CS032 2009 Lab 08
C

1 C - It's Legit

So most likely you've seen some C. Hopefully, you've done the homework already. If not, you should get on that, because it's due pretty soon. C is one of the most widely used programming languages. It has become such a popular choice due to its efficiency and the power it gives the programmer over the machine. Almost any electronic device that uses a microprocessor can be programmed in C. Whether its a typical computer, a specialized gaming console, or a onboard real-time system controlling a military jet, C gives you the power to tell a machine what to do, and how to do it. Of course, there are more exact ways to program, such as MIPS Assembly, but C is a good middle ground between Java and a magnifying glass, a magnet, a stick of ram, and a very steady hand.

This lab is going to cover some C basics. We can't cover enough C in two hours to actually prepare you for the coming assignments, so we trust that you have read or at some point will read the CS123 C mini course at http://www.cs.brown.edu/courses/cs123/resources/c_mini_course.pdf.

2 Tools

2.1 gcc

gcc is just one tool in the larger GNU compiler collection. This set of compilers is typical for all Linux distributions because it is open source. It contains many compilers, including an open source Java compiler called gcj. We'll cover some gcc basics, but most of your compilation will be done with Makefiles.

2.2 make

You will inevitably have projects with tons of files, which will lead to tons of .o files and issues about binaries being out of date and what not. Make can be thought of like a script for your compiler, but with significantly more control than a shell script. Think of it like Ant, except instead of build.xml, your file will be called Makefile, and instead of sucking, it'll be awesome. True story.

2.3 man

This is Unix's Hitchhiker's Guide To The Galaxy. Any information you need about a shell command, a system function, a built-in program, random facts about Andy Van Dam, etc...it's all there.

2.4 gdb

GNU Debugger. The best program ever. This will save your life. It's like the debugger in Eclipse, but faster and better.

3 gcc

3.1 The Code

For this part of the lab, you'll be using the code in the `make/` directory. This is a copy of an old CS36 assignment called myalloc, implemented by Josh Dawidowicz in 2007. It is a simplified implementation of

malloc, about which you will learn more than you want to know in CS167. It has two different run modes, Task 1 and Task 2, which can be switched in `myalloc.h`. None of this matters for your lab.

3.2 gcc is Way Complicated

"UNIX is the answer, but only if you phrase the question VERY carefully."
-Button on the bulletin board in the Sunlab

gcc is very robust, but very complicated. It has one of the longest `man` pages, which should tell you something. It will do a ton of stuff for you on its own, but you need to guide it. It helps to know how gcc works before you start compiling stuff with it, but this is not a systems class, so if you'd like to investigate, we've included the old CS36 lab[1] for your reading pleasure. For now, we'll just be dealing with compiling, taking pre-processing and linking as givens.

3.2.1 Compiler Flags

Once the C pre-processor has had its turn, the compiler takes over. The compiler is responsible for generating assembly or machine code from source code. Binary files produced by gcc are known as object files and have a file extension of `.o`. It's important to note now that unless specific flags are passed into gcc it will attempt to build an executable from source, rather than the intermediate step of building an object file. In order to build an object file, we pass the `-c` flag to gcc. In order to give a specific name to the outputted file, we pass the -o flag. Here is an example of the above.

```
/u/jdawidow % gcc -o file.o -c file.c
```

In the example above, we are trying to compile the file file.c, requesting object file output, renaming the output file to file.o. Remember to specify the new filename after the -o flag if you're trying to rename it.

3.2.2 Linking and Loading

The final step in building an executable is the linking and loading stage. All of the references to other files are taken care of in the linker. To link object files into a larger executable, invoke gcc without the -c option. Consider the example below.

```
/u/jdawidow % gcc -o theProgram file.o file2.o
```

In this example, myExecutable is built by linking the two object files file.o, and file2.o, which were built separately.

Alternatively, there are several instances when a user may want to compile a program linking against an external library. For example, suppose you want to use a support code library called, appropriately, `support`. In this instance, if the library is in a central location, you would simply use the `-l` option in the compiler

```
/u/jdawidow % gcc -o theProgram -lsupport file.o file2.o
```

3.2.3 Checkpoint 1

Compile myAlloc and show a TA. Remove the .o files and the executable and you'll be ready for part 2.

