Welcome!

COMP2521 19T0
Data Structures + Algorithms

COMP2521 19T0
Week 1, Tuesday: Hello, world!

Jashank Jeremy
jashank.jeremy@unsw.edu.au

course introduction
more C syntax
linked lists, redux
tools of the trade

What?
Course Aims

thinking like a *computer scientist*
not just a programmer

know and understand
fundamental techniques,
data structures, algorithms

reason about
applicability + *effectiveness*

What?

Course Syllabus

Over the next few weeks...

- ADTS: stacks, queues, lists, trees, hash tables
- algorithm analysis: complexity, performance, usability
- sorting and searching techniques
- graphs, graph algorithms

Who’s Teaching?

Dr John Shepherd (jas@)
is the lecturer-in-charge

Jashank Jeremy (jashankj@)
is the lecturer

Sim Mautner	Olga Popovic
Hayden Smith	Elizabeth Willer
Clifford Sesel	Gal Aharon
Deepanjan Chakrabarty	Kristian Nolev

are your tutors and lab assistants

Who’s Learning?

recent students from...
COMP1511 (andrewt, andrewb, jas, ashesh)
COMP1917 (richardb, blair, salilk?, angf, simm)
COMP1921 (mit, ashesh, anymeyer?)

some C experience,
familiarity with pointers, ADTS,
style, and testing

(also a sense of humour)

Who’s Learning?

...and what are they supposed to know?

At the start of this course, you should be able to

- produce a correct C program from a specification
- understand the state-based model of computation
  (variables, assignment, addresses, parameters, scope)
- use fundamental C data types and structures
  (`char`, `int`, `float`, arrays, pointers, `struct`)
- use fundamental control structures
  (sequence, selection (`if`), iteration (`while`))
- use and build abstraction with function declarations
- use linked lists

Who’s Learning?

...and what are they supposed to learn?

By the end of this course, you should be able to

- analyse performance characteristics of algorithms
- measure performance behaviour of programs
- choose + develop effective data structures (DS)
- choose + develop algorithms (A) on these DS
- reason about the effectiveness of DS+A
- package a set of DS+A as an ADT
- develop + maintain C systems <10 kLoC.

How?

by lecturing at you!
in interactive tutorials!
in hands-on laboratories!
in assignments and exams!

How?

...by lecturing at you!

- present a brief overview of theory
- demonstrate problem-solving methods
- give practical demonstrations
- lectures are based on text-book.
- slides available as PDF
  (*usually* up before the lecture... :-)
- feel free to ask questions...
  but No Idle Chatting, please.

Tue 14–17, Thu 10–13
Ainsworth G03

How?

...in interactive tutorials!

- clarify any problems with lecture material
- work through problems related to lecture topics
- give practice with design skills
  ... think before coding
- exercises available (usually) the week before
  please read and attempt *before* your class

Webster252 ...[MTW]10, [MW]14, T16
GoldsteinG01 ...F10
GoldsteinG02 ...[HF]14

How?

...in hands-on laboratories!

- build skills that will help you to
  ...complete the assignment work
  ...pass the final exam
- give you experience applying tools + techniques
- small implementation/analysis tasks
- some tasks will be done in pairs
- don't fall behind! start them before your class if needed
- usually up in advance, due by Sunday midnight

J17–306 sitar
[MTWF]11–13; [MWHF]15–17; T17–19

How?

...in assignments!

- give you experience applying tools/techniques
  to larger problems than the lab exercises
- assignment 1 is an individual assignment
- assignment 2 is a group assignment
- will *always* take longer than you expect
- organise your time
  ...don't leave it to the last minute!
  ...steep late penalties apply!

How?

...in exams!

- practical exams in weeks 5, 8; each worth 5%
- 3h theory + practical extravaganza; worth 55%

How?

...in exams!

- Supplementary exams are only available to students who
  ...do not attend the exam **AND**
  ...have a serious documented reason for not attending
- If you attend an exam
  ...you are making a statement that you are 'fit and healthy enough'
  ...it is your only chance to pass (i.e., no second chances)

Assessment

5% + 5% prac exams
10% lab marks
10% assignment 1
15% assignment 2
55% final exam

Course Evaluation and Development

assessed with myExperience

also, we’d love to hear from you...
provide feedback throughout the session!

