In Java all sources are compiled down to Java Byte Code. Java Byte Code is a made up language that runs in the Java Virtual Machine (JVM). The JVM itself is implemented in a different language that can run directly on the machine.

C on the other hand compiles down to a language that can run directly on the machine. This means that if you pass C code that has been compiled on one machine to a different machine it may not run. (unless the machines share the same architecture / processor instruction set).

The C build process is broken down into four steps, the processing step, the compiling step, and the assembling step. We use gcc, the GNU Compiler Collection to run through these four steps.

The preprocessing step runs the preprocessor on the source files. The preprocessor is a simple find and replace utility that is independent of the C language.

To interact with the preprocessor we use preprocessor directives (sometimes called macros). All preprocessor directives are preceded by a #. Keep in mind, preprocessor directives are not a part of C, so they do not end with a ;

#include is a very powerful preprocessor directive. It allows us to include whole other files at the top of our code. When the preprocessor sees this directive it replaces it with the contents of a specified file. Here is an example of its syntax:

#include <stdio.h>

Using the #include preprocessor directive we can effectively import files, like would in java.

If we run gcc with the –S flag, it will run the preprocessor and then the compiler. The compiler will convert our sources into assembly source code. Assembler languages are specific to the hardware of the machine. Being able to compile down to assembly code makes C very fast, but also prevents compiled code from being swapped between machines. (Many C programs have to be rebuilt locally before they can be run)

If we run gcc with the –c flag it will run the preprocessor, then the compiler, and then the assembler. The assembler converts the assembly source code into object code.

The final step of the build process is the linking step. In the linking step the linker takes the separate object files that the assembler has created and stitches them together. The linker only runs if we pass multiple files into gcc. For example:

gcc file1.c file2.

The result of this call is an executable called a.out.

Remember #include <stdio.h> ?

This statement includes stdio.h at the top of our file.

stdio.h is what we call a header file (because it goes at the head of our code).

stdio.h defines some standard input output functions (as well as a few other things). Using these I/O functions is how we handle printing to the screen, reading input from the user, and reading and writing to files.

To print to the screen we use printf (which stands for print formatted). This function takes a string and prints it to the command line. For example:

printf("hello world\n");

Notice the \n. This is called an escape sequence. It allows us to print things that we could not normally print. Here are some common escape sequences.

\n for a new line (like System.out.println() in Java), \' for ' , and \" for "

printf also allows us to use format specifiers. These are special strings which allow us to print out the values of variables. For example:

```
int myInt=5;
printf("%i",myInt);
```

Some common format specifiers are:

%i for int, %c for char, %f for float, and %d for double. I will introduce more format specifiers as we work through the course.

To read input from the user we use scanf (scan formatted). This function takes a format string (just like printf) and a series of variables' addresses. For the moment don't worry about what a variable address is, for now you can pretend that they're just variables with a & in front. For example:

```
int myInt;
scanf("%i",&myInt);
```

This will wait for the user to input a number, and then it will make myInt equal to whatever number they enter.

To open a file we use the fopen function. This function takes the name of the file and the file mode. The file mode is one of “r”: for reading, “w”: for writing, and “a”: for appending to the end of the file. It is also worth noting that fopen returns a FILE pointer. Don’t worry about what a pointer is yet, for now you can treat it as though it is just a FILE with a * after it. FILE is a datatype defined in stdio.h, it represents a file. For example:

```
FILE * readingFile = fopen(“input.txt”,“r”);
```

Now that we’ve opened a file we can read and write to it (depending on the file mode). To write to a file we use fprintf. This is exactly like printf except that it takes a FILE * as its first argument. For example:

```
FILE * writingFile = fopen(“output.txt”,“w”);
fprintf(writingFile, “hi”);
```

To read from a file we use fscanf. This is exactly like scanf except that it takes a FILE * as its first argument. For example:

```
FILE * readingFile = fopen(“input.txt”,“r”);
int myInt;
fscanf(readingFile,“%i”, &myInt);
```

fscanf also returns a value. When it has reached the end of the file fscanf returns EOF (which stands for End Of File). This is a value defined in stdio.h.

Finally, when we’re done with a file we need to close it, just like we would close a Scanner in java. We can close files by using the fclose function. For example:

```
FILE * myFile = fopen(“myfile.txt”,“r”);
fclose(myFile);
```

References

Commands

gcc: runs the GNU Compiler Collection, converting C source files into an executable.

gcc –E: runs gcc, but stops after the preprocessor

gcc –S: runs gcc but stops after the compiler

gcc –o: runs gcc but stops after the assembler

gcc –c PROGRAM: runs gcc and renames a.out to PROGRAM

Functions

Included in stdio.h

int printf(const char * format, …)

prints the format string with format specifiers swapped out for the passed arguments.
Returns the number of characters printed.

int scanf(const char * format, …)

scans an input string with the expected format. The values entered in place of format specifiers are read into the variables passed in. You must pass the address of a variable by using &.
Returns the number of variables read.

FILE * fopen(const char * filename, const char * filemode)

Opens the file given by filename in the given filemode. The three filemodes are “r” for reading, “w” for writing, and “rw” for reading and writing.
Returns a pointer to the file.

int fclose(FILE * file)

Closes a file that was opened with fopen.
Returns 0 on success, EOF on failure.

int fprintf(FILE * file, const char * format, …)

Like printf but prints to a file opened with “w” or “rw” filemode.

int fscanf(FILE * file, const char * format, …)

Like scanf but reads from a file.
Returns the number of variables read in or EOF if the end of the file is reached.

Format specifiers

%i – int

%c – char

%f – float

Escape sequences

\n – new line

\’ – ‘

\” – “

\\ - \