4 Manual Pages

While you won't actually write any code during this part, it's a good time to tell you about the manual pages. The manual pages, often referred to as manpages, are like Javadocs, but for all of UNIX. Manpages exist for UNIX commands as well as C standard library function. Often, if you're not sure how to invoke

[1] gcc_man.pdf, in this lab directory

a function, you can look it up for more information than you could possibly use. Try `man function` with your favorite C function or UNIX command[2]

When you type `man topic`, you will get the manpage for `topic` in the first available category. However, this might not give you what you want to know. For example, try looking at the manpage for printf, a standard C library function. When you look at the manpage, the manpage for the UNIX shell system call is specified instead of our friendly C library function. Try `man 3 printf` instead.

For more information, read the manpage for man. Info is a similar tool, but you will rarely use it.

5 Make

If you've got a medium to large sized project, you'll waste a lot of time every time you rebuild all of the source files. It would be better to only update the binary files that depend on source files that have been modified. The `make` program does exactly this and it is a very useful tool. You may have already used `make`, but now we will take a close look at how Makefiles function and how to write them. Obviously this does not encompass everything that `make` does for you, but this is a good beginning.

WARNING Historically, there exist a few tools that are white-space dependent. Absurd as this idea is, make is one of these arcane programs, and it is vital that you TAB and do not space before the body of rules and also between the target and the dependency. Don't blame us, blame Richard M. Stallman.[3]

5.1 `make` Variables

The structure of most makefiles is similar. First comes the declaration of variables. Commonly, this is used to specify programs and files that are going to be processed later by the rules section. A variable assignment looks like the following:

```
FIRSTNAME = Jon
LASTNAME = Natkins
FULLNAME = $(FIRSTNAME) $(LASTNAME)
OBJECTS = File.o Main.o
CC = gcc
```

Variables in make are usually all upper case for clarity. The first and second lines create variables FIRSTNAME and LASTNAME with values Jon and Natkins. The third line accesses the values of FIRSTNAME and LASTNAME to create a third variable called FULLNAME with value Natkins. Remember that variables are referred to by $(NAME) and not just (NAME), like in a shell script.

5.2 Comments

Just like in shell scripts or perl, the pound sign[4], #, is the comment character.

5.3 `make` Rules

The next part of a Makefile is usually the rules section. This section tells make how to turn one file type into another. For now we will be looking at rules that use one command to convert from one file into another but you can put any shell script into a make rule, which is one of the things that makes make so powerful. A typical set of rules looks like:

[2] ls and strcpy are good ones, gcc and fvwm2 are the length of Lord of the Rings. The extended versions.
[3] http://xkcd.com/225/
[4] Or, if you want to be a dick about it, the octothorpe

```
all: my_executable

my_executable: $(OBJECTS)
        $(CC) -o $@ $(OBJECTS)
```

where the $(OBJECTS) variable is defined earlier to be all the object files (.o's) to be compiled for this executable. You will also commonly see a variable called $(TARGET), which will be where my executable is right now.

This can get complicated so let's break down the components of a rule. Every rule has two required components, a "target", and a "dependency". Rules can also have an optional "command" component. The "target" is always on the left side of the ":" The "dependency" is always on the right side and specifies what the "target" depends on. If the "command" component exists, it's executed by make to turn the "dependency" into the "target."

The first rule in a Makefile is the rule executed by just typing make. A good Makefile convention that you should follow is to make the first rule `all:` which results in the entire Makefile being run. Here, `all:` depends on my executable, which is the target of another rule. `make` realizes that in order to complete the rule, it must complete `my_executable` first.. You can also access just the `my_executable` rule from the command line by typing `make my_executable` which will bypass the first rule, and go directly to `my_executable`. In other words, you can access a rule from the command line by referring to its target. The second rule is a typical rule as described above. The second rule tells the compiler what object files or .o's to link together to make your executable and tells make how to get the compiler to link them together.

Finally, you may have noticed some strange symbols in the rules example above. These are special (dynamically maintained per rule) variables that come in very handy while writing rules. $@ evaluates to the current target, $* evaluates to the basename of the current target, and $<evaluates to the dependency file.

Oh, and one more **RIDICULOUSLY IMPORTANT THING**: The tab between the target and the dependency is not required, but the tab before the rule body is *absolutely* required.

5.4 Pattern Matching

One of the most important capabilities of make is text substitution. This is simple to do, and it can be used in variables or rules. The symbol for a wild card or item to substitute is % (offcially dubbed the "pattern matching wild card metacharacter"). In a variable you might use % to turn a list of files with one extension (e.g., .c files) into a list of files with another extension (e.g., .o files). Such a variable would look like this:

```
OBJECTS = Board.o Main.o
SOURCES = $(OBJECTS:%.o=%.c)
```

In this case, the SOURCES variable will have every file name in the OBJECTS variable that ended in .o, but each file name will now have a .c extension. Thus, the SOURCES variable will contain Board.c Main.c. In a rule you might use % to match one type of file to another like this:

```
%.o: %.c
     $(CC) -c $<
```

In this case, the rule will compile any file ending in .o that it "sees" (through variables like OBJECTS) by replacing the .o extension with .c. This results in the object file actually being created.

