CSSE2310/7231 – Lecture 1

Course Introduction
Introduction to Linux and C

Peter Sutton
School of Electrical Engineering and Computer Science
The University of Queensland

Welcome

- CSSE2310 / CSSE7231
  - Computer Systems Principles and Programming
- Teaching Staff:
  - Associate Professor Peter Sutton – course coordinator and lecturer
  - Large team of tutors

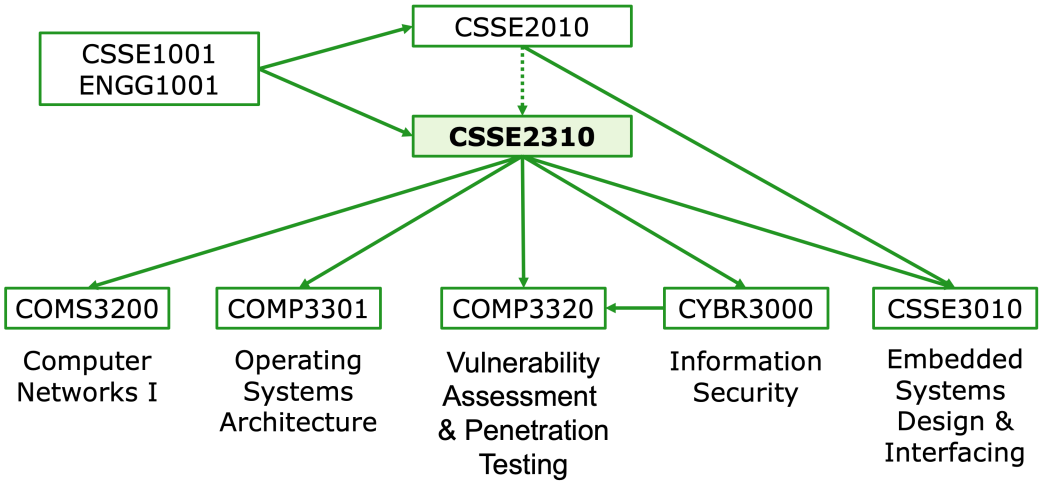
Today

- Course Overview
- Brief Intro to Linux on moss
- Brief Intro to C

What's This Course All About?

- Underlying principles of
  - Operating systems
  - Computer networks
- Systems programming in C
  - How to write programs that interact with the operating system and with other programs
  - Starting to practice good software engineering techniques
- Exposure to UNIX (Linux) operating system

Where This Course Fits in the (Undergraduate) Curriculum



Course Profile

- Describes
  - The course in detail
  - What you can expect
  - What we expect of you
- **You should read the course profile**
- Now for *some* of the details

Learning Activities

- **Lectures** – 2 hours per week (Tuesday 10am – 11:50am)
  - Introduce the content
  - Take notes! Lecture slides do not capture everything
  - Ask questions!
  - There will be a 5 -10 minute break in the middle
- **Contacts** – 1 hour per week (Friday 1pm – 1:50pm)
  - Will be used for additional lecture time in some weeks
  - Will allow time for demos, problem solving, exercises, sample exam questions, ...
- It will help if you bring a laptop – follow along, try things out
  - Experiment with the code examples after the lecture also!
- Lectures and contacts will be recorded and made available via Blackboard
- See week-by-week outline on Blackboard

Learning Activities (cont.)

- **Pracs** – 2 hours per week
  - One $\times$ 2 hour session (choice of nine sessions – signup to one)
  - Work on exercises, software tutorials and assignments
  - Opportunity to get individual help from tutors
  - You can attend extra sessions if space permits – but students signed up have priority
  - Queuing system used for asking questions in later weeks – go to https://q.uqcloud.net/csse2310
- Note – public holiday timetable variations over the semester
  - Fri 29 March – no contact or prac
  - Thu 25 April – no pracs

What we expect from you ...

