Lecture notes: Week 2

In Week 2, we covered:

1. Makefiles
2. gdb
3. Program layout

Makefiles

Makefiles let you define complicated sets of commands to build your projects.

Makefiles consist of a series of rules:

```
[target] : [dependences]
	[command 1]
	[command 2]
	…
```

A rule *target* is the name of the rule. The *dependences* are the files the rule depends on. The *commands* are what to do when the rule is “fired.” Note: there must be a tab before each command.

A rule is fired in one of two ways: (i) it is directly invoked (by calling “make [target]”) or (ii) it is invoked by another rule that is fired.

When a rule is fired, it goes through the following process:
1. If a *dependence* has a rule in the Makefile, fire that rule (using this same process)
2. Once all dependences have been fired, check to see if *target* is “out of date”: interpret *target* as a filename, and see if the timestamp on the file is older than the time stamp on any of its dependences. If it is, the target is “out of date.” If there is no file named *target*, the target is always assumed to be out of date. If there are no dependences and *target* exists, target is assumed to be up to date.
3. If the target is out of date, execute the list of commands

You can use Makefiles to orchestrate complicated build processes.

If you type “make” without a target, make will fire the first rule in the Makefile.

We usually define a target called “clean” whose job it is to clean up any intermediate files generated during the build process. This can also be used to remove all generated targets to force recompiling everything.

Macros

Makefiles also let you define macros to reuse the same commands over and over. For example, we can define GCC as a macro that invokes gcc the way we want:

```
DEBUG = -DDEBUG
CFLAGS = CFLAGS = -std=c99 -g -Wall -Wshadow --pedantic -Wvla -Werror
GCC = gcc $(CFLAGS) $(DEBUG)
```

Note that we use $(MACRO_NAME) to insert the macro into other places, including commands.

When make executes its commands, those commands are printed out as they execute (including expanding out any macros). A Makefile will not do anything you can't do by hand: you can execute exactly what a Makefile does by typing in those commands at the command line.

If any command in a Makefile recipe fails, make will exit with an error. Running the commands one by one is a good way to figure out what went wrong.

Common Makefile structures

A very common recipe for Makefiles is to compile a bunch of C files into *object* files (.o files), then link those together into an executable:

```
eliminate: eliminate.o main.o
      gcc $(CFLAGS) -o eliminate eliminate.o main.o

eliminate.o: eliminate.c
      gcc $(CFLAGS) -c eliminate.c

main.o: main.c
      gcc $(CFLAGS) -c main.c
```

This simple set of rules says: "to make an executable called `eliminate`, you need the files `main.o` and `eliminate.o`. To make *those* files, you need their associated C source files." Now, if you modify `eliminate.c`, Make will use its firing rules to first build `eliminate.o` and then create the executable `eliminate.` Note that Make will *not* compile `main.c` again, because `main.o` is not out of date.

Generic rules

But if you have a lot of C source files, writing these kinds of Makefiles can get tedious. For each C file, you need to write a rule for how to build the object file. Imagine what your Makefile would look like if you have 100 source files! To get around this, we can provide *generic* rules:

```
%.o : %.c
      gcc $(CFLAGS) -c $<
```

This generic rule says: to make *any* file that ends with `.o` (say, `foo.o`), you need a file with the same name ending in `.c` (in this case, `foo.c`). To create the target, you need to run `gcc -c`, and pass in the *input file name* (that's what `$<` means). So this command would execute

```
gcc -c foo.c
```

To build `foo.o`. Now we can create a simpler Makefile:

```
OBJFILES = eliminate.o main.o

eliminate: $(OBJFILES)
      gcc $(CFLAGS) -o eliminate $(OBJFILES)

%.o : %.c
      gcc $(CFLAGS) -c $<
```

Using the `OBJFILES` macro to list all the `.o` files we need. Now if we need to add another source file, we only need to update the macro!

Makefiles can get even more complicated than this, but looking into those issues is beyond the scope of this class.

GDB

The debugging process

We talked about how debugging a program is a little like being a detective. You have seen the crime (your program crashes), and you have some evidence (what your program did before it crashes). Solving the crime (fixing your program) involves formulating a *hypothesis* about what went wrong (what might your program be doing that made it crash?)

You can then collect more evidence to either confirm or disprove your hypothesis. As you collect more evidence, your hypothesis gets more refined, until eventually you can figure out what is wrong with your program.

One way to collect more evidence is with *printf debugging*: add print statements at various places in your program so that your program gives you more information about the values of variables, or where in the program you are when it crashes. But another thing you can do is use *gdb* to *debug* your program: you can essentially pause the execution of your program and investigate various parts of it.

