COMP2129 Week 4 Tutorial

Files, arguments, and Dynamic Memory

Notices

We welcome your feedback and thoughts about the course. Please complete the mid-semester survey, available on eLearning.

Exercise 1: Command-Line arguments

When a program is executed, it can be passed command-line arguments from whatever invoked it. This is the standard way of interaction using a shell. When the line

```
$ grep kindergarten /usr/share/dict/words
```

is entered into the shell, `grep` is invoked with three arguments: the strings `"grep"`, `"kindergarten"`, and `"/usr/share/dict/words"`. The program can then interpret those arguments how it likes (`grep` interprets the first as a pattern, and the second as a filename).

The `main` function can be one of the following two prototypes:

```c
int main(void);
int main(int argc, char *argv[]);
```

The second prototype can capture command-line arguments. When the program is run, the `argv` array will contain the command-line arguments, and `argc` will be the length of `argv`.

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i;
  for (i = 0; i < argc; i++)
    printf("argv[%d]: \"%s\"\n", i, argv[i]);
  return 0;
}
```

Furthermore, C guarantees that `argv[argc]` is the null pointer, so we could alternatively write the loop condition as `argv[i] != NULL`. Try compiling the above example and running it with some arguments, or even without any arguments. What happens if you rename the executable (`mv file1 file2`) and run it?

Exercise 2: Standard out, standard error

The C standard library provides the type **FILE ***, called *file pointers*. All programs start with three of these open: **stdin**, **stdout**, and **stderr**. **printf** and **scanf** by default will use **stdout** and **stdin** respectively. They have some related functions, **fprintf** and **fscanf** which can take another argument, allowing results to be printed/scanned from other files, or file-like objects.

Typically error messages are reported on **stderr**, so that they don't get mixed up with the (standard) output of a program. Copy the following C program:

```c
#include <stdio.h>

int main(void) {
  fprintf(stdout, "Standard out\n"); // Same as printf()
  fprintf(stderr, "Standard error\n");
  return 0;
}
```

And try running it, then redirecting its output to a file. You can redirect standard error using the **2>** symbol. Explain where **stdout** and **stderr** go using that program, when run in the following five ways:

```
  $ ./a.out
  $ ./a.out > out
  $ ./a.out 2> err
  $ ./a.out > out 2> err
  $ ./a.out | wc -l
```

So now you know of two *out-of-band* ways of returning data: via the return code of **main**, or via **stderr**. Obviously, **stderr** is much more desirable for human-readable messages.

The 2 in the above command stands for *file descriptor* 2. All programs start with three file descriptors open: 0 (**stdin**, open for reading), 1 (**stdout**, open for writing), and 2 (**stderr**, open for writing). The token **2>** tells bash to redirect the output of file descriptor 2 somewhere.

You can even redirect **sterr** into the same place as stdout:

```
  $ ./a.out > out 2>&1
```

Exercise 3: File I/O

C can open files on disk, using the **`fopen()`** function. Its capabilities, as well as how to handle its errors, are listed in the manual: **`man fopen`**. To open a file pointer for a file named **`foobar`**:

```
FILE *fp = fopen("foobar", "r"); // "r" for "read mode"
// Do things with the file...
fclose(fp);
```

Anywhere you see **`stdin`** or **`stderr`** in code, you can replace it with an opened file like **`fp`**, and the code will work as expected. (In fact, if you ever do network programming on *nix, you can write to network sockets as if they were files! There are many things in *nix systems which present a file-like interface).

We'll be using the **`fgets()`** function a lot during this tutorial, so make sure you read the manpage for it! It has some behaviours you might not be used to, if you have not done much low-level programming before.

Make a program which uses the **`fgets()`** function to count the lines in the file given as the first argument to the program. If there was no argument given, or there was an error opening the file, your program should print an error message to **`stderr`**. Here are some examples of interactions with the program:

```
 $ ./countlines /usr/share/dict/words
 479828
 $ ./countlines
 Usage: ./countlines <filename>
 $ ./countlines nonexistent-file
 ./countlines: nonexistent-file: No such file or directory
```

Below is an example which counts lines from **`stdin`**:

```
#include <stdio.h>

char buf[1024];

int main(void) {
  int count = 0;
  while (fgets(buf, sizeof(buf), stdin) != NULL) {
    count++;
  }
  printf("%d\n", count);
  return 0;
}
```

Modify this program so that it has the described behaviour.

Hint: Rather than hardcoding the string `"./countlines"` into your program for the error message, use **`argv[0]`** to get the name of the executable. This means if the file is renamed, the message will update itself!

Exercise 4: A dynamic integer array

The C standard library contains the **malloc** function to allocate a number of bytes in memory. **malloc** will return a pointer to the newly allocated memory, the value of which is indeterminate (the memory may not be zeroed, and could contain anything at all). The program can proceed to use this memory, then relinquish it with a call to **free**. A typical call to **malloc** to allocate space for 24 elements of type `int` looks like:

```c
#include <stdlib.h>

int arraylen = 24;
int *array = (int *) malloc(arraylen * sizeof(int));
```

The array can then be indexed and used just as if it were declared as an `int array[24]` (with one exception: `sizeof(array)` will just return the size of a pointer, rather than the number of bytes the array takes up, since `sizeof` is evaluated at compile-time).

Write a program which takes a single argument n (an integer) from the command-line, and allocates an array of type `int` which is n long. After that, use the array to calculate the sum of squares $1^2 + 2^2 + \cdots + n^2$ by writing 1^2 into **array[0]**, 2^2 into **array[1]** and so on, and summing them up afterwards.

You should use **malloc** to allocate the array, and call **free** to release the memory before your program exits.

An interaction with your program should look like:

```
$ ./sumsqs 0
0
$ ./sumsqs 1
1
$ ./sumsqs 2
5
$ ./sumsqs 3
14
$ ./sumsqs 4
30
```

Hint: You may want to look up the **sscanf** function (**man sscanf**), which can convert a string containing digits to an `int`. It works just like **scanf**, except it reads from a C string instead of from an input stream.

After you have your program working, use the **valgrind** utility to verify that you have freed all of the memory you allocated (in other words, you have not *leaked* any memory).

```
$ valgrind ./sumsqs 4
```

If valgrind finishes with the string “All heap blocks were freed – no leaks are possible”, then your program freed all its memory. Otherwise, it will come up with a “HEAP SUMMARY” and a “LEAK SUMMARY”, which means your program leaked memory.

We will come back to valgrind later in the course.

Exercise 5: Reading lines of any length

fgets will read a line from standard in into a given buffer. It is also a *safe* function (unlike **scanf**), in that it is given a maximum buffer length and will not write past that length.

The program below reads lines from standard in, and outputs the line again, prefixed by its length. However, it makes no attempt to resize buffers, and so has some rather silly output sometimes, for example on the input file **input**, which looks like

```
abc
zxywvu
abcdefghijklmnop
```

It produces the output:

```
$ ./linelens < input
(3) abc
(6) zxywvu
(6) abcdef
(6) hijklm
(2) op
```

It should really produce:

```
$ ./linelens < input
(3) abc
(6) zxywvu
(16) abcdefghijklmnop
```

Your task is to modify the function so that it works properly on lines which are any length at all. You should use **realloc** to resize the buffer if you detect that a full line has not been read. An easy resizing strategy is just doubling the size of the buffer. For example, if your program gets given a line of length 30, it should resize the buffer twice (8 → 16 → 32).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
  int buflen = 8;
  char *buf = (char *) malloc(buflen * sizeof(char));

  int len;
  while (fgets(buf, buflen, stdin) != NULL) {
    len = strlen(buf);
    buf[--len] = '\0'; // Trim trailing newline
    printf("(%d) %s\n", len, buf);
  }

  free(buf);
  return 0;
}
```

Exercise 6: Makefiles

Refer to lectures to find how to create a **`Makefile`** which allows you to save compilation arguments to **`gcc`**. Make rules for everything we've done in the tute so far, so that a single call to **`make`** will build all the **`C`** programs, and **`make clean`** will remove all the compiled files (but not the source files!).

A basic **`Makefile`** for the aforementioned **`countlines`** program is:

```
CFLAGS = -std=gnu99 -W -Wall

.PHONY: all clean

all: countlines

clean:
	rm -f countlines
```

Make sure the indent is a single tab character: anything else will not work.

Makefiles can express very rich dependencies and variables - this is barely scratching the surface. There is an excellent article on them at `http://oreilly.com/catalog/make3/book/ch01.pdf`. Be sure to take a look.

Exercise 7: SSHing in to servers

SSH stands for Secure SHell, and is widely used to log into and administer *nix systems. The university has a server put aside for IT students to use, called **`ucpu1`**. Try SSHing into it now, by opening a terminal and running

```
  $ ssh ucpu1.ug.it.usyd.edu.au
```

If you're at home, or on a computer where the username is not your unikey, you should specify which user you intend to login as:

```
  $ ssh abcd1234@ucpu1.ug.it.usyd.edu.au
```

If you're on Windows, you should download a program called PuTTY (google it), and use it to log in.

There's a catch: this server is only accessible from within the university network (it is not exposed to the internet). To connect to it from home, you should log into the university network by using a VPN (virtual private network). `http://sydney.edu.au/ict/internet/download-vpn.shtml`

There are pinned instructor notes on Piazza with further instructions as to how to log on to the university servers through a VPN, or transfer files to and from your home computer.