- **Attendance** at lectures and contacts (or watch the recordings)
  - You may be disadvantaged if you don't attend or review the content
- **Seek help if you're having trouble**
  - Don't leave it too late
- **Hard work** – 10 to 12 hours per week
  - Average time commitment for average student seeking an average grade (4 or 5)
  - Possible breakdown (150 to 160 hours over the semester):
    - Lectures and contacts – $26 + 12 = 38$ hours
    - Early pracs (not working on assignments) – 2 hours per week – 6 hours
    - Review, exercises, study – 25 to 30 hours
    - Four assignments – 10 to 20 hours each – 65 to 70 hours
    - Exam study + exam – 15 hours
- **Feedback and ideas**
  - What can we improve? What do you want to learn about?

Assessment

- **Four Assignments**
  - A1 – programming (C strings, files etc.) 15%
  - A2 – debugging (defusing the binary bomb) 10%
  - A3 – programming (multiple processes, pipes) 15%
  - A4 – programming (network client and multi-threaded server) 15%
- **Final Exam**
  - Open Book 45%
- To pass the course, you must achieve 50% overall and 40% on each of
  - the total assignment mark (NOT each individual assignment)
  - final exam mark
- CSSE7231 students will have an additional component for programming assignments 3 & 4

Programming Assignments

- You will be writing C programs, to run on a Linux operating system, and storing them in a Subversion (svn) revision control system

  Compiler: `gcc`
  Target: `moss.labs.eait.uq.edu.au`

- Your code must compile and run on moss
- You can access moss via ssh (Secure Shell) from a terminal program, e.g.
  - PowerShell or Windows Terminal or PuTTy on Windows
  - Terminal on MacOS or Linux or similar
    - See the Ed Lesson for how to login, set up key authentication, use ssh-agent, etc.
- You might be able to use your own system for some aspects, but we will not provide support for this

"Challenges"

Some (potentially) challenging aspects of the course:

- Code must compile
- Writing whole programs (i.e. start from an empty file)
- Writing larger programs
- Automated testing

Code must compile

- Programs are targeted at computers
  - Computers (generally) demand precision
- If your code does not compile, the computer can't do anything with it (regardless of how "small" the error is)
- **In this course, code that does not compile will get zero marks for functionality.**
  - Just because a program does compile, doesn't mean you will earn marks – it still needs to do the right thing!

Whole programs

- In previous courses you may have "filled in the blanks" or added functionality to an existing program
- **In this course, you will have to write some programs from scratch**
- A blank page/file/program can be intimidating
  - We will show you some strategies to adopt, e.g.
    - Think before coding!
    - Break problems into smaller problems
    - Try to get each bit working on its own
- "All of that is mine" is rewarding

Larger programs

- The programs you write in this course may be larger than those you've written before
- BUT, they are small relatively speaking, e.g. you'll use

A shell (e.g. `bash`)	$193K$ LOC[1]
Encrypted connection (e.g. `ssh`)	$128K$ LOC
Linux kernel	approx $28M$ LOC ! (2020)

- To get better at writing larger programs you need to write larger programs.

[1] LOC=lines of code, ignores blank lines and comments

Larger programs (cont.)

- Remember: Programming is a practical discipline:
  - theory is important
  - but to improve, you must *do* it
- Each element of this course supports the others
  - Linux skills make you more productive, e.g. automation and scripting, file system navigation
  - Debugging skills make you more productive
  - Revision control makes you more productive (no fear of 'breaking' a working program)
  - The more you program, the better you get
- Make a commitment to yourself to work steadily, and give yourself the best chance to succeed

Larger programs (cont.)

- Software is more than code
- To be a good programmer, you need to be able to communicate about your code
  - How it is structured
  - Why it was designed that way
  - How it works (for the non-obvious bits)
- Sometimes, you are communicating with your future self!
  - Two weeks later... "Why did I make that change?!!"
  - "I'm sure it worked last week"
  - This is what revision control is all about...

Larger programs — Challenges

"It's hard to write that much code at the last minute."

In this course, assignments are *learning activities*.

- A lot of what you learn in this course will come from coding
- You will need to work consistently to be successful
  - Time to realise problems (debugging)
  - Time to ask questions
  - Time to replace code which is causing problems.

