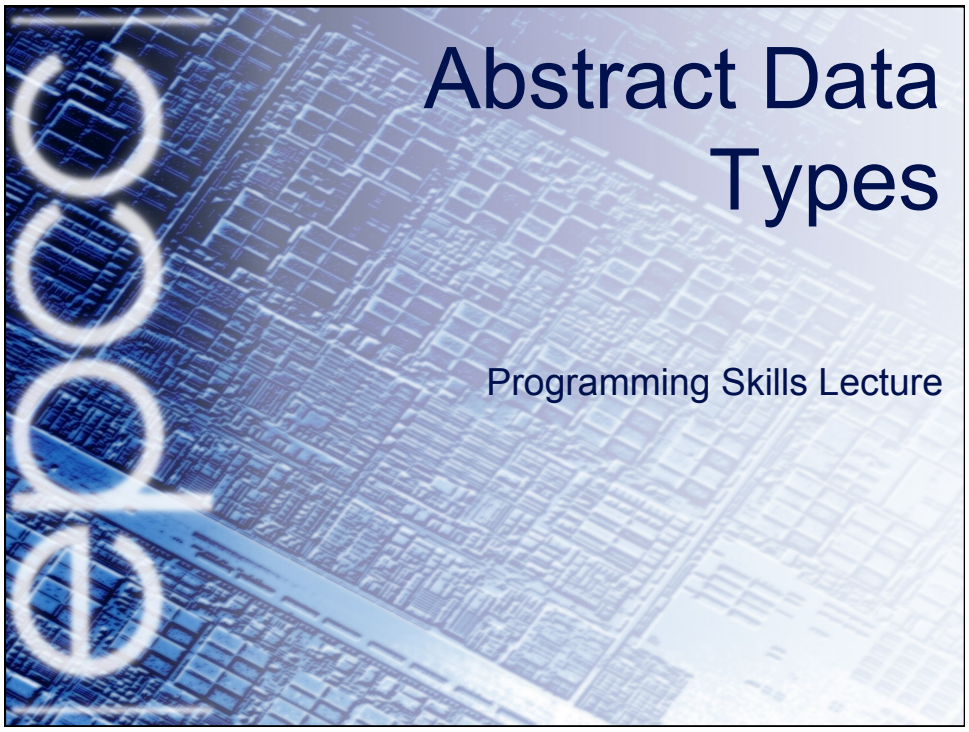


Overview

- What is an Abstract Data Type
- Why Do We Care
- Defining an ADT
- Using an ADT
- Isn't this all rather familiar?
- Next Time

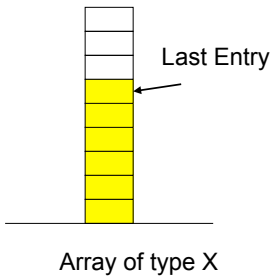
Example: Stack

- A Stack is a LIFO structure
  - LIFO is Last In First Out
  - Think of it like a piles of plates at a buffet
  - New plates are put on the top of the pile
  - Diners take a plate from the top
  - Try and take a plate from anywhere else?

- Uses include expression evaluation, parsing, backtracking algorithms
- What operations does a stack need?

Example Stack

- Could just use an array and track where the last entry is:
- Add to stack
  - Get last entry index
  - Add one to index
  - Put value in place
- What could go wrong if each time this is done it is up to each individual programmer?



What is an ADT?

- An method of encapsulating a set of data items and operations on those items.
- Exposes operations to a programmer via an interface
- Data and operation implementations are “hidden”

Advantages of ADTs

- Encapsulation
  - Remove the need for a user of a ADT to know how it implements its data or operations
  - Only need to know what it is expected to do
  - Low coupling - data is hidden behind interface and other parts of
  - the program can only use it via its interface
  - High cohesion – in this example: each stack-related function performs a well-defined task on the stack
- Change Localisation
  - Changes inside the ADT will not affect behaviour outside the ADT if and only if the specification of behaviour is adhered to
- Flexibility
  - Different Implementations suit different applications or environments
  - Related to Encapsulation

Problems of ADTs

- Can seem convoluted
  - A lot of hoops to jump through initially when designing
- Specifications not adhered to
  - Not just an ADT issue
- Information leaking
  - Often ADTs will leak information about how they are implemented and developers will latch onto this reducing the encapsulation and increasing coupling with other parts of the program
- Testing
  - Sometimes testing can be complicated when third party ADTs are involved and have bugs or erratic behaviours
  - Though this could be said about all testing

