Announcements

• Weekly readings will be assigned and available through the class wiki home page

• HW01 was assigned
  – Due Thursday 1 February
    • Electronic submission: by 12:00 noon EST

• Volunteer notetaker?

• This week: Lab02
  – Will be posted on Thursday

• Office hours: all held in our computer lab, room 022-C of this bldg
  – Me: After lecture 3:30-4:30 every Tuesday CANCELED TODAY! 😞
  – TAs:
    • Mondays: 3-5pm and 6-8pm
    • Wednesdays: 5-9pm

• Please see me: J. Barkstrom, R. O’Reilly
Back to Linux
## Some Common Linux Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ls</code></td>
<td>List the contents of a directory.</td>
</tr>
<tr>
<td><code>pwd</code></td>
<td>Show the name of the current directory.</td>
</tr>
<tr>
<td><code>cd</code></td>
<td>Change the current directory.</td>
</tr>
<tr>
<td><code>less</code> or <code>more</code></td>
<td>Show the contents of a file, one page at a time.</td>
</tr>
<tr>
<td><code>cp</code></td>
<td>Copy a file.</td>
</tr>
<tr>
<td><code>mv</code></td>
<td>Move (rename, relocate or both) a file.</td>
</tr>
<tr>
<td><code>rm</code></td>
<td>Delete (remove) a file.</td>
</tr>
<tr>
<td><code>mkdir</code></td>
<td>Make a new directory.</td>
</tr>
<tr>
<td><code>rmdir</code></td>
<td>Delete (remove) a directory.</td>
</tr>
<tr>
<td><code>man</code></td>
<td>Show docs (manual pages) for a command.</td>
</tr>
<tr>
<td><code>ln</code></td>
<td>Make a link to a file.</td>
</tr>
<tr>
<td><code>cat</code></td>
<td>Spits out the concatenated contents of one or more files, without paging.</td>
</tr>
<tr>
<td><code>touch</code></td>
<td>Change the timestamp on a file, or create an empty file.</td>
</tr>
<tr>
<td><code>which</code></td>
<td>Find a command in the search path.</td>
</tr>
</tbody>
</table>
• One of the main tasks you will do is edit human-readable files

• Many tools available to you to do this:
  – nano (you’ll use this in lab01)
  – pico (similar to nano)
  – nedit
  – emacs (lab01)
  – See [this wiki page](#) for some more discussion
Case Sensitivity:

Important Note: when typing commands, file names, etc...

Linux and C/C++ are

CaSe SeNsItIvE

So, for example:

This is not the same as this, VELOCITY is not the same as Velocity or velocity, MyFile.dat is not the same as MyFile.DAT

For best results, stick to all lower-case unless there's a good reason to do otherwise.
Local vs. Remote Computing
The “Galileo” Cluster:

In this class, we'll do our programming on a Linux cluster called “Galileo”. Galileo doesn't have any keyboards or monitors connected to it. So how do we use it?

[Image of the Galileo cluster web page]

http://galileo.phys.virginia.edu/
Remote Shells

• So far we have tacitly overlooked one key component:
  – What if I am not sitting in front of the specific computer / Linux machine I want to work on?
  – How do I establish a connection to that machine?
  – Use the network!

• Remote shell access:
  – We will use a program called ssh
    • Secure Shell
    • Replaced an old lovable program called telnet, which, in terms of security, had more holes than swiss cheese! Totally Unencrypted
    • Tip: Never ever type the command telnet or even consider typing it
  – FYI There is no ssh version for Windows by default but you can install a Windows version for free. See Software for home on the class home page.
Galileo: Where We Will Do Our Work

• The computer / Linux “machine” where we will do all our work is in the physics department and goes by the name galileo

• galileo is actually the head node of a cluster of computers running Linux
A Note on Our Computer Lab

Our Computer Lab:

Room 022-C

Lab Computers

Room 115

Galileo Cluster
The Computers in the Lab

- The desktop machines in our computer lab run Lubuntu v12.04, kernel v3.2.0-24-generic
  - a “lightweight” Linux operating system based on Ubuntu.
  - Ubuntu is a popular (most popular?) type of Linux OS distributions with its own desktop environment.
  - There are several such distributions popular in the community today, Ubuntu is just one.

