CS 352 (Fall 17): System Programming and UNIX

Project #4

Reading Files, `malloc()`, Structs

due at 5pm, Tue 26 Sep 2017

1 Overview

This project has three programs - each of which focuses on a new feature of the C language that we're exploring.

1.1 Standard Requirements

- Your C code should adhere to the coding standards for this class: `http://lecturer-russ.appspot.com/classes/cs352/fall17/DOCS/coding-standards.html`
- Your programs should indicate whether or not they executed without any problems via their **exit status** - that is, the value returned by `main()`. If you return 0, that means "Normal, no problems." Any other value means that an error occurred.

  In this class, we will always use the value 1 to indicate error; however, outside this class, many programs use many different exit status values - to indicate differnt types of errors.

  In `bash`, you can check the exit status of any command (including your programs) by typing `echo $?` immediately after the program runs (don't run **any** commands in-between).

2 Turning In Your Code

Turn in your code using D2L. Turn in the following files:

```
compareBytes.c
intRollingBuffer.c
structPacking.c
```

3 Grading

We have provided compiled versions of these programs for you; you should download them and run them (on Lectura) for comparison. The programs can be copied from `/home/russelll/cs352_website/projects/proj04/` on Lectura. (The UNIX commands you've learned, such as `ls, cd, cp` will all be useful here.)

We have also provided our grading script. (We'll do this for the first couple of assignments, in order to help you understand the requirements - but we will stop doing this as the programs get more complex!) You can copy it from `/home/russelll/cs352_website/projects/proj04/grade_proj04` on Lectura, or from the web.

Finally, we have provided a few example inputs to test your program with. (We will use additional ones when we grade your program.) **You should write additional testcases** - try to be creative, and exercise your program in new ways.

NOTE 1: Since this project has multiple programs, it needs testcases for each one. The testcases are named `test_<progName>_*`. In order for the grading script to see your new testcases, please name them according to this standard.

NOTE 2: Some of the programs in this project use `stdin` as their input; name those testcases `test_*.stdin` . Some use command-line arguments; name those testcases `test_*.args` .

3.1 Exact Output

In each Project this semester, most of your grade will come from automatic testing of the code. You must match `stdout` **EXACTLY**, byte for byte. Common mistakes include:

- Extra or missing spaces
- Extra or missing blank lines
- Misspelled words
- Different capitalization

Your code must also give the same return value (0 or 1) as the example executable in every case.

`stderr` will be handled a little more loosely. In each testcase, we will check to see if the standard executable reported **some** error message to `stderr` or not. If it did, then your program must as well; if not, then your program must not print anything, either. However, we will not be comparing the exact error messages.

NOTE: Each testcase either passes, or fails entirely. We do not give partial credit for any testcase - although you may, of course, pass some testcases but not others.

3.2 Running the Grading Script

To run the grading script, arrange your files like this, and then run `./grade_proj04`

```
grade_proj04

example_*
test_*

compareBytes.c
intRollingBuffer.c
structPacking.c
```

The grading script does a number of things:

- Confirms that we have an example executable for each program.
- Confirms that each of the required files has the proper name, in a directory with the proper name. If not, you lose all points for that program.
- Compiles your code. If your code doesn't compile at all, then you lose all points for that program. If it compiles but has warnings, then you will get a heavy deduction.
- Runs your code against all of the testcases. In each case, it runs both your code, and also the example executable. It checks to make sure that both have the same return code - and then also checks to make sure that the outputs match. If not, then it will give you a zero for that testcase.

  (If you have no testcases for some program, then the script will give you a zero for that program.)

At the end, the grading script reports a score; this score is determined by the number of testcases that you passed - modified for any deductions.

3.3 Hand Grading

In addition, each program will be looked at by a human, to check for things which we can't automatically check, like:

- Some of the details of the spec
- Good comments
- Good indentation
- Reasonable variable names

3.4 Point Distribution

In this project, your code will be graded by a script, with your score determined by how many testcases you pass for each program. After that score is calculated, a number of penalties may be applied.

3.4.1 Weight of each program:

If any program compiles and runs, but warnings were produced by the compiler, you will lose half your score for that program.

