Welcome!

COMP1511 18s1
Programming Fundamentals
Overview

_after this lecture, you should be able to..._

- have a basic understanding of the different types of i/o in C
- read and write files using C i/o functions and file redirection
- feel more comfortable working with argc/argv
- have a basic understanding of memory in C

_(note: you shouldn't be able to do all of these immediately after watching this lecture. however, this lecture should (hopefully!) give you the foundations you need to develop these skills. remember: programming is like learning any other language, it takes consistent and regular practice.)_
Admin

Don’t panic!

assignment 1 due YESTERDAY
nice work :)

week 5 weekly test due thursday
don’t be scared!

lab marks released
post in class forum || email your tutor
don’t forget about help sessions!
see course website for details
introducing: i/o

(input/output)
input/output?

i/o
(input/output)

various ways our programs can take input and give output
input/output?

you’ve already seen several ways

printf, scanf, getchar, putchar, fgets, ....
input/output?

these all work with **stdin** and **stdout**
(standard input and standard output)

**stdin**
scanf, getchar, fgets

**stdout**
printf, putchar
Other i/o

stdin and stdout go to the terminal

but there are other options

stderr
(standard error)

files
(e.g. “input.txt”)
stderr

still goes to the terminal

*semantically* different to *stdout*

used to print *errors*

can be redirected separately to *stdout*
can access with `fprintf`

```c
fprintf(stderr, "Uh oh, something went wrong!\n");
```
fprintf

just like printf, but works with **files**

```c
// stdout is a "file descriptor"
fprintf(stdout, "Hello, world!\n");

// stderr is also a "file descriptor"
fprintf(stderr, "Uh oh, something went wrong!\n");

// can also work with real files
FILE* output_file = fopen("output.txt", "w");
fprintf(output_file, "Hello!\n");
```
we can work with files in C

**printing** content to a file (just like printf)

**scanning** content from a file (just like scanf)
Files in C

we saw the syntax earlier:

```c
// opens a file called "output.txt" in "writing" mode
FILE* output_file = fopen("output.txt", "w");

// prints "Hello!" to the file
fprintf(output_file, "Hello!\n");
```
file redirection

we can also get the terminal to handle the file input/output for us

```c
// in a file "blah.c"
printf("Hello from a normal printf\n");
```

dcc -o blah blah.c

```bash
./blah > output.txt
```

```
$ cat output.txt
Hello from a normal printf
```

#include `<string.h>`

contains lots of functions to work with strings

man 3 string

note: many of these you can (and should, for practice) write yourself

e.g.: `int strlen(char *string)` – takes a string, returns the length.
how would you write it yourself?
well...

let’s try it out!
up next: memory
Memory

effectively a GIANT array
(4 GB big)

everything is stored in memory somewhere

variables are stored in memory

the code for your program is stored in memory

the code for library functions you use is stored in memory

(printf, scanf, …)

since everything is stored in memory, it has an address

(similar to your home address)
Memory Addresses

since everything is stored somewhere in memory, everything has an address.

we can get the address with & ("address of")

we print memory addresses in hexadecimal (why?)
Memory Layout

different things are stored in different places

variables are stored on **the stack**

dynamic memory (more on that next week) is stored on **the heap**
libraries & magic  0x00000000
your program   ~0x00600000
the HEAP       ~0x10000000
the STACK      0xBFFFFFFF
more magic     0xFFFFFFFF
some people draw memory “upside down”

(0xFFFFFFF at the top)

keep an eye out for which way up any given diagram is
up next: references
address of

from earlier:

we can get the address with & (“address of“)

```c
int number = 10;
printf("The address of number is: %p\n", &number);
```

we sometimes call this a “reference”
i.e. “&number” is a reference to the variable “number”
when we have the **address** of something, we know how to get to it

```c
int number = 10;
printf("The address of number is: %p\n", &number);

// might print something like:
// The address of number is: 0xffd53f50
```
going to the address of

when we have the **address** of something, we know how to get to it

we use the `*` operator for this: **dereference**

(“de-reference” – go to the reference, to get the value there)

```c
int number = 10;
printf("The address of number is: %p\n", &number);

// might print something like:
// The address of number is: 0xffd53f50

// we can print out the value at that address:
printf("The value at address %p is %d\n",
        0xffd53f50, *0xffd53f50);

// should print out "10"
```
going to the address of

we could do this directly in the printf:
int number = 10;
printf("The address of number is: %p\n", &number);
// might print something like:
// The address of number is: 0xffd53f50
printf("The value at address %p is %d\n", 
        &number, *(&number));

// should print out something like
// The value at address 0xffd53f50 is 10

// we can print out the value at that address:
printf("The value at address %p is %d\n", 
        0xffd53f50, *0xffd53f50);

// should print out something like
// The value at address 0xffd53f50 is 10
storing the address of

“but Andrew, why would we do that when we could just print out number directly?”

effectively.

addresses are most useful for telling things far away (i.e. other functions) about the address of something in your function.

we have a special type for storing the addresses of values

```c
int number = 10;
int *number_address = &number;

printf("The address of number is: %p\n", &number);

printf("The value at address %p is %d\n", 
        &number,  *(&number));

printf("The value at address %p is %d\n", 
        number_address  *number_address);
```
we call these **pointers** because they “point” to the variable whose address they store
storing the address of

pointer syntax:

```
[type] *[some_name] = &[something];
```

e.g.

```
int *my_pointer = &my_variable;
```

importantly: the **value** of the pointer is the **address** of the variable it points to
storing the address of

confused?

that's okay.

let's try it out!