CS 352 (Fall 17): System Programming and UNIX

Project #3

Strings and Characters

due at 9pm, Tue 19 Sep 2017

1 Overview

This project has three small programs. Each one deals with strings (which in C, are null-terminated arrays of characters) and with the basic characters themselves.

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:

```
dumpChars.c
intConvert.c
reverseWords.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/proj03/` 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/proj03/grade_proj03` 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: 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.

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_proj03`

```
grade_proj03

example_*
test_*
```

```
dumpChars.c
intConvert.c
reverseWords.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% - `dumpChars`
- 20% - `intConvert`
- 30% - `reverseWords`

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

4 Program 1 - Dump Characters

This program prints out the hex values of all of the characters that it reads from `stdin`.

Name your file `dumpChars.c`

4.1 Input

This program takes all of its input from `stdin`. You may read this using any function that you like; `scanf("%c",...)` is one that you've used before, but you are also welcome to investigate `getchar()` or `fgetc()`. You should not use any functions that read more than one character at a time, however (yet).

Read each character, and print out its hexdecimal value, using exactly 2 hex digits (zero padded). I recommend the following `printf()` format specifier:

```
%02x
```

. (Remember that `%x` means to print hexdecimal; the `2` means to print at least 2 characters; the `0` means to zero-pad, instead of padding with spaces.)

Separate each hex character with a single space, but for each group of 4 bytes, print out an extra space - so that the output is organized into columns, with one `int` (4 bytes) per column. Print out 16 bytes per line; do not put any spaces before the first byte in each line, or after the last.

If the number of bytes in the file is not a multiple of 16, the output should end with a partial line; do not have any trailing spaces, but include a newline at the end of that line.

To check the format precisely, run the example executable for comparison.

4.2 Error Conditions

This program does not have any error conditions.

5 Program 2 - Int Convert

One of the more terrible and wonderful things about C is its ability to re-interpret any variable as another type, through casting pointers. This program will illustrate that - and also illustrate how to use pointers as arrays.

This program will read a series of integers from `stdin`. (Each integer on the input will be encoded in decimal.) Read each number with `scanf()`, and store it into an `int` variable.

For each number that you read, print the number back out to `stdout`, first as a decimal integer, and then as an 8-character zero-padded hexdecimal value. Then, (as described below) print out information about the 4 bytes which make up the integer.

Name your file `intConvert.c`

5.1 Accessing the `int` as 4 bytes

After you read each `int`, create a `unsigned char*` variable, which points to the same location; this requires two steps:

- Get the address of the `int` variable (this is an expression of type `int*`)
- Cast the pointer to a `unsigned char*`
- Save the pointer as a `unsigned char*` variable

Now, use this pointer as an array. **DO NOT** actually allocate an array variable! Instead, use the `unsigned char*` pointer that you have created to access the 4 bytes of the integer, one at a time. For each of these 4 bytes, print it out in two ways: first as a 2-character zero-padded hexdecimal value, and then the character itself.

When you print out each character value, you must handle two different special cases:

- If the value is zero (you can check this with the `'\0'` character constant), then print out `<null>`
- If the value is not printable[1] (but is not zero), then print out a question mark.
- Otherwise, print out the character itself (with `%c`, in single quotes.

In addition, for each character that you print out, you must check for special types. Count how many times each of these happen, and print out a total at the end:

- Spaces
- Digits (use a standard library function to check this.)

[1] See the man page for `isalpha()`. The function you need to use is **not** `isalpha()`, but you can find the right function on that man page.

5.2 Non-Integer Input

Finally, if you ever attempt to read an integer with `scanf()` but it fails because it isn't an integer, then read that one character, and report a message to `stdout` - then, continue running. This is not an error (keep running), but keep a count of how many times this happened, and print out a message at the end of the program (along with the count of spaces and digits described above).

Check the example executable to see exactly how this line should look.

5.3 Exit Status

If any non-integer input is detected, then terminate the program with exit status of 1 (after all of the input has been read.) If not, then terminate the program with exit status of 0.

6 Program 3 - Reverse Words

This program illustrates buffer overflow. We will **intentionally** write buggy code, which will corrupt some variables - but we'll also see how to change our code, to make it work better.

I have provided the file `reverseWords.c` in the project directory. This is the base file, which has a little bit of code that you should keep - but of course you will need to add a lot more. The most important part of it is that it declares three different character buffers as local variables in `main()`, and initializes each one with `strcpy()`.

Name your file `reverseWords.c`

6.1 The 'bug' Switch

Your program will read from `stdin`. The first thing to read must be an integer. If you cannot read an integer, print an error message to `stderr` and terminate the program with a nonzero exit status. If you read an integer but it is anything other than 0 or 1, print an error message to `stderr` and terminate the program with a nonzero exit status.

This integer you read indicates whether your program should be buggy. If the integer is 1, then your program should be vulnerable to buffer overflow: when you call `scanf()` to read strings (see below), it must use `"%s"`. However, if the integer is 0, then it must include a limit on the number of characters to read, by using something like `"%10s"`.

Remember, when you place a limit on `%s`, this controls the **length** of the string that might be read. `scanf()` will always write one extra character to your buffer - which is the null terminator. You must set the limit such that, in the worst case, `scanf()` will never write anything past the end of the array.

6.2 Reading Words, and Output

After you read the "bug" switch (see above), you will then read words from `stdin`, until you hit EOF. Read using `scanf()`. Always read into the `buf1[]` variable. Remember than in C, arrays are (more or less) the same thing as pointers, so you **must not** use `&` to get the address of the buffer. Instead, you can simply pass an array to `scanf()`; C will give `scanf()` the address of the first byte in the array:

```
scanf("%s", buf1);
```

For each word that you read, you must perform the following tasks:

- Print out the characters of the word, one at a time, **in reverse order**, followed by a newline.
- Print out the contents of all three `buf*` variables (not reversed). Print this out on a 2nd line, with a tab at the head of the line.
- Check the length of the string in `buf1[]`. What is the maximum length that it can contain (while leaving space for the null terminator)? If the string is longer than that, then print out a third line - again leading with a space. (See the example executable to see the format.)

  NOTE: If your program is working correctly, then the 3rd line will never print out if `bug=0` - because you will limit `scanf()` properly, to prevent buffer overflows.

See the example executable to see the exact format of these lines.

6.3 Handling Long Words

Some of the words in the input will be longer than can fit into the `buf1[]` variable. **Do not change the size of** `buf1[]` to fix this.

If `bug=1`, then your code will suffer a buffer overflow when it reads long words. You will find that some of the characters in `buf2[]` are also modified - and if the string is very long, it might even modify `buf3[]`. Allow this to happen; this is the point of the program!

However, if `bug=0`, then your code will only read as many characters as there are spaces. To see how this works, take a look at the testcases I have provided - and run them through the example executable. Pay close attention to the contents of the `buf1[]` array after each line.

6.4 Error Conditions

Your program should normally print only to `stdout` (using `printf()`), and should return 0 from `main()`. However, you must print an error message to `stderr`, and return nonzero from `main()`, if any of the following conditions occur:

- You are not able to read an integer at the beginning of the program.
- The integer that you read has any value other than 0 or 1.