The actual nodes on galileo run a Linux distribution called CentOS, release 5-11. The kernel is v 2.6.18-407.el5
Use ‘lsb_release –a’
• The actual nodes on galileo run a Linux distribution called CentOS, release 5-11.
  – The kernel is v 2.6.18-407.el5 (Use ‘lsb_release –a’)
Remote Graphics:

- **X Server**: running on your local computer

- **X Client**: program to display graphics from any other program

- Can be on different computers!

Under Linux, graphical programs use a protocol called “X” to draw images on your screen. X is “network transparent”, meaning that X clients don’t care whether they’re running inside your computer or on a different computer, half a world away.
Connecting to galileo from other locations/laptops

• It would be nice to connect to galileo to do your work from say, your laptop while sitting in the library for instance

• Many hints on how to do this are located here: http://faculty.virginia.edu/comp-phys/phys2660/wiki/doku.php?id=software:sw_main

• If you run Windows
  – I recommend installing and using putty with Xming
  – Another option for Windows: NXClient for speedier graphics connections. See me if interested – might be better than putty+XMing

• If you run Mac OS X
  – ssh is installed but you still need an X Server to see graphics. See ssh and X11 for Mac
Computer Programming in C: The Beginnings
Programming Languages

• As we mentioned earlier, the hardware inside a computer does not speak a human-readable language

• But we as humans want to harness the power of the computer as a tool to solve problems

• Two options:
  – We communicate directly with computer using its language
  – We write programs in something more familiar to us, and figure out how to translate

• Option 2 is preferred!
Dennis Ritchie:
- defined the relatively simple syntax of C as part of original Unix R&D
- wrote the first compiler for C programs

Standard simple language meant people could easily write compilers for any CPU architecture AND users could easily write new programs
Compilers: 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.

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”.
Our compiler: g++

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:

**Frontends**
- g++ (C++)
- gcc (C)
- gfortran
- gcj (Java)
- others...

**Backends**
- IA-32
- Alpha
- ARM
- m68k
- others...

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 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.

```c
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.
The C Language: An Example

A Simple C Program:

```c
#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.
Variable Definition in C
Variable Types in C

- All variables in C must be declared as a particular type

<table>
<thead>
<tr>
<th>Integers</th>
<th>short</th>
<th>A “small” integer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int</td>
<td>A “medium” integer</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>A “large” integer</td>
</tr>
<tr>
<td></td>
<td>unsigned short</td>
<td>Positive-definite versions of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the types above.</td>
</tr>
<tr>
<td></td>
<td>unsigned</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
<td></td>
</tr>
<tr>
<td>Floating-point numbers</td>
<td>float</td>
<td>A real (floating-point) number</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>A “double precision” floating-point number</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td>Even higher precision.</td>
</tr>
<tr>
<td>Characters</td>
<td>char</td>
<td>A character of text.</td>
</tr>
</tbody>
</table>
Defining and Assigning Variables

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>An integer</td>
</tr>
<tr>
<td>float</td>
<td>A real (floating-point) number</td>
</tr>
<tr>
<td>double</td>
<td>A “double precision” floating-point number</td>
</tr>
</tbody>
</table>

These lines define the types of these variables.

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

These lines assign values to some variables.

