Computer Science 381
Programming Unix in C
The College of Saint Rose
Winter Immersion 2016

Lab 7: Structures in C

Due: Wednesday, January 6, 2016

In this lab you will learn about the `make` utility, then about C structures, a simple mechanism that allows heterogeneous data fields to be grouped into a single entity.

Using the `make` Utility

Any non-trivial software development involves many iterations of editing, compiling, linking, and running your programs. The code will be spread across multiple files. The most common mechanism for managing this process when programming in C in a Unix environment is the `make` utility. The actions of `make` are specified by rules in a `Makefile`.

Copy the following example to your account:

See Example:
`/home/cs381/examples/make-example`

You should find a small C program that demonstrates the use of multiple source files and a `Makefile`. Compile the program by issuing the `make` command. Capture the output of the command in `make.out`:

```
make > make.out
```

Output Capture:
`make.out` for 2 point(s)

Now, look at the contents of `make.out`, then at the rules and the description in the `Makefile`.

Question 1:
Briefly describe how `make` uses the rules in the `Makefile` to produce the executable `main`. Be sure to include the series of targets, their dependencies, and the commands used to satisfy those dependencies for each target. Your response should explain which lines in the `Makefile` cause each command to be executed. (5 points)

From this point forward, you should write a `Makefile` for each of your programs. You are strongly encouraged to do this when you first start each program rather than at the end - it is intended to be a tool to speed your development process, so use it that way!

Programs in Multiple Files

The `make` example above also demonstrated a very simple case of C code being separated into multiple `.c` (implementation) and/or `.h` (header) files.

We next consider an unnecessarily complicated C program that computes the greatest common denominator of two integer values that further illustrates this idea.

See Example:
`/home/cs381/examples/gcd`

There are lots of things to notice here:

- We have four files:
  - `gcd.c`: the implementation of the `gcd` function
  - `gcd.h`: a header file with a prototype for the `gcd` function
  - `gcdmain.c`: a main program that determines the input numbers, computes the GCD, and prints the answer, and
  - `Makefile`: a "make file" that gives a set of rules for compiling these files into the executable program `gcdmain`.

  When executing, functions from both `gcdmain.c` (`main`) and `gcd.c` (`gcd`) will be used. Both of these are included in our executable file `gcdmain`.

- Start with `gcd.c`:
  - This is a very simple recursive function to compute the greatest common denominator using the Euclidean Algorithm.
  - There is no `main` function here, so if we try to compile this by itself as we did with our single-file C programs, we will get an error.

    Question 2:
    What happens when you try this? Give the command you used and the error message that is produced. (2 points)

  - Instead, we have `gcc` use "compile only" mode to generate an *object file* `gcd.o` from `gcd.c`:

    ```
    gcc -c gcd.c
    ```

    `gcd.o` is a compiled version of `gcd.c`, but it cannot be executed.

    C (and many other languages) require a two steps for source code to be converted into an executable. The first step compiles source code into object code, the second takes a collection of object code files and *links* together the references in those files into an executable file. (There's much more to discuss here, but this should suffice for now.)

- Next up, `gcd.h`:
  - Much like `stdio.h` tells the compiler what it needs to know about `printf` (among other things), we have `gcd.h` to tell other C functions what they need to know about the function `gcd`. Namely, that it's a function that takes two `int`s as parameters and returns an `int`.
  - Any C file that contains a function that calls `gcd` should `#include "gcd.h"`.
- The driver program, `gcdmain.c`:
  - We include several header files to tell the compiler what it needs to know about C library functions (and our `gcd` function) that are called by functions defined here.
  - This is where our one and only `main` function is defined.
  - This file includes a `main` function, so we might think we could compile it to an executable as we did with the single-file C programs we've used so far. If we try, we'll find that it doesn't know how to find the `gcd` function.

    ? Question 3:
    Try this. Give the command you used and the error message you see. (2 points)

    Again, we'll have to compile but not link:

    ```
    gcc -c gcdmain.c
    ```

    This produces the object file `gcdmain.o`. We need to *link* together our two object files, which, together, have the function definitions we need:

    ```
    gcc -o gcdmain gcdmain.o gcd.o
    ```

    This gives us `gcdmain`, which we can run.
- The `Makefile` contains rules to generate a sequence of calls to `gcc` that will correctly compile and link the `gcdmain` executable.

? Question 4:
Draw a memory diagram for this program for the case where the numbers entered are 9 and 24. Your diagram should show the state of memory (including all copies of the parameters to each `gcd` recursive call that exist on the call stack) at the point where the "`return b`" statement is about to be executed during the base case of the recursion. (7 points)

C `struct`s

In addition to the readings from Chapter 6 of K&R, the following (somewhat silly) example should help you understand structures in C.

See Example:
`/home/cs381/examples/ratios`

Again, we have a number of C source code (`.c`) and header (`.h`) files. We will consider each in turn.

The files `gcd.h` and `gcd.c` are the same as the ones you saw earlier in this lab.

The files `ratio.h` and `ratio.c` define a structure and a number of functions that have to do with storing a ratio of two integer values.

In `ratio.h`, we have the definition of the structure that will hold our ratios:

```
typedef struct ratio {
  int numerator;
  int denominator;
} ratio;
```

There are two important things happening here. First, a structure called a `struct ratio` is defined. It consists of two `int` values: `numerator` and `denominator`. In many ways, these are like the instance variables of a Java class, but there are no access protections (*i.e.*, they are not "private" or "protected", but the equivalent of "public"). Second, we are giving another (shorter) name to our `struct ratio`: simply `ratio`. This is being accomplished by the `typedef`. In general, a `typedef` can define a new name for any type:

```
typedef x y;
```

would define a new type named `y` which is just another name for an already-existing type named `x`.

