**CPSC 1071 -- Lab 8   
Conditonal Compilation   
Makefiles**

**Lab Objectives**

* Gain some experience in using conditional compilation for your programs
* Gain some experience with Makefiles

**Preprocessor Commands**

The C compiler is made of two functional parts: a preprocessor and a translator. The preprocessor uses programmer-supplied commands to prepare the source program for compilation. You have already used one preprocessor command: file inclusion, e.g.

#include   
#include "filepath.h"

All preprocessor commands start with a hash mark (#).

There are essentially three uses of the preprocessor: directives, constants, and macros. Directives are commands that tell the preprocessor to skip part of a file, include another file, or define a constant or macro. Directives always begin with a hash mark (#) and for readability should be placed flush to the left of the page. All other uses of the preprocessor involve processing #define'd constants or macros. Typically, constants and macros are written in ALL CAPS to indicate they are special. We will not cover macros in this lab.

**Header Files**

The #include directive tells the preprocessor to get the text of a file and place it directly into the current file. Typically, such statements are placed at the top of a program, hence the name "header file" for files thus included.

**Constants**

If we write

#define [identifier\_name] [value]

then, whenever [identifier\_name] shows up in the file, it will be replaced by [value].

If you are defining a constant in terms of a mathematical expression, it is wise to surround the entire value in parentheses:

#define PI\_PLUS\_ONE (3.14 + 1)

By doing so, you avoid the possibility that an order of operations issue will destroy the meaning of your constant:

x = PI\_PLUS\_ONE \* 5;

Without parentheses, the above would be converted to

x = 3.14 + 1 \* 5;

which would result in 1 \* 5 being evaluated before the addition, not after. Oops! It is also possible to write simply

#define [indentifier\_name]

which defines [identifier\_name] without giving it a value. This can be useful in conjunction with another set of directives that allow conditional compilation

**Conditional Compilation**

There are several options that can be used to determine whether the preprocessor will remove lines of code before handing the file to the compiler. They include #if, #elif, #else, #ifdef, and #ifndef . An #if or #if/#elif/#else block or a #ifdef or #ifndef block must be terminated with a closing #endif.

The #if directive takes a numerical argument that evaluates to true if it is non-zero. If its argument is false, then code until the closing #else , #elif , of #endif will be excluded.

**Example 1:**

An application of conditional compilation is found in debugging. Most compilers have a debugger; however, if there is not a debugger, the programmer can insert *printf* statements in the code to check the values of variables during program testing. If the programmer simply inserts the *printf*statements as part of the code, the statements will have to be removed or commented out before the program is submitted.

A simpler solution is to insert them conditionally by enclosing the *printf* statements inside the ***#if ... #endif*** commands to print the debug statements only when we run in "debug" mode.

For example, the following code prints the value of x only when DEBUG is true:

#define DEBUG 1

#include <stdio.h>

int main()

{

int x = 1155;

#if DEBUG

fprintf(stderr, "%d\n");

#endif

**Avoiding including files multiple times**

Another use of conditional compilation is to avoid duplicate names. A common problem is that a header file is required in multiple other header files that are later included into a source code file, with the result often being that variables, structs, classes or functions appear to be defined multiple times (once for each time the header file is included). This can result in a lot of compile-time headaches. Fortunately, the preprocessor provides an easy technique for ensuring that any given file is included once and only once. By using the #ifndef directive, you can include a block of text only if a particular expression is undefined; then, within the header file, you can define the expression. This ensures that the code in the #ifndef is included only the first time the file is loaded.

#ifndef FILE\_NAME\_H #define FILE\_NAME\_H

/\* code \*/

#endif // #ifndef FILE\_NAME\_H

Notice that it's not necessary to actually give a value to the expression FILE\_NAME\_H. It is sufficient to include the line "#define FILE\_NAME\_H" to make it "defined". (Note that there is an *n* in #ifndef -- it stands for "if not defined").