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

- Variables can optionally be assigned a value when they're defined.
- Assigning the value of “c” to the variable “d”.
Interlude: Storing Variables
Bits and Bytes

• How is data stored on a computer?
  – Data is stored in **bits** – think of them as values that can have either the value 0 or 1, or switches
  – One value is called a bit
  – bits are grouped together in groups of 8, called a **byte**
  – 8 bits are convenient for binary arithmetic – using the binary number system (1s and 0s only)

– Examples: subscripts reflect base system
  • \(1_d = 1_b = 1 \times 2^0\)
  • \(5_d = 101_b = 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\)
  • \(12_d = 1100_b\)
  • \(21_d = 10101_b\)
  • \(182_d = 10110110_b\)
Familiar Storage Sizes

- **Byte**: $10^0$
- **Kilobyte**: $10^3$
- **Megabyte**: $10^6$
- **Gigabyte**: $10^9$
- **Terabyte**: $10^{12}$
- **Petabyte**: $10^{15}$
What is a Petabyte?

- **Library of Congress:**
  - Founded in 1800
  - Largest library in the world
  - Functions as the national library of the United States
  - Burned by the British in 1814
  - Restocked in 1815 when Congress approved the purchase of Thomas Jefferson’s personal library of 6,487 books for $23,950.

- As of 2012, it was **claimed by a curator** that the Library of Congress had 3 PB of digital archives – digitized books, films, audio, etc.
What is a Petabyte?

Facebook, in its IPO filing, said it stores over 100 petabytes (PB) of media (photos and videos). It’s not unrealistic to say that Facebook probably has a total storage of capacity well beyond that, once you factor in backups and other data (status updates, likes, and so on), possibly in the 300PB range.

Microsoft recently admitted that Hotmail stores over 100 petabytes, and that SkyDrive, with “17 million customers,” stores 10PB of data. Like Facebook, Microsoft’s total capacity, once we factor in the rest of Azure and its web properties, is probably well over 300 petabytes.

Megaupload is relatively tiny in comparison, apparently storing just 25 petabytes.

Amazon, rather than giving us a nice, easy number of petabytes, instead announces the total number of objects stored by its S3 cloud storage service. As of April 2012, Amazon S3 stored 905 billion objects. If we assume an average size of 100KB, that’s around 90 petabytes; if the average size is 1MB, that’s 900 petabytes — almost an exabyte!

Dropbox, a year ago, stored “10+ petabytes” of data. It had 25 million users then, and 100 million users today, so all things being equal the company now stores around 40PB of data.
What is a Petabyte?

- The Large Hadron Collider produces about 1 PB of data per second
- In a typical 20-hour run of the LHC,
  - 24,000 Libraries of Congress
  - 720 Facebooks
  - 80 amazons
Larger and Larger

https://www.quora.com/Is-one-terabyte-larger-than-one-gigabyte
Storing Variables

• When you run your program, an area in your computer’s memory is set up to store the value for each of the variables you define.

• Different variable types require a different amount of space.

• Hence, one must be careful to match variable types with the actual use one envisions … cannot put a decimal number in an int, for instance

```c
int main()
{
    double velocity = 0.0;
    float x = 0.0;
    int number = 0;
    char a = 'a';
    float y = 0.0;
    ...
}
```
Storage of an integer: Example

- For an integer we have up to 4 bytes = 32 bits of storage
- So the variable “num” defined as this:

```markdown
int num = 90937759;
```

is stored this way:

```
0 0 1 1 0 1 1 0
0 0 1 1 0 0 1 1
0 1 1 0 0 1 0 0
1 0 0 1 1 1 1 1
```

- Comments:
  - One bit is reserved for the sign +/- (leading bit):
    - leading bit = 1
    - leading bit = 0
  - So max value we can store is $2^{31} - 1 = 2,147,483,647$

Implication:
If your OS uses only 32-bit words for memory addresses, the max memory address is 4,294,967,296… meaning up to 4.3 GB of memory can be handled…

**hence 32-bit systems cannot have much more than 4 GB of memory.**
Storage of a non-integer: Example

Floating Point Number Representation - Conversion Example

https://www.youtube.com/watch?v=n-XozGu1viM
Back to Variable Definition in C
Defining Variables in C

- In C, variables should be defined at the beginning of the function in which they are being used:

```c
#include <stdio.h>
int main() {
    // start of main function
    int an_int = 100; // start of variable declarations
    float a_float = 0.1;

    a_float = a_float * 100.5; // start of program statements
    an_int = an_int / 7;
    printf("%d %f\n", an_int, a_float);
    return(0);
}
```

Note format specifiers for integer and floating-point numbers. We'll talk more about this later.
Defining Variables: Defaults?