To debug your program, you first need to compile it with debugging information turned on, by passing in the -g flag. You can then run the debugger:

```
> gdb a.out
```

From within the debugger, you can add breakpoints:

```
> b [function name]
```

Adds a breakpoint at the beginning of the function, while

```
> b [line number]
```

Adds a breakpoint at a particular line number. You can then run the program:

```
> r [command line arguments]
```

And the program will run until it gets to a breakpoint. You can then see where you are in the program:

```
> bt
```

Which will print out what function you're in, including the line number, as well as any other functions on the program stack (functions that are executing, but have not yet returned). [See below for information on the program stack]

To continue on to the next breakpoint, you can type `c`. To move to the next line in the code, you can type `n`. To *step* into the next line (including moving into a function that is called) you can type `s`. To step *out* of a function (continue until right when the function returns), you can type `fin`.

Note: on some systems, you might not have gdb, and instead will need to use the LLVM debugger, lldb. These commands will generally work there, too.

Printing out values

So how can we use gdb to gather more evidence? Well, when you get to a particular point in the program, you can print out values. Consider the following code from reverseArray.c (uploaded to Brightspace):

```
void reverseArray(int * arr, int size) {
    for (int i = 0; i < size / 2; i++) {
        int tmp = arr[i];
        arr[i] = arr[size - i];
        arr[size - i] = tmp;
    }

    return;
}
```

This code has a bug in it. We can place a breakpoint at the for loop:

```
> b reverseArray
```

And when we run the code

```
> r inp1
```

The debugger stops at the breakpoint:

```
Breakpoint 1, reverseArray (arr=0x602250, size=10) at reverseArray.c:5
5         for (int i = 0; i < size / 2; i++) {
```

And after stepping through the code one time (s) we can print out the value of I:

```
> p i
$1 = 0
```

We can also print out other things:

```
> p size
$2 = 10

> p size / 2
$3 = 5
```

We can even print out elements of the array:

```
> p arr[5]
$4 = 4
```

So now we can see when things go wrong. Step to the next line:

```
> s
7             arr[i] = arr[size - i];

> p arr[i]
$5 = 134545

> p size - i
10
```

A hah! We're indexing past the end of the array!

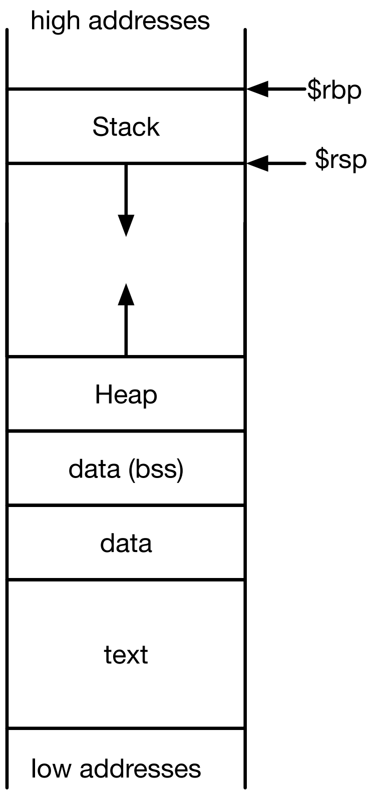
By setting breakpoints and then exploring the values of different values in the program, gdb gives you a lot of opportunities to collect evidence to find your bugs.

Program layout

When a program runs, your computer's memory is divided up into four *segments*:

1. The stack – this is where local variables for functions, function arguments, return values, and return addresses go. Every function has a *stack frame* that stores this information. When a function gets called, its *stack frame* is "pushed" onto the stack, and when it returns, the frame is "popped" off the stack.
2. The heap – this is where memory you allocate using `malloc` goes
3. The "text" – this is where the code of the program is stored
4. The "data" – this is where global variables of various sorts are stored. This space is broken up into smaller chunks:
   i. "data" holds global or static variables in the program that are initialized (e.g., if you declare a global variable `int x = 7;`)
   ii. "bss" holds global or static variables that are *uninitialized* (e.g., if you declare a global variable `int y;`) This whole segment is initialized to zero when the program starts (Why distinguish bss from data? Initialized variables need to have the correct values initialized for them, so they need to be stored in your program's binary. Uninitialized values don't need values stored, so the binary just tracks *how much space* the uninitialized variables take up.)

How are these segments placed in memory? One thing to note is that the text segment and the data segment(s) have fixed size, but the stack and the heap do not—as the program runs, you may call additional functions (requiring space on the stack) or allocate more memory using `malloc` (requiring space on the heap). To make room, the stack is organized as follows (higher addresses on top, lower addresses on the bottom):



When the program starts (i.e., we call `main()`), all of the local variables for main() are placed on the stack in a *stack frame*. We use the register $rbp (the *base pointer*) to mark the "bottom" of the stack frame, and the register $rsp (the *stack pointer*) to mark the "top" of the stack. (Note that when we draw the stack this way, the "top" of the stack has a lower address, and looks like it's lower than the base of the stack. Confusing, I know.)

