Lab 01 - Introduction to C Programming

Objectives

1. Write your first C program.
2. Learn how to build your C source code into an executable program.
3. Learn how to use the **`make`** utility to automate the build process.

Pre-reading

The following material contains commands and/or other information that could be essential to completing this and future labs. Students are strongly encouraged to take a look at the material below *before* coming to the lab, since knowing this material before you start will likely make task of completing the lab much faster and easier.

Web Links

`make` Manual: http://www.gnu.org/software/make/manual/make.html

A Simple "Hello World" Example

The following is a simple "Hello World" program written in the C programming language (the language we will use in this course).

Program: Hello World
Version: 1.0

File: helloworld.c

```c
#include <stdio.h>


/* Prints "Hello World!" to the console. */
void printLine1() {
      printf ("%s\n", "Hello World!");
}


/* Prints "This is Computer Science 2280!" to the console. */
void printLine2() {
      printf ("%s\n", "This is Computer Science 2280!");
}


/* The main function. */
int main() {
      printLine1();
      printLine2();
      return 0;
}
```

Function Declarations

When calling a function, you must always keep the following rule in mind:

Within a source (.c) file, a function must either be *implemented* or *declared* before it can be called.

That is, a function cannot be called unless it has been implemented or declared at some point within that file *before* the function call. For example, in Hello World version 1.0, the implementations of the functions `printLine1()` and `printLine2()` appear before they're called.

When programming in C, you have the ability to call functions that are implemented in the *same file* as well as functions that are implemented in *other files*. If you're calling a function from within the *same file*, then you **declare** the function as follows:

```
void myfunction(int x);
```

On the other hand, if you're attempting to call a function that is implemented within some *other file* (such as the file **math.o** in the Standard C Library), then you **declare** the function using the **`extern`** keyword as follows:

```
extern double cos(double x);
```

In general, you should include a list of function declarations at the top of every source (**.c**) file you write. This list should contain a declaration for *every* function that you call in the given file.

Multiple Source (.c) Files

As you progress through the course, your programs will quickly become larger and more complex. As a result, attempting to store all of your program's code in a one source (**.c**) file may become messy. If this happens, you may find it helpful to divide your functions into groups, and place each group in a separate file. That is, your source code can be split into multiple files, where each file contains a *subset* of the original functions.

The set of all functions stored within a given file form the **interface** for that file. That is, if you want to execute *any* of the code stored within that file, your only choice is to call one of the functions presented in its interface. Hence, after splitting a file, the *interface* of each resulting file is smaller and more manageable than that of the original file.

Header Files

A **header (.h) file** stores the complete list of *function declarations* for *every* function that is implemented within a particular file. That is, a header file defines the *interface* for a given file. The name chosen for a header file is the same as that of the file whose interface it defines, but with a .h extension (for example, **foo.h** and **foo.c**).

If you have a source (**.c**) file that calls one or more functions that are implemented within the file **foo.c**, then *instead of* declaring each of these functions at the top of your source code, you can simply **include** the header file for **foo.c** as follows:

```
#include "foo.h"
```

If the file that contains the implementations of the functions that you wish to call is in the Standard C Library (for example, **math.o**) then you must use the syntax:

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

For example, the code within **helperfunctions.c** (see the example below) makes multiple calls to the **`printf()`** function, which is stored in the file **stdio.o** in the Standard C Library. Hence, **helperfunctions.c** includes the corresponding header file **stdio.h** at the beginning of the file. Of course, you could just declare the function manually using the **`extern`** keyword instead, as in the preceding section.

During the build process, each of the **`#include`** statements at the top of your source code file are replaced by the contents of the specified header file. Hence, including a header file full of function declarations has the exact same effect as declaring each of those functions yourself.