• What if I do not initialize (give an initial value to) a defined variable?

  What does the following code print?

  ```c
  #include <stdio.h>
  int main() {
    float a;
    a = a * 100.0;
    printf("%f\n", a);
    return(0);
  }
  ```

• In C, this could be 0.0, could be anything, WILL BE GARBAGE.
  – C does not initialize variables for you!
  – It reuses memory space when defining a new variable though – and this memory space is not “flushed” before the assignment: stale contents

• It is best to always initialize your defined variables to something appropriate
Aside: Preprocessor Directives

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.

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

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...
Variables vs. Preprocessor Definitions

• You can define oft-used constants (think the speed of light, etc.) using preprocessor definitions

• Use the `#define` command

Comparison:

With preprocessor definitions, the compiler sees all instances of `PI` and replaces with `3.14159`, etc…
Preprocessor Definitions vs Constants

- The virtue of using preprocessor definitions is speed – can speed up your program at run time, since fewer vars are stored in memory, meaning fewer accesses.

- But there are dangers:
  - universal replace might go poorly: What if you had a function called SAPInterestCalc()?
  - other preprocessor definitions are hidden from you – hard to see if yours clobbers another say in a header file

- Instead, one can use const:

```c
#include <stdio.h>
// Define constant values. Compiler will protect these:
const float RADIUS_OF_EARTH = 6378.1;  // in km
const float PI = 3.14159;

int main() {
    printf(“The circumference of Earth = %f\n”,
            2.0*PI*RADIUS_OF_EARTH);
    return 0;
}
```

**Using const is the best coding practice – compiler won’t let it be altered!**
Variable Storage: Examples

A variable declaration determines how its data are physically stored in memory.

In general the **details of this storage differ** from machine type to machine type, OS to OS, and programming language to programming language.

All data are ultimately stored as binary patterns, but the format differs depending on the variable's type.

Here's how one compiler, on one computer, stores the value "4" when it's an int, float or char:

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Binary Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 4;</td>
<td>000000100 00000000</td>
</tr>
<tr>
<td></td>
<td>000000000 00000000</td>
</tr>
<tr>
<td>float f = 4;</td>
<td>000000000 00000000</td>
</tr>
<tr>
<td></td>
<td>100000000 01000000</td>
</tr>
<tr>
<td>char c = '4';</td>
<td>00110100</td>
</tr>
</tbody>
</table>

- int: 4 bytes
- float: 4 bytes
- char: 1 byte
Note the last two lines: compound types. 5/2 is rightfully evaluated as a decimal number but has two int’s as args – hence it is assigned the size of an integer. So don’t try to divide integers and expect the right answer – unless you define things properly!

And the last one – the size of the result is given by the highest precision argument.
Using `sizeof()`

```c
#include <stdio.h>
int main()
{
    float b;
    // return size of a variable or data type in bytes
    printf("sizeof(int) = %d bytes\n", sizeof(int));
    printf("sizeof(b) = %d bytes\n", sizeof(b));
    printf("sizeof(5/2.0) = %d bytes\n", sizeof(5/2.0));
    printf("sizeof((int)(5/2.0)) = %d bytes\n",
            sizeof((int)(5/2.0)));
    return(0);
}
```

- `sizeof(int) = 4 bytes`
- `sizeof(b) = 4 bytes`
- `sizeof(5/2.0) = 8 bytes`
- `sizeof((int)(5/2.0)) = 4 bytes`
Casting Variables

- **Casting** is the conversion from one variable type to another – can either increase or decrease the precision of a stored value

Example:

```c
float a = 101.1;
int i = 0;
i = a;
a = i + 1;
```

Downward cast, setting i equal to 101 (**lower precision**). A cast to int always **truncates**!

