ENGG1003 - Tuesday Week 7

File I/O
More Pointers
2D Arrays

Brenton Schulz

University of Newcastle

April 9, 2019

Che C Documentation

- Linux systems have a program called "man"
  - Short for "manual"
- It is used to display a wide variety of documentation called "man pages"
- To install it type this in the terminal:

```
sudo apt update
sudo apt install man
```

  and press `y` (or `<enter>`) when prompted to confirm installation
- Afterwards, C documentation can be accessed by typing `man <topic>`

Che C Documentation

- For example, all library functions have a `man` page you can read by typing:
  `man <function name>`
- eg, try:
  - `man fopen`
  - `man printf`
  - `man sin`
  - `man string`
  - etc..

Correction: String Initialisation

- This is totally fine:

```
1 char string[] = "initial value";
```

- The compiler copies the string literal into `string[]`
- The length is automatically calculated
  - You may specify a length *longer* than necessary:

```
1 char string[1024] = "initial value";
```

Correction: String Initialisation

- This is totally fine:

```
1 char string[] = "initial value";
```

- The compiler copies the string literal into `string[]`
- The length is automatically calculated
  - You may specify a length *longer* than necessary:

```
1 char string[1024] = "initial value";
```

- A constant string is created with:

```
1 char *str = "some string";
```

  - We will study this *pointer* syntax later

File I/O

- A stream is kept in a variable of type `FILE *`
  - Read as "pointer to `FILE`" or "`FILE`-star"
- Three already exist in your C programs:
  - `stdin`
  - `stdout`
  - `stderr`
- Additional streams are declared like other variables, eg:

```
1 FILE *input, *output;
```

File I/O - Quick Review

- Before a file can be accessed you must *open* it with the `fopen()` function
- In order to open files you need two pieces of information:
  - The file's name
  - The data direction (mode)
    - Reading
    - Writing
    - Both

File I/O

- `fopen()`'s function prototype is:

```
1 FILE *fopen(const char *name, const char *mode);
```

- `const char *name` is a string holding the file's name
- `const char *mode` is a string describing the desired data direction
- Both of these can be passed as variable strings or hard-coded

File I/O

- The `*mode` argument can be one of the following:
  - `"r"` (reading)
  - `"r+"` (reading and writing)
  - `"w"` (writing)
  - `"w+"` (reading and writing, file truncated)
  - `"a"` (appending)
  - `"a+"` (reading and appending)
- Read documentation for details
- `fopen()` example:

```
1 FILE *input;
2 input = fopen("data.txt", "r");
```

`fopen()` Errors

- The return value of `fopen()` is `NULL` on error
- Check it! Attempting to access a `NULL` stream will result in a segmentation fault!

```
1 FILE *input;
2 input = fopen("data", "r");
3 if(input == NULL) {
4   perror("fopen()");
5   return;
6 }
```

- `perror()` prints a user-friendly error message

File I/O

- Once opened, a file can be accessed with:
  - `fscanf()`
  - `fprintf()`
- These functions behave just like `scanf()` and `printf()` except they take an extra argument:

```
1 int fscanf(FILE *stream, const char *format,
    ...);
```

- The first argument is a `FILE *`
- The rest is identical to `printf()` and `scanf()`

File I/O - Position Indicators

- Concept: bytes in files have an address known as a *position indicator*
- The address is the number of bytes, starting at zero, from the start of the file
- Unless otherwise controlled, files are only read from and written to *sequentially*
- The position indicator automatically increments when a byte is read or written

File I/O - Position Indicators

- Some useful functions:
  - `ftell()` - Returns the position indicator
  - `fseek()` - Sets the position indicator
  - `feof()` - Returns TRUE if the position indicator is at the end of the file
- For example, to process data until the end of file is reached:

```
1 FILE *stream;
2 // open file etc
3 while(!feof(stream)) {
4   // Read from file
5   // Do stuff
6 }
```

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

- Declare `FILE *input;`

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

- Declare `FILE *input;`
- Use `fopen()` to open it for reading

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

- Declare `FILE *input;`
- Use `fopen()` to open it for reading
- Write a loop which reads and writes characters until the whole file has been read
  - Read with: `fscanf(input, "%c", &c);`
  - Write with: `printf("%c", c);`

File I/O Example 1

Write a C program which opens a file, `input.txt`, then reads and prints each character to the console on a new line, indicating the position indicator's value *after* reading each character.

