ENCE 260 Computer Systems C Programming Section

Richard Lobb, room 211
Email: richard.lobb@canterbury.ac.nz
Web: www.cosc.canterbury.ac.nz/richard.lobb

Dr Richard Clare

ENCE260 Course Co-ordinator

Department of Electrical & Computer Engineering

Link Building Room 511

richard.clare@canterbury.ac.nz

Phone extension 93721

Teaching embedded systems in Term 4

Who is this Richard Lobb guy?

- Room 211, Jack Erskine Building
- Email richard.lobb@canterbury.ac.nz
- Adjunct Senior Fellow
- "Retired" from full-time academia
  - Was in CS dept at Auckland from 1978 to 2003
    - Here ever since
  - Computer graphics was my area
- Passionate about programming
- Teach lots of different languages (Python, C, C++, C#, Java, JavaScript, PHP, Matlab, ...)

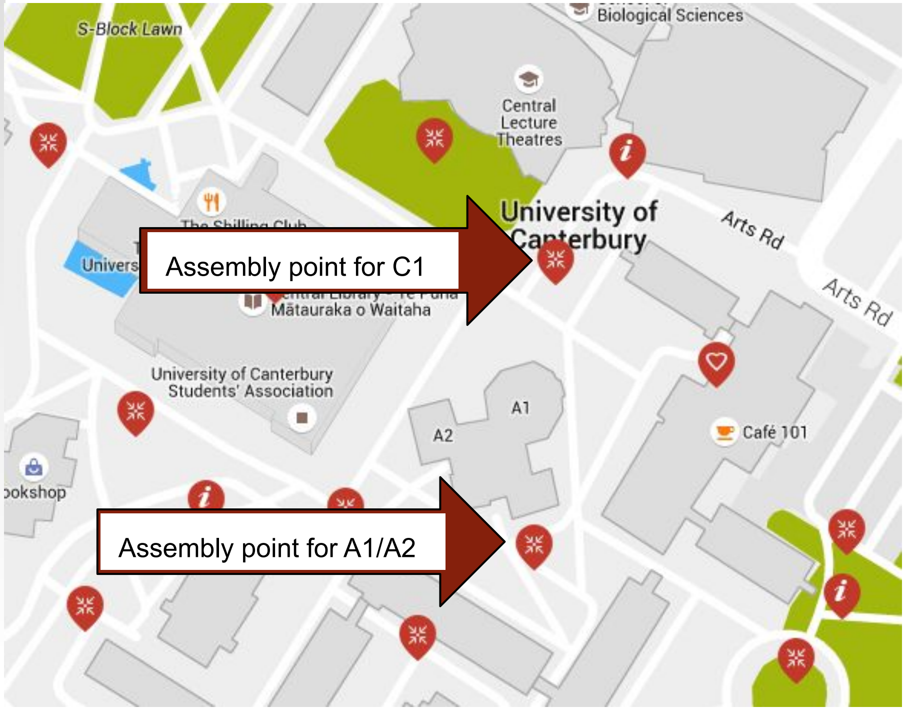
0. Administrivia

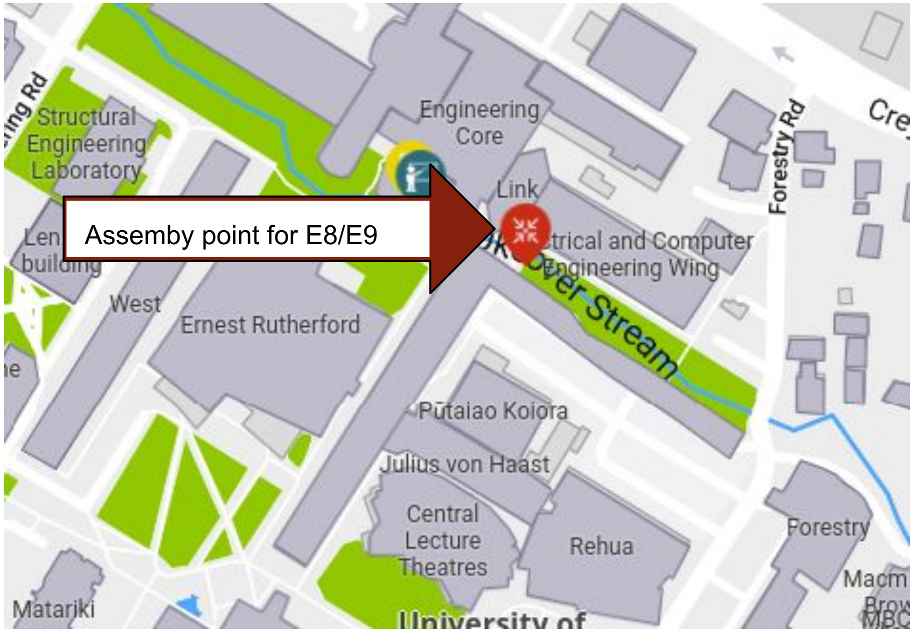
- Emergency procedures
- About the course
- Assessment
- Textbooks
- Timetable
- Intro to the C section

Emergency

www.canterbury.ac.nz/emergency/

- **University Security Emergency**
  - Ext. 6111 (campus phone) or 0800 823 637 (from mobile)





Earthquake

If inside:

- Stay inside
- Stop, drop, hold
- Under desks or down beside an internal wall

If outside:

- Stay outside
- Take shelter in the nearest open space/car park

New to NZ?

FREE EVENT

SETTLING INTO NEW ZEALAND

30 July, 6.15pm – 8pm

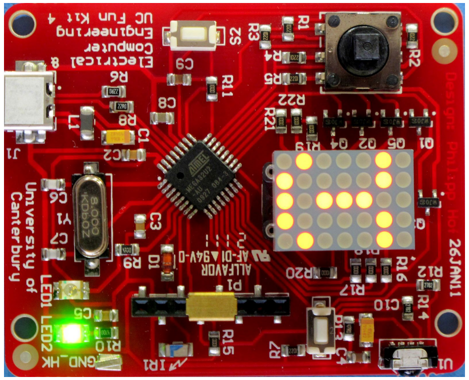
Visit the Diversity Agenda's Facebook page to secure your spot now.

THE DIVERSITY AGENDA.