**Common Conditional Commands**

|  |  |  |
| --- | --- | --- |
| **Command** |  | **Meaning** |
| #if expression |  | When expression is true, the code that follows is included for compilation. |
| #endif |  | Terminates the conditional command |
| #elif |  | Specifies alternate code in multi-way decisions. |
| #else |  | Specifies alternate code in two-way decisions. |
| #ifdef name |  | Abbreviation for #if defined name |
| #ifndef name |  | Abbreviation for #if !defined name |

**Getting the files for today's lab**

1. Create a **"lab8"** directory
2. Use *cd lab8* to move to that directoyr
3. copy the lab files to the directory using:

lab1071copy 7

Next untar the file, i.e.:

**tar -xvf lab8.tar**

You should now have the files  *data8.txt*  *image.h*  *lab8.c*  *main.c*  *pixel.c*  and  *redgreen.c* in your lab8 directory.

**Task 1:**

1. Open lab8.c and review the code.
2. Notice the use of *#if DEBUG* conditional compilation around the debug print statements to turn on and off the debugging print statements.
3. To see how conditional compilation works, run the program with DEBUG off and also with DEBUG on. Initially, the *DEBUG* flag should be off (value 0). Check the *#define DEBUG* directive near the top of the file. If the value is 1, change it to 0.
4. Compile the program using *gcc -Wall -o lab8 lab8.c*
5. Now run the program using *./lab8 < data8.txt*

You should not see any of the debugging print statements scrolling on your monitor. Now, let's turn on *DEBUG* as follows:

1. open lab8.c and change the value of *DEBUG* to 1.
2. compile and run the program again.
3. This time, you should see several debugging print statements scrollin on your monitor.

Note: sometimes, when you are working with multiple files and want to use conditional compilation, it would probably be useful to define a compiler flag *-DDEBUG*, rather than have to include the #define statement at the start of each file. Let's try this:

1. delete the *#define DEBUG 0* directive near the top of lab8.c
2. compile lab8.c using *gcc -Wall -o lab8 lab8.c*
3. now execute the program using *./lab8 < data8.txt*

You should not see any of the debugging print statements scrolling on your monitor. Now, let's turn on *DEBUG* as follows:

*gcc -Wall -DDEBUG -o lab8 lab8.c*

Execute the program as before:

*./lab8 < data8.txt*

You should see debugging print statements scrolling on your monitor.

You can turn off *DEBUG* again by using

*gcc -Wall -UDEBUG -o lab8 lab8.c*

This time when we execute the program, we will not see any of the debugging print statements.

**Task 2:**

One of the files that you copied is a redgreen.c file that produces a .ppm image with one half red and the other half green.

1. Compile the program using

*gcc -Wall -o img main.c pixel.c redgreen.c*

1. Execute the program using

./img out.ppm

You should see all pixel values scrolling on your monitor. Let's use conditional compilation so we can turn the debugging print statements on or off as desired.

1. Open the redgreen.c and find the call to the *print\_pixel()* function.
2. Surround the call to *print\_pixel()* with #if DEBUG ...#endif directives to be able to turn the print statements on or off.
3. Compile the program using *gcc -Wall -DDEBUG -o img main.c pixel.c redgreen.c*
4. When you run the program, using ./img out.ppm, the pixel values should print
5. compile the program again using *gcc -Wall -UDEBUG -o img main.c pixel.c redgreen.c*
6. when you run the program this time using

*./img out.ppm*

, the pixel values should not print.

-DDEBUG turns compilation of the call to print\_pixel()   
-UDEBUG turns off compilation of the call to print\_pixel().

**Makefiles for multi-module programs:**

The Unix make program is a handy utility that can be used to build entities ranging from programs to documents.

Consider the following makefile that you have for program 2:

C=gcc

CFLAGS=-g -Wall

OBJS = getnum.o getimage.o newimage.o outimage.o prog2.o mirror.o

mirror: ${OBJS}

@echo Linking $@

gcc -g -o $@ ${OBJS}

${OBJS}: image.h makefile

clean:

rm -f \*.o mirror

test:

./mirror flower.ppm mirror-flower.ppm

Elements of significance with a makefile include:

**targets**: labels that appear in column 1 and are followed by the character ":" . A makefile may have multiple targets. The desired target may be specified on the make command, e.g.

make test

**dependencies**: are files that are enumerated following the name of the target. This list should include all files that are components of the executable. If you forget to include one (like image.h here) there is no automagic test that will find it. If any dependency is newer than the target, the target will be rebuilt, i.e., changing any dependency file will trigger execution of the rule(s) that follow the target line the next time the command "make" is issued. Rule lines must start with a tab (ctl-I) character.

