ECE 264: Advanced C Programming

Programming Assignment 3 - Structures, Due 6/29/19

In PA01, you learned how to

- compile programs using gcc (remember turning on the warning messages)
- compile and test a program with multiple files using make
- evaluate test coverage using gcov
- compare performance using gprof

In PA02, you learned the structure of the stack memory.

PA03 is the first part of two related assignments. PA04 is built on PA03. The instructors will not provide solutions for PA03. Your PA04 will need your solution for PA03.

1 Learning Goals

You will learn how to:

- Create new data types using struct
- Pass arguments of the new type
- Understand the difference between passing a structure object and passing a pointer to a structure object
- Integrate a function in a range
- Write a Makefile
- Remove untested code marked by gcov

2 Background

2.1 Creating New Types in C

C has some built-in data types, such as char (characters), int (integers), float (single-precision floating point numbers), and double (double-precision floating point numbers).

C also allows programmers to define new types. There are multiple ways to define new types. ECE 264 uses the following style:

```
typedef struct
{
    // attributes: type - name pairs
    // for example
    int ival;
    double dval;
} TypeName; // remember the ending ;
```

The naming convention we will use in class is that the name of a new type is a noun and it starts with a capital letter.

After creating a type, it can be used to create "objects" (using the term from C++). For example,

TypeName obj1;

When we declare an int variable, we are telling the compiler: "this variable holds one value: an integer." Similarly, when we create an object of type TypeName, we are telling the compiler: "this object is a TypeName, which means that it holds two values: an integer and a double."

The attributes of obj1 can be referred to using ".", for example,

obj1.ival and obj1.dval

2.2 Pointers in C

In C programs, a pointer means a way to refer to a memory address. For example,

int a = 5; // an integer and its value is 5

int * pt; // an integer pointer

pt = & a; // pt's value stores a's address

You can merge the two lines into one:

int * pt = & a;

After assigning a's address to pt, pt can be used to refer to a's value, for example,

int b = * pt; // pt at the right hand side (RHS) of =, read the value of a

// b's value is 5 because a's value is 5

It is equivalent to

int b = a;

If pt is at the left hand side (LHS) of =, the value of a is modified:

* pt = -3; // a's value is -3

You cannot assign an arbitrary address to a pointer (except NULL). If you do something like the following, you will get a compiler warning

int * pt = 5;

The reason is that the operating system determines the addresses available to each program. You do not know whether 5 is a valid address available to your program (in fact, 5 is never a valid address).

NULL is a special address.

int * pt = NULL;

means pt points to nowhere. This is useful when you do not know where pt should point to yet. In later parts of ECE 264, you will learn that NULL can serve as a "sentinel" value that indicates important things, like the end of pointer chains.

If you create a pointer and do not know where it should point to yet, initialize it to NULL. Uninitialized pointers store garbage addresses and the program's behavior is unpredictable. Remember, C does not initialize values for you. If you do not initialize a pointer, it stores garbage (not NULL).

2.3 Pointers and Structures

Earlier, you learned that if TypeName is the name of a structure, you can create an object using this syntax:

TypeName obj;

Then, you can use . to refer to an attribute:

obj.ival = 2017; // assign 2017 to the integer attribute

It is also possible to create a pointer to an object.

TypeName * opt = obj;

To refer to an attribute, you need to replace . by ->

opt -> ival = 2017;

Note: you could also write (* opt).ival to get the same effect. Why? Recall that (* opt) is like referring to whatever opt points to. Since opt points to obj, (* opt).ival is the same as saying obj.ival. We recommend that you use the -> syntax, because it is easier to read.

One thing to notice is that (* opt).ival is not the same as *opt.ival. Why? The order of operations in the C language states that the member operator . has a higher precedence than the dereference operator *. So *opt.ival refers to whatever opt.ival points to!

2.4 Function Arguments

When an argument is passed to a function, a copy is created. Thus, modifying the value inside the called function (callee) does not affect the value in the caller when the called function finishes. Consider this example:

```
void f1(int a) // callee
{
    a = 264;
}

// caller

int b = 2017;
f1(b);
// b is still 2017, not 264
```

To modify the value of b, it is necessary to pass the address of b:

```
void f2(int * a) // callee, a is a pointer
{
    * a = 264;
}

int b = 2017;
f2(& b); // pass b's address
// b is 264
```

The same rule applies when the argument is an object:

```
void f3(TypeName obj) // callee
{
    obj.ival = 264;
}

// caller
TypeName obj;
obj.ival = 2017;
f3(obj);
// obj.ival is still 2017, not 264
```

To modify the value, it is necessary to pass the address;

```
void f4(TypeName * opt) // callee
{
    opt -> ival = 264; // notice ->
}

// caller
TypeName obj;
obj.ival = 2017;
f4(& obj); // notice &
// obj.ival is 264
```

2.5 Integrate a Function

This assignment asks you to write two C functions to calculate the definite integration of a function. Suppose f is a function that can be integrated over the range [lowerlimit, upperlimit]. If it is possible to find a closed form for the integration of f, let's call it int_f, then the answer would be simple:

int_f(upperlimit) - int_f(lowerlimit)

In many cases, however, a closed form is unavailable and we can use numeric methods to approximate the answer.