File I/O Example 2

Write a C program which copies a file, `input.txt`, into a new file, `output.txt`. While copying, the program should count how many spaces there are in the input and print the final count to the terminal before exiting.

File I/O Example 3

Write a C program which opens a file, `input.txt`, and counts the number of times the string `"the"` appears.

The program should include a function, `isThe()`, which tests if a string is equal to `"the"` or not.

Pointers

- A *pointer* is the memory address of a variable
  - This includes "first element of an array" pointers
- Pointers can be stored in variables of type "pointer to data type"
  - Declaration syntax:
    ```
    data_type *variable_name;
    ```
  - eg: `int *p;`
- Pointers also implicitly exist when using arrays
- All pointers are the same size
  - The memory address of a `char` is the same size as the memory address of a `double`

Pointers - Why?

- In ENGG1003:
  - Passing a pointer to a function lets the function modify the variable
    - This lets functions "return" more than one value (ie: modify multiple variables given as pointer arguments)
  - String functions mostly accept `char *`'s
  - Help you understand computer memory organisation
  - Pointers are the only way to send "large" amounts of data to a function

Pointers - Why?

- Beyond ENGG1003:
  - Pointer "casting" can be used to interpret one variable as a different type in a specific way
    - eg: break a 32 bit `int` into two 16 bit chunks for transmission on an SPI bus
    - Interpreting "file header" data chunks as complex data structures
  - Pointers are required for dynamic memory allocation
    - For getting large amounts of RAM after a program has begun executing
  - Pointers are required to build advanced memory structures such as trees and linked lists

Pointer Terminology

- A pointer is *declared* with the syntax:

```
1 datatype *pointerName;
```

- A pointer is *assigned* with the syntax:

```
1 pointerName = &variable;
```

- A pointer is *dereferenced* with the syntax:

```
1 *pointerName = 12;
```

  - This allocates `12` to the variable `pointerName` is pointing to

Pointer Declaration

- Declaring a pointer variable allows you to store and manipulate pointers
- Declaration examples:

```
1 int *p; // Pointer to an integar
2 char c, *a; // char c and pointer-to-char a
3 char * str = "string" // Pointer to string
```

- Explicit declaration like this is mostly beyond ENGG1003
  - We will mostly just use them as function arguments

Pointer Assignment

- A pointer is "created" by using the `&` operator before a variable name:

```
1 int k, *x;
2 k = &x; // k holds the address of x
```

- Array names are implicitly pointers

```
1 char string[] = "Hello";
2 char *p = string; // Pointer to string[0]
```

Pointers and Strings

- When using double quotes "..." pointers are created by the compiler
  - The pointer is of type `const char *`
  - The string data gets stored in the program
    - ie: in the same area of memory as the binary machine code
  - The compiler creates a pointer to that memory address
- Example: `printf()` has a prototype:

```
1 int printf(const char *format, ...);
```

  - `const` implies that `printf()` won't modify a string passed to it

Pointers and Strings

- Example 2: `fopen()`'s prototype is:

```
1 FILE *fopen(const char *pathname,const char *mode);
```

- The string arguments are both `char *`
- We can use constant strings:

```
1 fopen("input.txt", "r");
```

- Or we can use string arrays:

```
1 FILE *input;
2 char fn[256];
3 scanf("%s", fn); // Read filename from user
4 input = fopen(fn, "r");
```

Pointer Dereferencing

- This is the conceptually tricky one:
  - If `p` is a pointer to `x`, then `*p` makes it "appear" as `x`

```
1 int *p, x;
2 p = &x;
3 *p = 12; // Makes x 12
```

- Be careful: the function of the `*` character is context dependent!
  - It could multiply
  - It could declare a pointer type
  - it could dereference a pointer

Pointer Example

(Toy example) What will this code print?

```c
#include <stdio.h>

int main() {
  int x = 5; *p;
  p = &x;
  x++;
  printf("%d\n", *p);
  *p++;
  printf("%d\n", x);
  return 0;
}
```

Pointer Types

- Pointers to different types are all "the same", right?
  - They are all memory addresses
  - Memory addresses are all the same size

Pointer Types

- Pointers to different types are all "the same", right?
  - They are all memory addresses
  - Memory addresses are all the same size
- So why do pointers have different types?

