Week 08 Laboratory Exercises

Objectives

- learning how to access files
- learning how to work with binary data
- learning how to use lseek

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab08`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab08
$ cd lab08
$ 1521 fetch lab08
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE — INDIVIDUAL:

Create a File of Integers

Write a C program, `create_integers_file`, which takes 3 arguments:

1. a filename,
2. the beginning of a range of integers, and
3. the end of a range of integers;

and which creates a file of this name containing the specified integers. For example:

```
$ ./create_integers_file fortytwo.txt 40 42
$ cat fortytwo.txt
40
41
42
$ ./create_integers_file a.txt 1 5
$ cat a.txt
1
2
3
4
5
$ ./create_integers_file 1000.txt 1 1000
$ wc 1000.txt
1000 1000 3893 1000.txt
```

Your program should print a suitable error message if given the wrong number of arguments, or if the file can not be created.

HINT:

Use *fopen(3)* to create the file and *fprintf(3)* to output to the file. If you need some help starting off, read this example program to see how to use these functions to create and write to a file.

HINT:

For this exercise, you can use the simple function *atoi(3)* to convert a null-terminated string to an `int`. In general, *strtol(3)* is a more useful function because it allows error handling — but this is not required for this exercise.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest create_integers_file
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab08_create_integers_file create_integers_file.c
```

You must run `give` before **Sunday 26 July 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

EXERCISE — INDIVIDUAL:

Print the Bytes of A File

Write a C program, `print_bytes`, which takes one argument, a filename, and which should should read the specifed file and print one line for each byte of the file. The line should show the byte in decimal and hexadecimal. If that byte is a an ASCII printable character, its ASCII value should also be printed.

Assume ASCII printable characters are those for which the `ctype.h` function *isprint(3)* returns a non-zero value.

Follow the format in this example exactly:

```
$ echo "Hello Andrew!" >hello.txt
$ ./print_bytes hello.txt
byte    0:  72 0x48 'H'
byte    1: 101 0x65 'e'
byte    2: 108 0x6c 'l'
byte    3: 108 0x6c 'l'
byte    4: 111 0x6f 'o'
byte    5:  32 0x20 ' '
byte    6:  65 0x41 'A'
byte    7: 110 0x6e 'n'
byte    8: 100 0x64 'd'
byte    9: 114 0x72 'r'
byte   10: 101 0x65 'e'
byte   11: 119 0x77 'w'
byte   12:  33 0x21 '!'
byte   13:  10 0x0a
```

HINT:

Use *fgetc(3)* to read the file, and *printf(3)* for output.

The *printf(3)* format you need will be similar to `"byte %4ld: %3d 0x%02x"`

NOTE:

Note that, in the above example, the last byte has value 10 (ASCII "linefeed", '\n') which is *not* a printable character — `isprint('\n')` returns 0 — so its ASCII value is not printed.

The useful *nix utility *hexdump(1)* does what this program does and more.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest print_bytes
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab08_print_bytes print_bytes.c
```

You must run `give` before **Sunday 26 July 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

EXERCISE — INDIVIDUAL:

Create a Binary File

Write a C program, `create_binary_file`, which takes at least one argument: a filename, and subsequently, integers in the range 0...255 inclusive specifying byte values. It should create a file of the specified name, containing the specified bytes. For example:

```
$ ./create_binary_file hello.txt 72 101 108 108 111 33 10
$ cat hello.txt
Hello!
$ ./create_binary_file count.binary 1 2 3 251 252 253 254 255
$ ./print_bytes count.binary
byte    0:   1 0x01
byte    1:   2 0x02
byte    2:   3 0x03
byte    3: 251 0xfb
byte    4: 252 0xfc
byte    5: 253 0xfd
byte    6: 254 0xfe
byte    7: 255 0xff
$ ./create_binary_file 4_bytes.binary 222 173 190 239
$ ./print_bytes  "%02X\n"' 4_bytes.binary
byte    0: 222 0xde
byte    1: 173 0xad
byte    2: 190 0xbe
byte    3: 239 0xef
```

Your program should print a suitable error message if given the wrong number of arguments, or if the file can not be created.

> HINT:
>
> Use *fopen(3)* to create the file and *fputc(3)* to output to the file. If you need some help starting off, read this example program to see how to use these functions to create and write to a file.

> HINT:
>
> For this exercise, you can use the simple function *atoi(3)* to convert a null-terminated string to an `int`. In general, *strtol(3)* is a more useful function because it allows error handling — but this is not required for this exercise.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest create_binary_file
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab08_create_binary_file create_binary_file.c
```

You must run `give` before **Sunday 26 July 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

EXERCISE — INDIVIDUAL:

Extract ASCII from a Binary File

We are distributing programs as binaries, and would like to know what if the C compiler is leaving any confidential information in the binaries as ASCII strings.

Only 95 of 256 byte values correspond to printable ASCII characters, so several byte values in a row corresponding to printable characters probably will occur infrequently in non-ASCII data. There is only a 2% chance that four (independent, uniform) random byte

values will correspond to ASCII printable characters.

Write a C program, `hidden_strings`, which takes one argument, a filename; it should read that file, and print all sequences of length 4 or longer of consecutive byte values corresponding to printable ASCII characters. In other words, your program should read through the bytes of the file, and if it finds 4 bytes in a row containing printable characters, it should print those bytes, and any following bytes containing ASCII printable characters.

Print each sequence on a separate line.

Assume ASCII printable characters are those for which the `ctype.h` function *isprint(3)* returns a non-zero value.

Do not read the entire file into an array.

Use the `create_binary_file` program from the previous exercise to create simple test data. For example:

```
$ dcc hidden_strings.c -o hidden_strings
$ ./create_binary_file test_file 72 101 108 108 111 255 255 65 110 100 114 101 119
$ ./hidden_strings test_file
Hello
Andrew
```

When you think your program is working, try extracting strings from a compiled binary. For example:

```
$ cat secret.c
#define secret_hash_define 1

// secret comment

int secret_global_variable;

int main(void) {
    int secret_local_variable;
    char *s = "secret string";
}

int secret_function_name() {
}
$ gcc secret.c -o binary1
$ gcc secret.c -g -o binary2
$ gcc secret.c -s -o binary3
$ ./hidden_strings binary1
/lib64/ld-linux-x86-64.so.2
libc.so.6
__cxa_finalize
__libc_start_main
GLIBC_2.2.5
...
$ ./hidden_strings binary1|grep secret
secret string
secret.c
secret_function_name
secret_global_variable
$ ./hidden_strings binary2|grep secret
secret string
secret.c
secret.c
secret_global_variable
secret_function_name
secret_local_variable
secret.c
secret_function_name
secret_global_variable
$ ./hidden_strings binary3|grep secret
secret string
```

The above example shows that, by default, *gcc(1)* leaves function names, global variables names and the filename in the binary.

If you specify the `-g` command line option, variable names are also left in the binary. This is part of information left for debuggers such as *gdb(1)* (which dcc uses). This information allows debuggers to print the current value of variables.

If you specify the `-s` command line option, all names are stripped from the binary but the string remains.

> **HINT:**
>
> Use *fopen(3)* to open the file, and *fgetc(3)* to read the file.

> **HINT:**
>
> Use a 3-element array as a buffer.

> **NOTE:**
>
> You do not have to consider locale or Unicode in this exercises.
>
> Tab ('\t') and newline ('\n') are not printing characters for our purposes: *isprint(3)* returns 0 for them.
>
> You are not permitted to read the entire file into an array.
>
> The useful *nix utility *strings(1)* does almost exactly what your program does.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest hidden_strings
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab08_hidden_strings hidden_strings.c
```

You must run `give` before **Sunday 26 July 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

CHALLENGE EXERCISE — INDIVIDUAL:

Print The Last line of Huge Files

Write a C program, `last_line`, which takes one argument, a filename, and which should print the last line of that file. For example:

```
$ dcc last_line.c -o last_line
$ echo -e 'hello\ngood bye' >hello_goodbye.txt
$ cat hello_goodbye.txt
hello
good bye
$ ./last_line hello_goodbye.txt
good bye
```

You program should not assume the last byte of the file is a newline character.

```
$ echo -n -e 'hello\ngoodbye' >no_last_newline.txt
hello
goodbye$ ./last_line no_last_newline.txt
goodbye$
```

Your program should handle extremely large files. It should not read the entire file. As this is a challenge exercise, marks will not be awarded for programs which read the entire file.

For example, it should be able to print the last line of a one-terabyte file:

```
$ echo -e 'Hello\nGood Bye'|dd status=none seek=1T bs=1 of=/tmp/gigantic_file$$
$ ls -l /tmp/gigantic_file$$
-rw-r--r-- 1 z5555555 z5555555 1099511627791 Oct 26 17:27 gigantic_file12345
$ ./last_line /tmp/gigantic_file$$
Good Bye
```

The gigantic file created above is a sparse file, consisting almost entirely of zero bytes: it uses little actual disk space, but, to be safe, remove it when you finish the exercise.

The commands above create the sparse file in /tmp to avoid it accidentally being backed up or otherwise copied.

Sparse files can create problems if they are accidentally copied by a program which doesn't handle them specially — and most programs don't.

BTW the *$$* in the above command is replaced by the shell process id. This is because /tmp is shared so we'd like to use a filename that is (more or less) unique.

> **NOTE:**
>
> Assume line are separated only by the byte corresponding to `'\n'`.

> Assume bytes can contain any value; you cannot assume bytes are ASCII values.

> **HINT:**
>
> Use *fseek(3)* and *fgetc(3)*.

> **NOTE:**
>
> If the last byte of file is not `'\n'`, you should print all bytes after the last `'\n'` byte.
>
> If there is no `'\n'` byte, you should print the entire file.
>
> If the last byte of the file is `'\n'`, you should print every byte after the previous (second last `'\n'`) `'\n'` byte.
>
> If there is no previous `'\n'` byte, you should print the entire file.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest last_line
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab08_last_line last_line.c
```

You must run `give` before **Sunday 26 July 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Sunday 26 July 21:00** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```
