in this way
// to make the extended comments more
// clear in the text of your code

  int a =15; // C99 and C++ allow single line comments w/ the double slash
  int b = 6; /* this type of comment is allowed, but not preferred */
```


Comments are also useful for debugging your program, which we'll discuss later. If you suspect that a section of your program is causing a problem, you can just “comment it out” (insert // in front of each of the possibly-offending lines) and see if it makes any difference. This is easier than actually removing the lines and putting them back.

Preprocessor directives:

By default, most modern C "compilers" actually do more than just translate source code into machine code. For example, before compiling your code, they typically run a preprocessor program. The preprocessor program scans the code, looking for special instructions.

```
#include <stdio.h>
#include <math.h>
#define PI 3.14
```

Some preprocessor directives.

Preprocessor directives begin with a pound sign (#). These statements are not part of the C/C++ language, per se. They form a small separate language of their own. We'll introduce the parts of it you need as we go along.

#include <stdio.h> directs the preprocessor to include the file called stdio.h into the text of your program. This is a header file. These are used to define the meaning of statements you use that are not intrinsically part of the C/C++ language, standard functions from the C library, functions from your own code base, etc...

Using Functions:

The core functionality of the C language can be extended by adding functions. Many of these are available in the Standard C Library, or other libraries. You'll also be writing your own functions.

Here's how you could use the function "printf" (which prints out text) in a program:

```
#include <stdio.h>
int main() {
  printf("Hello World\n");
  return (0);
}
```

The header file we included, stdio.h, defines the calling syntax of printf. The compiler checks that the function is used properly and stops with an error message if there is a usage mistake.

"\n" is the special character "newline" (also known as "linefeed").

(Note that the code above is a complete C program.)

Note: not defining printf through the header file will also cause the compiler to stop with an error message.

More About printf:

The printf function takes one or more arguments. The first argument is called the "format", and it's just a text string that specifies a sort of template for the things that printf will print out.

printf(format, arg1, arg2, ...);

The format string contains plain text and any number of format specifiers (like "%d" below). Each of the format specifiers will be replaced by one of printf's extra arguments, then the result will be printed out.

```
#include <stdio.h>
int main() {
  int a =15;  // integer variable
  int b = 6;
  printf("Results:\n");
  printf("a/b is: %d\n", a/b);
  printf("a%%b is: %d\n", a%b);
  printf("a, b = %d, %d\n", a, b);
  return(0);
}
```

The program would print:

```
Results:
a/b is: 2
a%b is: 3
a, b = 15,6
```


"%d" means that the argument should be treated as a "decimal number" (meaning an integer, in C).

Note that you can print a literal "%" by writing "%%".

Format specifiers aren't required with the "cout" syntax available in C++.

We'll see more format specifiers later.

Integer Arithmetic:

Pay attention to the types of your variables. For example, when you try to store a floating-point value in an integer variable, C will round the value down to the next integer. This may not be what you want.

Note that operations on integers produce integer results. Here, the result is truncated to make it an integer.

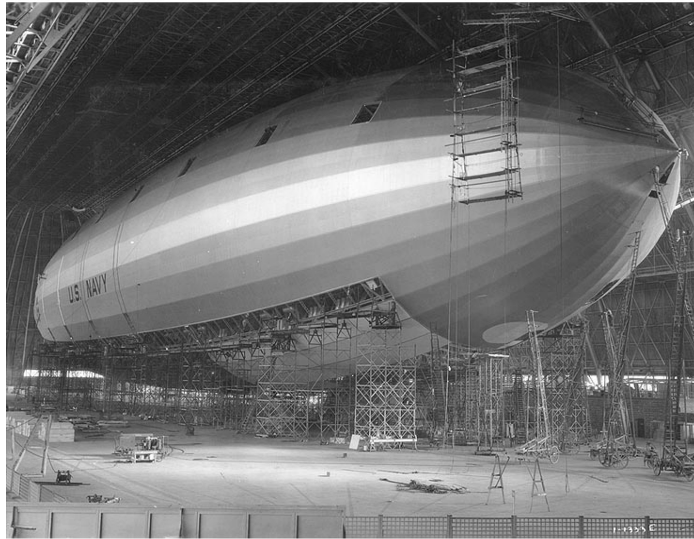
```
#include <stdio.h>

int main() {
  int a =15; // integer variable
  int b = 6;
  printf("a/b is: %d\n", a/b);
  printf("a%%b is: %d\n", a%b);
  return(0);
}
```

The program would print:

```
a/b is: 2
a%b is: 3
```

Part 6: Constructing Your First C Programs

Photo # NH 42023 USS Macon under construction at Akron, Ohio, circa early 1933



Now we know enough to start writing programs in C.

Compiling and Running Your Program:

```
#include <stdio.h>
int main() {
  printf("Hello World\n");
  return (0);
}
```

Let's say you've saved the C code for our "Hello World" program in a file called "hello.cpp". To compile this into an executable binary file that you can run, you could type:

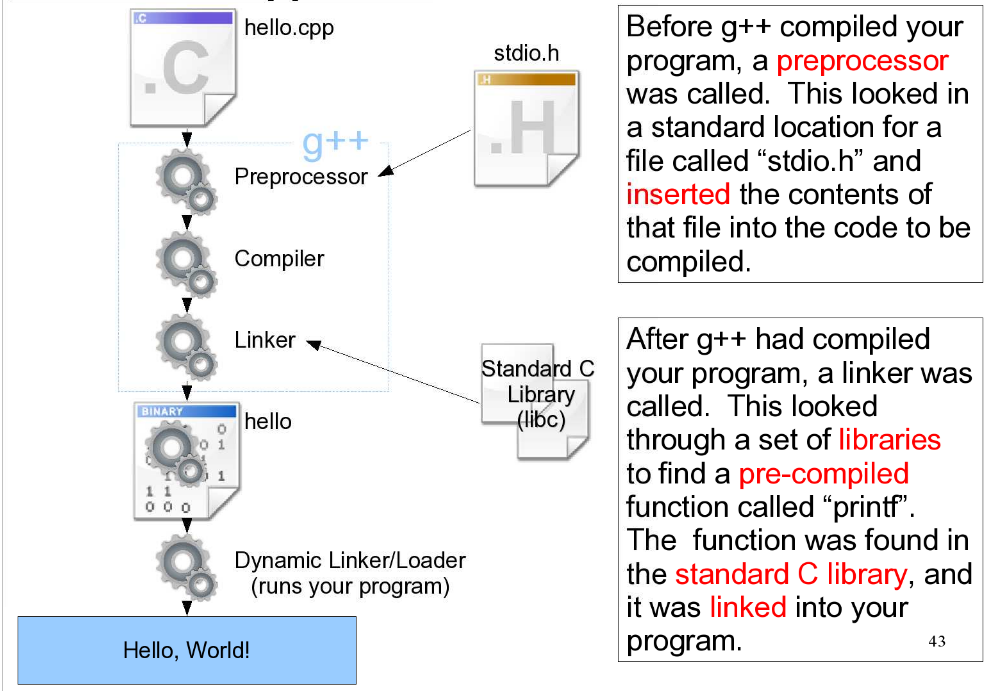
```
~demo> g++ -o hello hello.cpp
```

The qualifier "-o hello" tells the compiler that we want the output executable file to be named "hello". Otherwise your program will be given the default name "a.out". We can run this executable file by just typing its name. Here's what will happen:

```
~demo> hello
Hello World!
```

42

Note that on Galileo, by default, your current working directory is included in the shell's search path. This is why we can just type the name of the program and be confident that the shell will be able to find it. If the program were elsewhere, or if we changed to a different working directory, we'd probably need to type out the full path to the file, instead of just its name. For example, this might be "/home/bkw1a/hello".



We could have specified other libraries on the command line, if we wanted to use functions that weren't included in libc.

Trying Out C++-style Programming:

"Hello World", using C:

```
#include <stdio.h>
int main() {
  printf("Hello World\n");
  return (0);
}
```

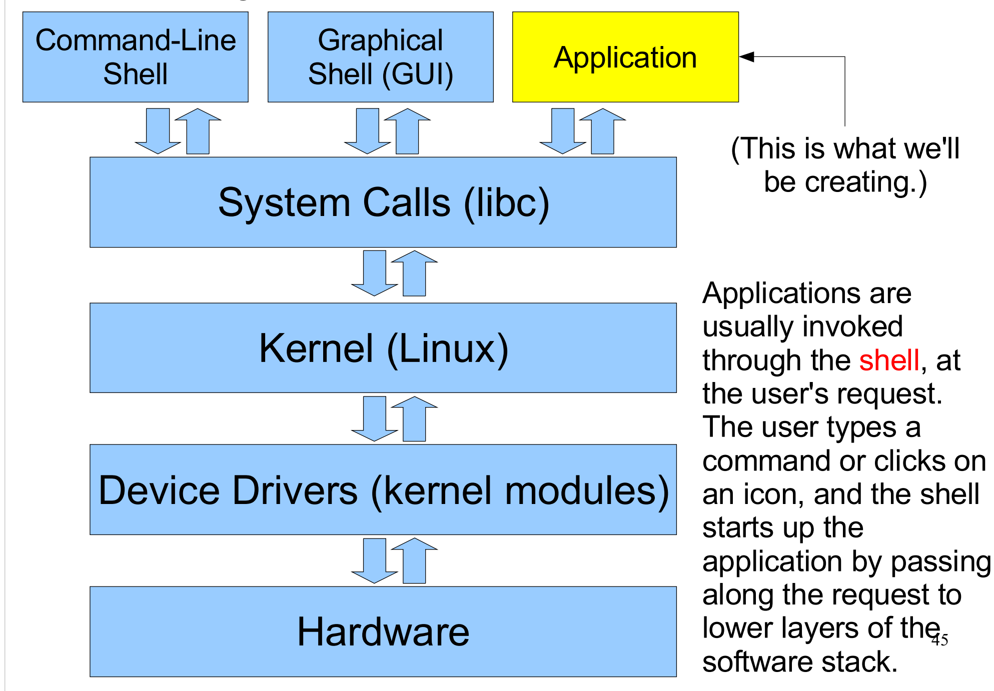
"Hello World", using C++:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Hello World" << endl;
  return (0);
}
```

