Step 12: Compilation and Makefiles

This assigment is done in class on (monday)04-07-14 AND (friday)04-04-14.

This assignment is due for monday class 04-07-14 AND for the firday class 04-04-14.

Getting Started

Create a new directory called hanoi:

```
cd cse251
mkdir hanoi
cd hanoi
```

Download the file hanoi.c (right click and save the file into your hanoi directory. Be sure the program compiles and runs:

```
gcc -o hanoi hanoi.c
./hanoi
```

This program implements the simple game: Tower of Hanoi. You have 6 disks of varying size and three pins. At any move you have move one disk from the top of a pile on one pin to the top of the pile on another pin. The rule is that you can never put a larger disk onto a smaller disk. The object of the game is to move all of the disks from the first pin to the third pin. Try as a first few moves: 1 to 3, 1 to 2, and 3 to 2. See if you can figure out the solution. You can enter 0 as the pin to move from and the program will display a complete solution one step at a time.

> There is a legend of a Tower of Hanoi with three pins and 64 disks in a monastery in what is now Vietnam. There, monks are moving the disks in accordance with the rules of the game. When all 64 disks have been moved to the third pin, the world will end. Since Tower of Hanoi with three pins requires 2^D-1 moves for D disks, that will require 2^{64}-1 seconds, assuming they move one disk per second. That's about 585 million years. Who knows, they may be right. Our version requires 63 moves.

 Be sure it compiles and runs and you know how to play the game. I'll not indicate specific places to compile very often. You will need to do so regularly.

1. Compilation and Makefiles

We have been using gcc to compile our programs in one step. Instead, today, we are going to start breaking the process down a bit. Try the following command:

```
gcc -c hanoi.c
```

This *compiles* the program hanoi.c into an object file hanoi.c. If you do an **ls**, you should see that file.

We can then use the object file to create the executable program using this command (try it):

```
gcc -o hanoi hanoi.o
```

This command took the object file created by compiling and *linked* it with other libraries needed by C and created an executable file: hanoi.

Makefile

As programs grow larger, they need to be broken into pieces for them to be manageable. hanoi.c is, in fact, too big and hard to read as it stands. When you have more than one .c file for a given program (and this is most of the time), you have to compile each .c file to a .o file, then link them all together. If I had 5 .c files, this would require 6 commands. That would quickly get tedius. This is one of the reasons for makefiles. We used makefiles I provided for the elevator and garage projects. Note how much easier it was to just type "make" each time instead of a lot of commands.

Use gedit to make a new file named **Makefile**. Note that the M must be capital. This is a special name that the make utility looks for.

We are going to create our first Makefile rule. For each rule you need to know three things:

1. The target. This is what is being made. In our case, that will be **hanoi**.
2. The dependencies: What the target is made from. In our case, this will be hanoi.o.
3. The command: How to make the target from the dependencies. In our case: gcc -o hanoi hanoi.o

Add this rule to your Makefile:

```
hanoi:  hanoi.o
        gcc -o hanoi hanoi.o
```

It must be an actual tab character before the command. If you have gedit set to insert spaces instead of tabs, you would have to change that. If you get this message when you run make:

```
Makefile:2: *** missing separator.  Stop.
```

That is an indication you have spaces in front of the command instead of a tab.

Save this file and type at the command line:

```
make
```

You should get a message something like this:

```
cbowen@ubuntu:~/cse251/hanoi$ make
make: `hanoi' is up to date.
```

Make is smart. It looks at the time stamp for hanoi.o and hanoi. If the time stamp for hanoi is later than hanoi.o, it means hanoi was built after hanoi.o, so it was build from the most current version of hanoi.o. Try this command:

```
touch hanoi.o
```

The touch command sets the time stamp for a file to the current time. So, now hanoi.o is more recent than hanoi, so it needs to be built. Type make again and you should see something like this:

```
cbowen@ubuntu:~/cse251/hanoi$ make
gcc -o hanoi hanoi.o
```

This created the file hanoi again from hanoi.o. Typing make one more time should indicate that hanoi is up to date.

A Rule to Compile

Building an executable program is a two step process: we compile all of the .c file, then link them into an executable. Your Makefile does the linking, but not the compilation. For that we need another rule. Remember, a rule needs three things:

1. The target. This is what is being made.
2. The dependencies: What the target is made from.
3. The command: How to make the target from the dependencies.

First, insert a blank line after the rule you just created. Always put a blank line between rules in your Makefile.

Our rule for compiling hanoi.c (the dependency) into hanoi.o (the target) using command gcc -c hanoi.c will be:

```
hanoi.o:        hanoi.c
        gcc -c hanoi.c
```

The order is important. The first rule has to be what we are finally making, the final product. That will be our executable. The following rules are what that is made from. Their order does not matter, so long as they are *after the first rule*. Your entire Makefile should look like this:

```
hanoi:  hanoi.o
        gcc -o hanoi hanoi.o

hanoi.o:        hanoi.c
        gcc -c hanoi.c
```

When you type **make**, it should indicate that everything is up to date. Type these commands:

```
touch hanoi.c
make
```

It should do the two steps to create the executable hanoi:

```
cbowen@ubuntu:~/cse251/hanoi$ make
gcc -c hanoi.c
gcc -o hanoi hanoi.o
```

You have just made the most basic Makefile. You can create these from now on (and I'll expect you to). Then you just type make to compile.

