Reminder

• HW02 due Thursday 16 Feb electronically by noon

• My office hours:
  – 3:30-5pm Tuesdays in Room 022-C (our computer lab)

• TA office hours
  – In Room 022-C
    • Mondays 7-9:30pm
    • Tuesdays 4:30-6:30pm
    • Wednesdays 7-9:30pm

• Grading of HW01 coming!
Review and Outline

• Last time:
  – C program structure
  – Intro to a basic C program
  – Defining simple variables and doing arithmetic
  – Formatted input/output via printf() and scanf()
  – How variables’ values are stored on a computer
  – More on formatted I/O

• Today:
  – Operators
  – Functions
  – Conditional structures:
    • if, if/else, if/elseif/else
    • switch/case
  – Loops
    • Count-controlled loops: for()
    • Conditioned controlled loops: while(), do
  – Random numbers
  – Scope of variables
Storing Variables
• 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
    • \(1_{d} = 1_{b} = 1 \times 2^{0}\)
    • \(5_{d} = 101_{b} = 1 \times 2^{2} + 0 \times 2^{1} + 1 \times 2^{0}\)
    • \(12_{d} = 1100_{b}\)
    • \(21_{d} = 10101_{b}\)
    • \(182_{d} = 10110110_{b} \rightarrow 10110110\)
Storing Variables

- Recall last time we introduced the most-common C variable types:
  - int
  - float and double
  - char

- 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
For an integer we have up to 4 bytes = 32 bits of storage

So the variable “num” defined as this:

```c
int num = 90937759;
```

is stored this way:

![Binary representation of 90937759](image)

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.**
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>Type</th>
<th>Representation</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int</strong></td>
<td>000000100 000000000</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td>000000000 000000000</td>
<td></td>
</tr>
<tr>
<td><strong>float</strong></td>
<td>000000000 000000000</td>
<td>4 bytes</td>
</tr>
<tr>
<td></td>
<td>100000000 010000000</td>
<td></td>
</tr>
<tr>
<td><strong>char</strong></td>
<td>00110100</td>
<td>1 byte</td>
</tr>
</tbody>
</table>
Formatted Input / Output

I/O
I/O Format Specifications

• Recall from lab03, you used `printf()` to print values to screen
• Similar function `scanf()` is used to read in values from the user:

```
Here are some more examples:

printf("%d\n", im_an_int); // print an integer
scanf("%f\n", &a_float); // read a float
```

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

```
<table>
<thead>
<tr>
<th>Some common format specifiers:</th>
</tr>
</thead>
<tbody>
<tr>
<td>i,d,ld,li</td>
</tr>
<tr>
<td>f,lf</td>
</tr>
<tr>
<td>e,E</td>
</tr>
<tr>
<td>g,G</td>
</tr>
<tr>
<td>c,s</td>
</tr>
</tbody>
</table>
```

“i” and “d” are similar but very different. Use “d” for integers until we cover in more detail.
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 with 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.
Controlling the Appearance of Output: Escape Characters

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>&quot;</td>
<td>Print the character &quot;</td>
</tr>
<tr>
<td>\</td>
<td>Print the character \</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");
```
output v. input

- Essentially all of the lessons about controlling output can be applied to controlling/designing **input** to your program.

- `printf()` and `scanf()` are not very different:
  - `printf()` – prints output to screen
  - `scanf()` – accepts input from the keyboard

- These are examples of *interactive I/O*
Interactive v. Passive I/O

• What about passive I/O?
  – Passive I/O = use of files

  – One can write output to a file:
    • fopen(), fclose(), and fprintf()

  – One can also accept input from a file:
    • fopen(), fclose(), and fscanf()

  – Some experience in last week’s lab03
Operators
Operator Types

• What kind of operations do we need to be able to do in a computer program?

• Here are some:
  – Arithmetic (e.g., +, -, *, / )
  – Assignment (e.g., = )
  – Increment / decrement (e.g., ++, --) used in counters
  – Logical / Conditional (e.g., &&, || )
  – Bitwise operations…
  – Pointer operations…
Arithmetic Operators

C and C++ support the following arithmetic operators. Note that some operators are binary (operating on two numbers) and some are unary (operating on one number).

**Binary Operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>a+b</td>
</tr>
<tr>
<td>-</td>
<td>a-b</td>
</tr>
<tr>
<td>*</td>
<td>a*b</td>
</tr>
<tr>
<td>/</td>
<td>a/b</td>
</tr>
<tr>
<td>%</td>
<td>a%b</td>
</tr>
</tbody>
</table>

**Unary Operators:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-a</td>
</tr>
</tbody>
</table>

Arithmetic inverse
The simplest assignment operator is “=”, which is used to set the value of a variable equal to some expression (e.g., “a = b”).

C also offers an array of additional assignment operators that combine assignment with the various arithmetic functions:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>a += b</td>
<td>a = a+b</td>
</tr>
<tr>
<td>-=</td>
<td>a -= b</td>
<td>a = a-b</td>
</tr>
<tr>
<td>*=</td>
<td>a *= b</td>
<td>a = a*b</td>
</tr>
<tr>
<td>/=</td>
<td>a /= b</td>
<td>a = a/b</td>
</tr>
<tr>
<td>%=</td>
<td>a %= b</td>
<td>a = a%b</td>
</tr>
</tbody>
</table>

Be careful when you're typing these. It's easy to type “=+” instead of “+=”!
Increment/Decrement Operators

The unary operators `++` and `--` add or subtract 1 from the operand:

**Usage:**

<table>
<thead>
<tr>
<th>Increment</th>
<th>a++ or ++a</th>
<th>→ a = a+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement</td>
<td>a-- or --a</td>
<td>→ a = a-1</td>
</tr>
</tbody>
</table>

Notice that these operators can be used either before or after the variable. Their action differs slightly, depending on which of these is chosen. Here are some examples:

```c
int a = 1;
a++;    // Set a to a+1 before moving to the next line.
++a;    // Set a to a+1 immediately upon entering this line.

x = a++ * 2;  // Set x = a*2, then set a = a+1.

x = ++a * 2;  // Set a = a+1, then set x = a*2.
```

It's best to avoid statements like the last two unless you have a good reason to use them.
Logical / Conditional Operators

These operators test or combine logical expressions. The answer to a test is either true (not 0) or false (0). Any non-zero value is considered true.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>Equality</td>
<td><code>a==b</code></td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
<td><code>a!=b</code></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td><code>a&lt;b</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td><code>a&gt;b</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less or equal</td>
<td><code>a&lt;=b</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater or equal</td>
<td><code>a&gt;=b</code></td>
</tr>
<tr>
<td><code>!</code></td>
<td>Logical NOT. Invert a test or true/false value</td>
<td><code>!a</code></td>
</tr>
<tr>
<td><code>&amp;&amp;</code></td>
<td>Logical AND</td>
<td><code>(a==b) &amp;&amp; (c==d)</code></td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>`</td>
</tr>
</tbody>
</table>
Operator Precedence: Which Comes First?

When we see a statement like this, we know that we should first multiply “b*c” and then add “a”.

\[ x = a + b \times c \]

By convention, multiplication & division **precede** addition & subtraction.

Here’s a table showing the order of precedence of some common operators in C and C++:

<table>
<thead>
<tr>
<th>First</th>
<th>a++, a--, type casts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Second</td>
<td>a*b, a/b, a%b</td>
</tr>
<tr>
<td>Third</td>
<td>a+b, a-b</td>
</tr>
<tr>
<td>Fourth</td>
<td>a&amp;&amp;b, a</td>
</tr>
</tbody>
</table>

Operations of equal precedence are evaluated from **left to right**.

To clarify statements, you can use **parentheses** as needed or split statements into several steps. Make your intentions clear and you’ll be much happier.
Bad Coding Example: I

Don't even think about writing code like this! What does this even do?

```
a=1;
b=2;
c=3;
d=4;
f= ++a + c*d/a++ + b;
```

Here's a better way to do the same thing:

```
a += 1;
f = (a+b) + c*d/a;
a += 1;
```

```
a = 2
f = (2+2) + 3*4/2 = 10
a = 3
```
**Another Bad Coding Example:**

```c
float a = 10.0;
float b = 5.0;
float c;

c = 1 / 2 * a * b;
```

What value does `c` have?

Precedence rules dictate:
1) $1/2 = 0$ by integer division
2) $0*10.0 = 0.0$
3) $0.00*5.0 = 0.0$

So, `c` is **zero!**

Some possible fixes:

```c
float a = 10.0;
float b = 5.0;
float c;

c = 1/2.0 * a * b;
c = 0.5 * a * b;
c = a * b / 2;
```

Of course, these assume that you didn’t really mean to type:

```c
c = 1/(2*a*b);
```
Functions
Introduction to Functions

• C is a simple language

• Its utility is extended through the use of re-usable functions

• Some functions are found in standard libraries like `stdio.h` and `math.h`

• Users can write functions too – for many good reasons!
"Intrinsic" Functions:

While the text makes reference to **intrinsic** functions, it's more accurate to call them the **C Standard Library Functions**.

These are **not part** of the C language, but reside in a library of useful functions that evolved along with C and is now also standardized across compiler distributions and hardware platforms.

To use these functions it was necessary to first include a **header file** so the compiler would recognize their input/output interfaces. The actual code is **pre-compiled** and is **linked** to your source code to make a working program.
C Standard Library Functions

“Intrinsic” Functions:
While the text makes reference to intrinsic functions, it’s more accurate to call them the C Standard Library Functions. These are not part of the C language, but are evolved over time and now also part of the compiler and hardware.

Common C Standard Libraries we will use:
- `stdio.h`
- `math.h`

Best to get in the habit of including these in every program you write!

You don’t even know WHERE these files are on galileo in order to use them.
Functions from the C math library

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

