Now that we've looked at how an assembly language program runs on a computer, we're ready to move up a level and start working with more powerful programming languages, in particular C.

Why learn C?
- most modern programming languages are based on C
- there is tons of existing (“legacy”) code written in C
- C is a “low-level” language that relies on the programmer's understanding of the hardware on which it runs

Here is a simple C program:

```c
/*
 * Program to add two numbers.
 */
#include <stdio.h>
#include <stdlib.h>
main() {
    int num1 = 0, num2 = 0, sum = 0;
    printf("Please enter a number: ");
    scanf("%d", &num1);
    printf("Please enter another number: ");
    scanf("%d", &num2);
    // calculate the sum and print it
    sum = num1 + num2;
    printf("The sum is %d\n", sum);
}
```

Here’s what’s going on in this program:
- Lines 1 – 3 are a **multi-line comment**. This is text that is ignored by the compiler. It starts with “/*” and finishes with “*/”. It’s good practice to start your program with a comment explaining what it does.
- Lines 5 and 6 are used to **include** other libraries (collections of code) that are used to perform common functions like reading from the keyboard or writing to the display. You will need these two lines in pretty much every C program you write.
- Line 8 is the start of the **main function**, which is the “entry point” or “starting point” of your program. We will see different variants of this line as we go along.
- Line 9 is a **variable declaration**. A variable is a named piece of memory that holds some value as the program executes. When you declare a variable, you specify its name and its type. Here, the type is “int”, meaning a 32-bit signed integer. There are three variables being declared here: “num1”, “num2”, and “sum”. Note that, in C, you **must** declare a variable before you
start using it. Each variable is also **initialized** to 0.

- Line 11 is a **printf** statement. It displays the string (collection of characters) in the display.
- Line 12 is a **scanf** statement. It reads in a value from the keyboard. In this case, the “\%d” means that we intend to read a decimal value (the “d” stands for “decimal”). The “&num1” means that the variable num1 should be set to whatever value is entered from the keyboard. Don’t forget the ampersand (“&”) in front of the variable name! It means “the address of” the variable that follows it; we need it here because scanf is going to change the variable’s value, so it needs its address.
- Lines 14 and 15 are just copies of lines 11-12, except that now we’re getting the second value and putting it in num2.
- Line 17 is a **single-line comment**. It starts with “//” and goes to the end of the line.
- Line 18 is an **assignment statement**. It assigns the variable sum to the value of num1 + num2.
- Line 19 is a printf statement but it is slightly different from lines 11 and 14, as you can see. In this case, we want to include the value of the variable sum in our output. So we need to use the “\%d” placeholder to hold the value, and we specify the variable sum after the string. Note here that we also use “\n” to display a “newline” character, which is the same as hitting the Enter key.

**Compiling a C program**

After you write your C program, you need to compile it. On the surface, this is a pretty simple task: converting the code to an executable. However, what really happens is quite complicated. There are three steps: the preprocessor; the compiler; and the linker.

**Step 1. Preprocessor**

The preprocessor is responsible for checking that the code uses good syntax. That is, that what is in the program is legal C code. For instance:

```
x = a + b
```

is not legal C code because it does not end with a semi-colon. This is a “syntax error”.

The preprocessor also takes care of including header files (that is, when you do “#include <stdio.h>”, the code from the file stdio.h is actually considered to be part of your program) and addressing some other issues that we'll see later in the course.

Last, the preprocessor creates a symbol table based on the variable names. This is a much more complicated task than in LC-3, but we will look at this later, too.

**Step 2. Compiler**

Although we think of this whole process as “compiling”, the compiler is really just the second step in the process. It first checks for “semantic errors”. For instance, consider the line:

```
x = a + b;
```

Even though that line is syntactically correct, it may be semantically incorrect if the variables x, a, and/or b have not been declared. This step is referred to as “analysis”.

The compiler then takes the symbol table created by the preprocessor and converts the C code to assembly language code for the particular instruction set architecture. This is known as “synthesis”.
The output is an “object file” which contains the machine language code for this particular source file.

Step 3. Linker
The last step is to combine the object file into a single executable. That is, in C you may have multiple files that comprise your program, and each will have its own object file. The linker puts them all together.