**rules**: are specified in lines following the target and specify the procedure for building the target. Rules must start with a tab character. In the example below the tab has been expanded as spaces but you may not enter spaces.

Makefiles should be named either "makefile" or "Makefile" and the make operation is initiated by simply entering the command:

make

Consider the following makefile with the following contents: **A basic makefile:**

img: main.c pixel.c redgreen.c

gcc -g -Wall -o img main.c pixel.c redgreen.c

* In this simple makefile, img: is the name of a "target". A makefile may have multiple targets. The desired target may be specified on the make command

make img

* The file lists that follow the target are the "dependencies". This list should include all files that are components of the executable "img". If you forget to include one (like pixel.c here) there is no automagic test that will find it.
* Changing any dependency file will trigger execution of the rule(s) that follow the target line the next time the command "make" is issued. Rule lines must start with a tab (ctl-I) character.

**A more sophisticated makefile**

The previous makefile will build the program but it has two shortcomings:

1. When only one C source file is modified, all source files must be recompiled.
2. Compiler error messages from all modules are merged together and sent to *stderr*.

The makefile below solves both of these problems. It builds object (.o) files containing the machine language code for each .c file.

It also sends any compiler error messages from x.c to x.err.

The target .c.o: is called a suffix target. It is telling make to use the rules that follow the suffix target whenever it needs to convert a .c file into the corresponding .o file. Notice that two rules (*gcc* and *cat*) follow the dependency.

**There are a number of predefined macro based names:**

Macros are names that begin with the character $. When make processes a makefile the macro name is replaced by the current value of the macro in a manner similar to what is done in Unix script or Windows .cmd files. **$@**-- the current target's full name   
**$?** -- a list of the target's changed dependencies   
**$<** -- similar to $? but identifies a single file dependency and is used only in suffix rules   
**$\*** -- the target file's name without a suffix

Another handy macro based facility permits one to change suffixes on the fly. The macro $(@:.o=.err) says use the target name but change the .o to .err. Thus when you build the program you will see that files named listmain.err and list.err. These files will contain the compiler warnings and errors messages.

**Building a program with make**

Assuming you have correctly constructed the file named *makefile* or *Makefile* in your working directory and you wish to build the first target (here the *img* executable) you simply enter the command:

make

**Task 2**

Using your favorite editor, create a makefile for *img*, named *makefile* (or *Makefile*). Sorry, you need to manually type this -- you can try cutting and pasting but be careful!):

# makefile for img

OBJS= main.o \

pixel.o \

redgreen.o

img: ${OBJS}

@echo

@echo Linking $@

gcc -o $@ -Wall -g ${OBJS}

.c.o: $<

@echo

@echo Compiling $\*.c

gcc -c -Wall -c -g $< 2> ${@:.o=.err}

@cat $\*.err

${OBJS}: image.h makefile

NOTES:

* the "\" at the end of the line "OBJS" definition says that the files are continued on the next line. We could have put all names on the same line, but by separating them we can easily add new filenames later,
* the "rules" (for example the lines after "img") MUST start with a TAB, not spaces!
* the "@" before the echo tells the "make" facility to not print the command itself before executing the given command (otherwise, in this case, we would see the "echo" command printed followed by the output of "echo" which would look rather strange).
* several pre-defined macros (such as "$@") are used. These are recognized as having the folloiwng meaning:

$@ -- the current target's full name   
$? -- a list of the target's changed dependencies   
$< -- similar to $? but identifies a single file dependency   
$\* -- the target file's name without a suffix

* Another handy macro based facility permits one to change prefixes on the fly. The macro "${@:.o=.err}" says use the target name but change the .o to .err.

The same result effect may be obtained using "$\*.err" as is done in the subsequent cat command.

**Task 3:**

Add a dependency called "clean" and a rule to the makefile to remove all .o files, .err files, and all executables. type the command make, then make clean, the program to clean up your directory.

**Submission**

Submit your lab8.c, redgreen.c, and makefile files to [Handin](https://www.cs.clemson.edu/course/cpsc1070/labs/lab08/https/handin.cs.clemson.edu), unless your` lab instructor gives other directions.