CS 352 (Fall 17): System Programming and UNIX

Project #11

Parsing, Function Pointers, and `gcov`

due at 5pm, Mon 27 Nov 2017

1 Overview

In this project, you will write two programs. The first will be a parser for a config file; I'll show you how to read in a file line by line, and you will then parse it character by character (instead of using `scanf()` to do the work for you). At the end, you will print the config file back out - first, in the original order, and then again, but sorted alphabetically (and groups with duplicate names joined together). However, your output will have the output in a very specific format, with any extra whitespace removed.

In the second, you will read input from a user (perhaps interactive input), which are mathematical expressions. You will call a function that I've provided, which will parse this input into a parse tree, representing the mathematical expressions; you will then evaluate the expression by recursing through the tree. The trick here is that each node will use a **function pointer** to show how to evaluate that node.

You will use `gcov` to check for coverage on both of the programs you write; you will have to generate enough testcases to cover all of the code (except for `malloc()` error checking).

1.1 Loosened Makefile Restrictions

For this project, you must build a `Makefile` from scratch. You must write it such that the commands

```
make parseConfigFile
make calculator
```

will build your programs; include all of the necessary dependencies so that they will only rebuild when necessary.

You must ensure that `gcc` runs with the correct options to generate `gcov` reports (see ICA 13-2 or slide deck 08); the grading script will be running a `gcov` report after the testcases are executed.

1.2 Testcases Required

For this project, you must turn in some testcases for each program (you may assume that we will include the standard testcases when we grade your program - but you will need to provide some more, as well).

Part of your grade will come from covering all of the lines of both programs (except for `malloc()` error handling).

1.2.1 Grading

While you must provide testcases to cover your programs, we will be using a standard set of testcases to check your program for correctness (so that everyone is graded the same way).

As before, each testcase will be broken into quarters. Half of the testcase score comes from matching `stdout` (and you can't gain any of the other points if you don't match this). One-quarter comes from `stderr` and the exit status; one-quarter comes from a `valgrind` check. (As before, you must free **everything** when the program ends.)

The points will be broken down as follows:

- 40% for the `parseConfigFile` program
- 10% for the `calculator` program
- 10% for `gcov` coverage, based on the **testcases you provided** (along with the testcases I provided)
- 40% hand grading

1.3 Turning In Your Code

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

```
parseConfigFile.c
calculator.c
expr_student.h

<various testcases.  See my examples for the naming style>

Makefile
```

Please submit them as separate files; do not zip them up into a single zip file or tarball.

Due to limitations in D2L, please rename your `Makefile` to `Makefile.txt` in order to turn it in. I'll change it back when I collect the files.

2 Function Pointers

A **function pointer** is a variable which stores the address of a function. The variable has a type, which gives the types of the parameters and also the return type. Function pointers can be called anywhere that an ordinary function would be called; they use parameters in the same way as ordinary functions, and also return values as normal. The key distinction is that the function pointer is a variable - so it can be stored inside a struct, copied to other locations, etc.

To declare a function pointer, we write a declaration like this:

```
int (*fp)(char,float);
```

In this declaration:

- The name of the variable is `fp`
- The function takes two parameters: a `char` and a `float`
- The function returns an `int`

Note that the parens around the variable name are **required** - without the parens, this would be a function **prototype** that returns a pointer to an integer - instead of a function pointer to a function that returns an integer.

We initialize function pointers by using `&` on a function name[1]:

```
int exampleFunc(char,float);    // function prototype
fp = &exampleFunc;
```

We call a function pointer by using `*` as if we were **dereferencing** a pointer[2] - but then we pass parameters as well:

```
int returnValue = (*fp)('a', 1.5);   // call the function pointer
```

If (as is usually the case) the function pointer is a field inside a struct, you access the **variable** exactly as normal - but then use it as a function pointer. For instance, if `fp` above was a field inside a struct, and we had a pointer to that struct named `obj`, we would call the function pointer like this:

```
int returnValue = (*obj->fp)('a', 1.5);   // the fp is inside 'obj'
```

3 `gcov` and Testcases

In class, we're introducing `gcov` - which is a tool that reports how many of the lines of your program have been "covered" (that is, executed) by a set of testcases. While `gcov` can't tell you if your code is correct or not, it can tell you whether it's been well tested - if there are any lines that have not been executed, then you don't really have any reason to believe that they are correct.

`gcov` works in a multi-step process:

- You build your program with extra compiler flags.