```c
double sqrt(double x); // prototype for sqrt function
```

You'll find a line like this in the math.h header file.

The compiler reads this from `<math.h>`, then 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:

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

This will generate a warning.

This will generate an error.
User-Defined Functions

Why would one want to write their own functions?

- Avoid duplicating the same code many times within a program: streamlined simplicity
- Re-usable functions are easier to maintain and modify
- Functions are portable – can be used in other programs
- Simplicity: code something intricate once and call it via a simple single line rather than multi-line complications
User-Defined Functions: Example Use Case

No functions – repetitive!

#include <stdio.h>
#include <math.h>

int main () {
    double x0 = 0.0;
    double y0 = 0.0;

    double x1 = 1.0;
    double y1 = 2.0;

    double x2 = 4.0;
    double y2 = 1.0;

    double x3 = 3.0;
    double y3 = 0.0;

    double d01 = sqrt( (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) );
    printf("d01 is %f\n",d01);

    double d12 = sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );
    printf("d12 is %f\n",d12);

    double d23 = sqrt( (x3-x2)*(x3-x2) + (y3-y2)*(y3-y2) );
    printf("d23 is %f\n",d23);

    return(0);
}
Three main parts:
Prototype, Definition, and Calls

User-Defined Functions: Use Case Example

Designing a Function:

```c
#include <stdio.h>
#include <math.h>

double distance(double xstart, double ystart, double xend, double yend);

int main () {
    // (coordinates omitted for brevity)...
    double d01 = distance(x0, y0, x1, y1);
    printf ("d01 is %f\n", d01);
    double d12 = distance(x1, y1, x2, y2);
    printf ("d12 is %f\n", d12);
    double d23 = distance(x2, y2, x3, y3);
    printf ("d23 is %f\n", d23);
    return(0);
}
```

Function prototype

To make things better, we can create a new function, called "distance", to calculate the distance.

Using ("calling") the function

We could easily modify the distance function to return, say, travel time (adjusted for a headwind from a given direction!). We'd only need to make the change in one place: the function definition.

```c
double distance (double xinit, double yinit, double xfinal, double yfinal) {
    double d;
    d = sqrt( (xfinal-xinit)*(xfinal-xinit) +
              (yfinal-yinit)*(yfinal-yinit) );
    return(d);
}
```
User-Defined Functions: Use Case Example

Prototype, Arguments and Return:

```c
#include <stdio.h>
#include <math.h>

double distance(double xstart, double ystart,
                double xend, double yend);

int main () {
    // (coordinates omitted for brevity)...
    double d01 = distance(x0,y0,x1,y1);
    printf("d01 is \$f\n",d01);
    
    double d12 = distance(x1,y1,x2,y2);
    printf("d12 is \$f\n",d12);
    
    double d23 = distance(x2,y2,x3,y3);
    printf("d23 is \$f\n",d23);
    
    return(0);
}
```

- The prototype defines the syntax for the function. (What arguments it takes, and what type of data it returns.)
- The names of the arguments in prototype, function call and function definition don’t need to match, but the types do.
- Our function takes four “doubles” as arguments, and returns a double.

```c
double distance (double xinit, double yinit,
                double xfinal, double yfinal) {
    double d;
    d = sqrt((xfinal-xinit)*(xfinal-xinit) +
             (yfinal-yinit)*(yfinal-yinit));
    return(d);
}
```
User-Defined Functions: Reusable?

Making Functions Re-useable:

```c
#include <stdio.h>
#include <math.h>

double distance(double xstart, double ystart, double xend, double yend);

int main () {
    // (coordinates omitted for brevity)...
    double d01 = distance(x0,y0,x1,y1);
    printf ("d01 is %f\n",d01);
    double d12 = distance(x1,y1,x2,y2);
    printf ("d12 is %f\n",d12);
    double d23 = distance(x2,y2,x3,y3);
    printf ("d23 is %f\n",d23);
    return(0);
}

double distance (double xinit, double yinit, double xfinal, double yfinal ) {
    double d;
    d = sqrt((xfinal-xinit)*(xfinal-xinit) +
              (yfinal-yinit)*(yfinal-yinit) );
    return(d);
}
```

Some day, the prototype for your function could be moved into an **external header file**, to be **included** as needed...

and the function itself could be added to your **own library** of functions, for later use. We'll see how to do this later.
We’ll pick up from here next time.

See you Thursday in labs!