Register at https://www.eventbrite.co.nz/e/settling-into-new-zealand-christchurch-tickets-62965739221

Course Structure: 3 Sections

1. C Programming, Dr Richard Lobb
   - Term 3:
     - Lecture/Tutorial Friday
     - No tutorials
     - One lab per week (book yourself in)
2. Computer Architecture, Dr Ciaran Moore
   - Terms 3 & 4:
     - Lectures Thursday
     - Tutorials: Every week in term 3 starting in week 3; every second week in term 4
3. Embedded Systems, Dr Richard Clare
   - Term 4:
     - Lectures Wednesday and Friday
     - One lab per week (book yourself in)
     - Tutorials: Every second week in Term 4
     - UCFK microprocessor boards available from mid-semester break



*Lectures and Tutorials**

Lectures (attend all 3)

- Wednesdays 17:00 - 18:00 in C1 **term 4 only, except week 1**
- Thursdays 10:00 - 11:00 in E8/E9
- Fridays 10:00 - 11:00 in E8/E9

All lectures are recorded on Echo360.

Tutorials (attend 1, starting week 3)

- Mondays 13:00 - 14:00 in A1
- Thursdays 13:00 - 14:00 in C2

Tutorial type problems, in a smaller lecture theatre.

* Subject to change. The most up-to-date schedule is always on CIS: ENCE260 2019

Assessment

- 10% on C programming “assignment”
  - Actually it’s a 2-part “super quiz”; details later
- 10% on embedded systems assignment (Term 4)
- 10% on Learn/quiz.cosc quizzes (weekly)
  - C, Comp. Arch, Embedded Systems
  - Roughly 13 quizzes in total, equally weighted (ie 0.8% each)
- 20% on test (Friday 13 Sept, evening, 6 pm)
- 50% on final exam (does not include C programming)

Warning re collaboration on labs/assignments

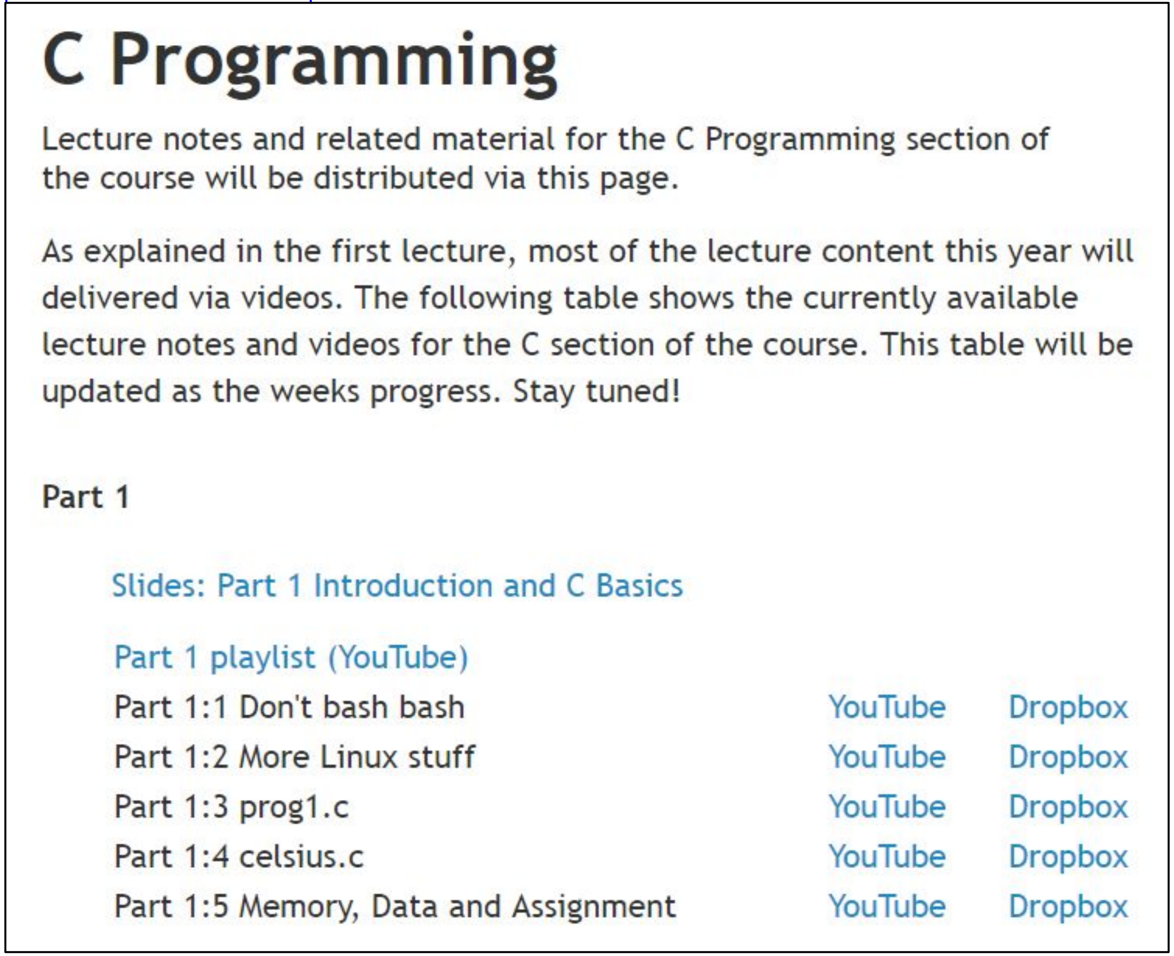
Warning re minimum 45% on invigilated components

Textbooks

- Prescribed text:
  - None, each lecturer provides detailed handout material
- Recommended text (for C section of course):
  - K. N. King, C programming: a modern approach (2nd Edition)

C: videos + 1 lecture per week

Learn page



C Programming

Lecture notes and related material for the C Programming section of the course will be distributed via this page.

As explained in the first lecture, most of the lecture content this year will delivered via videos. The following table shows the currently available lecture notes and videos for the C section of the course. This table will be updated as the weeks progress. Stay tuned!