In our case, the `typedef` just means we can refer to variables and parameters of type `struct ratio` as simply `ratio`.

The rest of the contents of `ratio.h` defines function prototypes for the functions that will be defined in `ratio.c` that can be called from elsewhere.

As a whole, the information in `ratio.h` tells a C source file that would like to work with these `ratio` structures everything it needs to know to compile.

Also notice that the meaningful (*i.e.*, non-comment) contents of `ratio.h` are enclosed in the following block:

```
#ifndef _H_RATIO
#define _H_RATIO

// the rest of the stuff in ratio.h

#endif
```

This uses C's preprocessor to ensure that the "rest of the stuff in ratio.h" above only gets included once, no matter how many times we end up including the `ratio.h` file. There are circumstances (such as in the programming assignment below) where a header file ends up including another header file, and the code that includes the first also includes the second (and that's a simple case – it gets really messy in large projects). Without this, we will get errors about things like redefining types or function prototypes.

In `ratio.c`, the four functions that operate on `ratios` are defined: `create_ratio` constructs a new `ratio` given a numerator and a denominator, `add_ratios` takes two existing `ratios`, adds them and constructs and returns a new `ratio` that represents their sum in lowest terms, `reduce_ratio` takes an existing `ratio` and reduces it to lowest terms, and finally, `print_ratio` takes an existing `ratio` and prints it in a reasonably nice format.

There are a number of things to consider in these functions. The first two functions return a value of type `ratio *`. This indicates a **pointer** to a `ratio` structure. The last three functions take one or two parameters of this same type, `ratio *`.

Perhaps the most important thing to note here is how we allocate the memory for these structures. In both `create_ratio` and `add_ratios`, we see the line:

```
ratio *r = (ratio *)malloc(sizeof(ratio));
```

You have already seen `malloc`, but this usage is C's way of doing the equivalent of a Java `new` operation. This line:

1. declares a variable `r` of type `ratio *`
2. initializes `r` to the return of the function `malloc`
3. `malloc` reserves a chunk of memory of the requested number of bytes and returns a pointer to the start of the memory segment
4. the `sizeof` operator determines the number of bytes in the type to which it applies – in this case `ratio`, which should be a total of 8 bytes
5. since `malloc` does not return a `ratio *` (it returns a `void *`, which is a generic pointer), we need the cast to tell the compiler that we will be treating this newly-allocated chunk of memory as a `ratio *`

Note also the way we refer to the fields of the `ratio` structure when the variable `r` contains a pointer to a `ratio`:

```
r->numerator = numerator;
```

This is functionally the equivalent of the Java statement:

```
r.numerator = numerator;
```

However, since C allows a variable referring to a structure to be either a pointer or the structure itself, there are two different notations. If we had a variable `r` of type `ratio` rather than `ratio *`, we would use the "dot" notation like we use in Java. But here, since we have pointers, we use the "arrow" notation.

Recall the very important difference between C and Java that dynamically allocated memory in C is **not** garbage collected. That means that every chunk of memory we obtain with `malloc` must be returned to the system for reuse by a call to the function `free`. In our case, these `free` calls are made in `ratio_example.c`. For each call to `create_ratio` or `add_ratios`, which each contain a call to `malloc`, there must be a corresponding call to `free`.

This brings us to the file `ratio_example.c`, which is a `main` function that makes use of the `ratio` structure and functions to demonstrate the complexities of C memory management.

Read over the comments in `ratio_example.c` and see if you can understand how the memory is being allocated and managed.

? Question 5:

Draw a series of memory diagrams showing the contents of memory (both stack variables and the memory allocated in the heap) right before the `return` statement in each call to `add_ratios` (so, 2 separate diagrams), and then right before the `return` statement at the end of `main`. (12 points)

Programming Assignment

Write a new driver program (C file with a `main` function) `sum_ratios.c` that reads in a series of lines representing ratios from an input file and prints the sum of those ratios in lowest terms at the end.

- The program should take a single command-line parameter which is the name of the file that contains the list of ratios.
- Properly formatted lines in the file should look like this, where `n` and `d` are `int` values:

  ```
  n/d
  ```

- Your program should stop reading input and print the final result when it encounters an incorrectly formatted line or the end of the file. (Hint: `fscanf`'s return value is your friend.)
- You should use the `ratio.c`, `ratio.h`, `gcd.c`, and `gcd.h` files, unmodified, from the example.
- Provide your own `sum_ratios.c` and a working `Makefile` that will build your program on mogul.

- Be sure to perform appropriate error checking including meaningful error reporting, and free all memory your program allocates.
- Your program should compile with no warnings using `gcc`'s `-Wall` flag.

The program is worth 20 points.

The executable for the reference solution to this program is available on mogul in `/home/cs-381/labs/structs`.

Submission

Please submit all required files as email attachments to *terescoj@strose.edu* by Wednesday, January 6, 2016. Be sure to check that you have used the correct file names and that your submission matches all of the submission guidelines listed on the course home page. In order to email your files, you will need to transfer them from mogul to the computer from which you wish to send the email. There are a number of options, including the `sftp` command from the Mac command line.

Grading

This lab is graded out of 50 points.

Grading Breakdown	
Lab questions and output captures	30 points
`sum_ratios.c` correctness	10 points
`sum_ratios.c` error checking	2 points
`sum_ratios.c` memory management	2 points
`sum_ratios.c` documentation	3 points
`sum_ratios.c` style	2 points
`Makefile` for ratios program	1 point
Total	50