COMP2521 19T0

Week 1, Thursday: Abstraction, Your Honour

Jashank Jeremy

jashank.jeremy@unsw.edu.au

abstract data types, redux
fundamental data structures
testing

IMPORTANT

UNSW will have rolling short network outages
from **6am Sat 1 Dec** to **6pm Sun 2 Dec**.
save your work often if you're using VLAB!
CSE workstations may be affected.

Abstract Data Types

Abstraction

Meaning and Mechanism

"...the purpose of abstracting is not to be vague,
but to create a new semantic level in which
one can be absolutely precise."

— from *The Humble Programmer*
by E. W. Dijkstra (EWD 340), 1972

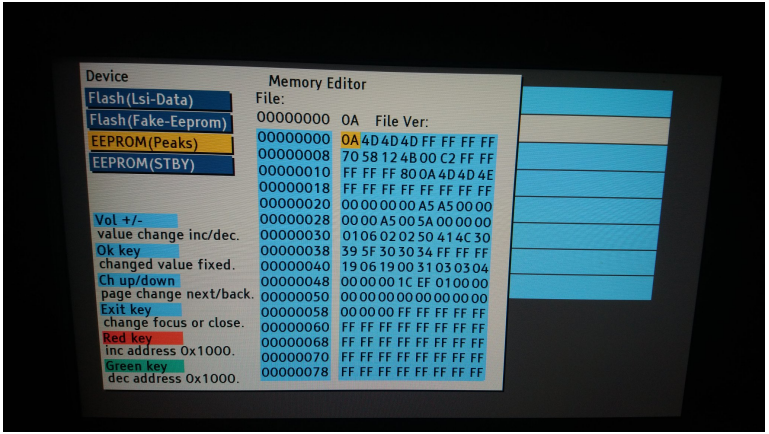
distinguish meaning and mechanism
...don't lose the forest for the trees

Abstraction

Abstraction in systems

To understand a system,
it should be enough to
understand what its components do
without knowing how...

e.g., we operate a television through its interface:
a remote control, and an on-screen display
... we do not need to open it up
and manipulate its innards



Good news: my parents TV has a hex editor
Bad news: I'm buying them a new tv

@0x47DF 2018-10-06 2142z

twitter.com/0x47DF/status/1048342591668965377

Data Types

Values and Operations

a set of values —

PRIMITIVE

`int`
(`char`, `short`, `long`, `long long`),
`float`
(`double`, longer!),
`void*`

COMPOSITE

`struct` *T*,
`enum` *T*,
`union` *T*

operations on those values —

`->`, `.`, `!`, `~`, `++`, `--` $+_2$, $-_2$, $*_1$, $\&_1$, $*_1$, `/`, `%`, $+_1$, $-_1$,
`<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, $\&_2$, `^`, `|`, `&&`, `||`, `?:`,
`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`

Abstraction

Abstraction in the Machine

When designing a new library,
it is important to decide...

what are the abstract properties
of the data types we want to provide?

which operations do we need to
create, query, manipulate, destroy
objects of these types?

FOR EXAMPLE...
we do not need to know how
`FILE *` is implemented to use it

Abstraction

Creating Abstractions

We want to distinguish:

- DT (non-abstract) data type (e.g. C strings)
- ADO abstract data object
- ADT abstract data type (e.g., C strings)
- GADT generic abstract data type

ACHTUNG!
ADTs are not algebraic data types!
see COMP3141 / COMP3161 for more

ACHTUNG!
Sedgewick's first few examples
are ADOs, not ADTs!

Abstract Data Types

facilitate decomposition, encapsulation of complex programs

make implementation changes invisible to clients

improve readability and structuring of software

Abstract Data Types

Interface and Implementation

ADT interfaces provide

- an opaque view of a data structure
- function signatures for all operations
- semantics of operations (via documentation, proof, etc.)
- a contract between ADT and clients

ADT implementations provide

- concrete definition of the data structures
- function implementations for all operations

ADTs in C

Interfaces

- an opaque view of a data structure
  - ... via `typedef struct t *T`
  - ... we do not define a concrete `struct t`
- function signatures for all operations
  - ... via C function prototypes
- semantics of operations (via documentation, proof, etc.)
  - ... via comments (e.g., Doxygen)
  - ... via testing frameworks (e.g., ATF-C)

ADTs in C

Implementations

- concrete definition of the data structures
  ... the actual `struct t` (and anything it needs)
- function implementations for all operations
  ... interface and internal functions

Stacks and Queues

Stacks and queues are

- … ubiquitous in computing!
- … part of many important algorithms
- … good illustrations of ADT benefits

Stacks

A stack is a collection of items, such that the last item to enter is the first item to leave:

Last In, First Out (LIFO)

(Think stacks of books, plates, etc.)