5.5 Scripting

As previously mentioned, the rules in a Makefile can contain scripts. These scripts must be Bourne Shell[5] scripts, which means the commands used in any make rule may be anything that can be executed by the

[5]The Bourne Identity was a good movie.

Bourne Shell[6]. This comes in handy if one wants a rule to call make in another directory to compile a library or other module. Rules like these can get really convoluted, especially since each script command must be parsed as one line by make. Luckily, you've probably made a typo in your shell where you type \by accident and the shell gives you a ?. That's because the \character tells the shell "Hey bro, there's more to this command, but I ran out of room, so hold on." Go ahead and try `ls; \ls`. It's baller.

5.6 Checkpoints 2-5

- Write a Makefile to compile myAlloc. Use dependencies and variables whenever possible.
- Modify your Makefile, unless you did it already on your first try, to use pattern matching.
- Create a target, `tidy`, to remove all .o files
- Create a target, `clean`, to do what tidy does, as well as get rid of backups and executables

5.7 Additional Information

Anything you may ever need about Makefiles is at http://www.gnu.org/software/make/manual/make.html. Epic win.

6 GDB

As software projects get bigger and bigger, debugging becomes more and more difficult. You may have gotten used to debugging by printing out every single variable, but trust us, there are much easier ways to fix your programs. With gdb you can pause the execution of your program, print out variables at run time and check for errors in memory usage, all without recompiling. [7]

6.1 Background and Command Summary

Once you learn a few of the basic commands of gdb, you will be able to do a lot. One important command to know is help. If you type help you will get a list of broad help topics. If you wanted to get more information about running programs within gdb typing help will show you that running is a help topic. You can then type help running to get sub-topics related to running programs. Typing help <command> will usually give you a great deal of information about that command and its use. The gdb help system is a great resource, do not forget about it!

6.2 Starting gdb and running programs

Loading up a executable in gdb is relatively simple. Simply type

```
gdb <your executable name>
```

to start gdb up on the executable. In order to use gdb you MUST have compiled your program with the -g compiler option. When you are in gdb and have a program loaded, these commands will help you get started:

- `help running` – Will give a list of commands related to running the program.
- `run` <args> – Will run the current program being debugged with the given arguments.

[6] Read the manpage for `sh` if you're curious

[7] For the manual on gdb, *http : //sources.redhat.com/gdb/current/onlinedocs/gdb_toc.html*

- `set args <args>` – Will permanently set the arguments being sent to the program.

One thing to note is that gdb has a very nice tab completion system. It will tab complete commands, file names, and even symbols within your program! One slight caveat to this is that to complete C++ symbols you need to start an expression with a single quote when you tab complete. C symbols will work fine without the single quote.

Another tip is that when invoking a gdb command you only have to type as many letters as is necessary to disambiguate that command from any others. For instance, if you type 'whe' and hit enter gdb will assume you are just being lazy and you want to run the 'where' command.

6.3 Commands to control execution

Here are some basic commands that will let you pause the execution of your program and examine data.

- `info <action type>` - Displays all breakpoints or auto-display expressions in effect. The 'action type' argument can be several things, for instance, you can use 'breakpoints' or 'display' depending upon whether you wish to examine breakpoints auto-display expressions. You can even use 'locals' to show all local varaibles of the current stack frame. Try help info for more information.
- `break <location>` - This command sets a breakpoint at a specified location.

  Some examples:

  `break main` - Sets a breakpoint in the main() function.

  `break Foo::myFunc` - Sets a breakpoint in the Foo::myFunc method. [8]

  `break main.c:100` - Sets a breakpoint in the main.c file, at line number 100
- `break <location> if <condition>` - This command sets a breakpoint at a specified location with a conditional requirement such that the breakpoint is only hit if the conditional is true. For example:

  ```
  break myFunc if m_variable==0
  ```