Note: all functions defined in a header should use the **`extern`** keyword. When this keyword is specified, the task of verifying that the implementation corresponding to a function declaration actually exists is performed by the **linker**. Otherwise, it is performed by the **compiler**. After the **`#include`** statements are replaced by function declarations from the specified header files, the **compiler** verifies that a function implementation exists for every declaration that appears at the top of the modified source code file. However, the **compiler** only looks for the implementation within that particular file, and no others. Hence, if you're calling a function whose implementation is within another file, then the compiler will not find it and will cancel the build. Using the **`extern`** keyword, you can pass this responsibility to the **linker**, which looks for the implementation in every file included in the link, instead of just the file containing the declaration. Hence, the **`extern`** keyword is required for functions that reside in another file. For functions in the same file, including the **`extern`** keyword makes no difference (it doesn't matter whether the linker or the compiler verifies the existence of its implementation).

Hello World Version 2.0

For our hello world example, we can split the functions into two groups: the main function and the helper functions. The resulting program is presented below.

Notice the file **helperfunctions.c** does not **#include** the file **helperfunctions.h**. Why do you think that is (the answer is on the bottom of the page)?

Program: Hello World
Version: 2.0

File: helloworld.c

```c
#include "helperfunctions.h"

/* The main function. */
int main() {
	printLine1();
	printLine2();
	return 0;
}
```

File: helperfunctions.h

```c
#ifndef HELPER_FUNCTIONS_H_
#define HELPER_FUNCTIONS_H_

/* Prints "Hello World!" to the console. */
extern void printLine1();

/* Prints "This is Computer Science 2280!" to the console. */
extern void printLine2();

#endif
```

File: helperfunctions.c

```c
#include <stdio.h>

void printLine1() {
	printf ("%s\n", "Hello World!");
}

void printLine2() {
	printf ("%s\n", "This is Computer Science 2280!");
}
```

You don't need to declare a function at the beginning of a source (**.c**) file if your code doesn't call it. Hence, **helperfunctions.c** does not include the header file **helperfunctions.h** because it doesn't actually *call* any of those functions.

Building an Executable Program

Step 1: Preprocess

Use the **preprocessor** to produce preprocessed C code (.i) files. Preprocessed C code is a modified form of the original C code where all of the macros (the lines starting with a # character) have been replaced with actual C code the compiler can understand. In particular, the macro **`#include`** is replaced by the contents of the specified header (.h) file. The command to use is:

```
gcc –E helloworld.c –o helloworld.i
gcc –E helperfunctions.c –o helperfunctions.i
```

Step 2: Compile

Use the **compiler** to produce an assembly (.s) file, which is an assembly language translation of the program. The command to use is:

```
gcc –S helloworld.i –o helloworld.s
gcc –S helperfunctions.i –o helperfunctions.s
```

Step 3: Assemble

Use the **assembler** to produce an object (.o) file, which is a machine language translation of the program. The command to use is:

```
gcc -c helloworld.s –o helloworld.o
gcc -c helperfunctions.s –o helperfunctions.o
```

Step 4: Link

Use the **linker** to read the object files that the machine language translation of each function used in this program, and merge the code into a single executable file. Object files used from the standard C library are *automatically* included in the linking process. The command to use is:

```
gcc helloworld.o helperfunctions.o –o helloworld
```

Note: The **`gcc`** options **`-E`**, **`-S`**, and **`–c`** tell the **`gcc`** program to stop the build process after completing the preprocessor step, the compiler step, and the assembler step, respectively. If you do not specify one of these options, then **`gcc`** will perform all of the steps without stopping. That is, all of the steps can be completed by a single command:

```
gcc helloworld.c helperfunctions.c –o helloworld
```

Some Useful gcc Command-Line Options

`-pedantic`	This will cause **`gcc`** to **strictly** enforce every rule in the standard C language.
`-Wall`	This will enable all **`gcc`** warning messages. Warnings are not errors, but if you pay attention to them, they can help you produce better code.
`-o`	Allows you to specify the name of the output file.

Preprocessor:

- Replaces all macros (lines starting with the # character) with actual C code that can be compiled.
- Replaces each #include directive in the C source code with the contents of the corresponding header file, so that compiler will know the exact list of available functions. If the code contains a call to a function that is not in the above list, then the compiler returns an error (function does not exist).