Why do We Care

- Reasons, can you come up with any?
- Think about it and we will come back to it

Defining an ADT

- Create an abstract specification
  - Don't mention any implementation details
    - Don't say 'get element of array at index I'
  - Identify any invariants that must hold about the representation
    - In a set -> No duplicates
    - Representation invariants should hold after initialisation and after any operations
    - Note: An invariant is any condition which can be relied upon to be true during program execution – so a set has no duplicates is an example. Invariants can be parts of loops, data structure or methods.
  - Don't allow a user to get at the underlying structure
    - Don't return your data structure as a changeable object (see above points)

Stack Operations

- Essential
  - Pop – retrieve top element of stack
  - Push – put new element at top of stack
  - Create
  - Delete
- Useful
  - Size
  - Peek – look at the top element but don't remove it
  - IsEmpty

Writing a Spec

- To write a spec for an ADT operation consider the following template:
- Pre-conditions
- Post-conditions
- Inputs
- Outputs
- Algorithm (in general terms)

Example: CreateStack

- Pre-conditions: none
- Post-conditions: Stack S is defined, initialised and empty
- Inputs: none
- Outputs: Stack S
- Algorithm: Create Stack data objects and returns stack S

- Now choose operations from the stack ADT and write a specification

An Example C Version: Header

```
/* This module implements a stack of characters. */
#include <stdlib.h>  /* Needed for malloc etc. */
#include <stdio.h>   /* Needed for printf in stackPrint. */

/* Private data. */
struct Stack {
  char* stk;       /* Array of characters forming the stack. */
  int   top;       /* Index of the topmost character in the stack. */
  int   size;      /* Size of the array. */
};

/* Public interface. */
struct Stack* stackNew         (void);
void          stackDelete      (struct Stack* s);
void          stackPush        (struct Stack* s, char c);
char          stackPop         (struct Stack* s);
void          stackPrint       (struct Stack* s);
void          stackPrintBasic  (struct Stack* s);
void          stackPrintReverse (struct Stack* s);

/* Private interface. */
/* Declaring this as 'static' means that it cannot be called from outside */
/* the file in which it is defined (ie. outside Stack.c).  So it is      */
/* 'private' to the functions of Stack. */
static void   stackGrow(struct Stack*);
```

Using an ADT

- We have an ADT implementation that adheres to your specification
- We don't bother about how it implements the operations only that it does so correctly
- How do we use it?
- Like any other library or module in the language of our choice
  - Java, Python, C, etc all use ADTs at some level so you will have used them

An Example

- Suppose you have an ADT and implementation
- How do you check that the implementation is valid?
- Answer: Test it against what you know it should do
- How could we test the stack ADT?

Test the Stack

- Can take the operations and write individual and combined tests for them all.
- Example (in pseudocode) TestPop:
  - Create stack
  - Push characters ‘ABCDEF’
  - While expecting characters
    - Pop top of stack and compare to expected character
    - Note that characters should come out ‘FEDCBA’
    - If not expected then fail
  - Check stack is empty -> if not then fail
  - Delete stack
- Write a sample test pseudocode for your operation

Why do We Care?

- Removes quick hacks
- Reduces the chances of bad dependencies being introduced
- Allows for implementation changes without affecting large swathes of code
- Protects data
- Helps developers

Why Do We Care?

- Because we already use them
- Can you identify some ADTs you use?
- Floats, Stacks, Lists, Queues, Graphs, Maps, Integers
- But aren't these classes?
- In some languages yes, in others no, but all are based round a specification of operations

Isn't this all rather familiar?

- What does all this remind you of?
- Object Orientation maybe?

It is and it isn't

- ADTs can be used in OO Programming – many primitives and common structures are implemented using ADTs
  - Examples: Floating points, Stacks
- ADTs do/may not support everything that OO does
  - ADTs may not support polymorphism or inheritance
- ADT Languages which aren't OO are often called Object-Based
- ADTs are characterised by the operations, OO is characterised by the data and observations
- Different Types of Abstraction

Next Time

- Object Orientated Programming
- A different but complementary technique
- Commonly found in a range of languages
  - C++
  - Java
  - Python