- 20% - `compareBytes`
- 30% - `intRollingBytes`
- 20% - `structPacking`

The last 30% of the score will come from hand grading by the TAs.

4 Program 1 - Compare Bytes

This program opens two files, reads them both, and reports the first place where the two bytes diverge.

Name your file `compareBytes.c`

4.1 Input

This program reads its parameters from **the command line** - not from `stdin`. The program must have two command-line arguments, which are the names of files; you will open both with `fopen()`.

Read one byte from each file at a time. You can do this using `fscanf()` - but it's generally easier to use a specialized function, such as the `getchar()` family. Check the man page for `getchar()`; as I've done before, the "right" function you need isn't `getchar()` - but it's something related to it, which is on the same man page.

So long as the two files are identical, continue to read them, byte by byte, until either you hit EOF, or some byte which is not the same in the two files. If the files are absolutely identical (meaning that they have the same size **and contents**), then print out a certain message (see the example executable to see what it is).

If the two files are identical up to one point - and then one ends (but not the other), then you will print out a different message; this message includes the next 4 bytes in the longer file (fewer if there are less than 4 bytes left in that file). Again, check the example executable to see how this is formatted.

Finally, if the two files are the same up to some point (but then both continue), you will print a third type of message - along with (up to) the next 4 bytes from the file.

4.2 Required Functions

You must write two functions[1]. Use them in your implementation!

[1] You are allowed to write more if you want - but these two are required.

4.2.1 `dumpNext()`

This function must be called, when it's time to report a difference between two files (either because they had different contents, or because one was shorter than the other). It takes three parameters, and returns nothing:

```
void dumpNext(char *filename, FILE *fp, unsigned char c);
```

The first two paramemters are obvious: they are the name of the file which is being printed, and the file handle which you have been using to read from it. The third parameter is the **first character which is different** - this is necessary because (of course), your program has **already read** one character from the file - and this was the one that you found was different!

This function should print out a single line, including the newline at the end. If the file has at least 4 bytes left (including the one that was passed as a parameter), then print out the following message:

```
The next 4 bytes of ...
```

(see the example executable for the exact format).

However, if the file has fewer than 4 bytes left (including the one parameter), then you will print out something like this (again, check the example for the exact format):

```
<filename> has only <n> bytes left ...
```

4.2.2 `printChar()`

This function must be called, as a utility function, from `dumpNext()`. It prints out the information about a single byte from the file; `dumpNext()` may call it as many as 4 times. This function takes a single parameter, which is the byte to print:

```
void printChar(unsigned char c);
```

This function will print out the hex value of the character, an equals sign, and then one of three things:

- The character itself, if it is printable.
- A special message if the character is null.
- A different message if the character is non-null but not printable.

As always, check the example executable for the exact format.

4.3 Error Conditions

Your program should normally print only to `stdout` (using `printf()`). This includes even output which says that "the files are different," and gives information about that - since having two files which are different is **not an error.**

However, we will **do something unusual with the exit status.** Your program should **only** return 0 exit status if the two input files are **absolutely identical.** If your program detects any difference, then it must report it (as described above), and then **return 1.**

In addition, your program must detect things which are actually errors. Print out a message to `stderr`, and then immediately terminate the program (with exit status of 1), if any one of the following happens:

- There are anything other than exactly 2 command-line arguments (plus the name of the program itself). Note that too many arguments should be handled as an error, just like too few.
- Your program is unable to open one or both of the input files.

5 Program 2 - Int Rolling Buffer

It's time to call `malloc()`! In this program, you will allocate an array of integers (based on a size you read from `stdin`) and then gradually insert and delete values in the array. You will add two **extra** elements in the array - in addition to the ones required - and use them to help you show that you have not overflowed the array you allocated.

Name your file `intRollingBuffer.c`

WARNING: You MUST check the return code from `malloc()`. Write code that will print out an error message - and terminate the progrm - if `malloc()` fails. In all likelihood, this condition will never occur in your program - becuse your program is small - but this is a **critical** habit for later. When you have a large program, failing `malloc()` is a **real possibility**, and you must plan for how you're going to handle it.

