C Pointers & Structures

CS 350: Computer Organization & Assembler Language Programming
Lab 4, due Fri Feb 20

A. Why?

- Pointers let us share large memory objects without copying them.
- Structures give us a way to define data values that contain named components.

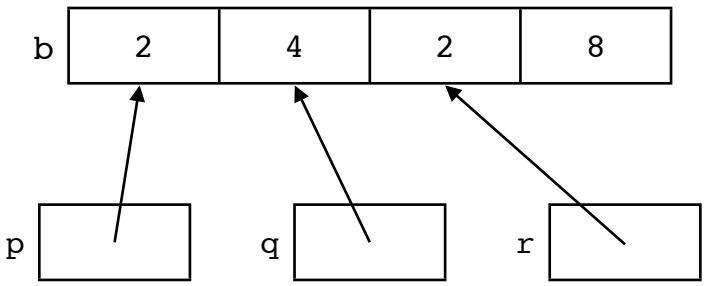
B. Outcomes

After this lab, you should be able to:

- Take a C expression or assignment that uses arrays and pointers and determine its value or action given a state of memory.
- Write simple C routines that take/modify structure arguments using pointers to the structure values.

C. Written Problems [60 points total]

1. [10 pts] Write some C declarations and code to establish the memory diagram below. (There are multiple right answers.) `p`, `q`, and `r` should be pointers to integers.



To answer Questions 2 and 3, you'll need some facts about arrays and pointers in C: Say `b[…]` is an array of (oh, say) `int` and `p` is a pointer to `int`. Then,

(1) `&b[i] == &b[0] + i`

(2) `b == &b[0]`

(3) If `x` is a pointer or array name, then `*(x+i)` is equivalent to `x[i]`; in other words, `x+i` is equivalent to `&x[i]`

(4) You can't assign `b` = …, so `&b` is illegal

2. [30 = 10 * 3 pts] Using the memory diagram for Problem 1, answer the following question for each of the expressions below: Does it cause a compile-time warning or error (and if so, which one), or does it cause a runtime error (and if so, which one), or does it evaluate to true or false? [Hint: Try typing these into a program and compiling them.]

   (a) `p == b`
   (b) `q == b+1`
   (c) `q == (&b)+1`
   (d) `*q == *(r–1)`
   (e) `p[1] == r[–1]`
   (f) `r-p == 2`
   (g) `p != r && *p == *r`
   (h) `q-b == &b[3] - &p[1]`
   (i) `p < q && q < r`
   (j) `2*q - 2*p == 2`

3. [20 = 2 * 10 points] Consider the C declarations and code below. (a) Draw a memory diagram that shows the state at position 1. (b) Draw a memory diagram that shows the state of memory at position 2.

```
int b[4] = {12, 13, 14, 15};
int u = 20, v = 30, *x = &u, *y, *z;
y = &u;
z = &b[2];
// <----- Position 1
++ *x;  // (i.e., *x = *x + 1)
y = &v;
--z;
z[1] = 20;
// <----- Position 2
```


D. Programming Problem [40 points]

You are to write a C program that allows you to manipulate rational numbers defined by the following `struct`

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

Definition: Two integers c and d are **relatively prime** if they have no common factor (other than 1 or -1). For example, 12 and 8 are not relatively prime because you can factor out 4 and get 3 and 2, which are relatively prime.

Definition: A rational r is **in lowest terms** if it is 0/1 or its numerator and denominator are relatively prime and the denominator is positive.

The full program is comprised of three files

- `Lab04_rational.h`, is a "header" file — it contains the definition of `Rational` and the prototypes of the various functions, but it contains no actual code. This file is given to you; if you want more utility functions, you're welcome to add their prototypes to this file.
- `Lab04_rational.c` is the implementation of the functions listed in the `*.h` file. (It's typical in C for the `*.c` file to implement the same-named `*.h` file.) **This is the file you have to write.**
- `Lab04_rat_client.c` contains the main method. It tests the functions in `Lab04_rational.c`. This file is also given to you, but you're welcome to extend it.

We `#include "Lab04_rational.h"` in the `*.c` files so that the code files know the prototypes of the various rational functions. (Note the difference with this and our usual `#include <stdio.h>` where the angle brackets indicate a system library.)

The `Lab04_rational.h` uses an idiom to avoid problems if you accidentally `#include "Lab04_rational.h"` more than once. (It's not an issue for these particular files, but for more complicated libraries, it's common to see one `.h` file include some other `.h` file) The idiom is


```
#ifndef RATIONAL_H
#define RATIONAL_H

… (actual content of file) …

#endif
```

These are preprocessor commands: if the symbol `RATIONAL_H` is not defined, then define it and pass the actual content of the file to the compiler. If we `#include` this file again, then since `RATIONAL_H` is defined, we don't pass the file contents to the compiler a second time.

What You Need to Do

A skeleton of the `Lab04_rational.c` file is given to you. It includes the definition of a gcd routine (greatest common divisor), which you need to normalize a rational number. You should augment so that it implements all of the following routines (these prototypes come from the `.h` file).

```
void set_rat(Rational *r, int n, int d);  // Set r to n/d
void copy_rat(Rational *r, Rational *s);  // Set r to s
double value(Rational *r);                // return r as a double

void add_rat(Rational *r, Rational *s);   // Set r to r + s
void sub_rat(Rational *r, Rational *s);   // Set r to r - s
void mul_rat(Rational *r, Rational *s);   // Set r to r * s
void div_rat(Rational *r, Rational *s);   // Set r to r / s

void print_rat(Rational *r, int p);   // Print r to p decimal places
void print_ratio(Rational *r);        // Print r as a ratio num/denom

void normalize(Rational *r);      // Put r into lowest terms
int gcd(int x, int y);            // greatest common divisor
```

To Test Your Program

In Unix-like operating systems, you can compile the two `.c` files and create an executable file using

```
> gcc -Wall -std=c99 -lm Lab04_rational.c Lab04_rat_client.c
```

This produces the usual `a.out` file for you to run.

What to Turn in

Create a folder, name it with your name and `"Lab4"`, copy the `.h` and two `.c` files into it. Zip the folder and submit the zip file. (Don't use rar etc.)