Variables in C
As you saw in the sample program above, an important part of a C program (or a program in just about any language, for that matter) is the variables that are used to hold data.

As you're learning more about C, keep in mind that a “variable” is just like a “label” in an LC-3 program: it refers to some memory address and is a convenience to the programmer so that he or she doesn't need to know exactly which address the data is stored in.

Datatypes
In C, each variable must have a specific type, which indicates how much storage is used to hold it, and what operations you can perform on it. C supports six main datatypes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8-bit (1-byte) ASCII character</td>
</tr>
<tr>
<td>short</td>
<td>2-byte signed integer</td>
</tr>
<tr>
<td>int</td>
<td>4-byte signed integer</td>
</tr>
<tr>
<td>long</td>
<td>8-byte signed integer</td>
</tr>
<tr>
<td>float</td>
<td>4-byte IEEE floating point</td>
</tr>
<tr>
<td>double</td>
<td>8-byte IEEE floating point</td>
</tr>
</tbody>
</table>

A few important notes:
- unlike Java, there is no “boolean” datatype to indicate true or false (we'll come back to this next time)
- there is also no “string” datatype; we'll see strings later on
- the signed integers use 2's-complement; there is also a type called uint, which is a 4-byte unsigned int.
- the IEEE floating point specification is described in your textbook on page 37

One last important note: for better or for worse, there are many slightly different implementations of C out there, so you may find yourself working in an environment in which an int is 8 bytes, not 4. In this class, we will try to use “standard” C to whatever extent possible, but just keep in mind that there are lots of variants of it.
Variable names
When naming variables, the rule of thumb is that someone reading your code should be able to understand what the variable means/represent just from its name. Although we may use variable names like x, y, a, b, etc. in class, that's just for convenience: try to use meaningful names whenever possible.

There are, however, some actual rules about what is and is not allowed in C:

- variable names can consists of letters (uppercase or lowercase), numbers, and underscores; they cannot, however, start with a number
- variable names are case sensitive, so name and Name are considered two different variables
- a variable cannot have the same name as a “reserved word”, which is a word that has a special meaning in C; some examples are switch, continue, break, and default

Beyond the hard-and-fast rules, there are some “guidelines” that you should try to adhere to and that are generally followed within the C programming community:

- variable names should start with a lower-case letter, e.g. age is good, but Age is bad
- the only time you'd start a variable name with an upper-case letter is if the whole word is upper-case, and that would mean that it's defined as a “constant”
- when you have a variable name that consists of two or more words, you should join them with an underscore instead of using concatenation, e.g. student_name is good, but studentname and studentName are bad (the latter is acceptable in Java, though!)

Declaring variables
Before you begin to use a variable (meaning, assign it a value or read its value), you need to declare it, using one of the datatypes above (or one that you create; we'll see that later, too). Note that you can only declare a variable once; you don't need to re-declare it every time you use it.

Generally, you declare a variable and initialize it (set its initial value) together:

```c
int a = 4;
```

Note that it is possible to declare multiple variables of the same type on one line:

```c
int b = 8, c = 11;
```

Some people don't like that (because it's easier to see all the declarations if each is on a separate line), but technically it's okay.

Another style issue is, at what point in the code do you declare a variable? For instance, you could do this:

```c
int a, b, sum;
scanf("%d\n", &a);
scanf("%d\n", &b);
sum = a + b;
```
or you could do this:

```c
int a, b;
scanf("%d\n", &a);
scanf("%d\n", &b);
int sum = a + b;
```

As you can see, in the first case, all the variables are declared at the beginning; in the second, sum is declared right when it's first used. Both are okay, and there are some good reasons for each; personally I usually end up doing the second but either is fine.
When declaring a floating point number, you can specify the entire number:

```cpp
float k = 12.3456789;
```
or use exponent notation:

```cpp
float m = 6.42E-28;
```
This means that the variable `m` is initialized to `6.42 * 10^{-28}`, which of course is a lot easier than writing

```cpp
float m = 0.000000000000000000000000000642;
```

For declaring chars, you can use their ASCII value or the actual character:

```cpp
char c = 88;
char k = 'X';
```
Since the ASCII value of 'X' is 88, both of these have the same value.