> When you move to using Makefiles, you should put a project in its own directory instead of having many in one directory. Note how we put this project in the hanoi directory.

Breaking a Program Into Modules

Edit the file hanoi.c. If you go to the end of it, you will see that it is 266 lines long. That's really long for a single program file. When programs get that large, they get hard to work on. So, we are going to break this program into pieces. Create a new file: display.c. Just leave it empty for now.

> Right now the entire program is in one file and is very big. I'm going to move all of the code responsible for displaying the Towers into the file display.c. Then I can go right to that code when I need to work on it instead of searching in a single monolithic file.

We need to tell the Makefile we have a new .c file as part of this program. Add this rule to you Makefile to compile the new file we just created:

```
display.o:      display.c
        gcc -c display.c
```

Be sure you put a blank line between this rule and the previous one and that this is after the rule that builds the executable.

Now, we need to tell the rule that makes the executable to *include the new display.o*. Change the first rule to the following. I highlighted the changes.

```
hanoi:  hanoi.o display.o
        gcc -o hanoi hanoi.o display.o
```

My entire Makefile with the changes highlighted is:

```
hanoi:  hanoi.o display.o
        gcc -o hanoi hanoi.o display.o

hanoi.o:        hanoi.c
        gcc -c hanoi.c
```

```
display.o:      display.c
        gcc -c display.c
```

We are saying:

1. hanoi is built from hanoi.o and display.o
2. hanoi.o is built from hanoi.c
3. display.o is built from display.c

Another way of looking at this is:

1. If hanoi.o or display.o change, we need to rebuild hanoi.
2. If hanoi.c changes, we need to rebuild hanoi.o
3. If display.c changes, we need to rebuild display.o

Note how we had to tell the command to *link* about both hanoi.o and display.o.

If you type make, it should build display.o by compiling display.c and rebuild hanoi:

```
cbowen@ubuntu:~/cse251/hanoi$ make
gcc -c display.c
gcc -o hanoi hanoi.o display.o
```

Header Files

We are going to move C code into display.c. So, first add these three header file includes to display.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
```

It should still make okay.

A Header File

When you break a program into modules like we are doing, you need a way to share information among the modules. We do this using *header files*. These are files with a .h extension. We've been using ones provided by the system up till now. Now we will make one of our own.

Create a new file hanoi.h with this content:

```
#ifndef HANOI_H
#define HANOI_H


#endif
```

This is special pre-processor code called an *include guard*. Because include files may be included by other include files, an include file may get included more than once. The first time it is included, the #ifndef is true, so the code between #ifndef and #endif is compiled. This include a subsequent definition of HANOI_H. The next time this file is seen by the compiler, HANOI_H exists, so it does not include the contents a second time. Suffice it to say:

1. Always put an include guard in your header file.
2. The name after #ifndef and #define must be the same and unique to this header file.
3. Never put anything before the first #ifndef
4. Never put anything after the #endif

I generally use the name of the file in upper case with an underline instead of a dot. If I asked you to create a new header file: prog.h, you would start it with (don't add this code):

```
#ifndef PROG_H
#define PROG_H


#endif
```

If you just paste in that save include guard in a later .h file that we did before it will fail.

This header file is going to be using by both hanoi.c and display.c. So, add the following #include to hanoi.c after the other #includes.

```
#include "hanoi.h"
```

Just to be sure, it goes here:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "hanoi.h"
```

Do the same thing in display.c. Both hanoi.c and display.c should start with:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "hanoi.h"
```

Note that you put these in quotation marks. Use <> for things from the C library and "" for header files you create yourself.

The idea of a Makefile is that if we change a file, everything dependent on it has to be re-built. For example, if we change display.c, we need to rebuild display.o. Since display.o changed, we need to rebuild the executable. That's what the dependencies in the Makefile are for. But, what if we change hanoi.h? Since it is included by display.c, its effectively a part of display.c. So, if we change hanoi.h, we need to rebuild display.c (and for the same reasons, hanoi.c). We tell this to the Makefile by adding the dependencies to the rules. Make these changes (highlighted) to your Makefile:

```
hanoi:  hanoi.o display.o
        gcc -o hanoi hanoi.o display.o

hanoi.o:        hanoi.c hanoi.h
        gcc -c hanoi.c

display.o:      display.c hanoi.h
        gcc -c display.c
```

I am telling the Makefile that if either hanoi.h or display.h are more recent than display.o, we need to rebuild it.

> If you fail to add a dependency like this, your program will be buggy in ways that are hard to understand and even completely correct code may not run. When you add a header file to a .c file, be sure to add it to the dependency list for that rule that compiles that header file! If you forget, you will be sorry.

Now we can make again. Try the command sequence:

```
make
make
touch hanoi.h
make
make
```

The output should look something like this:

```
cbowen@ubuntu:~/cse251/hanoi$ make
gcc -c hanoi.c
```

```
gcc -c display.c
gcc -o hanoi hanoi.o display.o
cbowen@ubuntu:~/cse251/hanoi$ make
make: `hanoi' is up to date.
cbowen@ubuntu:~/cse251/hanoi$ touch hanoi.h
cbowen@ubuntu:~/cse251/hanoi$ make
gcc -c hanoi.c
gcc -c display.c
gcc -o hanoi hanoi.o display.o
cbowen@ubuntu:~/cse251/hanoi$ make
make: `hanoi' is up to date.
```

Moving the Constants

The idea of a header file is that it is a place to put things that are shared among program modules. We have three constants in hanoi.c:

```
#define NumPins 3
#define NumDisks 6
#define MaxDiskSize 13
```

This information is going to be of interest to just about every part of the program. So, if we move code to display.c, it may be dependent on these values. So, we will move these to our header file. Cut them from hanoi.c and paste them into hanoi.h between the include guards:

```
#ifndef HANOI_H
#define HANOI_H

#define NumPins 3
#define NumDisks 6
#define MaxDiskSize 13

#endif
```

Don't put anything before the #ifndef and don't put anything after the #endif.

This should make and run just fine. Be sure you save all of the parts of your program each time, not just the last file you worked on.

All we have done is move something from one .c file to a place where it can be shared by both of our .c files.

> Don't duplicate code between two .c files. For example, don't have the same #define in two files. That's a recipe for disaster if you change the value in one place and forget the other place. Use a header file to share the code between the two files.

Moving a Function

We are now going to move C code to display.c. Find the function DisplayDisk in hanoi.c:

This function draws a disk of a given size on the output.

```
/*
 * Display a disk of width p
 */
void DisplayDisk(int p)
{
    int i;
    int spaceAround = (MaxDiskSize - p) / 2;

    for(i=0; i<spaceAround;  i++)
        printf(" ");

    if(p == 0)
    {
        printf("|");
    }
```

```
    else
    {
        for(i=0;  i<p;  i++)
            printf("O");
    }

    for(i=0;  i<spaceAround;  i++)
        printf(" ");
}
```

Cut this from hanoi.c and paste it into display.c. Don't leave it in both places. It should only be in one place. Add it to display.c after the headers.

When you share a function between two modules, you put the function declaration in a header. Remove the function declaration for DisplayDisk from hanoi.c and put it into hanoi.h. After that, hanoi.h should look like this:

```
#ifndef HANOI_H
#define HANOI_H

#define NumPins 3
#define NumDisks 6
#define MaxDiskSize 13

void DisplayDisk(int p);

#endif
```

This should make and run ok.

Now find the function: DisplayTower in hanoi.c. Cut it from hanoi.c and paste it into display.c. Then cut the function declaration from hanoi.c and paste it into hanoi.h. Be sure you save all three files before you make.

What Have We Done?

We have moved all code responsible for displaying our pins into the file display.c. If I am working on this program and need to work on what the display looks like, I can go to that file to find the functions instead of searching one big file. It's common that every file will have only a few functions in it.

make clean

A very common addition to a makefile is a rule that makes the target "clean". It's not actually going to make a file named clean, so it will always execute when you try it. Add this rule to the end of your Makefile:

```
clean:
        rm -f *.o hanoi
```

To use this, type this command:

```
make clean
```

I am telling it to make a specific target. What this does is delete the executable and intermediate files. Executables and intermediate files are made from your code. You can always make them again. The clean target removes these things so all that is left is your work. We'll use this later.

Final Tasks

1. Create a new header file autosolve.h. This file will be included ONLY by hanoi.c. Be sure you do the dependencies right in the Makefile. We are not going to include this in display.c, so display.o is not dependent on autosolve.h.

Note: You will have to include hanoi.h in autosolve.h. The reason is that the function declarations we will put in autosolve.h are dependent on the constants contained in hanoi.h. It is perfectly normal and reasonable for a header file to include other header files it depends upon. Of course, if autosolve.c includes autosolve.h and autosolve.h includes hanoi.h, autosolve.o is dependent on both autosolve.h AND hanoi.h.

2. Create a new file autosolve.c that also include autosolve.h. Move the functions necessary for the automatic solver to autosolve.c. These are the functions Autosolve and AutoMove. Put the function declarations for Autosolve and AutoMove into autosolve.h, not hanoi.h.

The idea of what we are doing here is to make autosolve.c a module that does a solution to the puzzle. To use that module, we include the header autosolve.h. Only the file hanoi.c uses the autosolve functions, so it's the only file that will include autosolve.h other than autosolve.c. autosolve.h is a way for hanoi.c and autosolve.c to communicate with each other.

When you are done, do the following:

A: Do a make clean to remove the executable and intermediate object files.

B: Move up one directory (cd ..) and type this command:

```
tar cvzf hanoi.tar.gz hanoi
```

This will create a file hanoi.tar.gz. This is the file you will turn in.

Turn in hanoi.tar.gz via http://secure.cse.msu.edu/handin/

Don't Do

Don't put the #defines for NumPins, NumDisks, and MaxDiskSize in autosolve.h.

CSE 251