Upward cast, setting a to a value of 102.0 (**higher precision**).

These are known as **implicit casts** – the compiler knows how to increase/decrease precision for you!

Implicit casts are generally not a good idea – anything done for you by the compiler is generally not ideal.
Avoid Implicit Casts

Upward casting usually proceeds without complaint, but automatic or implicit downward, resolution-reducing casts, can generate a compiler warning:

```
cast.cpp

10:  float a=101.1;
11:  int i = 0;
12:  i = a;
```

Implicit downward cast, giving `i` a value of 101 (lower precision).

```
~/demo> g++ cast.cpp
```

```
cast.cpp: In function `int main()':
cast.cpp:12: warning: assignment to `int' from `float'
```

Try to avoid implicit casts. Good programming style uses explicit casts, where data are consciously managed by the programmer.
Explicit Casts

• The preferred way to do this is the following:

```java
10: float a=101.1;
11: int i = 0;
12: i = (int) a;
```

Explicit downward cast \((i = 101)\).

The syntax for implicit casts is “\((type)\)variable”. For example:

```java
i = (int) a;
g = (float)i;
h = (double)a;
```

• Virtues:
  – no warning messages
  – you are more aware of the precision of the stored variables / values
Formatted Input / Output
Recall from lab00, you used printf() and scanf() to print/read values to/from screen/keyboard:

- % is the control key indicating some I/O formatted data
- followed by an I/O specifier:

Some common format specifiers:

| i, d, ld, li | Integer data or long integer data. |
| f,lf | Floating-point number in decimal notation ("float" or "double"). |
| e,E | Floating-point numbers in Scientific Notation, like "6.02e+23". You can choose upper or lower case by picking "e" or "E". |
| g,G | Floating-point numbers, using either Scientific Notation or regular notation, whichever is shorter. |
| c,s | Single characters, or strings of characters. |

Read Brooks page 33 re: the difference between “i” and “d” and leading zeroes.
I/O format specifiers are **important**. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will **misinterpret** the data in translating it for output.

```c
printf("%f",5/2)    // Error
printf("%d",5/2)    // Error
printf("%f",(float)(5/2))    // Correct
printf("%f",5/2.0)  // Error
printf("%d",5/2.0)  // Error
printf("%d",(int)(5/2.0)) // Error
```

Similar care must be taken with scanf statements.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```
printf("%f",5/2) → 0.000000  OOPS (integer data!)
printf("%d",5/2)  
printf("%f",(float)(5/2))  
printf("%f",5/2.0)  
printf("%d",5/2.0)  
printf("%d",(int)(5/2.0))  
```

Similar care must be taken with scanf statements.
Format Mismatches

I/O format specifiers are **important**. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or `printf` will **misinterpret** the data in translating it for output.

<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;%f&quot;,5/2)</code></td>
<td>0.000000</td>
<td>OOPS (integer data!)</td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;,5/2)</code></td>
<td>2</td>
<td>OK</td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;,(float)(5/2))</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;%f&quot;,5/2.0)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;,5/2.0)</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>printf(&quot;%d&quot;,(int)(5/2.0))</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similar care must be taken with `scanf` statements.
Format Mismatches

I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```
printf("%f",5/2) → 0.000000   OOPS  (integer data!)
printf("%d",5/2) → 2            OK
printf("%f",(float)(5/2)) → 2.000000   OK
printf("%f",5/2.0)             OK
printf("%d",5/2.0)             OK
printf("%d",(int)(5/2.0))      OK
```