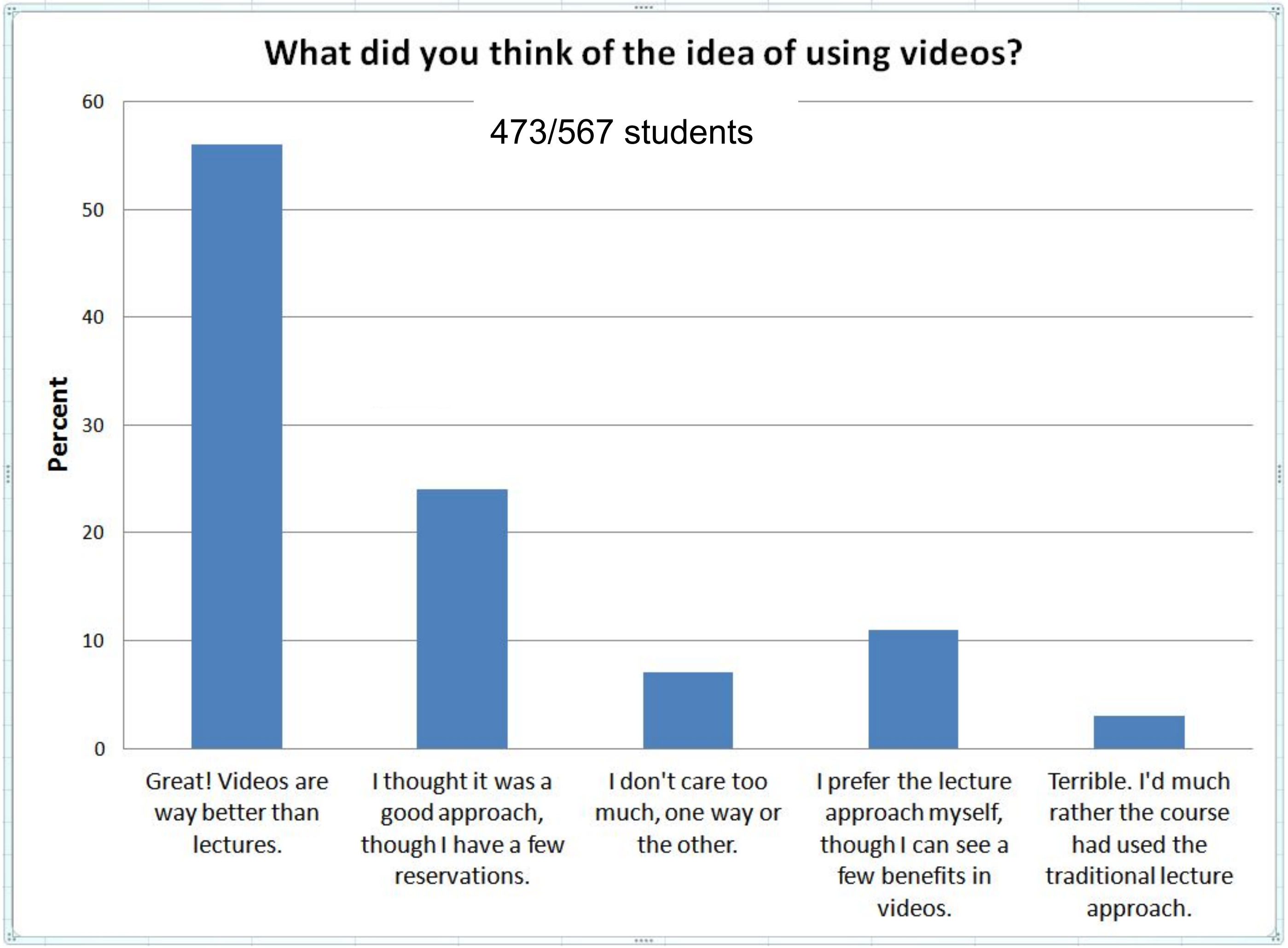
Part 1

Slides: Part 1 Introduction and C Basics

Part 1 playlist (YouTube)

Part 1:1 Don't bash bash	YouTube	Dropbox
Part 1:2 More Linux stuff	YouTube	Dropbox
Part 1:3 prog1.c	YouTube	Dropbox
Part 1:4 celsius.c	YouTube	Dropbox
Part 1:5 Memory, Data and Assignment	YouTube	Dropbox

COSC121-18S1 video feedback



Getting help

- Class forum (*Learn*)
- Talk to tutors or me in the labs
- Help on C:
  - C Textbook (King)
  - cppreference website
  - Google
- On line help for Linux
  - The *man* program displays manual pages
    - *man ls* displays the manual for the *ls* command
    - *man -k blah* displays all manual pages containing the keyword *blah*
  - Debian on-line tutorial/reference manual
    - https://www.debian.org/doc/manuals/debian-reference/ch01.en.html

Doing labs at home

- You are strongly advised to attend scheduled labs
  - Keeps you in touch with the course, tutors and me
- But you can do extra lab work at home:
  - Run *linux* (e.g. Ubuntu) under Windows via *vmware* or Oracle *VirtualBox.* Google for instructions (lots of hits).
    - Usually the easiest solution
  - Install *linux* on your home machine instead of Windows
    - Good soln but a bit drastic for most
  - Install *linux* as well as Windows (dual-boot)
    - Good soln but need to partition disk or need 2 disks
  - Get a Raspberry Pi 3/4! [$63/$80 + SD card + other accessories]
    - The fun option, and probably the most instructive, too
    - But be aware that data type sizes are different from lab machines

Linux install fest

Tuesday 23rd, Lab 2 (probably), preferred time … ?

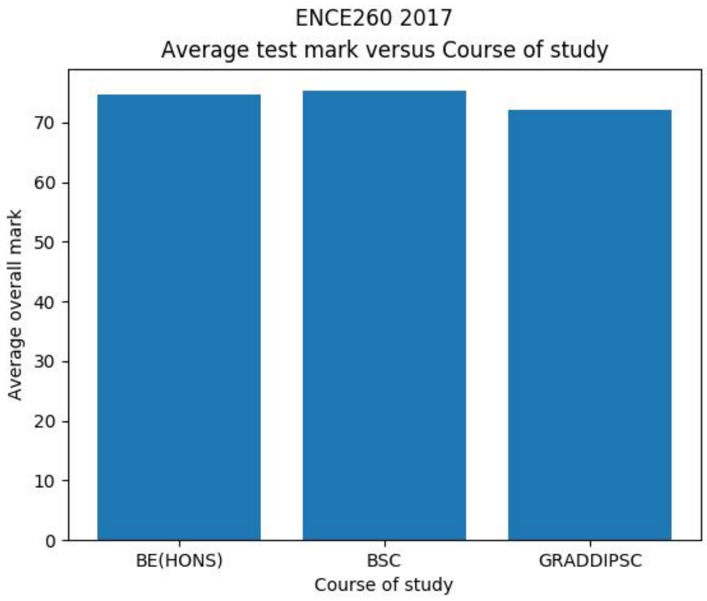
Who are the helpees?

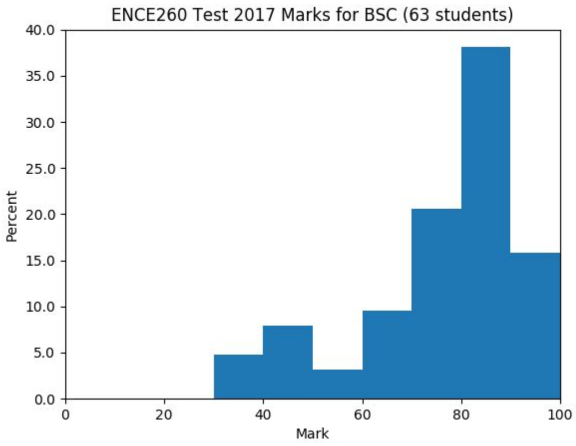
Who are the helpers?