Pointer Types

- Pointers to different types are all "the same", right?
  - They are all memory addresses
  - Memory addresses are all the same size
- So why do pointers have different types?
- When *dereferencing* the type is crucial!
- The type controls how much data will be read beyond the pointer
  - eg: an `int` is typically 4 bytes
  - An `int *` points to the *first* of the 4 bytes
  - Dereferencing an `int *` reads all 4 bytes

Pointers and Functions

- A pointer function argument uses the same syntax as an array argument:

```
1 void f(int *x); // Pointer to int argument
```

  - Aside: This is often just read as "int star"

- Inside the function the thing `x` points to can be modified or accessed with `*x`:

```
1 void f(int *x) {
2   y = 2 + *x;
3   *x = 2 * (*x); // This syntax is painful
4 }                // ()'s for clarity
```

Pointers and Functions

- Wait, is used *exactly the same* syntax as an array argument?

Pointers and Functions

- Wait, is used *exactly the same* syntax as an array argument?
  - Yes! Array arguments are a pointer to the first element
  - This is the same amount (and type of) data as a pointer to a single variable
  - It is up to you, the programmer, to interpret pointer arguments correctly!

Example

Write a C function which takes two unsigned integers as arguments, zeros the variable holding the larger value, and returns what the larger value was.

eg: If the function was passed `x=2` and `y=10` it would zero `y` and return 10.

Function prototype:

```
1 int zeroLarger(unsigned int *a, unsigned int *b);
```

Better Example

Write a C function which takes two integer arguments and swaps them.

Function prototype:

```
1 void swap(int *a, int *b);
```

Advanced Pointer Example

- When using command line arguments, `main()` is written as:

```
1 int main(int argc, char *argv[])
```

- What on Earth is `char *argv[]`?!?!

Advanced Pointer Example

- When using command line arguments, `main()` is written as:

```
1 int main(int argc, char *argv[])
```

- What on Earth is `char *argv[]`?!?!
- `char *` means it points to `chars`

Advanced Pointer Example

- When using command line arguments, `main()` is written as:

```
1 int main(int argc, char *argv[])
```

- What on Earth is `char *argv[]`?!?!
- `char *` means it points to `char`s
- `argv[]` means it is an array

Advanced Pointer Example

- When using command line arguments, `main()` is written as:

```
1 int main(int argc, char *argv[])
```

- What on Earth is `char *argv[]`?!?!
- `char *` means it points to `chars`
- `argv[]` means it is an array
- All together: `char *argv[]` is an array of pointers to `chars`
- Interpretation: it holds an array of strings!

Advanced Pointer Example

- Syntactically:
  - `argv[0]` is a pointer to a string
  - `argv[1]` is a pointer to a different string
  - ...etc
- So, you can do things like:

```
1 printf("First argument: %s\n", argv[0]);
```

- Or use `atoi()` to convert a numerical string to an `int`:

```
1 int x;
2 x = atoi(argv[1]);
```

Command Line Arguments Example

Write a C program which opens a file and prints the first 10 words from the file to `stdout`. The filename should be given as a single command line argument.

Multidimensional Arrays

- So far, all arrays have been 1D
- What if you want to store a matrix or image?

Multidimensional Arrays

- So far, all arrays have been 1D
- What if you want to store a matrix or image?
- In C, arrays can have multiple *dimensions*
- 2D example:

```
1 int x[3][3]; // a 3x3 matrix
```

- 3D example:

```
1 int y[10][3][3]; // 90 ints in 3 dimensions
```

- There is no *theoretical* dimension limit but compilers will have limits

Multidimensional Arrays

- Each unique combination of dimension indices indexes a unique element:

```
1 int x[2][2]; // four integers total
2 x[0][0] = 1;
3 x[0][1] = 2;
4 x[1][0] = 3;
5 x[1][1] = 4;
```

- This is one way of holding greyscale image data
- We will use this in the second programming assignment

Multidimensional Arrays

- Images typically use the RGB colour space
- Each pixel gets a red, green, and blue intensity
- An RGB image can therefore be stored as a 3D array:

```
1 unsigned char image[xres][yres][3]
```

- Here, a image of size `xres` by `yres` can have 3 `unsigned char` values for each pixel
- The red, green, and blue channels each have 255 intensity levels
  - Each `unsigned char` is 8 bits
  - 3x8 = 24b per pixel

2D Array Example

Write a C program which reads a 5x5 array from a file and finds the maximum value in the array. The file stores `float` data in csv format. Each line contains 10 `float`s, separated by commas. There are 10 lines in the file. There are no commas at the end of lines.

You may assume that the file does not contain errors.