Compiler:

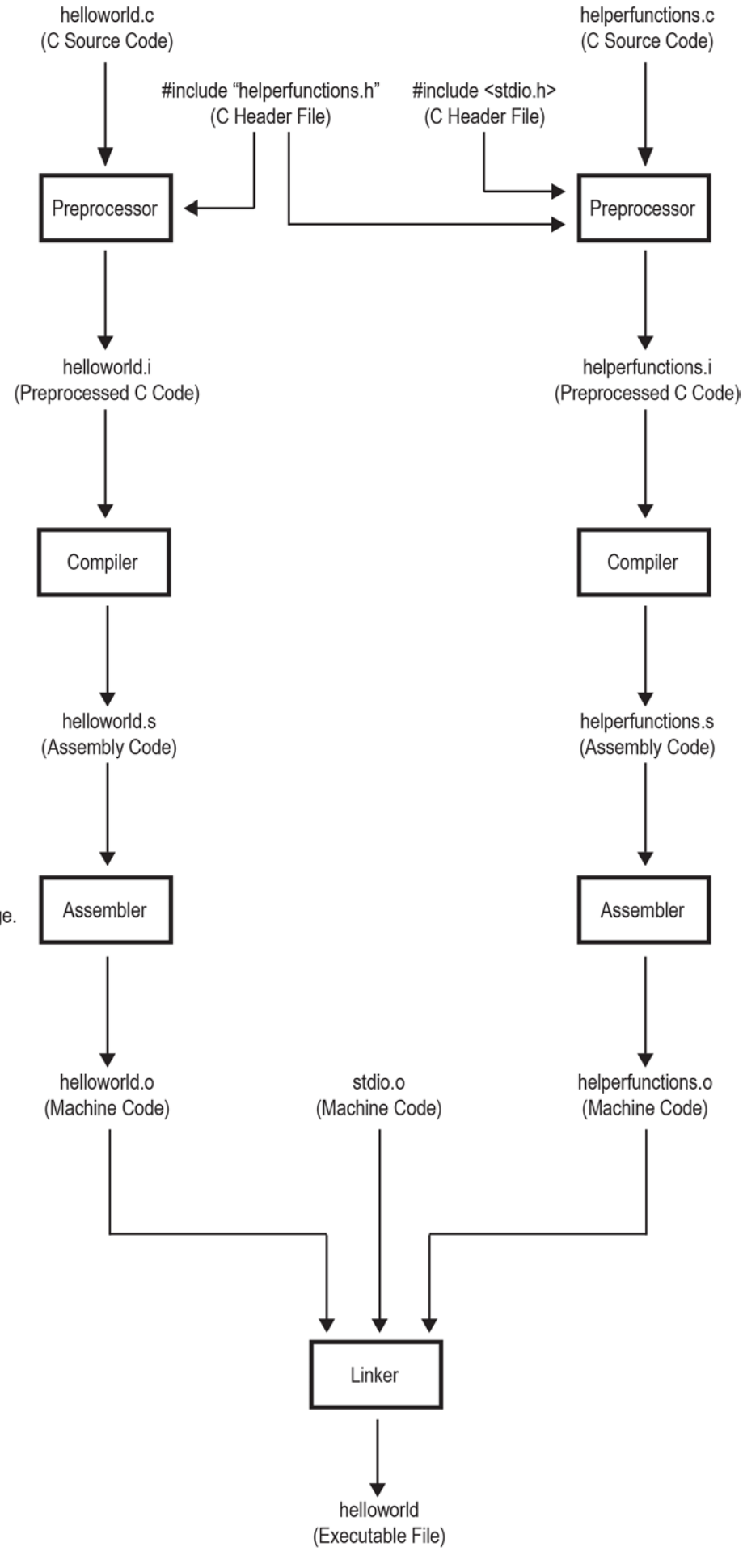
- Translates the preprocessed C code into assembly language.
- Assembly language is a low-level (but still human-readable) programming language in which each instruction has a one-to-one correspondence to a machine language instruction.

