31251 – Data Structures and Algorithms

Week 1

Luke Mathieson

What’s Data Structures and Algorithms about?

What's an Algorithm?

- A *well defined* computational process that takes some *input* and produces an *output*.
- Really it's a tool for solving a *well specified* computational problem.
- You don't actually need a computer for them though.

What is a Data Structure?

- A way to store and organise data.
- Really just a special type of algorithm.
    - (But calling the subject “Algorithms and some special types of algorithms” isn’t as catchy.)

But you've seen this before!

- All the little patterns you learnt in App. Prog. are algorithms or templates for algorithms.
- All the code you've written was made up of algorithms (at least the bits that worked).
- What we're really doing in this subject is starting to build a formal awareness of algorithms as separate from computer programs.

Some Adminstrative Matters

The Learning Support System

- Ed!
  - Kind of like a mix between PLATE and UTSOnline.
  - Gives better support for first time C++ programmers.
  - Can avoid wrangling with CxxTest. (But if you want to play with it...)
  - Also means we have seamless support for up-to-date C++.

Lecture Recording

- Both lectures will likely be recorded.
- As it's me doing the whole thing, this may be a bit rough to begin with.
- I will still make supplementary videos as necessary.

Other Important Things

- Assessment:
  1. 3 short multiple choice quizzes.
  2. 2 programming assignments.
  3. 1 *open book* exam (multiple choice and short answer).

U:PASS

- U:PASS runs for this subject.
  - Get extra help from a student who did well in the subject.
  - Attending U:PASS correlates with improved grades (even better when you attend regularly).
  - Places are limited, but the more people attend, the more slots they can open.
  - See `http://www.tinyurl.com/upass2018` to sign up.

“Office” Hours

- 2-4pm every Thursday starting Week 2.
- In the learning precinct. (Building 11, Level 5, right up the Wattle Street end)

Plagiarism & Misconduct

A little advice...

Now Some C++

And now for something complete different.

Today's topics:

- Strings (briefly)
- Arrays, or "who though this was a good idea?"
- Pointers
  - References vs Pointers
  - Dereferencing a pointer
- Classes
- Headers and Source files, or "at least this isn't as bad as arrays"
- Lists and Linked Lists! (Yay, an actual data structure)

Strings in C++

- Strings in C are just null terminated `char` arrays.
  - Aside: ... what is null in C++?
  - Mostly just `0`, or something that looks like it (yay C).
  - Since C++11, an actual `null_ptr` type exists, so you can have a proper null that isn't just `0`.
- Where could that possible go wrong?
  - What if you forget the null?
  - What if you want to know the length?

Strings in C++

C++ has a proper `string` class (`std::string`) that conceptually wraps a `char[]` and fixes these problems: `http://www.cplusplus.com/reference/string/string/`

Arrays in C++

- Arrays in C++ look a lot like Java arrays:
  - `int a[4] = {1,2,3,4};`
  - `int a[] = {1,2,3,4};`
  - `int a[4] = {};`
  - `int a[4];`
- Note that these are all statically created.
- ..huh? What does that mean?

Static and Dynamic Allocation

- C++ has a more complex allocation system than Java (at least from the programmer's perspective).
- Things can be statically allocated:
  - They are automatically deallocated when they go out of scope.
  - What does this mean for return data?
- Or dynamically allocated:
  - Created on the heap with the `new` keyword.
  - C++ has no garbage collection, so you have to manage it yourself.
- Short version, don't use `new` unless you mean it!

Arrays in C++

- In all the previous array examples, the size was known at declaration.
- The program does its own memory management – you need to know the size!
- What if we don't know the size?
- Arrays decay to pointers to the first element.
  - So an `int[]` can be treated as a `int*`.
  - Wait, what's a pointer?

Pointers!

- Pointers are what make C++ programming annoying!
- Actually they're not so bad, they're just variables that tell you where something is in memory.
- *i.e.* they *point* to something.

Pointers!

- To create a pointer to type `t`:
  - `t * foo;`
  - The spaces around the `*` don't matter (*i.e* `t* foo`, `t * foo` and `t *foo` are all the same).
- A pointer is really just number that is the address of whatever it's pointing at.
- To get what it's pointing at we *dereference it*:
  - `t bar = *foo;`
  - If you're derefencing to get a member `(*foo).bar`, you can write the alternative `foo->bar`. This can be nicer in many situations.
- To get the address of something, use the address operator:

  -
    ```
    int foo = 5;
    int * bar = &foo;
    ```

References!

- C++ also has references.
- References are like pointers, but:
  - They can't change *where* they're pointing after initialisation.
  - They're transparently dereferenced:
- They're created with the `&` operator:
  - `int & foo = ...;`
- But then they work like the thing at the other end:
  - `foo = foo + 5` does what you'd expect (what would it do to a pointer?).
- References are good for passing data around without copying it (this should be familiar from Java – it does essentially the same thing).

Back to Arrays

- So if we want to create an array where we don't know the size (*e.g.* as a parameter or return type), we need a pointer:
  - `int * tabulate(Data dataObject)...`
- But how do we know that we're getting an array?
- We don't!
- Well... that's not so good... how do we fix it?
- `std::vector`!

Classes

Classes in C++ look a lot like Java classes:

```
#include <string>

using std::string;

class myClass :  public parentClass {

  private:

    int privateInt;

  public:

    int getPrivateInt();
    void setPrivateInt(int newValue);
    string toString();
};
```

Classes

- Notice that the methods have no content there.
- They can, but they don't have to.
- C++ routinely separates definition from source code.
    - It expects a single pass compiler, so you have to have all the names in the right order!
- Typically definitions are put in *header* files (usually with a `.h` extension, but not necessarily).
- Source code is normally in source files (usually `.cpp`, but again, that can change).
- Sometimes code is put in the header file (sometimes it even makes sense to do so!).

Header Files

- So what do we do with header files if they have no code?
  - Declare things in the right order for `#includes`.
  - Create the equivalent of interfaces (virtual classes!)

A Data Structure!

A Data Structure

- We now have almost enough to build our first data structure!
- But first: Abstract Data Types
    - ADTs are specifications of behaviour of Data Types.
    - They don't specify *implementations*.
    - Adhereing to an ADT allows us to code without having to know implementation details (good for teams, reusability and modularity).
    - In Java, we'd achieve this with an `Interface` and abstraction.

The `List` ADT

- A list stores data in a sequential order.
- So what methods should a list have?
    - Something to check if it's empty?
    - Something to add to the front of the list?
    - Something to add to the end?
    - Something to get the first element?
    - Something to get the rest of the list?
- We should be able to manage that!

An intList Virtual Class

```
class intList {
public:
  virtual ~intList() {};
  virtual bool isEmpty() = 0;
  virtual void prepend(int c) = 0;
  virtual void append(int c) = 0;
  virtual int head() = 0;
  virtual intList tail() = 0;
};
```

Now the implementation (of some of it)

See C++ files.