CSc 352 (Summer 21): System Programming and UNIX

Project #9

Function Pointers, and `gcov`

due at 11:30pm, Thu 05 Aug 2021

NO LATE DAYS

1 Overview

In this project, 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 of the program 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 command

```
make calculator
```

will build your program; 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 Lecture 15); 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).

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:

- 50% for the `calculator` program
- 20% for `gcov` coverage, based on the **testcases you provided** (along with the testcases I provided). P.S. This will also be graded manually, so Autograder grade will be only 50% from the calculator program.
- 30% hand grading.

1.3 Turning In Your Code

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

```
calculator.c
expr_student.h

student_test_*  // See my examples for the naming style

Makefile
```

Number your testcases like- `student_test_01`.

Please upload them *together* as separate files; do not zip them up into a single zip file or tarball. When Gradescope prompts to upload files, you should select the files together or drop them together to make sure they all are uploaded.

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 parenthesis around the variable name are **required** - without them, 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. You can also use the alternate option discussed in the Lecture.

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'
```

[1] gcc also allows you to omit the `&`

[2] Similarly, some programmers like to skip the parens and asterisk, and simply call the function pointer. However, it can too easily cause confusion, making it look like you're just calling an ordinary function.

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.
- 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 convert 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 two different files (except for `malloc()` error handling code):

- `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 (just some examples):

- 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 `strdup()` or `strndup()`.

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

4 Program 1 - 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.)

4.1 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, and note that you should not print the quotes themselves) 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`.

4.2 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.

4.3 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.

4.4 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()`.

4.5 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.

4.6 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).

4.7 Extending `calculator_parser.c`

Do not submit `calculator_parser.c` to Gradescope; 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.

4.8 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.

5 Standard Requirements

- Your C code should adhere to the coding standards for this class: `https://developer.gnome.org/programming-guidelines/stable/c-coding-style.html.en`
- 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 different 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).

6 `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.

7 `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!)

8 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);
```