
CS 702, Spring 2007
Assignment #1
Getting to know C and Unix

Due 1/31/07, in class

Introduction

The goal of this assignment is to get acquainted with the C programming language and Unix. It is shorter than the handout makes it seem, since it is broken into many small parts. You can simply do the last part if you'd like, but if you're new to C you may want to work through each part.

You will be graded on part 7 of this assignment. Name the program you write for part 7 `shell` and put it in the top level of your course home directory until the project is graded. Turn in the complete source listing for `shell`.

Part 1: Hello World. <yawn>

What would learning C be without the canonical welcome program?

Objectives

- log into your account
- create and edit text files
- use the C compiler
- terminate a UNIX program nicely

Specification

Create a small program that prints `Hello, World` and terminates. Here is the complete text:

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello, World\n");
    exit(0);
}
```

Make sure you understand what each part of this program does.

2. Mini cat

Objectives

- Learn some basic functions of the standard I/O library
- Learn the UNIX end-of-file convention
- Understand rudimentary C storage allocation
- Handle a loop with an exit in an inconvenient place

Specification

Create a program that reads from the standard input (normally the terminal) a line at a time, and writes the line it just read to the standard output (normally also the terminal). Terminate when the end-of-file is reached without writing anything.

The full-blown version of this program is named `cat` from *concatenate*, because it can glue several inputs together.

Things to Think About

The end-of-file from a terminal is expressed in UNIX by typing a `^D` (control-D), not by a blank line.

C strings are terminated by a `'\0'` character, while non-existent strings are represented by a `NULL` pointer. Don't confuse characters with pointers to characters.

Use the standard routines `gets` [don't use `gets` in any later parts] and `puts` to read and write whole lines at a time.

`gets` requires a buffer in which to place incoming characters. Where does this buffer come from? What happens if the input line is bigger than the buffer you provide? (Hint: where do the extra characters get deposited?)

This was one of the three security holes exploited by 1988's "worm" attack on the Internet, so it is a serious problem.

There are many ways to write a "do something; quit if test; do something else; repeat" loop in C. Find three ways. Find two ways to write an infinite loop.

Try adding a prompt to cue the user to type something by printing a string just before trying to read a line from standard input.

Documentation

The entries for `stdio` and its individual routines is in Section 3 of the on-line UNIX manual (type `'man 3 stdio'`).

3. Mini echo

Objectives

- Understand how UNIX programs receive their initial arguments
- Exercise basic C string and pointer handling

Specification

Create a program that takes its initial arguments and prints them on the standard output, one to a line. The "zereth" argument, which is the name of the program, counts as an argument. If you call your program `miniecho`, and you invoke it like this:

```
miniecho one two three
```

then the output should be

```
miniecho
```

```
one
```

```
two
```

```
three
```

Things to Think About

A standard C idiom for iterating over a program's arguments is

```
while (argc-- > 0)
{
    process(*argv++);
}
```

Explain what the *, --, and ++ do. You may find it clearer to achieve the same effect with a conventional for loop:

```
int i;

for (i = 0; i < argc; i++)
{
    process(argv[i]);
}
```

Explain why a UNIX program will always have at least one argument.

4. Tokenize Input

This is the largest and trickiest part of the assignment. Be careful with C pointers.

Objectives

- Learn more C string and pointer handling
- Perform more involved storage allocation
- Create a reusable software *module* and a main *driver* program to invoke it.

Specification

Create a program that reads the standard input a line at a time, identifies the *tokens* (sequences of characters that do not include blanks or tabs), and prints the tokens on the standard output, one to a line.

For example, given the input

```
this is a      lineof lots-of-things      to read %&^^*^^foo() ()bar
```

your program should print

```
this
is
a
lineof
lots-of-things
to
read
%&^^*^^foo()
()bar
```

Things to Think About

When UNIX starts your `main` program, it provides a count of the number of arguments (`argc`), and a vector of pointers to characters (`argv`). These pointers point to the beginning of each successive argument. UNIX had to make space both for the characters representing the arguments, and for the vector holding the pointers.

I recommend that you create a similar count and vector of pointers when you tokenize an input string. The procedure that does the real work might have an interface something like

```
char **tokenize(char *input, int *token_count)
```

Why is the parameter `token_count` a pointer to an integer instead of just an integer? What happens if more tokens are entered than you allocated space for in the initial `token_vector`? Why does `tokenize` return a pointer to a list of pointers?

It will be convenient to add an additional `NULL` pointer at the end of the counted vector of pointers. That is, if you find 5 tokens, return 5 in the count, fill in the appropriate 5 pointers in the vector, then make the next pointer in the vector `NULL`. That way, the tokens can be processed either by counting, or by proceeding until `NULL` is encountered.

You know how to read and process a line of input at a time from the mini cat program, and you know how to process a counted list of strings from the mini echo program. Create a new main program reusing those parts that calls `tokenize` in between, as in