5.1 Input

The input to this program is a series of integers, which are read from `stdin`. The first integer is special; it indicates the size of the buffer that you must allocate. This integer must be ≥ 1. (Also, there must not be any error reading this value.)

The rest of the values in `stdin` are integers which will be written **into** the array; these can have any value, positive or negative. In addition, while you are reading these additional integers, you must handle non-numeric characters; print out an "OOPS" message any time that you are not able to read an integer (but you haven't hit EOF). See the example executable to see how this works -

but I'll tell you one thing: the OOPS message in this program prints the value in hex, and then simply **prints the character** - there is no checking to see if the character is printable, or if it is null. So some of the outputs might come out looking a little odd!

5.2 The Buffer and Its Magic Values

When you read the first integer from `stdin`, it is telling you how many integers your array needs to store. However, we also would like to be able to debug a bit. So for this reason, you must actually allocate an array which is two elements **larger** than the input asks for. Set the first and last values to certain constants (run the example executable to find out what they are).

The middle values in the array - that is, the "real" values - you must initialize to be sequential integers, starting at 1. (These values will change as soon as you start running in the algorithm - but the first and last values should **never change.** They are just there for debugging!

5.3 The Algorithm

After you have allocated the array of integers (and filled it in), but before you read anything else from `stdin`, print out the current state of the array. Do this with two calls to `dumpInts()`, a function you will write (described below).

Then, as you read each integer from `stdin`, do the following:

- Remove the first element from the array
- Insert the new element based on its value (that is, keep the array sorted at all times)
- Print out the array again.

(Never change the first and last values - they are just there for debugging, to make sure that you don't modify anything outside the range of the array.)

As we've done in some projects before, leave a blank line between the various printouts - but do **not** have a blank line at the beginning or end of the output.

5.4 Required Function: `dumpInts()`

Each time that you want to print out the contents of the array, you must do it by calling `dumpInts()` **twice** - once to print out decimal values, and once to print out hex values.

You must implement `dumpInts()`. This function takes three parameters, and returns nothing:

```
void dumpInts(int *array, int count, int hex);
```

The first parameter is the pointer to the first element of the array (it should point to the unchanging debug element at the head). The second parameter is number of elements in the array (including the debugging elements at the front and back). The third parameter is an int, which indicates whether you should print the valuse as decimal or as hex. If the value is nonzero (meaning true), then print hex; if the parameter is 0, then print decimal.

Use the format specifiers in `printf()` - in particular, the length modifier - to make sure that the columns of your output line up well. Since we are using spaces to align the columns, you will have leading spaces on most of your lines. However, you **must not** have trailing spaces on any line.

5.5 Error Conditions

In this program, errors come in two categories. While reading the first integer, if you have any error, report it on `stderr` and immediately terminate the program with an exit status of 1. Likewise, if you successfully read that integer but it is less than 1, then report an error on `stderr` and terminate the program with 1. Finally, if you fail to allocate the array with `malloc()`, report an error and terminate the program with 1.

However, once these initial checks have been passed, no other errors should terminate the program, or be printed to `stderr`. In the rest of the program, the only error to handle is a character that cannot be interpreted as an integer - as detailed above, print a warning to `stdout` (**NOT** `stderr`), and keep running the program.

At the end of the program (after you have hit EOF), return 0 if there were no warnings at all - but return 1 if you ever had to report one (or more of these warnings.

6 Program 3 - Struct Packingr

Let's look at the internals of memory again. This time, we'll look at how structs are laid out. You will include a header that I provided - with three different structs defined inside it. All three structs have the same fields - but different arrangements. In some, I changed the order; in others, I changed an array into individual variables.

You will declare a local variable, which contains an array of the first type, and will fill it by reading from `stdin`. You will then print out the raw data (ignoring the fields). Last, you will cast the array to the two other struct types, and read those structs - and see that you will see different values (because the structs are laid out differently in memory).

Name your file `structPacking.c`

6.1 Input

The input to this program is a little subtle: it starts with an integer, which must be in the range 0 to 255 (inclusive), because you are going to use this value to initialize an array of bytes.