Note that when declaring a character, you use single quotes, i.e. 'X' not “X”. Double-quotes are for strings, which we'll see later.

---

**One last super-important thing about declaring variables!!**

A variable declaration like this is perfectly legal:

```cpp
int x;
```
But what is the value of `x`? The answer is “it depends”. Sometimes it will be initialized to 0, sometimes its value will be undefined! Later on we'll answer the question “depends on what?” but for now, assume that the value of `x` is undefined if it's not initialized when it's declared. *In general, you should always specify an initial value!*  

---

**Casting**

In some cases, we need to temporarily change the type of a variable before performing some operation on it. This is known as *casting* and it is very important when doing operations on variables of different types.

For instance, let's say we have the following code:

```cpp
int x = 5;
float y = 7.9;
float sum1 = x + y;
```
What is the value of `sum1`? In this case, the variable `x` is temporarily considered to be 5.0, because when you add an `int` to a `float`, the result is a `float`. Thus, the right side of the assignment will evaluate to 12.9, and that's the value that will be put into `sum1`.

Okay, what about this?

```cpp
int a = 5;
float b = 7.9;
int sum2 = a + b;
```
Now what is the value of `sum2`? In this case, we still have `5.0 + 7.9 = 12.9` on the right side. However, the variable `sum2` cannot hold the value 12.9 because `sum2` is an `int`, not a `float`. So 12.9 is *truncated* (not rounded!!) to 12, which is the value that is put into `sum2`.

In this case, even though the sum of `a + b` is a float, it is *cast* to an integer in order to to store it in `sum2`. This is known as an “implicit cast”; the programmer didn't explicitly say to do it, but it just
happens automatically.

Alternatively, we could temporarily change the type of the float variable before doing the addition:

```c
int s = 5;
float t = 7.9;
int sum3 = s + (int)t;
```

Now, the variable `t` is temporarily changed ("cast") to an int, specifically the value 7. So now the right-hand side becomes \(5 + 7 = 12\), which is the value stored in `sum3`.

It seems like it's the same thing, right? But it's not! Check this out:

```c
int q = 5;
float r = 7.9;
float sum4 = q + (int)r;
```

What happens here is that `r` is cast to an int, making it 7. It's added to 5, which gives us 12 on the right-hand side. Now, when we try to put 12 into `sum4`, it becomes the float 12.0. Note the difference between this example and the first one, in which `sum1` is 12.9.

**Formatted Printing**

As you saw in the example at the start of this document, you print to the console by using the command `printf`. Here is a bit more about how that works:

When you call `printf`, you need to give it a “format string” that specifies what you want to print. In that format string, you have placeholders for any variables.

As we saw before, here's how you'd print an int:

```c
int x = 9;
printf("the value of x is %d\n", x);
```

The placeholder “%d” means “print it as a decimal”. You could also have multiple placeholders, like this:

```c
int x = 9, y = 14;
printf("the value of x is %d and the value of y is %d\n", x, y);
```

To print a character, you use the “%c” placeholder:

```c
char k = 'M';
printf("the character is %c\n", k);
```

Note that you could also print the character like this:

```c
char k = 'M';
printf("the value is %d\n", k);
```

What will happen? It will treat `k` not as a char, but as an int. What is its value as an int? It's the ASCII value of 'M', of course, which is 77. Why does this happen? Because the value held at the address labeled `k` is 01001101 (the binary representation of 77). When you tell `printf` to print it as a char, it will print 'M'; but when you tell `printf` to print it as an int, it will print 77.

So, what happens when you do this?

```c
int y = 63;
printf("the value is %c\n", y);
```

The answer is left as an exercise for the reader!
One last thing: printing a float. You can control the precision (number of places after the decimal point) using the placeholder, like this:

```c
float s = 12.748;
printf("s is %10.2f\n", s);
```

The number 10 in the placeholder means “use 10 spaces to display this number”. The 2 after the decimal point means “display two values after the decimal point”. So the output (including the extra spaces) would be:

`s is      12.748`