Similar care must be taken with scanf statements.
Format Mismatches

I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```
printf("%f",5/2) → 0.000000
printf("%d",5/2) → 2
printf("%f",(float)(5/2)) → 2.000000
printf("%f",5/2.0) → 2.500000
printf("%d",5/2.0)  OOPS (integer data!)
printf("%d",(int)(5/2.0))
```

Similar care must be taken with scanf statements.
Format Mismatches

I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

```c
printf("%f",5/2) → 0.000000   OK
printf("%d",5/2) → 2          OOPS (integer data!)
printf("%f",(float)(5/2)) → 2.000000   OK
printf("%f",5/2.0) → 2.500000   OK
printf("%d",5/2.0) → 0          OOPS (double float data!)
printf("%d",(int)(5/2.0))
```

Similar care must be taken with scanf statements.
I/O format specifiers are important. They translate the internal representation of the data into the text on your screen.

The data type must match specifier, or printf will misinterpret the data in translating it for output.

\[
\begin{align*}
\text{printf("\%f",5/2)} & \rightarrow 0.000000 & \text{OOPS (integer data!)} \\
\text{printf("\%d",5/2)} & \rightarrow 2 & \text{OK} \\
\text{printf("\%f",(float)(5/2))} & \rightarrow 2.000000 & \text{OK} \\
\text{printf("\%f",5/2.0)} & \rightarrow 2.500000 & \text{OK} \\
\text{printf("\%d",5/2.0)} & \rightarrow 0 & \text{OOPS (double float data!)} \\
\text{printf("\%d",(int)(5/2.0))} & \rightarrow 2 & \text{OK}
\end{align*}
\]

Similar care must be taken with scanf statements.
In general, the structure of a format specifier is:

\%[parameter][flags][width][.precision][length]type

All elements except “\%” and the type are optional.

Examples:

```c
int ia=12, ib=13;
float fx = 123.456;
printf("%10d %10d\n", ia, ib);
printf("%8.4f\n", fx);
printf("%-d %-d\n", ia, ib);
```

ints printed in 10 columns w/ spaces between.

float printed in 8 columns, 4 numbers after decimal.

int printed left justified.

By default, data are right-justified. The flag “-” causes them to be left-justified.
Some sequences of characters beginning with a `backslash` have a special meaning when used in printf's format string. These are sometimes called "escape sequences".

Here's a list of commonly-used escape sequences. Among other things, these control the cursor on your monitor before/between/after characters are printed.

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Add a new line</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed (new page)</td>
</tr>
<tr>
<td>\b</td>
<td>Move back one character</td>
</tr>
<tr>
<td>\r</td>
<td>Go to beginning of line</td>
</tr>
<tr>
<td>\t</td>
<td>Go to next tab stop</td>
</tr>
<tr>
<td>\a</td>
<td>Ring the bell</td>
</tr>
<tr>
<td>\</td>
<td>Print the character \</td>
</tr>
<tr>
<td>&quot;</td>
<td>Print the character &quot;</td>
</tr>
</tbody>
</table>

Some usage examples:

```c
printf("This is a line.\nThis is another line\n");
printf("This is a double-quote: \"\n");
```
Who Did More?

Your attention, please:

Both of these men died in the same month of the same year. Steve was largely considered a hero, while Dennis was largely ignored by the world. Only a handful of programmers who know the real value of Dennis’ work even know of his death.

Without Steve Jobs there is no iPhone, iPad, iPod or Macintosh. Without Dennis there is no C. Without C, there is no Unix, Windows, or Linux. Without C there is no C++ nor Objective C. There is no MacOS X, no iOS, no Photoshop, no FLStudio, no Firefox, no Safari, no Google Chrome, no Playstation, no XBox. In fact, 90% of the applications in the world are written in C or C++ or Objective C. If you think Dennis deserves our respect, please pass this along.
We’ll pick up from here next time.

Don’t forget lab on Thursday!
See you then.