Following that integer are 4 different groups of inputs. Each group has the same form: 3 integers, followed by 2 characters. So for instance, one of our testcases has these contents:

```
204   1 2 3xy    10 20 30ab    -1 -2 -3[]    2 3 5.-
```

This input is interpreted as:

- Init value of 204=0xcc
- Integers 1,2,3, followed by characters 'x','y'
- Integers 10,20,30, followed by characters 'a','b'
- Integers -1,-2,-3, followed by characters '[',']'
- Integers 2,3,5, followed by characters '.','-'

(You'll notice that this is designed to work with `scanf()` - where you would have three `%d` specifiers, followed by two `%c`.)

6.2 The `Foo` Array

Your program must declare an array of 4 `Foo` objects. `Foo` is declared in `proj04.h`, which I will provide; you don't have to do anything to get it - except to include the header in your C code.

These 4 Foo objects will be the buffer in which we will be working. We will fill in the elements, and then take pointers and cast them - and see what this array looks like, on the inside.

6.2.1 Initializing the Array

After you declare the `Foo()` array, you must initialize it. Normally, we would fill it with zeros, and there are good C functions for this. But in this case, we want to see if there are any "holes" inside the struct which are not being used.

Therefore, we will initialize it to an integer (and the value we use is the first thing we read from `stdin`). Cast the array-of-`Foo` object to a pointer to an `unsigned char`. Then write a `for()` loop to fill it.

It is importatnt that you use `sizeof()` to find the size of the array - instead of hard codign it! NO only is this better style, it also means that your code will work if the structure is changed. (I might change `proj04.h` when I grade your code!)

6.2.2 Filling in the Array

(After you have filled all of the **bytes** of the array of `Foo` structs, you will fill in the values. Since the struct has some holes, there will be some parts of the struct which were written to only by the byte-wise initialization.)

To fill in this array, you **must** use a function named `fillOneFoo()`. This takes a single parameter (a pointer to a `Foo` object), and returns an integer.

Inside `fillOneFoo()`, call `scanf()` to read data from `stdin`; have it write what is read directly into the `Foo` object you've been given.[2]

However, if `fillOneFoo()` has any problems with `scanf()`, it must print out an apporpriate message to `stderr` and then return 1 to indicate error. Remember, however, that this is **not** `main()`, and so returning 1 from this function does **not** kill the program. Instead, `main()` should check the return code from `fillOnFoo()` and terminate the program if an error occurred.

6.3 Printing Sizes

The first 3 lines of the output will show the size (in bytes) of all 4 structs. (Again, don't hard-code these numbers, since I might change it when I do the grading.)

6.4 Printing Raw Data

Next, print the entire `Foo` array, as a series of raw integers. Use a helper function to do this. Write the function `printInts`, which takes two parameters (a pointer to `int`, and an integer representing the length of the array. (Remember that this must be measured in `ints` - not in bytes!)

6.5 Printing `Bar,Baz`

Finally, get a pointer to the `Foo` array, and cast it to a pointer of type `Bar`. Pass this pointer to the `printOneBar()` function, which takes no other parameters, and returns nothing. This will print out the contents of the **exact same memory** - but interpreting it as if it was a `Bar` struct, instead of `Foo`. However, if you look closely, everything should still line up properly - because while `Foo` has an array of three `ints`, `Bar` just simply has three `int` variables - which consumes exactly the same space in memory.

After you have printed the first `Bar`, use array indexing (on your casted pointer) to print a second `Bar` object. That is, pretend that you have an array of `Bar` structs - you should print the first two.

Finally, cast the `Foo` array to a `Baz` pointer. Use the `printOneBaz()` function to print two different `Baz` objects (again, pretending that we had an array of two of them). Look closely - things don't line up as nicely here. Why is the size of `Baz` larger than `Foo` and `Bar`? Why don't the fields likne up properly?

6.6 Error Conditions

Print an error message to `stderr`, and immediately terminate the program with an exit code of 1, if you encounter any error reading the various inputs from `stdin`. In addition, if the very first input (the init value) is not in the range 0

[2]HINT: How do you find the address of a given field inside a struct? Remember that the `&` operator will work on **any** l-value.

to 255 (inclusive), then report an error and terminate the program with an exit code of 1.

Otherwise, you should be able to run straight to the end of the program, and return 0.