Automated testing

- Functionality is tested by running your programs against test cases and checking that the output matches exactly.
- Why?
  1. Much of the time, code (programs/libraries/...) is used by other code, not people.
     - Computers are not good at flexibility / inconsistency
  2. Allows for much faster marking
     - get feedback and results back faster
     - if there are problems, remarking is faster
  3. Allows for more consistent marking
- Style marking will be partially automated
- Tutors will assess some aspects of style and documentation

Yes, it's challenging, but ...

Some comments from recent SECaT course evaluations ...

- *I came into this course slightly worried since all the past students that I had talked to only mentioned how hard this course [is]. But after that, I soon realised that if you manage to stay on top of the content and start the assignments early like the lecturers and tutors mentioned, you'll be fine.*
- *As someone who almost failed CSSE1001 (couldn't code), I was very sceptical going into this course ... However, I was pleased to find that the course, given you stay up to date with the content, is more than manageable. Definitely, the most useful resource was the EdStem lessons ...*
- *While challenging, I learnt so much from this course and became a better programmer.*
- *assignments were difficult but rewarding once completed.*
- *I have learned a lot about this new OS (or perhaps I should say this new power)*

Tips (1/2)

- Keep a notebook or logfile of your questions as the occur to you
  - More efficient use of tutor time
- Remember - coding is a muscle memory skill! Copy/paste, autocomplete, fancy GUIs might seem easier, but they **will impact your learning**
- Be willing to try things out
- Use test scripts when we make them available.
  - Remember that the test scripts are not the marking scripts – it's your responsibility to implement the specification
- Allow time to deal with problems.

Tips (2/2)

- Test and debug as you go.
  - It's easier to remove eggshell before you've mixed it into the cake.
  - Write a test program if necessary.
- Don't be more than an hour from working code.
  - Comment out blocks of code
  - Get a working version from version control.
- Commit *working* code when you've made a reasonable improvement.
  - Only commit working code.

Let us help you

- If something is happening that:
  - you don't understand;
  - that seems to contradict what we've already told you;
  - ...
- Ask questions!
  - In class (lectures, contacts and pracs)
  - Via Ed Discussion Board
  - At consultation times

Information and help

- Discussion board (Ed Discussion, access via link on BlackBoard)
  - Questions about all aspects of the course
  - Do **not** post any of your assignment code publicly
    - You can use private posts (visible to you + staff) if your post involves assignment code or something personal
    - Do not use private posts for general questions
  - You can remain anonymous to other students if you wish
- Email csse2310@uq.edu.au
  - Questions / issues that are specific to you.
    - e.g. queries about assignment marks
    - BUT – extension requests must always be submitted to myUQ

Blackboard Ultra Course View

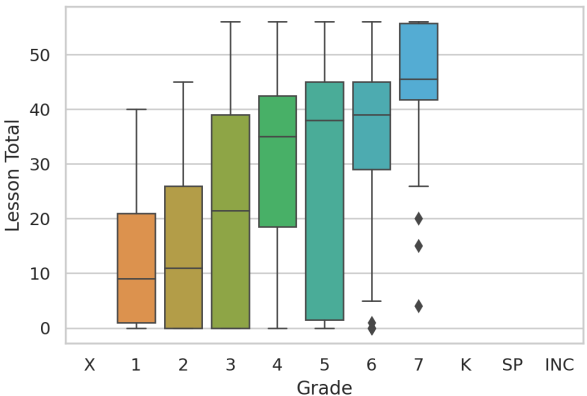
- Being trialled in CSSE2310/CSSE7231 this semester
  - One of 50 UQ courses to try it out
- Help is available from UQ Library (AskUS)
  - Mention that this is a Blackboard Ultra course and ask for tier 2 support
- You'll be surveyed about it towards the end of semester
- Let me know in the mean time if there are issues that are bugging you

- Quick demo ...

Ed Lessons

Introduced in Semester 1, 2022

- Not assessable, but see below...