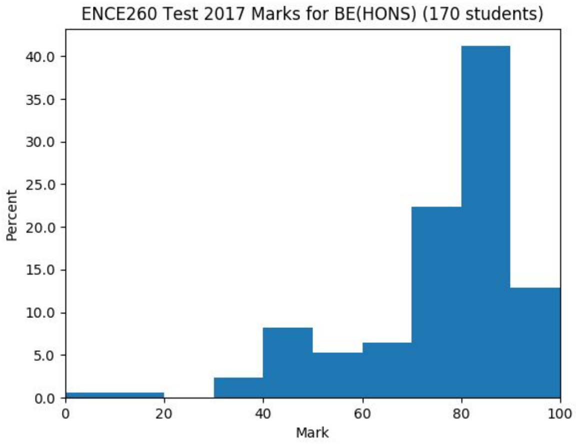
How to succeed in this section

- Do all the labs, starting early – labs will take well over 2 hours
- Watch the videos before the labs
- Get started early on the assignment questions
  - It’s a 2-part “superquiz”
- Don’t copy code; write your own
- Try to solve problems by yourself
  - Read the book
  - Experiment with code
  - Google
- Don’t just hack at code until it works
  - Work out what’s wrong before continuing
  - Every failure is a learning opportunity
- Don’t give up when the going gets tough

“I’m not doing COSC
So this course is hard for me”


ENCE260 2017
Average test mark versus Course of study


ENCE260 Test 2017 Marks for BSC (63 students)


ENCE260 Test 2017 Marks for BE(HONS) (170 students)

Class Reps

- Volunteers?
- Elected volunteers: please register at
  - http://ucsa.org.nz/classreps/signup/

Goals of C section of 260

- To teach you C
  - Still widely used for
    - systems programming
    - microcontroller development
- To improve your programming skills
  - More practice
  - Learning different languages
  - Use of various tools
- To expose you to Linux

Goals of C section of 260 (cont'd)

- To give you a lower-level understanding of computers and programming
  - No interpreter or virtual machine in the way
  - You get to see:
    - Compilation to raw machine code
    - Layout of code in memory
    - Linking and loading
    - Dynamic memory management
    - Calls to the operating system

Tentative timetable (C)

Week #	Begins	Topics
1	15/Jul	Linux and C fundamentals
2	22/Jul	Selection and iteration
3	29/Jul	Arrays and functions
4	5/Aug	Pointers and strings
5	12/Aug	Structures
6	19/Aug	Dynamic memory

1. C Basics

First lab and tutorial includes an introduction to Linux, too.

- Why Linux?
- Why C?
- History of C
- C versus Python
- C versus Java
- A simple C program
  - The C preprocessor
  - Declaring data
  - Formatted I/O

Why Linux?

- It's free
- It underpins Android
- It underpins a large percentage of other smart devices
- It runs ~70% of the world's websites
  - https://news.netcraft.com/archives/2017/09/11/september-2017-web-server-survey.html

Why use a command-line interface?

- Why would I learn a clunky old command line interface? Everything nowadays uses GUIs.
- But … really?
  - Is this how everyone programs nowadays?
- Sometimes text interfaces are just better
  - Faster and more-powerful
  - Certainly they're easier to write!
- But they *do* take longer for the user to learn

Why C?

- Brings you much closer to the hardware
- Used in other courses (e.g. networking, graphics)
- Close relationship to Unix/Linux
  - All the "classic" Unix system software is written in C (including kernel)
  - Extensive set of tools for support of C programming
- Widely used in industry
  - Embedded systems
  - Legacy applications development

Note: above motivations are not based on C's qualities as a programming language!

History of C

- First there was assembly language ("assembler"). Then ...
- BCPL → B (Ken Thompson, 1970) → C (Dennis Ritchie, 1973)
- Developed specifically for implementing Unix
  - Easier and more portable than assembler
- Book by Kernighan and Ritchie (1978) defined "Classic C" (K&R C)
  - But "loose" spec
- ANSI standardisation 1989 (C89), 1999 (C99) ← We do this
  - Newer standard 2011 (C11)
- C++ 1980's onwards (Bjarne Stroustrup)
  - Classes, overloaded operators, templates, exceptions, library classes, ...

Strengths and weaknesses of C

See King, section 1.2

- Strengths
  - Efficiency
  - Portability
  - Power (?)
  - Flexibility
  - Standard library (?)
  - Integration with Unix

- Weaknesses
  - Error-prone
  - Hard to understand
  - Hard to maintain

Obfuscated C contest winner: input *n* and it prints all solutions to *n-queens* problem (see King):

```
int v,i,j,k,l,s,a[99];main(void){for(scanf("%d",&s);*a-s;v=a[j*=v]-a[i],k=i<
s,j+=(v=j<s&&(!k&&!!printf(2+"\n\n%c"-(!l<<!j)," #Q"[l^v?(l^j)&1:2])&&++
l||a[i]<s&&v&&v-i+j&&v+i-j))&&!(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&
++a[--i]);printf("\n\n");}
```

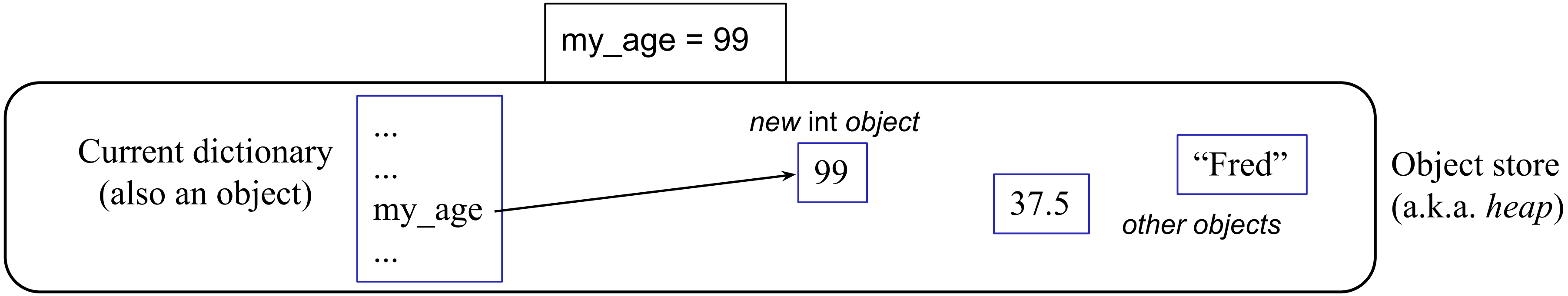
C versus Python

- Chalk and cheese
  - Python is a modern programming language; C is structured assembly language
- C compiles to run on CPU, not interpreted
- Statically typed – all variables must have their type declared before they can be used
- Insensitive to layout
  - Statements terminated by semicolon
  - Blocks enclosed in braces, e.g. { this-is-a-block }
- The only data types for structured data are *array* and *struct*
  - No (real) lists, strings (just *char* arrays), dictionaries, sets, OO, modules
- Minimal set of standard library functions