[1]Some compilers allow you to omit the `&`, but I don't know if that is official C or not. I don't recommend the style.

[2]Similarly, some programmers like to skip the parens and asteris, and simply call the function pointer. Again, I'm not 100% whether this is standard C, or an extension - but either way, I don't advocate the style. It can too easily cause confusion, making it look like you're just calling an ordinary funciton.

- You run your program several times, with different testcases. The `gcov` code (which was added by the compiler) keeps track of which lines are executed as your program runs.

  The records keep track of how many lines have been covered across all of the runs of your program.

- You use the `gcov` tool to conver the records into a human-readable format.

In this project, **you must write sufficient testcases to test all of the code.** Your testcases must cover **every line** of three different files (except for `malloc()` error handling code):

- `parseConfigFile.c` - the entire `parseConfigFile` program, which you will write.
- `calculator.c` - the part of the `calculator` program which you will write.
- `calculator_parser.c` - the part of the `calculator` program which I have written.

Yes, I did say that you have to write testcases to cover my code! Examine the `gcov` output - and then devise testcases which will cover all of the various lines in my code.

3.1 `malloc()` Failures and `gcov`

Since it is not possible, in practice, to force a C program to have a `malloc()` failure, you are not required to cover the lines of the `malloc()` error handling code. However, you must still **write** this code.

Note that there are several ways that you might hit a `malloc()` failure in this project:

- Directly, when some code (such as my calculator parser) calls `malloc()`
- Indirectly, through `getline()` (although you won't realize that this happened, since it will look like EOF to you)
- Indirectly, through `realloc()`.

  If you call this function, you must check the return value and write error-handling code.

- Indirectly, through `strdup()` or `strndup()`.

  If you call either of these functions, you must check the return value and write error-handling code.

4 Program 1 - Parse Config File

You must write a program `parseConfigFile`, which reads in a text file. The text file is formatted as a config file for some program - and a common way of doing these files is to have a set of variables, or organized into groups.

In this file, there are two types of lines: group headers and variable declarations. A group header line has a single name (which must be a string of letters - no digits or symbols are allowed), with no whitespace inside it. The name is followed by a colon, and then the end of the line (nothing after it, except for (possibly) whitespace).

Variable declarations have a single name (as with group declarations), but they are followed by an equals. We then ignore any whitespace after the equal sign, up to the first non-whitespace character. From there, we take the **entire line** (including whitespace) as the value of the variable.

Empty strings are allowed as variable values, but never as group or variable names.

In both types of lines, whitespace is allowed anywhere: before the name, and before or after the colon/equals. The file format also allows for blank lines (that is, lines with only whitespace); these are ignored. Blank lines are allowed **anywhere** - including in the middle of groups.

The first non-blank line of the file must be a group header; after that, there can be any combination of group headers and variable declarations.

In this program, we will not sort the groups or variables - and we will not detect duplicates. (In a more official version of this program, for use in the Real World, we probably would.)

4.1 Data Structure

In this program, you have to types of collections: a set of groups, and many sets of variables (one set of variables per group). You should declare structs (at least two of them) which reflect this; one to represent a group, and one to represent a variable.

How you implement this is up to you. You've already used two classic options this semester: an array, which you extend over time (which is what I did for the example executable), or linked lists. Or, you can use something more interesting than either one.

4.2 `getline()`

In this program, you **must** use `getline()` to read the input. `getline()` works much differently than `scanf()`. First, instead of using a format specifier, it simply reads data into a character buffer. It reads in **all** of the data - including whitespace and the trailing newline; it ends after it has read the newline.

However, `getline()` is not stuck using a fixed-size buffer. Instead, it has the ability to resize the buffer as often as it needs - meaning that it can read a line of **any** length. However, as you've seen with your own resizable arrays, this means that it needs at least two variables: a pointer to the buffer, and a length variable. Both change when it adjusts the size of the buffer.

C doesn't allow you to return two values from the same function - but there's a classic workaround for that: you can pass pointers. `getline()` takes (in addition to the `FILE*` parameter) two pointers: a `char` double-pointer (pointer-to-pointer-to-char), and a pointer to a `size_t`[3]

You must declare a `char*` and a `size_t` as local variables. Initialize the `char*` to `NULL`, and the size to zero; then call `geline()` and pass it pointers to both variables. `getline()` will call `malloc()` to create a buffer to hold the line; when you return, your pointer will point at that `malloc()` buffer (and the size will reflect the size of the buffer).

