What did we cover yesterday?

Exam

- Exam format
- Difficulty of Questions
- How to approach it

Course Recap Part 1

- The earlier parts of the course
What are we covering today?

Course Recap Part 2

- The non-technical part of the course
- The second half of the course (all the spikey bits)
Programming is much more than just code

COMP1511 Programming Skills Topics

- History of Computing
- Problem Solving
- Code Style
- Code Reviews
- Debugging
- Theory of a Computer
- Professionalism
Problem Solving

Programmers While Coding
It Doesn’t Work....... Why?

It Work....... Why?
Problem Solving

Approach Problems with a plan!

- Big problems are usually collections of small problems
- Find ways to break things down into parts
- Complete the ones you can do easily
- Test things in parts before moving on to other parts
Code Style

Half the code is for machines, the other half for humans

- Remember . . . readability == efficiency
- Also super important for working in teams
- It's much easier to isolate problems in code that you fully understand
- It's much easier to get help if someone can skim read your code and understand it
- It's much easier to modify code if it's written to a good style
Code Reviews

No one has to work without help

- If we read each other's code . . .
- We learn more
- We help each other
- We see new ways of approaching things
- We are able to teach (which is a great way to cement knowledge)
Debugging

[diːˈbʌɡɪŋ]

1. Being the detective in a crime movie where you are also the murderer.
Debugging

The removal of bugs (programming errors)

- Syntax errors are code language errors
- Logical errors are the code not doing what we intend
- The first step is always: Get more information!
- Once you know exactly what your program is doing around a bug, it's easier to fix it
- Separate things into their parts to isolate where an error is
- Always try to remember what your intentions are for your code rather than getting bogged down
Professionalism

There's so much more to computing than code

- What's the most important thing for a Software Professional?
- It's not coding!
- It's caring about what you do and the people around you!
- Even in terms of pure productivity, it's going to get more work done long term than being good at programming
- If you care about your work, you will be fulfilled by it
- If you care about your coworkers you'll teach and learn from them and you'll all grow into a great team
Break Time

A thought exercise . . . the future

● Why are you doing computer science (or related field)?
● Is there something you'd like to do with these skills?
  ○ Jobs?
  ○ Research?
  ○ Change the World?
● How do you want to use your time at UNSW to push yourself towards your goals?
● Note: You don't need all the answers yet, but it's useful to start thinking about these things!
Characters and Strings

Used to represent letters and words

- **char** is an 8 bit integer that allows us to encode characters
- Uses ASCII encoding (but we don't need to know ASCII to use them)

- Strings are arrays of characters
- The array is usually declared larger than it needs to be
- The word inside is ended by a Null Terminator `\0`
- Using C library functions can make working with strings easier
Characters and Strings in code

// read user input
char input[MAX_LENGTH];
fgets(input, MAX_LENGTH, stdin);
printf("%s\n", input);

// print string vertically
int i = 0;
while (input[i] != '\0') {
    printf("%c\n", input[i]);
    i++;
}
Structures

Custom built types made up of other types

- structs are declared before use
- They can contain any other types (including other structs and arrays)
- We use a . operator to access fields they contain
- If we have a pointer to a struct, we use -> to access fields
struct spaceship {
    char name[MAX_NAME_LENGTH];
    int engines;
    int wings;
};

int main (void) {
    struct spaceship xwing;
    strcpy(xwing.name, "Red 5");
    xwing.engines = 4;
    xwing.wings = 4;

    struct spaceship *myShip = &xwing;

    // my ship takes a hit
    myShip->engines--;
    myShip->wings--;
}
Our programs are stored in the computer's memory while they run

- All our code will be in memory
- All our variables also
- Variables declared inside a set of curly braces will only last until those braces close (*what goes on inside curly braces stays inside curly braces*)
- If we want some memory to last longer than the function, we allocate it
- `malloc()` and `free()` allow us to allocate and free memory
- `sizeof` provides an exact size in bytes so `malloc` knows how much we need
struct spaceship {
    char name[MAX_NAME_LENGTH];
    int engines;
    int wings;
};

int main (void) {
    struct spaceship *myShip = malloc(sizeof (struct spaceship));
    strcpy(myShip->name, "Millennium Falcon");
    myShip->engines = 1;
    myShip->wings = 0;

    // Lost my ship in a Sabacc game, free its memory
    free(myShip);
}
Linked Lists

Structs for nodes that contain pointers to the same struct

- Nodes can point to each other in a chain to form a linked list
- Convenient because:
  - They're not a fixed size (can grow or shrink)
  - Elements can be inserted or removed easily anywhere in the list
- The nodes may be in separate parts of memory
Linked Lists

Function: Removes first item from a linked list.