C++ headers drop the .h extension.

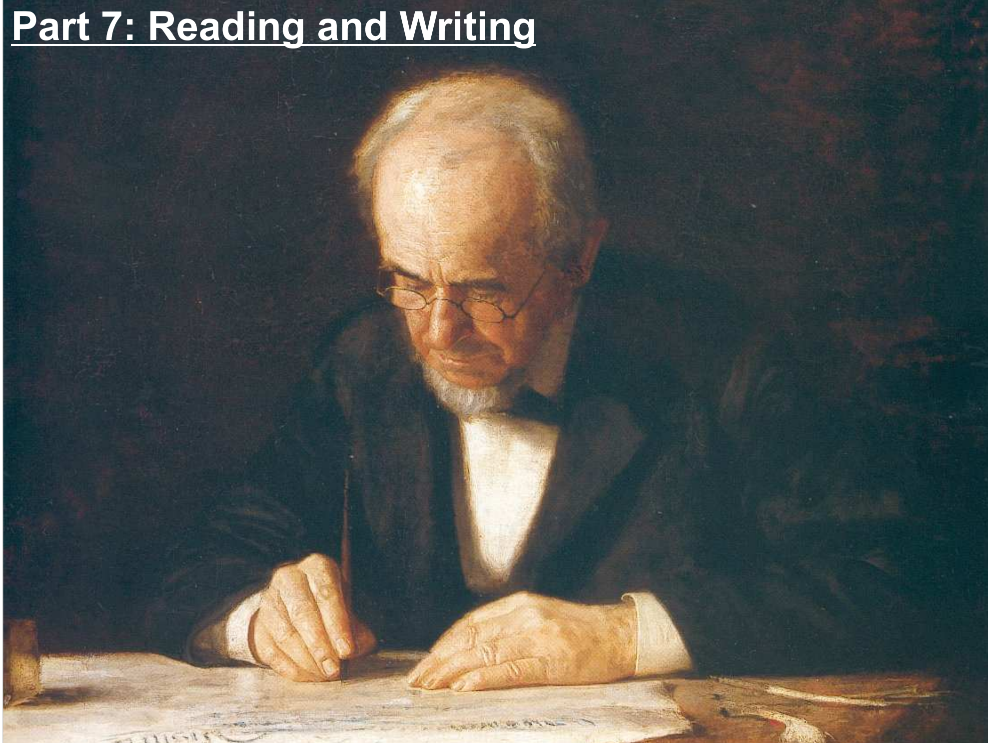
Members of the C++ library are confined to namespaces, the standard namespace contains the most common tools you will use.

Notice the very different syntax. Here we send two pieces of data to the output stream, cout. "endl" is equivalent to "\n".



Finally in this section, take a look at where the C programs we write fit in with the other software that's running on the computer. Note that the compiler (g++) is just another application.

Part 7: Reading and Writing



What if our program generates a lot of data? We could just ust printf to spit it all out, and then write down the results, but it would be a lot easier to write it directly into a file, where we can look at it later. Also, what if we need to give our program some data? We could set the values of some variables in the program and then re-compile the program, but it would be nicer if the program would just ask us for the numbers it needs, or read them from a file. Let's look at the other Input/Output functions in C for doing things like this.

Reading Data from the Keyboard:

scanf is the input analog of printf. It reads data from the keyboard, but you must tell it what kind of data it is reading, using a format string.

```
// Calculate the area of a square
#include <stdio.h>

int main() {
  int length;
  printf("Enter the length of each side\n");
  scanf("%d",&length);
  printf("The area = %d\n", length*length);
  return(0);
}
```

The textbook by Brooks begins with many Input/Output examples, and you'll get lots of practice in the labs and homework assignments.

Note the ampersand (&)...