Assembler:

- Translates the assembly code into machine language.
- Therefore, each original C source (.c) file now has a equivalent machine language (.o) file.

Linker:

- Merges all of the machine code from each object file into a single executable file.
- If a function from the standard C library was used, the corresponding object file must be included here as well.



The Definition of the Word "Compile"

The *actual* definition of the word ***compile*** is:

The process of translating code into another language.

In other words, a compiler is programming language ***translator***. It converts code written in one language into ***equivalent*** code in another language.

Recall the definition used by **`gcc`** (as well as most other build systems):

The process of translating pre-processed code into assembly code.

As you can see, the **`gcc`** definition is satisfies the general definition. It translates pre-processed code into its ***equivalent*** assembly language code.

One very popular way to use the word ***compile*** is: *the process of translating source code into machine code*. This popular use of the word is the combination of the pre-process, compile, and assemble steps from **`gcc`**. This is a good definition to follow when you're writing your makefiles (read the section on the **`make`** utility below).

The GNU Compiler Collection (gcc)

Why would we want to build a program at each step separately? That is, why not just use the command

```
gcc helloworld.c helperfunctions.c –o helloworld
```

and be done with it? Well, here are a few common scenarios where separating the build steps might be necessary:

1. We want to modify the code at some intermediate step. For example, we could preprocess and compile the source code, then make changes to the resulting assembly language files.

2. We want to link in object files from a third party. For example, there are a countless number of free and proprietary programming libraries online, each with a large set of functions that you can call to perform a given task. These libraries often consist of just a collection of object files and their associated header files. In order to use their functions in your program, you must specify their object files to the linker manually. Remember: by default, the linker only includes the object files produced by **`gcc`** during that *same* build and the object files in the Standard C Library.

3. We have source files written in ***different*** high-level programming languages, and we wish to **combine** them into a *single* executable program. As it turns out, the **`gcc`** utility can be used to build more than just C programs. In fact, **`gcc`** comes with a collection of pre-processors & compilers for a variety of programming languages.

   Hence, if we have multiple source code files, each written in any one of a variety of high-level languages, we can use **`gcc`** to:

   1. Individually translate (*pre-process* + *compile* + *assemble*) each source code file into its equivalent machine code (i.e. object) file.

   2. Link the object files together to produce a *one executable program*.

   For example, say we have a source file written in C and another source file written in FORTAN, and we want to *call* one of the functions we wrote in FORTRAN from within our C code. In the C source file, we need to declare the function we wish to call using the **`extern`** keyword at the top of the file, like we usually do when a function's implementation resides in another file. Then we simply preprocess, compile, and assemble the C source file using **`gcc`**, then do the same for the FORTRAN source file (also using **`gcc`**), then use **`gcc`** to link the resulting object files into the final executable.

   The **`gcc`** utility currently supports C, C++, Objective-C, Objective-C++, FORTRAN, Java, Ada, and Go among others. Some (but not all) of these languages can be built and linked together like in the above example.

Using the Make Utility to Automate the Build Process

The **`make`** utility allows you to clearly define each step in the build process for every file in the project. That is, the build steps can be customized for each file. Furthermore, to save time, the **`make`** utility will only re-build a source file if it has been updated since the previous build.

The build rules are stored in a text file called **`makefile`** (no file name extension). Once the make file is written, you can build the project simply by calling the **`make`** utility. The **`make`** utility will search the current directory for the make file, and then execute its instructions.

A **rule** in a make file has the general form:

```
target: dependencies
    command
```

where

- **`target`** is the name of the output file to be produced.
- **`dependencies`** is a list of source files used by the command from which to build the target file.
- **`command`** is the command to be executed by make in order to build the target file from the given source files.

The **`make`** utility will only execute a rule if one or more dependencies have been modified since the target file was last built.