Cheating and collusion

- All code which you submit for assessment must be, either
  - supplied by teaching staff **for this offering** of the course
    OR
  - written entirely individually by you
    OR
  - in some limited circumstances, from other sources (e.g. man pages)
- All code you submit will be checked for collusion and plagiarism (including but not limited to using M.O.S.S.) If similarity is found, we are obliged to start an integrity investigation
- Seek help from course staff in plenty of time if you are having difficulty with assessment.
- Be very careful when seeking outside help – outside tutors may facilitate cheating (and get you caught) rather than your learning.

Misconduct — Things not to do

1. Do not show your code to other students
2. Do not look at another student's code
3. Do not use code given to you in other semesters or courses
4. Do not get (or even try to get) other people to write code for you
5. Do not store your code in any online repository which **could** be accessible to others. (This will be deemed as if you have broken item 1)
6. Do not start a few days before the deadline and then claim that cheating was the only way to get it done in time

Use of Artificial Intelligence Tools

- AI tools are permitted to be used in CSSE2310/7231 programming assignments
- BUT
  - We'd prefer you didn't use them (it's important to learn the basics)
  - AI tools frequently get things wrong (and you may not have the knowledge to know)
  - Strict documentation/referencing requirements apply
    - More details in the assignment specs + guide on Blackboard
    - Misconduct charges may apply if you don't reference correctly
  - You must understand all the code you submit
    - A subset of students are interviewed about their code and you must be able to explain your code or your assignment mark will be scaled down
  - It may take you longer to do the assignments using AI tools
- Feedback from past students was that
  - AI tools were not particularly useful in assignments
  - AI tools were useful to explain some concepts

Questions?

Credits

Some of the slides for this course are built on those by:

- E.N. Elnozahy, U Texas
- R. Chandra, Cornell University

Many are based on those created by a previous course coordinator, Dr Joel Fenwick

Intro to Linux on moss

- Demo

Editing Files on moss

- Lots of text editors available
  - vi, emacs, nano, pico, ...
- We recommend (and expect you to learn)
  - vim (vi improved)
    - You can expect an exam question or two about vim usage
- Run **vimtutor** on moss to start learning

First C program

```c
// Function called "main" returns an int
// and takes two parameters
int main(int argc, char** argv) {
    return 0;
}
```

- C source files should end in `.c` e.g. `init.c`
- Filenames are case sensitive

Compile and run

Source is not executable – you need to compile it into a program

```
$ ./init.c      # Attempt to execute init.c from current directory
bash: ./init.c: Permission denied
$ file init.c
init.c: C source, ASCII text
$ gcc init.c
$ file a.out
a.out: ELF 64-bit LSB pie executable, x86-64, ...
$ ./a.out
$ echo $?
0
```

- The return value from main is sent outside the program - to the shell (e.g. bash)
- In bash this is available as `$?` – only works until the next command is run

Other Compiler Options

You can change what the compiler checks for with optional parameters (not an exhaustive list):

- `-std=c99` or `-std=gnu99` — Which version of the language. You'll *need* at least c99
- `-g` — Add debug information
- `-Wall` — Switch on all "avoidable" warnings.
- `-pedantic` — Warn about more problems
- `-O2` — optimise the build. Don't bother with this especially if you are debugging
- `-Werror` — a single warning stops the build. Don't use this unless you really mean it!

printf

A call to `printf` starts with a format string containing:

- Escape characters – start with \ (eg '\n', '\t')
- Place holders — start with % (Substitute in an expression)
- Normal characters

```
printf("Text here\n");  // Text + escape character
printf("3+5=%d\n", 8);
printf("3+5=%d\n", (3+5));
```

Place holders must (correctly) describe the type of the expression being substituted.