Acknowledgements

Always give credit if you use someone else’s work!
COMP2521 material drawn from...

- slides by Angela Finlayson (COMP2521 18x1)
- slides by John Shepherd (COMP1927 16s2)
- slides by Gabriele Keller (COMP1927 12s2)
- lectures by Richard Buckland (COMP1927 09s2)
- slides by Manuel Chakravarty (COMP1927 08s1)
- notes by Aleks Ignjatovic (COMP2011 ’05)
- slides and books by Robert Sedgewick

Academic Conduct and Integrity

On Academic Integrity

You'll be fired into space
or, at least, out of this course
if you're found to be using others' work as your own.

The lawyers would like me to remind you that
UNSW and CSE consider plagiarism as
an act of academic misconduct with severe penalties
up to and including exclusion from further study.

Academic Conduct and Integrity

On Academic Conduct

...don't be a dick.

The lawyers would like me to remind you that
UNSW and CSE consider bullying, harassment, ..
both on- and off-campus (including online!)
an act of student misconduct with severe penalties
up to and including exclusion from further study.

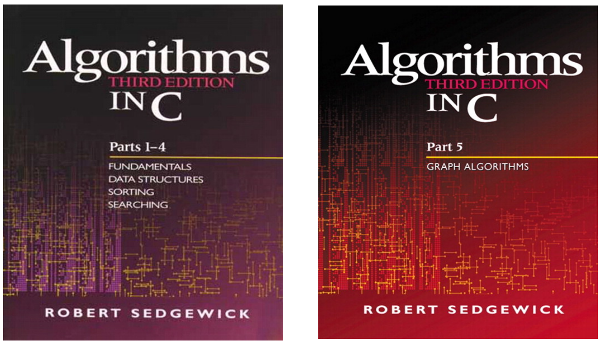
Course Website

`webcms3.cse.unsw.edu.au/COMP2521/19T0`

`cse.unsw.edu.au/~cs2521/19T0`

Resources

The Textbook



Algorithms in C, parts 1–4 and 5, by Robert Sedgewick

BEWARE!
there are *many* editions/versions of this book,
with various different programming languages
including C, C++, Java, and Pascal

Resources

Additional Face-to-Face Help

- weekly consultations...
  for extra help with labs and lecture material
  more time slots scheduled near assignments/exams
  email cs2521@ for additional consultations, if needed
- help sessions...to be advised
- WebCMS3 course forums

Advice

- Do lab exercises and assignments yourself
  (or with your pair partner when appropriate)
- Programming is a skill that improves with practice
  The more you practice, the easier labs/assignments/exams will be.
- Don’t restrict practice to lab times
  ...or two days before assignments are due.
- Make use of tutorials by
  ...attempting questions before the class
  ...participating!
- Go to consults if you need help or fall behind
- We want you to do the best you can!

More C Syntax

Compiling

LOOKING FOR dcc?
dcc held your hand in *many* ways.
the training wheels are now off! no *dcc* for you!
if you're desperate, try *3c*

- compiling for normal use

```
$ 2521 3c -o prog prog.c
```

- compiling multiple files

```
$ 2521 3c -o prog prog.c f2.c f3.c
```

- compiling with leak checking

```
$ 2521 3c +leak -o prog prog.c f2.c f3.c
```

Style in COMP1511/1917/1921

COMP1511, COMP1917, COMP1921
used a restricted subset of C

mandated layout, mandated brackets,
only `if` + `while`,
no side-effects, no conditional expressions,
functions with only one return…

… but this style is used in
no texts + no real code.

Style in COMP2521

the good
more freedom, more power!
more choice in how you express programs
can write more concise code

the bad
easy to produce code that's
cryptic, incomprehensible, unmaintainable

the style guide
available on the course website

Style in COMP2521

layout: consistent indentation
brackets: omit braces around single statements

control: all C control structures
(except `goto` ... that's how you get ants)

assignment statements in expressions
(but prefer to avoid side-effects ... that's how you get ants!)

conditional expressions ('ternaries') permitted
(use with caution! that's how you get ants!!)

functions may have multiple returns
(concise $\not\rightarrow$ clear! ants!!!)

