Step 7: Pointers

This assigment is done in class on (monday)02-24-14 AND (friday)02-21-14.

This assignment is due for monday class 02-24-14 AND for the firday class 02-21-14.

Getting Started

By now you should be all logged on. You should still have a cse251 directory in your home directory from last week. Please change into that directory.

You should be able to do a pwd and see that you are in the cse251 directory at this time.

Remember, start gedit with this command line:

```
gedit &
```

Create a new program called pointers.c with the following code:

```
#include <stdio.h>

/*
 * Name : <Insert name>
 * Program to experiment with hexadecimal
 * and pointers
 */

int main()
{
    int a = 0;
    int b = 5;
    int c = 1957;

    printf("%d\n", a);
    printf("%d\n", b);
    printf("%d\n", c);
}
```

Be sure it compiles and runs. The commands to compile and run are, of course:

```
gcc -o pointers -lm pointers.c
./pointers
```

1. Hexadecimal Numbers and Addresses

When you run the pointers.c program as is, it prints out the values of a, b, and c as normal decimal numbers:

```
cbowen@ubuntu:~/cse251$ ./pointers
0
5
1957
```

I would like to know the value of each of these numbers in hexadecimal (base-16) instead of decimal. Change the printf statements to this:

```
    printf("%d: %x\n", a, a);
    printf("%d: %x\n", b, b);
    printf("%d: %x\n", c, c);
```

You should now be able to see the hexadecimal value of each number.

You can input values in hexadecimal in a C program using this notation: 0x2251. Change line that declares the variable c to this:

```
    int c = 0x2251;
```

When you run this you should be able to see what the decimal value of 0x2251 is. Note that %d in a printf statement prints in decimal, while %x prints in hexadecimal. Are you sure which is which in the output?

Add this line of code to your program:

```
    printf("%x\n", (int)&a);
```

The operator & reads as "at" and means we are taking the address of the variable. So, this is going to tell us where the variable a is in memory. I have "cast" it to an integer, since that is what the %x format expects. If I leave off the (int), it will still work, but I'll get a compiler warning.

Note: You may get a compiler warning stating the following with the above code:

```
warning: cast from pointer to integer of different size
```

This is caused by the fact that you are on a 64-bit computer (with 64-bit memory addresses) and are casting the 64-bit memory address to a 32-bit integer. If you replace the above code with the following:

```
    printf("%lx\n", (long int)&a);
```

that will fix the warning. The pointer is cast to a *long int*, which is a 64-bit data type. Also note the use of %lx, rather than %x. The l means to expect a *long int* value.

Run this and see what you get.

Hint: You can do math on the address values if you cast them to integers. For example, (int)&c - (int)&b will work just fine.

2. Using Pointers

Add these variables to your program:

```
    int *pA = &a;
    int *pB = &b;
    int *p;
```

Each of these variables is a pointer variable. It will contain an address in memory of *another* integer variable. Right now pA is pointing to the variable a and pB is pointing to the variable b. I have not initialized p to anything yet.

Add this printf statement to the end of your program:

```
    printf("a=%d, pA=%x, *pA=%d\n", a, (int)pA, *pA);
```

This statement will print the address of the variable a and the value of the variable **a**. The * operator means to dereference. This means get what is at that address. So, ***pA** gives us what is at the address of **a**, in this case zero.

Try adding these lines to the end of your program and see what happens:

```
    a = 47;
    printf("a=%d, pA=%x, *pA=%d\n", a, (int)pA, *pA);
    *pA = 99;
    printf("a=%d, pA=%x, *pA=%d\n", a, (int)pA, *pA);
```

Because **pA** points to **a**, changing **a** also changes ***pA**. Because **pA** points to **a**, changing ***pA** also changes **a**.

Note that ***pA** can be used to get the value pointed to by **pA** and also to set the value pointed to by **pA**. So, I can **get** a value though a pointer and **set** a value though a pointer.

Now add these lines of code:

```
    printf("Next experiment:\n");
    p = pA;
    *p = 22;
    p = pB;
    *p = 18;
    p = &b;
    *p = 108;
    p = pA;
    *p = 2;
    printf("a=%d, pA=%x, *pA=%d\n", a, (int)pA, *pA);
    printf("b=%d, pB=%x, *pB=%d\n", b, (int)pB, *pB);
    printf("p=%x, *p=%d\n", (int)p, *p);
```

3. Using Pointers

Create a new program named quadratic.c and put this code into it:

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

/*
 * Name : <Insert name here>
 * Program to compute the zeros of a
 * quadratic equation
 */

int main()
{
    /* Values for the quadratic formula */
    double a, b, c;
    double z1r, z1i;    /* First zero */
    double z2r, z2i;    /* Second zero */
    double discriminant;

    printf("Input a: ");
    scanf("%lf", &a);
    printf("Input b: ");
    scanf("%lf", &b);
    printf("Input c: ");
    scanf("%lf", &c);