  Sets a breakpoint at the start of the myFunc function only if m_variable==0
- `condition <breakpoint number> <condition>` - This command sets an already created breakpoint to be conditional so that execution of the program only halts at the specified breakpoint when the the condition specified is true.
- `step` - If stopped in execution, this will execute the next line of code. If this line is a function call, it will enter the function call. It is common to repeatedly call step to advance slowly through your program's execution.
- `next` - If stopped in execution, this will execute the next line of code, not stepping into any function calls. It is common to repeatedly call next to advance slowly through your program's execution.
- `continue` If stopped in execution, this will begin to execute normally and run until the next breakpoint or until the program exits. It is common to call cont when you have finished using step or next to examine things closely and want to have the program execute at full speed.
- `kill` - This simply kills the execution of the program being debugged. Useful if the program won't stop running and you need to force it to end.
- `delete [<action type>] <number>` - This deletes one of the breakpoints or auto-display expressions displayed by the info command. The 'action type' argument can either be 'breakpoints' or 'display' depending on whether you are deleting a breakpoint or auto-display expression. If the 'action type' argument is not given that it is assumed you are deleting a break point.

[8] If you don't understand this :: syntax yet, don't worry. It's C++ syntax indicating a class method call as opposed to a straight C function call. In this case, we break at the myFunc method of the Foo class

- `clear [<location>]` - This clears a specified location of all breakpoints. If no line number is specified this clears all breakpoints.
- `call <function>(args)` - Instructs gdb to call the given function, with the given arguments. Use print to show the return value (see next section).

6.4 Displaying and Naming Data

Once you reach a certain point in the execution of a program, it is often useful to be able to print out information about the execution of the program up to that point. For example, say you have some functions: funcOne calls funcTwo which calls funcThree. If you stop at a breakpoint in funcThree, you can print out the stack trace using the where command. This will print out a list of the calls which led to reaching funcThree, and the values of their arguments. While you are stopped in funcThree, you can also print out the values of all variables within the scope of the function, which may include local variables, class variables, and memory addresses, by using the print, display, and examine commands. Within the scope of funcThree, you can't access the variables in the scope of funcTwo. However, you can move between stack frames by using the up and down commands. Using the up command will move you from the scope of funcThree to the scope of funcTwo, where you can view the stack trace and variables of funcTwo. Using the down command will bring you back to the scope of funcThree. The important commands to know for looking at data are:

- `where` - Prints a list of all stack frames, along with the function and argument values for each stack frame.
- `up [<number>]` - Move up one stack frame. An optional argument can move up multiple frames.
- `down [<number>]` - Move down one stack frame. An optional argument can move down multiple frames.
- `frame <number>` - Jump to an exact frame number.
- `print <expression>` Prints the value of the given expression. The expression can be a function call, local variable, or a constant.
- `list` - Prints the code that surrounds the current point of execution.
- `display <expression>` - Display the value of an expression every time execution is halted at a breakpoint. It should be noted that all the above expressions may be arbitrarily complex and contain any math, bit operations, etc. They need not use variables from the program (e.g. print (1291 & 15) << 2).

6.5 Abbreviations

- `n` is equivalent to `next`
- `s` is equivalent to `step`
- `c` is equivalent to `cont`
- `wh` is equivalent to `where`
- `do` is equivalent to `down`
- `b` is equivalent to `break`
- `p` is equivalent to `print`
- `l` is equivalent to `list`
- `bt`[9] is equivalent to `where`

[9] bt is short for *backtrace*, which is the same as where

6.6 Extended Example

The following is an example of the first few commands issued after a program crashes.

`> gdb buggy`
Begin debugging the program that has the bug.

`(gdb) run`

Run the program within gdb. Interact with the program to try to recreate the conditions that made the program crash earlier. After run is called, in all likelihood, the program will crash at the same point that it did before. At this point gdb will halt execution of the program and return you to the gdb prompt.

`(gdb) print the_value`

This prints out the value of the variable the_value in the main function.

`(gdb) list`

This lists the 10 lines of code around the current line that we are stopped at.

`(gdb) where`

The where command will show us the complete stack backtrace.

`(gdb) down 2`

This jumps down two function calls.

`(gdb) print tempInt`

This prints out the value of tempInt which might be a local variable in the current function being examined.

6.7 Exercises - Final Checkpoint

The first step of these exercises is to copy some buggy source code from the gdb directory of the lab. Compile the code with the -g flag. Begin by looking at the source without modifying it. As you complete the exercise's questions, you'll need to modify the source in order to fix the bug(s) associated with this program.

- What is the stack backtrace when rot13() has just been called?
- At what line does the program die because of a segmentation fault?
- Fix the bug.
- What happens when you try and step through isupper?

6.8 Additional Information

Anything you may ever need about GDB is at http://www.gnu.org/software/gdb/documentation/. Epic nguyen.

7 Go Home

Seriously. Go home.