Call `getline()` for each line in the input. If, at some point, it finds that it needs more space than the current buffer allows, it will re-allocate the buffer (freeing the old one), and update the `char*`.

So, after you call `getline()` one (or many) times, your `char*` will point to a buffer that was allocated with `malloc()`. Make sure that you free it before your program exits!

You must not use `scanf()` in this program. Experiment with `getline()` to see how it works.

4.3 `realloc()`

If you choose to implement your groups and/or variables as arrays of structs, you will be gradually re-allocating the array as you read more data from the input.

In this program, you may use `realloc()`. `realloc()` is a standard library function which automates the process of re-allocating a buffer, copying the data over, and freeing the old buffer. You pass it the old buffer pointer as a parameter, along with the new size that you want; it returns to you the new buffer pointer.

However, since this involves a `malloc()`, `realloc()` can fail. Do error checking on anything that `realloc()` gives you, just like you already do with `malloc()`.

And of course, make sure that you free the memory before the program ends.

4.4 `strdup()`, `strndup()`

One of the things you will often be doing in this project is selecting a few characters from a buffer - and then saving them as a string, for use later. This requires, of course, that you `malloc()` a new buffer, and copy the characters into that buffer.

For this, I suggest that you use `strdup()` - or, more likely, its length-limited version, `strndup()`. Both functions `malloc()` a new buffer, and copy a string into it (including the null terminator). They work almost identically - but the

[3]`size_t` represents a "size" in memory - it's basically an integer type. But you **must** declare it as `size_t` (not `int`), or the compiler will complain.

key difference is that the first version will copy the **entire** string into the new buffer, while the second will limit the length that it copies. So, for instance, if you are looking at a word which is 6 characters long inside a longer string, you can call

```
    strndup(ptr, 6);
```

and it will copy no more than 6 characters (7, if you include the null terminator).

As with `realloc()` and `getline()`, this calls `malloc()`, so you must check the return value, and `free()` the buffer before your program ends.

4.5 Output Format

For this project, I've again provided example executables for both programs. Run the example for details about exactly what this program should do. But at a high level, the output from this program is:

- Print out the same values as the input file (in the same order), but with the whitespace regularized.
- Print out a few notification messages as you sort the input.
- Print out the input a second time, now sorted.

4.6 Sorting

In this program, you need to perform two different sorts: you will sort the groups (based on group name, sorting them with `strcmp`), and you will sort the variables (sorted in the same way).

You may implement this sort however you want; even an $O(n^2)$ sort will be tolerated. However, if you want to investigate some standard C sorting, check out the man page for `qsort()`. The `qsort()` function requires that the input be an array, and it uses function pointers, so it may be overwhelming at this time. (You're not required to use it.) But if you want, you are welcome to experiment.

4.7 Combining Groups

After you sort the groups, you will find the duplicate group names (if any) and then combine them. That is, if there are two (or more) groups in the file with the same name, then you should join their two lists of variables together. You do this **before** you sort the variables.)

You may assume that the input file never has two copies of the same variable inside the same group (after you've combined all of the groups). However, it's perfectly OK for the same variable name to show up in two **different** groups (so long as the groups don't have the same name).

4.8 Error Handling

As with some of the previous projects, you need to match the `stderr` and exit status of the example executable - but there is flexibility. You must always match the exit status, no matter what; however, for the actual contents of `stderr` you only need to match the **existence** of a message. That is, if the example executable prints out a message, so must you - but if it doesn't print anything, then you must also print nothing.

5 Program 2 - Calculator

In this program, you will read lines, one at a time, from `stdin`. Remove the newline from the input (if it exists), and then call the function `parseExpr()` to parse the string into a tree of `Expr` objects.

If, at any time, you are not able to parse an expression, print out a message to `stdout` (not `stderr`); see the example executable for the exact format. (You will notice that the code I've provided will also print out a message, giving more details about what happened. These messages will go to `stderr`. Don't change that.)

5.1 Input Prompt

Since this might be an interactive program (with a user typing expressions live), I've added a prompt. Print the two character prompt `"> "` (without a newline) each time, before you read anything from `stdin`. (This means, of course, that your program will terminate without a trailing newline, since you should terminate the program (with rc 0) when you hit `EOF` on `stdin`.

5.2 What to do with the `Expr`

You must evaluate the expression that you've been given. To do this, you must recurse through the tree of `Expr` objects, calling the `eval` function pointer of each one in turn. The `eval` function will return to you a `float` value, which is the value of that entire subtree.