Second Item:
struct location {
    char name[MAX_NAME_LENGTH];
    struct location *next;
};

int main (void) {
    struct location *head = NULL;
    head = addNode("Tatooine", head);
    head = addNode("Yavin IV", head);
}

// Add a node to the start of a list and return the new head
struct location *addNode(char *name, struct location *list) {
    struct location *newNode = malloc(sizeof(struct location));
    strcpy(newNode->name, name);
    newNode->next = list;
    return newNode;
}
Complications in Pointers, Structs and Memory

What's a pointer?

- It is a number variable that stores a memory address
- Any changes made to pointers will only change where they're aiming

What does * do?

- It allows us to access the memory that the pointer aims at (like following the address to the actual location)
- This is called "dereferencing" (because the pointer is a reference to something)
Complications in Pointers, Structs and Memory

What about -> ?

- Specifically access a struct at the end of a pointer
- -> must point at one of the fields in the struct that the pointer aims at
- It will dereference the pointer AND access the field

Pointers to structs that contain pointers to other structs!

- We can follow chains of pointers like track->beat->note
int main (void) {
// create a list with two locations
struct location *head = addNode("Dantooine", NULL);
head = addNode("Alderaan", head);

// create a pointer to the first location
struct location *alderaan = head;

// set head to a newly created location
head = malloc(sizeof(struct location));

// What has happened to the alderaan pointer now?
// What has happened to the variable that the head and alderaan
// both pointed at?
}
Create a linked list of two locations with a head pointer aimed at the first location.

`head`:

```
Alderaan -> Dantooine -> NULL
```

Where:
- **Alderaan** is the first location in the list.
- **Dantooine** is the second location in the list.
- **NULL** is the end of the list.

This diagram illustrates the concept of pointer arithmetic in programming, where the `Next` pointer points to the next location in the list.
**Pointer Arithmetic**

A program's memory (not to scale)

```c
struct location *alderaan = head
```

This line creates a new pointer that's a copy of the head pointer. It is given the same value as head, which means it's aimed at the same memory address.
head = malloc((sizeof(struct location));
This line allocates new memory and assigns the address of
this new allocation to the head pointer.
Changing head doesn't change anything it was pointing at!
Keeping track of pointers

`beat->head->next->next->note = ???`

- This is code that might work in most Beats by CSE implementations
- **Remember:**
  - Changing a pointer changes its value, a memory address
  - Changing a pointer will change where it's aiming, nothing more!
  - Once you use `->` on a pointer, you're now looking at a struct field
  - This means you are not changing that pointer, you have dereferenced it and accessed a field inside the struct
Abstract Data Types

Me: calls stack.pop()
Item at the top of the stack:
Abstract Data Types

Separating Declared Functionality from the Implementation

- Functionality declared in a Header File
- Implementation in a C file
- This allows us to hide the Implementation
- It protects the raw data from incorrect access
- It also simplifies the interface when we just use provided functions
// ship type hides the struct that it is
// implemented as
typedef struct shipInternals *Ship;

// functions to create and destroy ships
Ship shipCreate(char* name);
void shipFree(Ship ship);

// set off on a voyage of discovery
Ship voyage(Ship ship, int years);
// ship type hides the struct that it is implemented as
struct shipInternals {
    char name[MAX_NAME_LENGTH];
};

Ship shipCreate(char* name) {
    Ship newShip = malloc(sizeof (struct shipInternals));
    return newShip
}
void shipFree(Ship ship) {
    free(ship);
}

// set off on a voyage of discovery
Ship voyage(Ship ship, int years) {
    int discoveries = 0, yearsPast = 0;
    while(yearsPast < years) {
        discoveries++;
    }
}
Abstract Data Types Main

- Including the Header allows us access to the functions
- The main doesn't know how they're implemented
- We can just trust that the functions do what they say

```c
#include "ship.h"

int main (void) {
    Ship myShip = newShip("Enterprise");
    myShip = voyage(myShip, 5);
}
```
So, you're programming now...

NOT SURE IF I AM A GOOD PROGRAMMING

OR GOOD AT GOOGLING
So, you're programming now …

Where do we go from here?

- There's so much you can do with code now
- But there's also so much to learn
- Programming has more to offer than anyone can learn in a lifetime
- There's always something new you can discover
- It's up to you to decide what you want from it and how much of your life you want to commit to it
- Remember to care for yourselves and your work
- Enjoy yourselves, keep working as hard as you can and I hope to bask in your future glory
Good luck, have fun :)