In my opinion, **`make`** is one of those commands where it's easier to learn by example, rather than studying a manual or long tutorial. Hence, below is a ***ridiculously*** detailed makefile for building the Hello World Version 2.0 program. This make file is overkill. It specifies a rule for every step in the build process (preprocess → compile → assemble → link). Nonetheless, it's a good example to study. Look at it carefully, and try to understand how the make file works. It's surprisingly easy.

File: makefile

```
#
# Default rule to execute when the make command has no arguments
#
all: helloworld


#
# preprocessor step (produces preprocessed C code files)
#
helloworld.i: helloworld.c helperfunctions.h
	gcc -E helloworld.c -o helloworld.i -Wall -pedantic

helperfunctions.i: helperfunctions.c
	gcc -E helperfunctions.c -o helperfunctions.i -Wall -pedantic


#
# compiler step (produces assembly code files)
#
helloworld.s: helloworld.i
	gcc -S helloworld.i -o helloworld.s -Wall -pedantic

helperfunctions.s: helperfunctions.i
	gcc -S helperfunctions.i -o helperfunctions.s -Wall -pedantic


#
# assembler step (produces object files)
#
helloworld.o: helloworld.s
	gcc -c helloworld.s -o helloworld.o -Wall -pedantic

helperfunctions.o: helperfunctions.s
	gcc -c helperfunctions.s -o helperfunctions.o -Wall -pedantic


#
# linker step (produces final executable file)
#
helloworld: helloworld.o helperfunctions.o
	gcc helloworld.o helperfunctions.o -o helloworld -Wall -pedantic


#
# The command-line 'make clean' executes the following command
# (removes all files created during build).
#
clean:
	rm -f helloworld helloworld.o helloworld.s helloworld.i
	rm -f helperfunctions.o helperfunctions.s helperfunctions.i


#
# Lists the "phony" rules in this file ('all' and 'clean').
#
.PHONY: all clean
```

In most cases, you can merge the first three steps into one, and then link the result (preprocess + compile + assemble → link). As mentioned in an earlier section, you could call this combined step the ***compiler*** step, since it translates source code into machine code and thus satisfies the general definition of the word *compile*. This is shown in the following make file. **This is what you should base your make files on.**

File: makefile

```
#
# Default rule to execute when the make command has no arguments
#
all: helloworld


#
# Preprocessor + compiler + assembler step (produces object files)
#
helloworld.o: helloworld.c helperfunctions.h
	gcc -c helloworld.c -o helloworld.o -Wall -pedantic

helperfunctions.o: helperfunctions.c
	gcc -c helperfunctions.c -o helperfunctions.o -Wall -pedantic


#
# linker step (produces final executable file)
#
helloworld: helloworld.o helperfunctions.o
	gcc helloworld.o helperfunctions.o -o helloworld -Wall -pedantic


#
# The command-line 'make clean' executes the following command
# (removes all files created during build).
#
clean:
	rm -f helloworld helloworld.o helperfunctions.o


#
# Lists the "phony" rules in this file ('all' and 'clean').
#
.PHONY: all clean
```

Procedure

Suppose that you have written a program consisting of 3 source files:

`main.c`
`f1.c`
`f2.c`

and two header files

`f1.h`
`f2.h`

All 3 source files include **`f1.h`**, but only **`f1.c`** and **`f2.c`** include **`f2.h`**. Write a make file for this program, assuming that the compiler is **`gcc`** and that the executable file is to be named **`demo`**.

Review Questions

1. **(10 marks)**
   According to your make file, what files are created when the program is built for the first time?

2. **(10 marks)**
   If **`f1.c`** is changed after the program has been built, which object file(s) will need to be updated?

3. **(10 marks)**
   If **`f1.h`** is changed after the program has been built, which object file(s) will need to be updated?

4. **(10 marks)**
   If **`f2.h`** is changed after the program has been built, which object file(s) will need to be updated?

Deliverables

1. **(60 marks)**
   The make file you wrote for the **`demo`** program.
2. **(40 marks)**
   Answers to the above review questions in a file called **answers.txt**.