But CodeRunner questions enforce a standard layout

“Memory” in C

In Python, we saw memory as an “object store”



In C we deal with the *actual* computer memory.

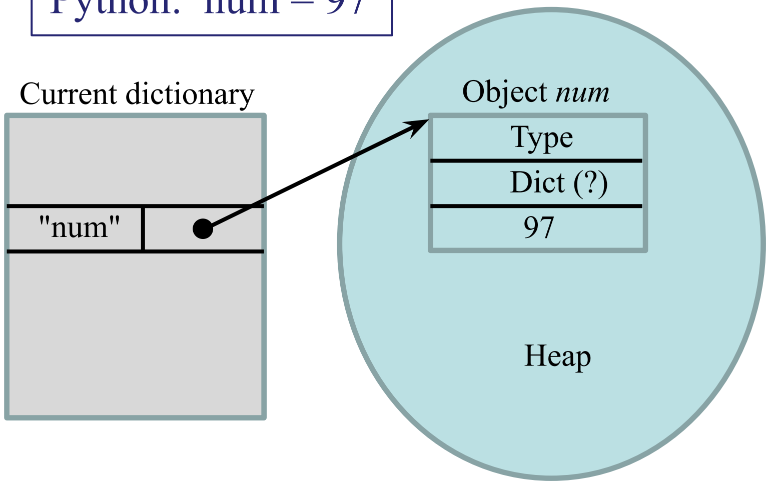
A linear array of several billion 1-byte numbered storage locations.

23	39	11	0	255	7	3	…	35	44	0	100	…
0	1	2	3	4	5	6		1001	1002	1003	1004	

C versus Python cont'd

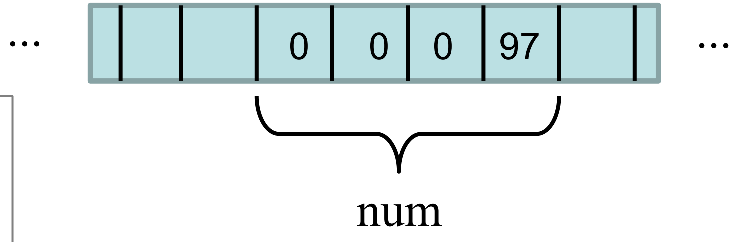
- **Variables are locations in memory, not references to objects**

```
Python:  num = 97
```



```
C:  int num;
     ...
     num = 97;
```

Compiler reserves 4-bytes of memory for num. Assignment then puts the number 97 at the target location.

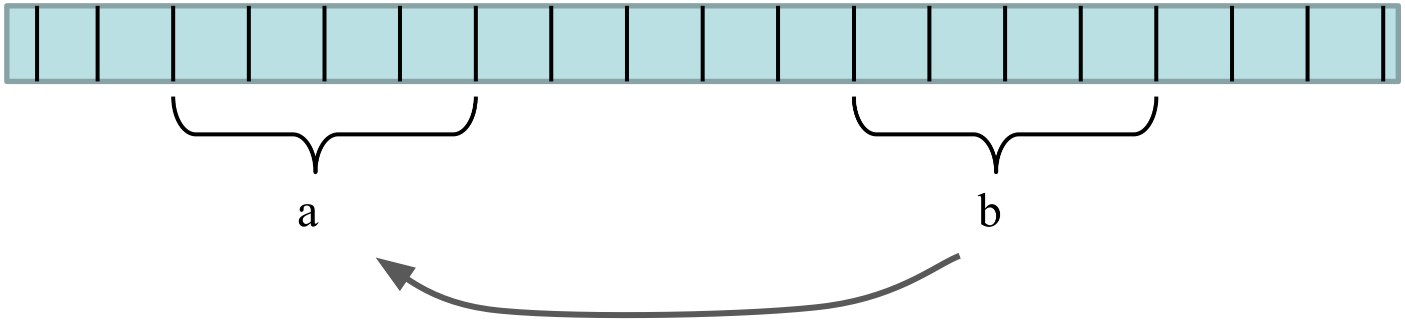


Few safety nets in C.

- Variables aren't initialized
- Array bounds not checked
- Pointer references not checked
- Easy to crash

C's "assign by value"

- In C, the statement `a = b` means *copy the contents of the memory block at the location* `b` *to the location* `a`.



- This is *not* the same as Python, where the LHS is the name of a new object reference - "assign by reference"
- It *is* the same as Java's assignment of primitive types
  – But not object types, which again are references

C versus Java

- C is a procedural language (like Pascal, FORTRAN etc)
  - All statements occur within procedures/functions
  - No classes/objects/methods
  - Data is either global (accessible from everywhere) or inside one file or inside one function
- C compiles to run on CPU, not on a virtual machine.

For SENG201 students

- Vaguely like writing a single nameless Java class with all methods being static
  - The class fields (also static) are all the *global* variables
  - But C pointers are something else!