More About scanf:

When using scanf to read in the value of most types of variable, we must prepend the character "&" to the variable name. Strings of text are an exception to this rule, as you can see in the example below. For now, let's treat this as a necessary piece of magic. The reason will become clear as we learn more.

Also notice how we define a string variable.

```
#include <stdio.h>
int main() {
   int length;
   char string[10]; // A 10-character text string
   printf("Enter a number\n");
   scanf("%d",&length);
   printf("Enter a string of text\n");
   scanf("%s", string);
   printf("len %d, string %s\n", length, string);
   return(0);
}
```

This is not covered immediately in the text, so for starting out make sure to follow the examples closely in writing your first programs

File Input and Output:

The functions fprintf and fscanf are analogous to printf and scanf, but they operate on data files instead of on the computer display and keyboard.

```
// Calculate the area of a square
#include <stdio.h>
int main() {
 int length;
 FILE *inFile; // define variables for our files
 FILE *outFile;
 inFile = fopen("input.dat","r"); // open input file
 outFile=fopen("outfile.dat","w");// open output file
 fscanf(inFile,"%d",&length);
 fprintf(outFile,"The area = %d\n", length*length);
 fclose(inFile);
 fclose(outFile);
 return 0;
}
```

Close both files when we're done.

Read from one file and write to the other.

Why close files? They're normally closed for you automatically when the program ends.

First of all, your program may not be as simple as the one above. It may take hours or days to complete, and during all that time your program would have files open. This uses up some limited system resources. Normally, there are limits on the number of files you can have open at one time. If you have lots of programs that leave files open while they're running, you may run into these limits.

Second, the last chunk of data isn't usually written to the file until it is closed. If one of your programs is running, and has the file open, the file may look incomplete to any other programs that try to use it.

It's good practice to close files as soon as you're done with them, BUT, don't close and re-open files. Opening and closing files takes a (relatively) long time, so your program should open and close files as few times as possible.

Next Time:

We’ll have a more systematic look at the language including

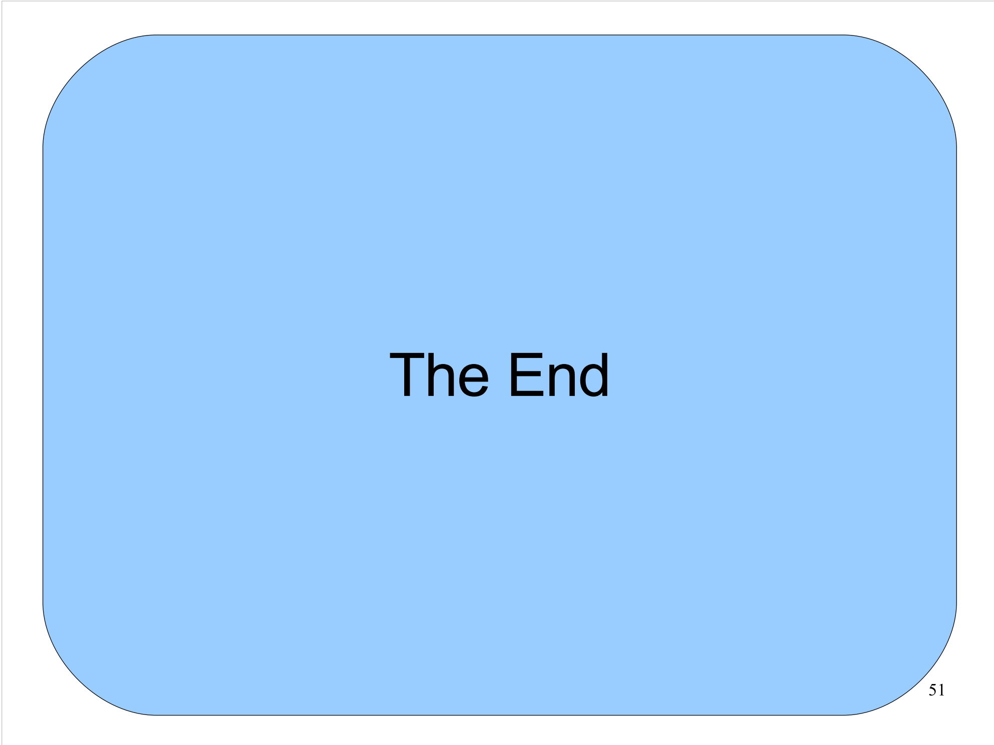
- data types
- making and using functions

This week's Lab:

- Building your first programs
- Basic calculations, input/output in C/C++

- Let me know if you don't have a Galileo account yet.
- Complete the Prelab BEFORE 10am Thursday
- Reading assignment: Brooks chapter 1, and sections 1-2.2 of chapter 2.

Brooks Ch. 1, Ch. 2 sections 1-2.2



Thanks!