'for' loops

with `while`

```
init;
while (cond) {
    /* … do something */;
    incr;
}
```

with `for`

```
for (init; cond; incr)
    /* … do something */;
```

‘for’ loops

with while

```
int sum = 0;
int i = 0;
while (i < 10) {
    sum = sum + i;
    i++;
}
```

with for

```
int sum = 0;
for (int i = 0; i < 10; i++)
    sum += i;
```

‘for’ loops

pros and cons

all interesting parts of the loop in one spot!
... but easy to write disgusting code

prefer `for` when *counting* or with *sequences*
... otherwise, use a `while` loop

‘switch’ statements

```
if (colour == 'r') {
    puts ("red");
} else if (colour == 'b') {
    puts ("blue");
} else if (colour == 'g') {
    puts ("green");
} else {
    puts ("invalid?");
}
```

```
switch (colour) {
case 'r':
    puts ("red"); break;
case 'g':
    puts ("green"); break;
case 'b':
    puts ("blue"); break;
default:
    puts ("invalid?");
}
```

the break is critical...
if it isn’t present, execution will fall through

‘switch’ statements

```
char *month_name (int);
```

Exercise: Switched On

Write a function `month_name`
that accepts a month (1 = Jan …12 = Dec)
and returns a string containing the month name
… assume the string will be read only
… use a switch to decide on the month

Exercise: Hip, Hip, Array

Suggest an alternative approach using an array.

jumping around: ‘return’, ‘break’, ‘continue’

avoid deeply nested statements!

return in a function
gives back a result to the caller
terminates the function, possibly ‘early’

break in `while`, `for`, `switch`
allows *early termination* of a block
jumps to the first statement after the block

continue in `while`, `for`
terminates one iteration… but continues the loop
jumps to *after* the last block statement

Conditional Expressions (‘Ternaries’)

`if` statements can’t return a value.

```
if (y > 0) {
    x = z + 1;
} else {
    x = z - 1;
}
```

… but what if they *could*?

```
x = (y > 0) ? z + 1 : z - 1;
```

Conditional Expressions (‘Ternaries’)

Rewrite these using ternaries, or explain why we can’t do that.

Exercise: Rewriting (I)

```
if (x > 0)
    y = x - 1;
else
    y = x + 1;
```

Exercise: Rewriting (II)

```
if (x > 0)
    y = x - 1;
else
    z = x + 1;
```

Assignment in Expressions

- assignment is really an expression
  … returns a result: the value being assigned
  … returned value is generally ignored
- assignment often used in loop conditions
  … combines test with collecting the next value
  … makes expressing such loops more concise

Assignment in Expressions

```
int nchars = 0;
int ch = getchar ();
while (ch != EOF) {
    nchars++;
    ch = getchar ();
}
```

…or …

```
int ch, nchars = 0;
while ((ch = getchar ()) != EOF)
    nchars++;
```

Assignment in Expressions

Exercise: Mystery Biscuits

```
void what_does_it_do (void)
{
    int ch;
    while ((ch = getchar ()) != EOF) {
        if (ch == '\n') break;
        if (ch == 'q') return;
        if (! isalpha (ch)) continue;
        putchar (ch);
    }
    puts ("Thanks!");
}
```

Function Pointers

- In C, you may point to anything in memory.
- The compiled program is in memory.
- The compiled program is made up of functions.
- Therefore...you can point at functions.

- Function pointers
  ... are references to memory addresses of functions
  ... are pointer values and can be assigned/passed
  ... are effectively opaque
  ... (unless you're interested in machine code)
  ... ((if you are, you'll enjoy COMP1521))

Function Pointers

`return_t (*var)(arg_t, …)`

`int` → `int`: `int (*fp)(int);`
(`int`, `int`) → `void`: `void (*fp2)(int, int);`

Function Pointers

```
int square (int x)    { return x * x; }
int times_two (int x) { return x * 2; }

int (*fp)(int);

// Take a pointer to the square function, and use it.
fp = &square;
int n = (*fp) (10);

// Taking a pointer works without the `&'.
fp = times_two;
n = (*fp) (2);

// Normal function notation also works.
n = fp (2);
```