C versus Java or *Python*

- C doesn't have reference variables
  - Instead have to use explicit pointer variables
  - Parameters are always passed by value
- Do-it-yourself dynamic memory allocation/deallocation
  - No "garbage collector"
  - Errors lead to "memory leaks" or heap corruption and crash

Interlude - odds and ends

- Does anyone have experience running Linux from a pen-drive?
- On home machines don't forget to set *geany* to use spaces for indentation everywhere
- Open brace for *functions* is on the next line
- Knowing signed binary numbers is really useful
  - More in Ciaran Moore's lectures
  - Or see

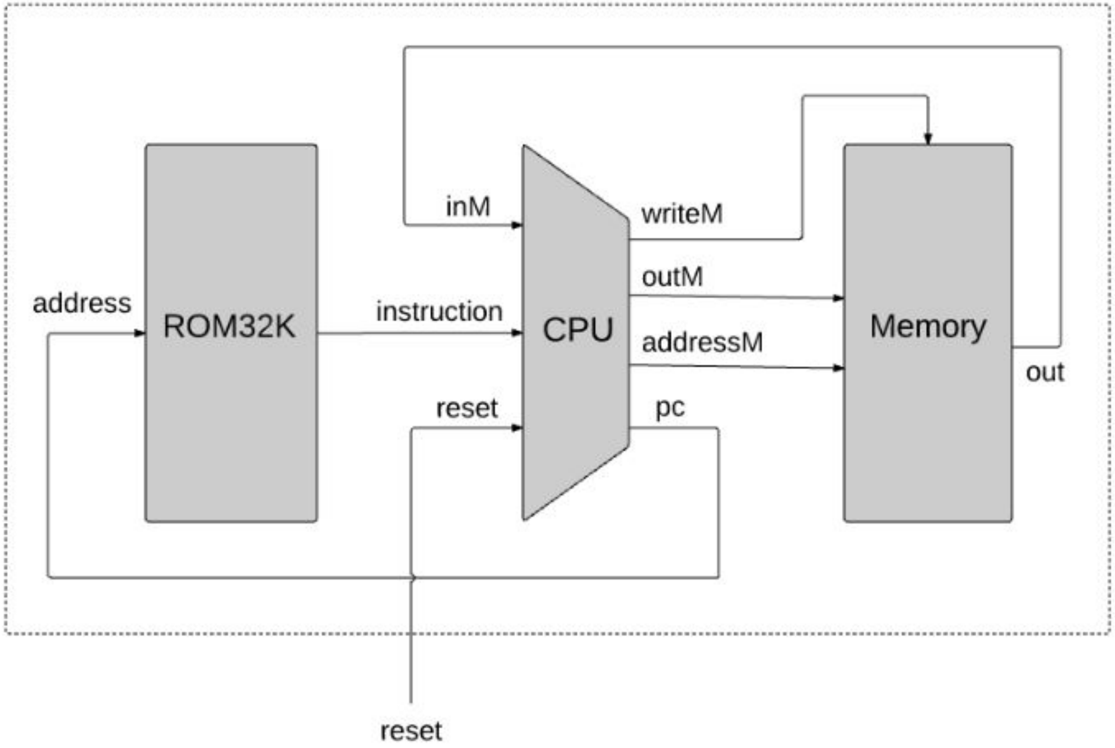
  https://www.swarthmore.edu/NatSci/echeeve1/Ref/BinaryMath/NumSys.html
- nand2tetris is a great course for people who want to understand computers from the *nand* gate through language implementation to games
- *CODE: The Hidden Language of Computer Hardware and Software* by Charles Petzold is a popular book that introduces half of nand2tetris.

Interlude (con'td)

Demo assembly-language output from C program

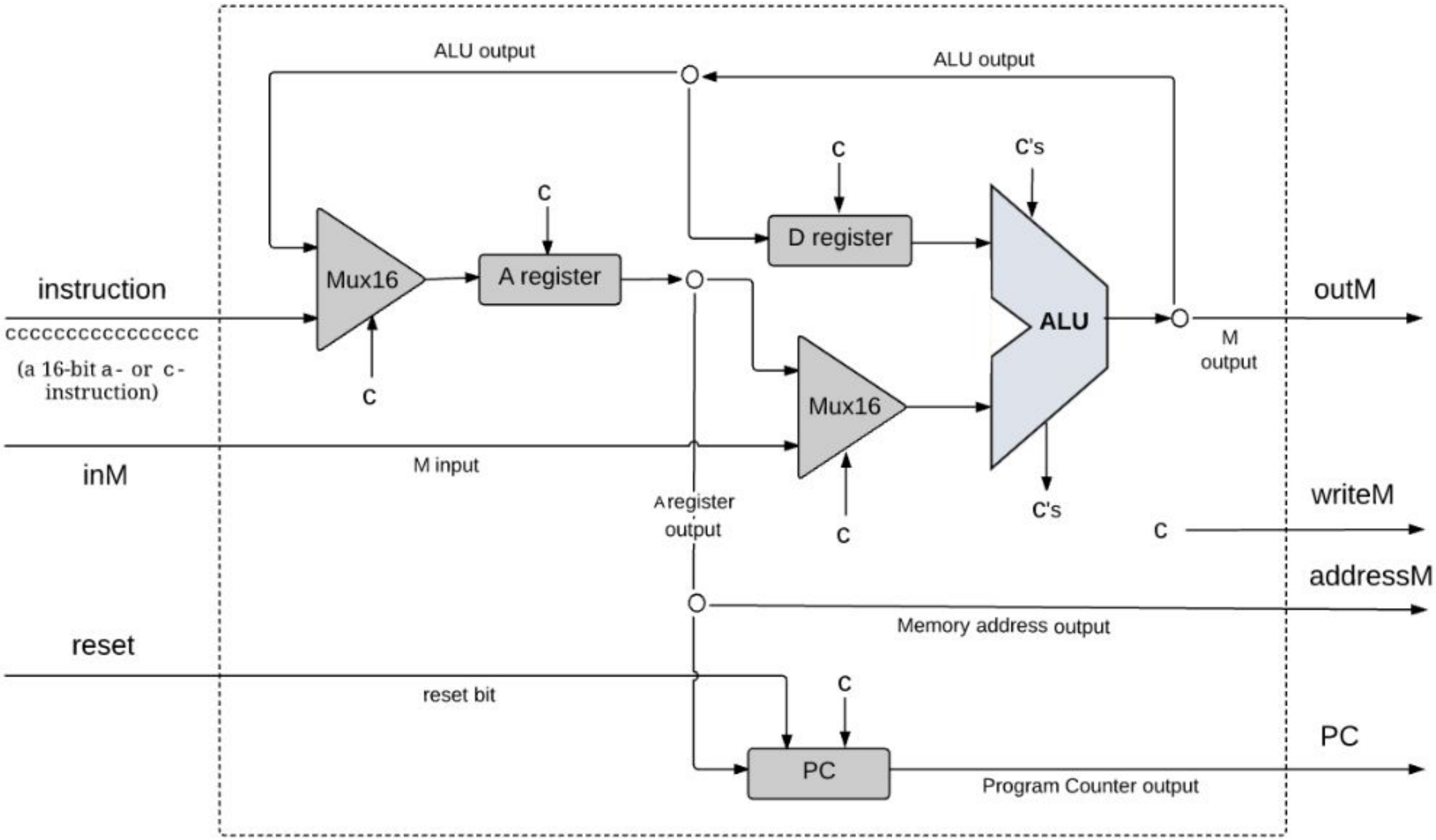
- Show demoassembly.c
- Compile: gcc -S demoassembly.c
- Show demoassembly.s
- Assemble: as -o demoassembly.o demoassembly.s
- Inspect:
  - file demoassembly.o
  - nm demoassembly.o

Interlude (cont'd): the nand2tetris computer ("hack")



"Harvard architecture": separate memories for code (ROM32K) and data (RAM Memory)
Like the UCFK AVR microcontroller.