```
int tokenc;
char **tokenv;
/* main loop */
/* get a line of input (from part 2) */
tokenize(input, &tokenc, &tokenv);
/* now tokenc, tokev are just like argc, argv */
/* process an argument list (from part 3) */
```

You will find several utility routines useful. Create one routine that takes a string (`char *`) and returns the pointer to the next space or tab, and another routine that takes a string and returns the pointer to the next non-space or non-tab. Be careful not to fall off the end of a string.

Where does storage for the tokens and the list of pointers to tokens come from? What's a reasonable size for a token? What's a reasonable number of tokens? What happens if a token is too big for the space you allowed? How can you tell how big a token is before getting space for it? (Hint: the whole token is already sitting in the input string.) How can you tell how many tokens there are before getting space for the list of tokens? (Hint: you don't have to copy tokens the very first time you see them.)

Allocate space for the tokens, and for the token list, dynamically, using `malloc` or `calloc`, instead of using fixed-size arrays. Find the length of the input and allocate space for a copy, leaving space for a final `'\0'` terminator. (You could allocate space for each token individually, but you don't have to.) Allocate space for a vector of pointers to tokens, leaving room for a final `NULL` pointer. Write a simple routine to take a token list and free the storage you allocated earlier. Be sure to free the storage for both the vector and the arguments.

Now is a good time to try separate compilation of two or more C source files, one for the `tokenize` routine, and one for the main program. You will need the `tokenize` routine for part 7, and if you have already separated it out, you won't have to tear it out and rewrite it later.

5. Exec a New Program

In the UNIX operating system, a program can simultaneously terminate itself and start a second program, using the `execve` operating system call.

Objectives

- Start a program and give its initial arguments
- Check for failures of system calls

Specification

Create a program that takes its first argument, which should be the name of another program and irreversibly becomes that other program. The complete list of arguments, starting with the first, are passed as arguments to the new program. If you call your program `startup`, and you invoke it like this:

```
startup /bin/date -u
```

then it should execute the program `/bin/date` with the argument `-u`. Output should look something like `"Sat Jan 15 22:10:36 GMT 2006"`

Check the return code on `execve` and report a failure to start the new program. For example, if you misspell the name of the new program, `execve` will be unable to start it. Use the standard error reporting routine `perror`.

Things to Think About

You should actually use the library routine `execvp` instead of the system call `execve` for the following reason. `execve` is very strict about where it looks for programs. If you give it a program name beginning with a slash (`/`), that must be the exact file name of the program file. (This is an *absolute* pathname in UNIX.) If you give it a program name beginning with anything else, that must be the file name of the program file relative to the current working directory. This means that `execve` will not accept most of the program names you normally use, for example `cc`, `vi`, or `emacs`, because they don't start with a `/` and those programs don't reside in your current working directory.

Normally the shell (command interpreter) takes a short name like `cc` and finds the complete absolute pathname. This is complicated, so you can use the `execvp` library routine, which finds the absolute pathname if necessary, and then calls `execve` for you.

Documentation

Section 3 of the on-line UNIX manual has entries for `execl`, `execve`, `execvp`, `perror`, and `errno`.

6. Hello World Again

Objectives

- Learn some basic (unbuffered) file system calls
- Learn how IO works.

Specification

- (1) Modify part 1 to output its message to a file without changing the `printf` to `fprintf`. A sample session might look like this

```
prompt> a.out
prompt> more foo
  Hello World
prompt>
```

Things to Think About

Standard output is file descriptor 1. Thus, `close(1); open(file);` should redirect io for you. Of course, some error checking would be nice. Also make sure and use `open(2)` and not `fopen`.

Documentation

Section 2 of the on-line UNIX manual has entries for `open`, `dup`, `perror`, and `close`.

7. Mini shell!

This is the heart of a UNIX command interpreter, normally known as a shell.

Objectives

- Start a new process and wait for it to terminate
- Integrate the previous five parts into one program
- Do some simple IO-redirection
- Celebrate the completion of the assignment

Specification

- (1) Create a program that prints a prompt, reads a line at a time from standard input (part 2), parses the input into tokens (part 4), creates a new process (new), makes the child process start the program specified by the first token (part 5), and waits for the child process to terminate (new), then repeats the whole sequence (part 2). When the end of the standard input is reached, the program will terminate.
- (2) Include simple IO redirection in your shell. The command line “foo > bar” should send the standard output of foo to the file bar and the command line “foo | bar” should make bar’s standard input come from foo’s standard input. Note that foo might have arguments.

Things to Think About

Your shell is allowed to quit only upon end-of-file. When the user types a blank line (actually a line with no tokens on it), just ignore it and print another prompt.

The `fork` system call duplicates the current process (and program). The two processes are identical except that the parent process returns from calling `fork` with the process identifier of the child, while the child returns from `fork` with the value zero. This means you can use the following block of code to determine which is which:

```
int pid;

pid = fork();
if (pid == 0) {
    /* I am the child */
}
else {
    /* I am the parent */
}
```

The child process should `execvp` the desired program. If there is an error, the child should print an error message using `perror` and then `exit`. What happens if it doesn’t exit?

The `wait` system call blocks the parent process until a child process terminates. Be sure to pass `wait` the argument(s) it expects.

Documentation

The entries for `pipe`, `fork`, and `wait` are in Section 2 of the on-line UNIX manual

What to Hand In

- (1) Your answer to Part 7.
- (2) the executable only in `~<your account name>/shell`