Higher-Order Functions

functions that take or return functions

e.g., traverse an array, applying a function to all values.

```
void print_array (size_t len, char *array[])
{
    puts ("[");
    for (size_t i = 0; i < len; i++)
        printf ("%s\n", array[i]);
    puts ("]");
}
```

Higher-Order Functions

functions that take or return functions

e.g., traverse an array, applying a function to all values.

```
void traverse (size_t len, char *xs[], void (*f)(char *))
{
    for (size_t i = 0; i < len; i++)
        (*f) (xs[i]);
}

void print_array (size_t len, char *array[])
{
    puts ("[");
    traverse (len, array, &puts);
    puts ("]");
}
```

Using Higher-Order Functions

```
void traverse (link l, void (*f) (link));

traverse (my_list, print_node);
traverse (my_list, print_grade);
```

```
void print_node (link l)
{
    if (l == NULL)
        puts ("NULL");
    else
        printf ("%d -> ", l->data);
}
```

```
void print_grade (link l)
{
    if (l == NULL)
        puts ("(nil)");
    else if (l->data >= 85)
        printf ("HD ");
    else
        printf ("FL ");
}
```

Linked Lists

Recap: Linked Lists

- a *sequential* collection of ‘nodes’ holding value + pointer(s)
  ...no ‘random access’ to individual nodes
- easy to add, rearrange, remove nodes
- list node references other list nodes
  ...singly-linked list: `next` only
  ...doubly-linked list: `prev` and `next`
- last node’s `next` may point to
  ...`NULL` — no ‘next’ node
  ...a ‘sentinel’ node without a value
  ...the first node (a *circular* linked list)

Recap: Linked Lists in C

```
typedef int Item;

typedef struct node *link;
typedef struct node {
    Item item;
    link next;
} node;

// allocating memory:
link x = malloc (sizeof *x);
link y = malloc (sizeof (node));

// what's wrong with this?
link z = malloc (sizeof (link));
```

Recap: Linked Lists in C

```
// traversing a linked list:
link curr = …;
while (curr != NULL) {
    /* do something */;
    curr = curr->next;
}

// traversing a linked list, for loop edition
for (link curr = …; curr != NULL; curr = curr->next)
    /* do something */;
```

Functions on Linked Lists

Exercise: ‘insert_front’

`link insert_front (link list, link new);`
Write a function to insert a node at the beginning of the list.

Would this prototype work?
`void insert_front (link list, link new);`

Exercise: ‘insert_end’

`link insert_end (link list, link new);`
Write a function to insert a node at the end of the list.

Functions on Linked Lists

Exercise: ‘reverse’

Write a function which reverses the order of the items in a linked list.

```
link reverse (link list) {
    link curr = list;
    link rev = NULL;
    while (curr != NULL) {
        tmp = curr->next;
        curr->next = rev;
        rev = curr;
        curr = tmp;
    }
    return rev;
}
```

Deletion in Linked Lists

Demonstration: ‘delete_item’

```
// Remove a given node from the list
// and return the start of the list
link delete_item (link ls, link n);
```

Deletion in Linked Lists

- deletion is awkward:
  ...we must keep track of the previous node
- can we delete a node if we only have the pointer to the node itself?
- we may need to traverse the whole list to find the predecessor
  ...and that’s if we even have a reference to the head

IDEA every node stores a link to both the previous *and* next nodes

The Doubly-Linked List

- Move forward and backward in such a list
- Delete node in a constant number of steps

```
typedef struct dnode *dlink;
typedef struct dnode {
    Item item;
    dlink prev, next;
} dnode;
```

The Doubly-Linked List

- Deleting nodes:
  easier, more efficient
- Other operations:
  ...pointer to previous node is necessary in many operations
  ...doesn't have to be maintained separately for doubly linked lists
  ...$2\times$ pointer manipulations necessary for most list operations
  ...memory overheads in storing an additional pointer

The Tools of the Trade

Documentation

learn how to access documentation ‘online’:
man(1), *info(1)* – available in exam environment!

you should even learn to *write* documentation:
mdoc, texinfo, doxygen, sphinx
all make it easy to document code and projects
(though are beyond the scope of the course)

man(1)