Interlude (cont'd): the hack CPU



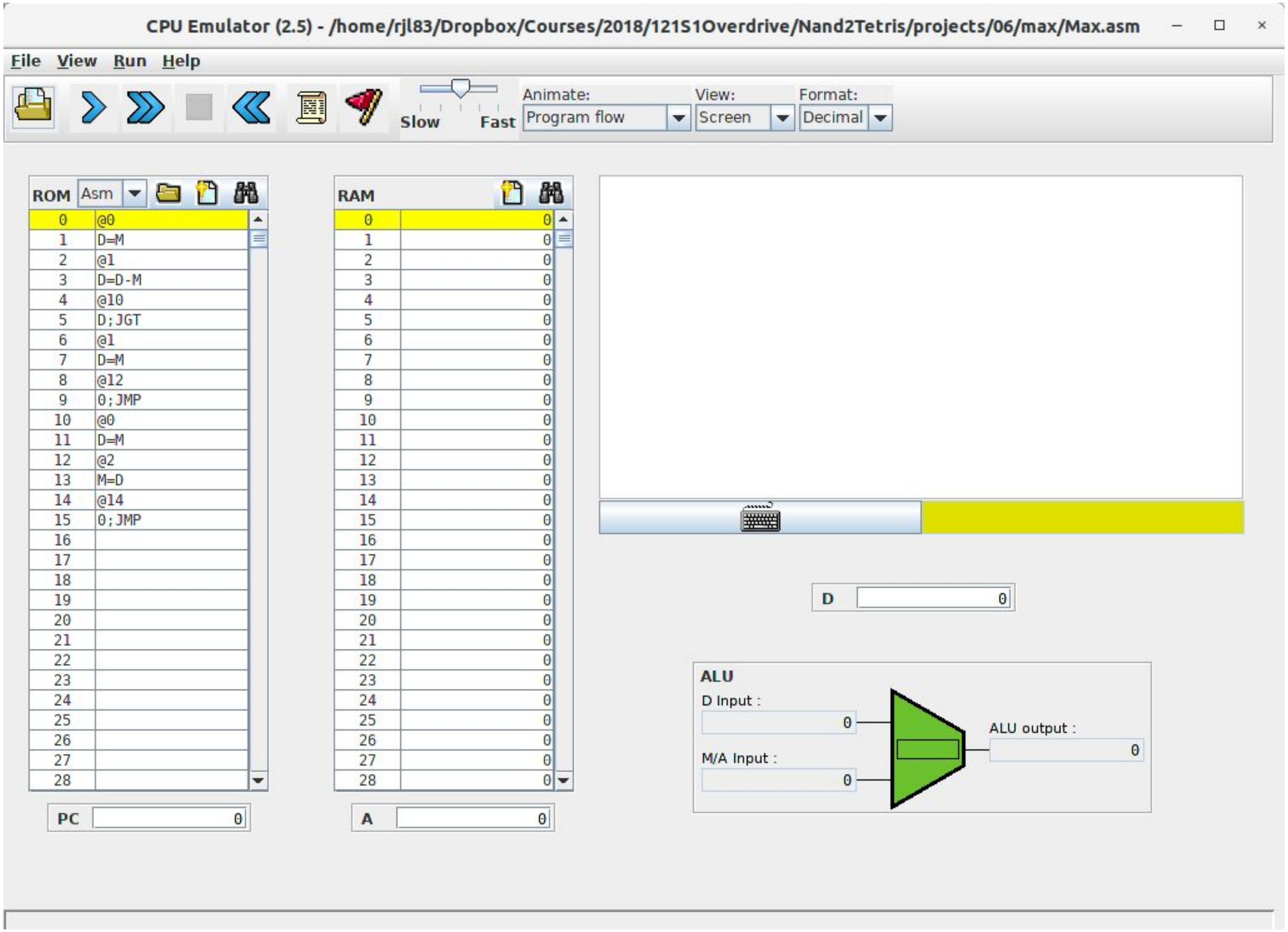
Interlude (cont'd) Hack assembler

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/06/max/Max.asm

// Computes R2 = max(R0, R1)  (R0,R1,R2 refer to RAM[0],RAM[1],RAM[2])

   @R0
   D=M              // D = first number
   @R1
   D=D-M              // D = first number - second number
   @OUTPUT_FIRST
   D;JGT            // if D>0 (first is greater) goto output_first
   @R1
   D=M              // D = second number
   @OUTPUT_D
   0;JMP            // goto output_d
(OUTPUT_FIRST)
   @R0
   D=M              // D = first number
(OUTPUT_D)
   @R2
   M=D              // M[2] = D (greatest number)
(INFINITE_LOOP)
   @INFINITE_LOOP
   0;JMP            // infinite loop
```

Interlude (cont'd)



CPU Emulator (2.5) - /home/rjl83/Dropbox/Courses/2018/121S1Overdrive/Nand2Tetris/projects/06/max/Max.asm

Animate: Program flow

View: Screen

Format: Decimal

ROM	Asm
0	@0
1	D=M
2	@1
3	D=D-M
4	@10
5	D;JGT
6	@1
7	D=M
8	@12
9	0;JMP
10	@0
11	D=M
12	@2
13	M=D
14	@14
15	0;JMP
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

PC 0

RAM	
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

A 0

D 0

ALU

D Input : 0

M/A Input : 0

ALU output : 0

DEMO using *pythontutor.com*

prog1.c *(slightly modified from lab version)*

```c
/* Your first mindless C program */
#include <stdio.h>
#define SECOND_NUMBER 20
int main(void)
{
    // First the declarations
    int number1;
    int number2;
    int total;

    // Now some code
    number1 = 10;
    number2 = SECOND_NUMBER;
    total = number1 + number2;
    printf("The sum of %d and %d is %d\n", number1, number2, total);
    return 0;
}
```

Note two different comment types

Compiling, building and running

Compile, link and run with Linux *bash* shell:

```
$ gcc -g -std=c99 -Wall –Werror -o prog1 prog1.c
$ prog1
The sum of 10 and 20 is 30
```

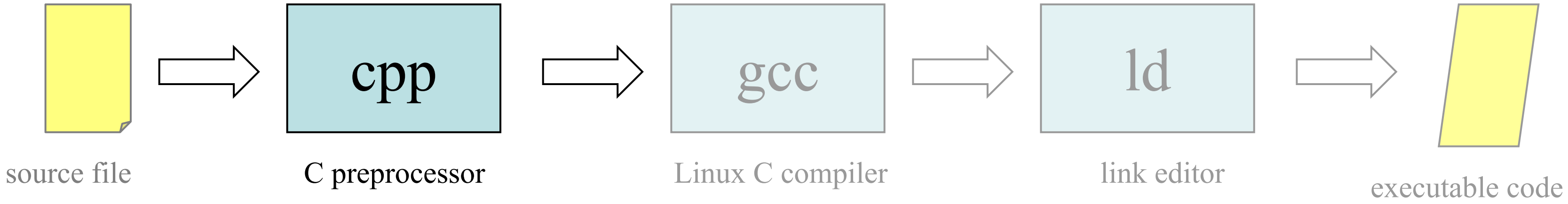
- `gcc` is GNU C compiler
- `-g` option outputs extra runtime debugging info
- `-std=c99` forces ISO C99 compliance
- `-Wall` outputs all warnings
- `-Werror` treats warnings as errors
- `-o filename` specifies name of output file
    - In this case it's an *executable* file

Standard layout of a C Program

1. #include header files
2. #define constants
3. Declare global variables
   - None in this example
4. Declare and define all the functions, each being:
   - Signature declaration
     - Return type, function name and parameters (each with type)
   - Body:
     - Local variables
     - Function code

The C preprocessor (cpp)

- *cpp* is a “macro preprocessor” (called by *gcc*)
- Before compilation begins, *cpp* transforms your **source** program



- Lines beginning with ‘#’ are *cpp* commands
  - `#include <blah.h>` inserts the file */usr/include/blah.h*
  - `#include "blah.h"` inserts the file *./blah.h*
  - `#define BLAH thing` replaces all BLAHs with string *thing*
- *cpp* just does simple text transformations
  - no syntax or semantics checks

Static typing

cf Python's dynamic typing

- In C, a variable name is a label for a pre-allocated chunk of memory
  - Variables must be *declared* before they're used
    - Recommend: declare all variables *first*, at the start of a block, before the statements [this is *required* by C89]
  - Declaration specifies *type* of data
    - And hence amount of memory to allocate
  - *Only* that type of data can ever be placed there
  - Declaration can include an *initialiser* to set initial value
    - **In ENCE260 you *must* initialise your variables**
    - Uninitialised variables result in random runtime behaviour!
- Functions must also be declared before they're called
  - Declaration specifies return type and type of all parameters

Interlude: how's your binary?

Assuming 8-bit binary integers:

1. What are the values (in binary) for 0, 3, 15, 127?
2. If the 8-bit number is *unsigned* what are the maximum and minimum values it can represent?
3. If the 8-bit number is *signed*
    a. What are the maximum and minimum values it can represent?
    b. What are the values (in binary) for -1, -2, -3, -127?

[Note: binary numbers will be covered in the architecture section of the course, but if you want to get a headstart, see

https://www.swarthmore.edu/NatSci/echeeve1/Ref/BinaryMath/NumSys.html]

Basic data types

- Integers: [*unsigned* | *signed*] [*long* | *short*] *int*
  - Platform dependent!!
    - In our labs (64-bit Linux Mint) *int* is 32 bits

Type qualifiers – next slide

- Characters: *char*
  - Always 8 bit ASCII
  - Treated as 8-bit *int*s (tho' may be signed or unsigned dep. on compiler!)
  - *char* constants like Java, e.g. 'x', '=' NB: enclosed in *single* quotes
    - Backslash codes for special ASCII chars e.g. '\n' is newline character
- Floating point types (*float* and *double)*
- Complex floating point numbers (not covered in ENCE260)
- [Weak] boolean type _*Bool* (or just *bool* if #include <stdbool.h>)

More on integer data types

- Integer data in C is binary: 1-, 2-, 4-, or 8-byte values

Type	**Size (bytes/bits)**
`char`	1-byte/8-bit
`short [int]`	2-byte/16-bit
`int`	4-byte/32-bit (on 32-bit and 64-bit machines)
`[long] long [int]`	8-byte/64-bit

- Additionally, can prefix type with `signed` or `unsigned` e.g.
  - `unsigned int` (a.k.a. `size_t`): *ints* in the range 0 to 4,294,967,295
  - Used by functions/operators that deal with sizes/lengths: `sizeof`, `strlen`
- `<stdint.h>` defines aliases: `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, `uint64_t`

printf

- Call function *printf* to do formatted output
  - `printf(string, expr1, expr2, ...)`
- Format string contains ordinary characters plus *conversion specifications*
  - %[number][*l*]letter e.g. %4d , %5.2f, %.3*l*f
  - *d*, *u*, *f*, *c*, *s* for *int*, *unsigned int*, *float*, *char*, *string* (= *char* array)
  - *l* is for *long* e.g. *long int* or *double*
  - number is of form *minimumFieldWidth.numDigitsPrecision*
    - e.g. **%5.1f** would print at least 5 characters (adding leading spaces as required) with exactly 1 digit after the decimal point.
- A conversion spec causes the next expression parameter to be appropriately formatted in its place

Demo: prog1.c

celsius.c

```c
/* celsius.c (King Chapter 2, page 24)
 * Converts a Fahrenheit temperature to Celsius
 * Slightly modified by RJL to use the EXIT_SUCCESS return code
 */
#include <stdio.h>
#include <stdlib.h>

#define FREEZING_PT 32.0
#define SCALE_FACTOR (5.0 / 9.0)

int main(void)
{
    float fahrenheit = 0;
    float celsius = 0;
    printf("Enter Fahrenheit temperature: ");
    scanf("%f", &fahrenheit);
    celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;
    printf("Celsius equivalent: %.1f\n", celsius);
    return EXIT_SUCCESS;
}
```

scanf

- *scanf* is “printf running backwards”
- Reads values from the input stream into nominated variables
  - `scanf("%d %f\n", &intVar, &floatVar);`
  - Note the '&' preceding the variables
    - Looking ahead ... this passes the *address* of the variable to scanf rather than its value
- *scanf* “pattern matches” first parameter with input from *stdin*
  - White space matches white space (any seq. of `' '`, `'\n'`, `'\t'` etc)
  - % denotes start of a format specifier
    - Defines an argument value
  - Other characters *must* exactly match or scanf aborts

Type *man scanf* for details

- Function return value is count of converted values