CS 352 (Summer 2021): Systems Programming and UNIX

Project #6

Extending Arrays, Arrays of Structs

due at 11:30pm, Thursday 22 Jul 2021

1 Overview

In this project, you will be reading in a bunch of data from `stdin`. You'll be practicing how to deal with a `malloc()`d buffer - only this time, you will be **extending** it as it fills up. In addition, it will be an array of `structs` - not of simple types.

2 Turning In Your Code

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

```
manyRecords.c
Makefile
```

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

3 Program 1 - Many Records

In this program, you will declare a structure that holds a few fields. You will then read many records from `stdin`, and store them into an array of `struct`. Since you don't know the size of this array, you will need to expand the buffer periodically.

After reading many records, you will find the keyword `END`. After that, the rest of the input will will be a series of integers, which are indices into the array you just read. You will print out each of those records pointed by those indices.

Name your file `manyRecords.c`

3.1 Input

This program reads from `stdin`. The first part of the input file will be a set of records to read. Each record will be two integers, followed by a word with no more than 15 characters. The three fields in each record are separated by whitespace, so that you can read them with `scanf()`[1].

At some point in the input, instead of another record, you will find the word `END`. This indicates the end of the records.

After `END`, there will be any number of integers. Use each one as an index into the array of records that you have saved in memory. If the index is not valid (that is, negative or too large), then print out a message about this to `stdout`; otherwise, print out the record itself, along with the index. Continue, printing out records (or warnings about bad indices), until you hit EOF.

3.1.1 Error Handling

If the file doesn't follow the format above, report an error (to `stderr`) and terminate the program immediately with exit code 1.

Note that, so long as the values after `END` are all integers, the input is not "invalid" and so your program should keep running - even if you have to print out warnings that some are invalid indices.

However, if you find non-numeric input after `END`, print out an error message to `stderr` and kill the program.

[1] Technically, the 2nd integer, and the word, might not have a space between them. `scanf()` will interpret `''123asdf''` as the integer 123, followed by the string "asdf". However, this all happens automatically, so long as you read your fields using `scanf()` - so you don't need to know the details yet.

3.2 Declaring a `struct`

You must declare a `struct` in your file, to store the records that you read from `stdin`. You may give it (and its fields) any names that you like, but it should store (at least) two `int`s and an array of characters.

3.3 Extending the Array

When your program begins, it must allocate a buffer with `malloc()`; it must be large enough to hold a single instance of your `struct` (no more).

Each time that you are about to read another record, check to see if you have already filled your array. If you have, then you must extend it by **doubling** its length.

The C standard library has a function which helps you do this, named `realloc()`. You should read its man page - but you **must not use this function for this project.** Instead, you must `malloc()` a new buffer, copy the values over (use `memcpy()`), and then free the old buffer. (Later, when you have more experience, I'll allow you to use `realloc()` - which basically does this steps for you.)

When you double the array, you will print out a message about what you are doing - including printing the contents of the first record in the array. You must print this twice: once before copying the array, and once after; this is simply to help you confirm that you have copied the values correctly.

Finally, you must extend your array **before** you call `scanf()`, and when your code calls `scanf()`, it code must read **directly into the appropriate record.** Occasionally, this will mean that expanding the array will be useless - as you won't be able to read any more into the array - but you must still do it, in order to match the example executable.

3.4 `printRecord()`

You must declare a function named `printRecord()`, which takes a pointer to your `struct` as its only parameter, and which returns nothing. It must print out the fields of the struct. You must use this function to print out the contents of records, every time that you need to do so.

(No, this is not a difficult function to write. In fact, it will probably be only one line long! This requirement is really about getting used to **passing** pointers to `struct`s as parameters. It's not really about the function itself.)

3.5 Output Format

All output should be to `stdout`, except for terminal errors (see above). To determine the exact output, test the example executable. Write testcases to see how it responds to various conditions, such as:

- Various types of broken input files - such as text where you expect integers, missing `END` markers, etc.
- Input files with different numbers of records, such as 0 records, 1 record, and many records. Be sure to check what happens when the number of records is one less, one more, or equal to powers of 2 (since those are the times when the array needs to be resized).
- Try negative, or very large, indices after the `END` marker.
- Try non-numeric input after the `END` marker.
- (There's more to test - be creative!)

4 Grading

In this project, we have some new elements which we'll be grading.

- We will use `valgrind` to confirm that your program has no memory errors.
- You must provide a Makefile, which compiles your code.

We'll give details below.

4.1 Testcase Grading

I have decided to subdivide the score from each testcase:

- You must exactly match `stdout` to get **ANY** credit for a testcase.
- You will lose one-quarter of the credit for the testcase if the exit status or `stderr` do not match the example executable.[2]
- You will lose one-quarter of the credit for the testcase if `valgrind` reports any errors (including failing to `free()` something before your program ended.

As always, you lose half of your testcase points if your code does not compile cleanly (that is, without warnings using `-Wall`).

5 `Makefile` - Requirements

In this project, you must provide your own Makefile. The Makefile must include a rule which builds your executable from your source file. In addition, your Makefile may (but is not required to) include other rules; common rules include `all` and `clean`.

Your rule must:

- Automatically build your executable from your `.c` file when we type

  ```
  make [programName]
  ```

  in the directory.
- Must automatically re-build the executable if the `.c` file is modified.
- **NOT** rebuild the executable if the `.c` file has not been modified.
- Use `gcc` as the compiler.
- Pass the `-Wall` option (because you always do!).
- Pass the `-g` option (so that we can run your executable under `valgrind`).

6 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/marahman/cs352/projects/proj06/` on Lectura. (The UNIX commands you've learned, such as `ls, cd, cp` will all be useful here.). Note that you can copy the files using the `scp` command. You can find more how to use the `scp` command using the man page. One example to download the files to your current working directory of your non-lectura Linux machine:

```
scp lectura.cs.arizona.edu:/home/marahman/cs352/projects/proj06/* .
```

(We have provided the same on Piazza; however, realize that these will only run on a Linux machine running the x86-64 architecture. If don't have a linux machine, you should work on Lectura and only copy the files to your lectura account and your choice of working directory.)

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/marahman/cs352/projects/proj06/grade_proj06` on Lectura, or from the web.

Note that the grading and testing scripts are executable binaries but only have read permissions! Once you download or copy them to your working directory, you need to modify their permissions to execute

[2] Remember, we handle `stderr` more loosely than `stdout`. See below.

them. You may need to review the UNIX lecture from Week 1 and are welcome to use google to see how you can change permissions.

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 test cases** - 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` .

6.1 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.

6.2 Running the Grading Script

To run the grading script, make sure you place the following files in the same parent directory (e.g., `proj06`) and then run `./grade_proj06`

```
grade_proj06

example_*

test_*
[any input files required by the testcases]

Makefile
manyRecords.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.
- Checks that your Makefile works properly.
- 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. It also checks to make sure that your program runs cleanly under `valgrind`.

  (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.

Note: The grading script may show you full points on lectura at it does not have all the testcases. To make sure you pass all the test cases, see gradescope output after submitting there.

6.3 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

6.4 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.

6.4.1 Score Distribution

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

- 10% - `Makefile` works properly
- 60% - `rollingStrings` testcases

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

7 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).

8 `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.

9 `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.

10 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);
```