The Unix Programmer’s Manual

the traditional ‘Unix manual’:
terse documentation in several sections
terrible tutorial, but great reference

commands *(1)*,
syscalls *(2)*,
library functions *(3)*,
file formats *(5)*,
the system *(7)*,
administrative tools *(8)*,
and more…

`man ls` gets *ls(1)*
`man printf` gets *printf(1)*
`man 3 printf` gets *printf(3)*

SOME USEFUL MAN-PAGES
intro in all sections,
stdio.h(0p), *stdlib.h(0p)*, *math.h(0p)*
printf(3), *ascii(7)*

info

GNU’s Online Documentation System

GNU decided *man(1)* wasn’t good enough
(a bundle of loose documents ≠ a good manual…)
so built the Texinfo system

SOME USEFUL INFO MANUALS
libc, *gdb*, *gcc*,
binutils, *coreutils*,
emacs, …

the *info(1)* command
will fall back to *man(1)*-pages

other renderings of info pages:
dead trees, PDFs, web sites …

Code, Compile, Crash, Confusion

Debugging in the Software Development Cycle

what's happening in your program as it runs?
why did that segfault happen?
what values are changing in my program?

"I'll just add some *printf(3)*s..."
clunky, not reliable, only gives what you ask for

a family of tools can help you find out:
debuggers

source debuggers: gdb/ddd/gud, lldb, mdb
specialist tools: valgrind, sanitizers

gdb

The Breaking-Point

`set args` *`args`*
    set command arguments

`run` *`args`*
    run the program under test

`break` *`expr`*
    set a breakpoint

`watch` *`expr`*
    set a watch expression

`continue`
    run the program under test

`print` *`expr`*
    print out an expression

`info locals`
    print out all local variables

`next`
    run to the next line of code

`step`
    step into a line of code

`quit`
    exit gdb

NOTE
you'll need to compile with -g
or GDB is very unfriendly indeed

Sanitizers

Programmers Must Wash Hands Before Returning To Codebase

{Address, Leak, Memory, Thread, DataFlow, UndefinedBehaviour}Sanitizer

a family of compiler plugins, developed by Google
which instrument executing code with sanity checks
use-after-free, array overruns, value overflows, uninitialised values, and more

you've been using ASan+UBSan already: *dcc* uses them!
usable on your own *nix systems (Linuxes, BSDs, 'macOS') too!
unfortunately... a bit of work to get going on CSE (hence *dcc* and *3c*)

```
clang -fsanitize=address,undefined -fno-omit-frame-pointer
    -g -m32 -target i386-pc-linux-gnu --rtlib=compiler-rt -lgcc -lgcc_s
        -o prog main.c f2.c
```

```
2521 3c -o prog main.c f2.c
```

valgrind

Have You Forgotten Something?

- finding memory leaks
  … not `free`’ing memory that you `malloc`’d
- finding memory errors
  … illegally trying access memory

```
$ valgrind ./prog
...
==29601== HEAP SUMMARY:
==29601==     in use at exit: 64 bytes in 1 blocks
==29601==   total heap usage: 1 allocs, 0 frees, 64 bytes allocated
==29601==
==29601== LEAK SUMMARY:
==29601==    definitely lost: 64 bytes in 1 blocks
```

Valgrind doesn’t play well with ASan. Compile without ‘3c’ if you really need it.

make

Making Everything Better

long, intricate compilation lines?
forgot to recompile parts of your code?

make lets you specify
rules, *dependencies*, *variables*
to define what a program needs to be compiled
doing only the necessary amount of work

implicit rules for compiling C (and more)
(`.c` → `.o`, `.o` → exec)

make

A Simple Example

```
CC       = gcc
CFLAGS   = -Wall -Werror -std=c99 -g
LDFLAGS  = -g -lm

# `prog' depends on `prog.o', `ADT.o'
prog: prog.o ADT.o
# `prog.o' depends on `prog.c', `ADT.h'
prog.o: prog.c ADT.h
# `ADT.o' depends on `ADT.c', `ADT.h'
ADT.o: ADT.c ADT.h
        ${CC} ${CFLAGS} -std=gnu11 -c $< -o $@
```