One easy approximation assumes that f can be approximated by a straight line. Thus, the integration can be approximated by using the middle of [lowerlimit, upperlimit]:

(upperlimit - lowerlimit) x f((upperlimit + lowerlimit) / 2)

This approximation is incorrect if f is not a a straight line. This assignment asks you to approximate the answer by dividing [lowerlimit, upperlimit] into several intervals. Let's use n as the number of intervals. The integration is approximated by

```
(upperlimit - lowerlimit) / n x
(
f(lowerlimit) +
f((lowerlimit) +       (upperlimit - lowerlimit) / n) +
f((lowerlimit) + 2 * (upperlimit - lowerlimit) / n) +
f((lowerlimit) + 3 * (upperlimit - lowerlimit) / n)  +
...
f(lowerlimit + (n - 1) * (upperlimit - lowerlimit) / n)
)
```

You can assume that n is a positive integer. You do not need to check whether n is zero. You can assume that upperlimit is greater than lowerlimit.

3 What do you need to do ?

The only file you need to modify in this assignment is answer03.c. Please read every file in this assignment and understand all details. PA04 will require that you write the entire program.

You need to implement two functions:

double integrate1(Range rng);

The argument provides lowerlimit, upperlimit, and the number of intervals (intervals). They are three attributes of the type Range, defined in pa03.h. integrate1 should perform a numerical integration of the

function func (which will be provided for you in files named func1.c, func2.c, etc.) using the method described above.

and

void integrate2(RangeAnswer * rngans);

This is the function that pa03.c calls to find the integral: it takes a RangeAnswer struct as an argument (also defined in pa03.h). Because the argument is a pointer, setting the answer field of rngans inside integrate2 will allow whoever calls integrate2 to retrieve the answer.

You are strongly encouraged to use integrate1 when implementing integrate2. Thus, when testing integrate2, integrate1 is also tested, i.e., when INTEGRATE_2 is defined, INTEGRATE_1 is also defined.

Both functions call the function to be integrated func.

3.1 Write DRY Code

When writing code, a general rule is "Don't Repeat Yourself" (DRY). In this assignment, you should implement integrate2 using integrate1 but these two functions are somewhat similar. This is important because you need to implement (and debug) the overlapped part only once.

If you implement integrate2 independently of integrate1, you are creating WET ("We Enjoy Typing") code. You have two similar but different functions. This invites mistakes. Consider the situation when you find a mistake in integrate1 and correct it. It is highly likely that you forget to correct the mistake in integrate2.

DRY code is good. WET code is bad.

3.2 Testing your integrator

Five functions are created for you to test. If a function uses the mathematics library in C (such as func4.c and func5.c), please include math.h and add -lm after gcc. (You can see how this works in the Makefile)

```
file      function          integration
func1     x                 x * x / 2
func2     x * x             x * x * x / 3
func3     x * x - 3 * x     x * x * x / 3 - 3 * x * x / 2
func4     sin(x)            -cos(x)
func5     cos(x) + sin(x)     sin(x) - cos(x)
```

Each of these functions defines its own version of func that gets compiled together with answer03.c and pa03.c to create your program. So, to integrate the function defined in func1.c, you would compile that and link it together with answer03.c and pa03.c. Here are the Makefile rules that build an integrator for the function in func1.c:

```
pa03-func1-1: pa03a.o answer03.o func1.o
    $(GCC) pa03a.o answer03.o func1.o -o pa03-func1-1


pa03a.o: pa03.c pa03.h
    $(GCC) -c -DINTEGRATE_1 pa03.c -o pa03a.o


func1.o: func1.c
    $(GCC) -c func1.c
```

```
answer03.o: answer03.c pa03.h
    $(GCC) -c answer03.c
```

(If you just run make you will get executable programs that will test the integrate1 and integrate2 versions of your integrator for all five of the provided test functions)

To test your integrator, run pa03-func1-1 on a test input:

> ./pa03-func1-1 tests/test-func1-1

The format of a test file is:

```
<lower bound (floating point)>
<upper bound (floating point)>
<# intervals (integer)>
<expected answer (floating point)>
```

The provided test files (in the directory tests/) all have the answer you are expected to get. Feel free to generate new tests, knowing what the "correct" answer is for the functions you are testing.

You can run through all of our test cases using

make pa03-test1

to test integrate1, and

make pa03-test2

to test integrate2.

4 Grading

You will receive zero if your program has error or warning from gcc.

The teaching staff will run your program using some test cases for the two integration functions. Your score is proportional to the number of test cases the program passes. For example, suppose 10 test cases are used for each integration function, there are 20 test cases. Passing each test receives 5% of the score. Passing means the program returns EXIT_SUCCESS and provides correct answers.

You need to use gcov to check test coverage. It is expected that every line in answer03.c is tested by these test cases. If any line is marked ##### by gcov, you will lose 5% of the score. If your answer03.c has more than 20 untested lines, you will receive a zero for this assignment