If `main` calls `foo`, a bunch of things happen: the arguments to `foo`, and space for its return value, are "pushed" onto the stack (this automatically decrements $rsp to move the top of the stack). The address of the *next* instruction in `main` is pushed onto the stack (this is where execution will go to when foo returns). The current value of $rbp is pushed onto the stack, and then $rbp is moved to $rsp. $rsp is then moved down. This new space between $rbp and $rsp is `foo`'s stack frame: it's where any local variables for `foo` can get stored. When `foo` returns, the process is rewound, "popping" the frame off the stack, and the program resumes from the return address saved on the stack.

Essentially, as functions are called, we push stack frames for them onto the stack, so the stack keeps growing as long as we call more functions. Whatever function is *currently executing* has its frame at the top of the stack. When a function returns, its frame is popped off the stack.

Exploring the stack with gdb

Let's explore this data layout a bit with gdb, by compiling and running `recursiveSort.c` (also on Brightspace). This function uses recursion in a silly way to perform selection sort:

```
/* Use recursive selection sort to sort the array */
void recursiveSSort(int * arr, int lo, int hi) {
    // if the array is only one element long, it's sorted
    if (hi - lo <= 1) return;
    //find the smallest element in the range [lo, hi) and swap it to 
the front
    for (int i = lo; i < hi; i++) {
        if (arr[i] < arr[lo]) {
            //swap elements
            arr[i] = arr[i] ^ arr[lo];
            arr[lo] = arr[i] ^ arr[lo];
            arr[i] = arr[i] ^ arr[lo];
        }
    }
    // sort the array [lo + 1, hi)
    recursiveSSort(arr, lo + 1, hi);
}
```

This is a silly use of recursion because it's not doing anything you can't do with a loop. In fact, this style of recursion, where the recursive call is at the very end of the function, is called *tail recursion*, and your compiler can recognize the pattern and turn it into a loop! Later in class, we'll cover recursive functions that are more interesting.

So let's compile this and run it in gdb:

```
(gdb) b recursiveSSort
Breakpoint 1 at 0x40081f: file recursiveSort.c, line 7.
(gdb) r inp1
Starting program: /home/dynamo/b/milind/264Fall21/gdb/a.out inp1
The file has 10 integers
Original:
2 7 3 10 9 4 5 8 6 1

Breakpoint 1, recursiveSSort (arr=0x603250, lo=0, hi=10) at recursiveSort.c:7
7        if (hi - lo <= 1) return;
```

Note that here we can see when we call the sort function, we have three arguments: the array we're sorting, and the bounds that we're sorting (`lo` and `hi`). If we print out the stack (by printing 20 words starting at the stack pointer — since it prints out higher addresses, that means it's printing from the stack pointer "up" in the stack), we can see this information!

```
(gdb) x/20x $rsp
0x7fffffffdcc0:   0x0000000a     0x00000000     0x00603250     0x00000000
0x7fffffffdcd0:   0xffffde00     0x00007fff     0x00000000     0x0000000a
0x7fffffffdce0:   0xffffdd20     0x00007fff     0x00400bdf     0x00000000
0x7fffffffdcf0:   0xffffde08     0x00007fff     0x00000000     0x00000002
0x7fffffffdd00:   0x00400c30     0x00000001     0x00603250     0x00000000
```

We can see at the base pointer (`0x7fffffffdcc0`) that we have address of the array (highlighted in purple), the value of `lo` (highlighted in red) and of `hi` (highlighted in pink; note the hexadecimal value of 10!). If we continue, moving to the next recursive call:

```
(gdb) c
Continuing.

Breakpoint 1, recursiveSSort (arr=0x603250, lo=1, hi=10) at recursiveSort.c:7
7        if (hi - lo <= 1) return;
```

We can see that this version of the function has a different value for `lo`. In fact, we can see the full stack of function calls, including the two invocations of `recursiveSSort`:

```
(gdb) bt
#0  recursiveSSort (arr=0x603250, lo=1, hi=10) at recursiveSort.c:7
#1  0x000000000040096b in recursiveSSort (arr=0x603250, lo=0, hi=10)
    at recursiveSort.c:18
#2  0x0000000000400bdf in main (argc=2, argv=0x7fffffffde08)
    at recursiveSort.c:78
```

And if we print out the stack, we can see the stack frame for the two invocations of `recursiveSSort` (highlighted with boxes, color coding the same as above). Note that the value of `lo` is different!

```
(gdb) x/24x $sp
0x7fffffffdc90:   0x0000000a  0x00000001  0x00603250    0x00000000
0x7fffffffdca0:   0x00400720  0x00000000  0xf7a84219    0x00007fff
0x7fffffffdcb0:   0xffffdce0  0x00007fff  0x0040096b    0x00000000

0x7fffffffdcc0:   0x0000000a  0x00000000  0x00603250    0x00000000
0x7fffffffdcd0:   0xffffde00  0x00007fff  0x00000000    0x0000000a
0x7fffffffdce0:   0xffffdd20  0x00007fff  0x00400bdf    0x00000000
```