Output — numbers

Type	Symbol		
int	%d or %i		
unsigned int	%u		
double	%e	$12.34 \rightarrow 1.234000e+01$	Scientific notation
	%f	$12.34 \rightarrow 12.340000$	
	%g	$12.34 \rightarrow 12.34$	Combination
char	%c	'c'→c	
		99→c	ASCII

- More C types later

Another program – `digit.c`

- Count digits in a positive integer:

```
int main(int argc, char** argv) {
    int number=54;
    int result=(int)log10(number)+1;
    printf("%d has %d digits\n", number, result);
    return 0;
}
```

- Note casting syntax: `(int)`*expression*
- But does it build?

manual pages

```
$ man log10
```

- The first line tells us which page `LOG10` and section `3`.
- `SYNOPSIS`
  - `#include` — which header file do we need to include?
  - `Link with` — do we need additional libraries?

more gcc options

```
$ gcc digit.c -lm -o digits
```

- `-lm` — link in the `m` (maths) library (`libm.so`)
- `-o digits` — call the output file `digits` intead of default (`a.out`)

- `gcc` is actually carrying out multiple steps here, including:
  - preprocessing — dealing with things that start with #
  - compiling — turn source into executable form
  - linking — get missing functions from libraries (eg `printf()`)
    - e.g. `stdio.h` tells the compiler that `printf()` exists but not what it does.

Aside: printf man page(s)

What about `printf()`?

```
$ man printf
PRINTF(1)        User Commands        PRINTF(1)
...
```

```
$ man -k printf
...
```

Man sections (from `man man`):

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libararies)

...

So we want `man 3 printf`

Expressions

- In programming an expression is a fragment of code which can be "evaluated" to get a value.
- e.g.

  Literals `1.2, "a string", 'A'`

  Variables `cost`

- Expressions can be combined:

  Operators
  ```
  a + b
  y+m*x              // Note precedence
  ```
  Function calls
  ```
  get_cost(3)
  f1(nested('A', x + 2))
  ```
  ...
  ```
  costs[3]
  ```

Types

- In C, expressions (and variables) have explicit types
  - `int` – integer
  - `unsigned int` – unsigned (non-negative) integer
  - `char` – character
  - `float` – single precision floating point number
  - `double` – double precision floating point number
  - ...
- Variables must be declared before use (unlike Python), e.g.
  - `int x;`
  - `int y = 5; // Can declare and initialise in one statement`
- Note – variables should be initialised before use – C does not guarantee initialisation (unless it's a global variable)

Arrays

- Examples:

```
int numbers[100]; //array of 100 integers
char str[50]; // array of 50 characters
```

- Accessing members - use `[]` notation, e.g.

```
int first = numbers[0]; // the first member is index 0
```

- Note
  - The size of the array must be specified when creating it
  - The size is part of the type, so `int a[3]` and `int a[4]` are different types
  - Whole arrays can't be compared – you must compare them element by element

Array Initialisation

- Arrays can be initialised when declared, e.g.

```
int a[3] = { 1, 3, 5 }; // or just int a[] = {1, 3, 5};
```

- Size need not be given when an initialiser is given – size can be inferred
- Strings are an array of characters – with a null character to terminate them

```
// The following declarations are all the same
char str[6] = "hello";
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char str[] = "hello";
char str[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

- `str[5]` is zero (0 or `'\0'`) in this example

for loop

C uses `for` not a "for each" loop (as in python and some Java loops).

```
for (int i = 0 ; i < 4; ++i) {
    totalA += values[i];
}
```

1. `int i = 0` — Done once at the beginning of the loop
2. `i < 4` — If this is false, stop the loop
3. `totalA +=...` — Loop body
4. `++i` — Done after the loop body. Now jump to 2

for

All three parts of the loop header are optional:

```
for (;remain > 0;)
```

```
for (;;) // loop forever
```

```
for (A; B; C) {
    BODY
}
```

is equivalent to:

```
A;
while (B) {
    BODY;
    C;
}
```

do while

```
do {
   BODY
} while (TEST);
```

This loop will execute BODY at least once. vs:

```
while (TEST) {
    BODY;
}
```

which might not execute BODY at all (If TEST fails first time).

Coming Up

- Friday Contact
  - C programming example
  - Work through the Week 1 C Ed Lesson before Friday if you can
- Pracs this Week
  - Make sure you can login to moss
    - see Ed Lesson on how to set-up key authentication
  - Work through Linux tutorial
  - `vim` tutorial – run `vimtutor` on moss
  - Week 1 C Ed Lesson
- Next week
  - More C
    - Pointers, strings, structs, multidimensional arrays
  - Intro to Subversion (revision control system) and `make` (for automating builds)