    /*
     * This code computes the quadratic equation
     * for both real and complex zeros
     */

    /* Compute the discriminant */
    discriminant = b * b - 4 * a * c;
    if(discriminant >= 0)
    {
        /* If the discriminant is greater than or
           equal to zero, the zeros are real */
        z1r = (-b + sqrt(discriminant)) / (2 * a);
        z2r = (-b - sqrt(discriminant)) / (2 * a);
        z1i = 0;
        z2i = 0;
    }
    else
    {
        /* If the discriminant is less than zero
           the zeros are complex  */
        z1r = -b / (2 * a);
        z2r = z1r;
        z1i = sqrt(-discriminant) / (2 * a);
        z2i = -sqrt(-discriminant) / (2 * a);
    }

    /* Display the results */
    printf("Zero 1: %f + %fj\n", z1r, z1i);
    printf("Zero 2: %f + %fj\n", z2r, z2i);

}
```

Be sure this runs. If you try a=1, b=1, and c=1, the answer should be:

```
cbowen@ubuntu:~/cse251$ ./filter
Input a: 1
Input b: 1
Input c: 1
Zero 1: -0.500000 + 0.866025j
Zero 2: -0.500000 + -0.866025j
```

The program currently computes the quadratic equation. The quadratic formula computes the zeros for the quadratic equation:

$$ax^2 + bx + c = 0$$

The quadratic equation is, of course:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This equation works fine if the answer is real. If we think of the answer as a complex number, the imaginary part is zero.

The equation within the radical is called the discriminant. If the discriminant is negative, the answer is complex and can be written this way:

$$\frac{-b}{2a} \pm \frac{\sqrt{-(b^2 - 4ac)}}{2a} j$$

In this equation, j^2=-1. I use j instead of i because you all are supposed to be electrical engineers and they prefer j instead of i because i is taken.

The program right now computes the first version if the discriminant is greater than or equal to zero and the second if it is negative. It then outputs the result. Be sure you understand how the program works.

The first thing we are going to do is to create a function to input the quadratic equation we want the zeros of. Add this function to your program at the end:

```
/*
 * Input a quadratic equation as a, b, and c
 */
void InputQuadraticEquation(double *a, double *b, double *c)
{
    printf("Input a: ");
    scanf("%lf", a);
    printf("Input b: ");
    scanf("%lf", b);
    printf("Input c: ");
    scanf("%lf", c);
}
```

Be sure to add the function declaration. Be sure this compiles okay before proceeding.

> In the past we have used an & when passing variable to scanf. Do you know why we don't do that now? Scanf expects to be passed a pointer to a variable, not the value of a variable. We did that before by using & to get the address of a variable to pass to scanf. We don't do that here because a, b, and c are already pointers to a variable. They are already what scanf expects, so we don't take their address. If you do add an & here, your program will probably crash.

Next, replace the printf/scanf calls which currently obtain these values with this call:

```
    InputQuadraticEquation(&a, &b, &c);
```

This should work as before.

It makes sense that we might need to use that quadratic equation solution code in more than one place. So, it makes sense to move it into a function. To do this, we need to answer two questions:

1. What does the function need to know?
2. What does the function need to return?

The function needs to know a, b, and c. So, we just pass them as parameters to the function like this:

```
  QuadraticEquation(double a, double b, double c,
```

The function needs to return two complex zeros. In my program right now the results are z1r, z1i, z2r, and z2i for the real and imaginary parts of two zeros. Since I need to return four values, I can't do that using the return value of the function. That only allows us to return one value. So, I'll make the function void:

```
void QuadraticEquation(double a, double b, double c,
```

I need to return four double values. To do that, I'll pass them by reference to the function. So, the arguments to the function will be pointers to where the result should go. I commonly avoid confusion with pointers by prefixing the variable name with a "p". That way I know it's a pointer rather than a variable. Using that convention, the entire function definition (other than the body, of course) looks like this:

```
void QuadraticEquation(double a, double b, double c,
                       double *pZ1r, double *pZ1i, double *pZ2r, double *pZ2i)
{
    /* Insert function body here */
}
```

Now, I'm going to leave it up to you to fill in this function and replace the code that originally computed this result with a call to this function.

> Hints:

Where you did z1r =, you will now do *pZ1r =.
Be sure you use & only on the variables you are passing by reference in your call to QuadraticEquation.
The variable discriminant will need to be declared in the function.

Turn in quadratic.c via **http://www.cse.msu.edu/handin** and we are done for the day.

CSE 251