See the detailed comments in `expr.h`, which describe how the tree is formed. Make sure that you call the `eval` function pointer on each node.

5.3 Recursion

If we wrote the `Expr` class in Java, then no doubt `eval` would recurse automatically into other nodes. Also, there would probably be many diffferent types, representing the various types of expressions.

However, to reduce the complexity of the new features you're seeing - and also, to require you to perform recursion - I've make `eval` kind of stupid. It takes `float` parameters, and cannot see the `Expr` object itself - meaning that **you** will have to recurse on its behalf.

5.4 What to Print Out

You will print out the value of the expression at **multiple places** as you recurse; for each of these (except the very last), you will print out the text of the expression (which is available in the `origText` field of `Expr`), as well as the value of the expression at that point in the tree.

You will print this on every node which had child nodes - that is, every node where `left,right` were not `NULL`. Do not print this for the "leaf" nodes - that is, nodes where `left,right` are `NULL` - since leaves represent simple values, with no operators.

At the very end of the calculation, when you have found the value of the entire expression, print out the value only (without the expression itself).

All value printouts should use the `%f` format specifier for `printf()`.

5.5 Memory Management

`parseExpr()` will allocate many `Expr` objects - as well as text buffers for teach of the `origText` fields. You must provide a function named `freeExpr()`, which will free an entire tree (including all of the `origText` inside it).

This function will be called by `parseExpr()` if there is a parse error (or if there is a `malloc()` failure). You probably should also use it in your own code, to clean up an expression after you evaluate it.

5.6 Code You Must Provide

You must write the `main()` function for `calculator`; you must also implement the `freeExpr()` function. You probably will find it handy to also write other helper functions - but those are up to you.

In addition to writing `calculator.c`, you must also write `expr_student.h`. This file **must include guards, just like all header files.** It must also include prototypes for both `parseExpr()` (which I have written) and `freeExpr()` (which you will write).

5.7 Extending `calculator_parser.c`

Do not check in `calculator_parser.c` to D2L; if you do, we'll ignore it.

I have provided two testcases to you, for `calculator`. The first one will work easily. The second one uses some parser features which I have not given you the code for. **You are not required to make the second testcase work.** However, I've provided it as a challenge: can you adapt my code in `calculator_parser.c` to support these new features?

The two new features to support are:

- Support for multi-character numeric literals. (The current only only allows single digits.)
- Support for the exponentiation operator.

5.8 Planning for the Future

Since we are using function pointers to evaluate the various nodes of the tree of `Expr` objects, it's quite possible that I might add new features in the future. For instance, I could add a square-root operator, or a trigonometric function, or logarithm, etc.

For this reason, don't make any assumptions about the input - or about how to evaluate the `Expr` you are returned. Always call `eval` (even on leaf nodes), and simply let my `eval` functions do whatever they want to do.

For the same reason, don't try to parse the string yourself. I might add new features in the future - your code should work fine even if I do so.

6 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. (Sometimes the error is serious - meaning "the operation cannot be performed." Sometimes, it's more minor, such as "two files are different" or "minor issues happened.")

  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).

7 `malloc()` Failures

As always, check the return code on `malloc()`, and have some sort of error handling routine for it. However, as we've discussed, it won't be practical to test this code.

8 `free()` all `malloc()`s

Starting with Project 5, you must now `free()` every buffer that you `malloc()`, before your program terminates. If you don't, then `valgrind` will report errors - and you will lose partial credit on your testcase.

This is, of course, a good habit to have! Always keep track of all memory that you have allocated with `malloc()` - and always be responsible to free it.

However, in the Real World, people often ignore the need to free memory, if the program is about to die. As I've mentioned in class, when your process dies, the OS will automatically free **all** of your memory - so it doesn't really matter whether you `free()`d everything or not.

But since we want you to practice `free()`ing your memory, we will require that you `free()` every buffer that you `malloc()`ed - even if you don't free it until right at the **end** of your program. (This is, probably, exactly what you'll do in Project 5!)

9 Special Note: `scanf()` and `%s`

Several programs in this class will require you to read strings from `stdin` using `scanf()`. **This can be dangerous, if you read more data than your buffer allows** - since it is possible to read right off the end of the array, and overwrite other memory.

To solve this, `scanf()` allows you to limit the number of characters that you read with the `%s` specifier. **You must always use this feature.** Remember that `scanf()` will also write out a null terminator - so this number **must** be less than the size of your buffer, like this:

```
char buf[128];
int rc = scanf("%127s", buf);
```