- Web browser history
- text editor undo/redo
- balanced bracket checking
- HTML tag matching
- RPN calculators
  (...and programming languages!)
- function calls

Stacks

Essential Operations

$\textsc{push} :: \mathcal{S} \to \texttt{Item} \to \texttt{void}$
add a new item to the top of stack $\mathcal{S}$

$\textsc{pop} :: \mathcal{S} \to \texttt{Item}$
remove the topmost item from stack $\mathcal{S}$

Stacks

Additional Operations

$\textsc{size} :: \mathcal{S} \to \texttt{size\_t}$
return the number of items in stack $\mathcal{S}$

$\textsc{peek} :: \mathcal{S} \to \texttt{Item}$
get the topmost item on stack $\mathcal{S}$, without removing it

a constructor and a destructor
to create a new empty stack, and
to release all resources of a stack

Stack ADTs

<stack.h>

```
typedef struct stack *Stack;

/** Create a new, empty Stack. */
Stack stack_new (void s);

/** Destroy a Stack, releasing its resources. */
void stack_drop (Stack s);

/** Add an item to the top of a Stack. */
void stack_push (Stack s, Item it);

/** Remove an item from the top of a Stack. */
Item stack_pop (Stack s);

/** Get the number of items in a Stack. */
size_t stack_size (Stack s);
```

Stack ADTs

An Implementation using Arrays

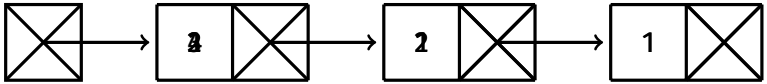
- Allocate an array with a maximum number of elements
  - … some predefined fixed size
  - … dynamically grown/shrunk using *realloc(3)*
- Fill items sequentially — `s[0]`, `s[1]`, …
- Maintain a counter of the number of pushed items

NEW PUSH (1) PUSH (2) PUSH (3) POP $\Rightarrow 3$ PUSH (4)

Stack ADTs

An Implementation using Linked Lists

- Add node to the front of the list on push
- Take node from the front of the list on pop



NEW PUSH (1) PUSH (2) PUSH (3) POP $\Rightarrow 3$ PUSH (4)

Stack ADTs

A Stack Client: Balancing Brackets

Sample input: `( [ { } ] )`

char	stack	check
		-
(	(	-
[	([	-
{	([{	-
}	([	{ = }
]	(	[=]
)		(=)
EOF		is empty

Stack ADTs

Sidenote: ‘Prefix’, ‘Infix’, ‘Postfix’

$$2 + 3$$
infix

$$+\ 2\ 3$$
prefix

$$2\ 3\ +$$
postfix

Many programming languages use infix operations.
Some (like Lisp) use prefix operations.
Some (like Forth, PostScript, *dc(1)*) use postfix operations.

Stack ADTs

A Stack Client: a Postfix Expression Evaluator

Given an expression in postfix notation, return its value.

```
$ ./derpcalc "5 9 1 + 4 6 * * 2 + *"
1210
$ ./derpcalc "1 5 9 - 4 + *"
0
```

Stack ADTs

A Stack Client: a Postfix Expression Evaluator

- We use a stack!
- When we encounter a number:
  1. push it!
- When we encounter an operator:
  1. pop the two topmost numbers
  2. apply the operator to those numbers
  3. push the result back onto the stack
- At the end of input:
  1. print the last item on the stack

Queues

A queue is a collection of items, such that the first item to enter is the first item to leave:

First In, First Out (FIFO)

(Think queues of people, etc.)

- waiting lists
- call centres
- access to shared resources (e.g., printers)
- processes in a computer

Queues

Operations

ENQUEUE :: $\mathcal{Q} \to$ `Item` $\to$ `void`
add a new item to the end of queue $\mathcal{Q}$

DEQUEUE :: $\mathcal{Q} \to$ `Item`
remove the item at the front of queue $\mathcal{Q}$

SIZE :: $\mathcal{Q} \to$ `size_t`
return the number of items in queue $\mathcal{Q}$

PEEK :: $\mathcal{Q} \to$ `Item`
get the frontmost item of queue $\mathcal{Q}$, without removing it

a constructor and a destructor
to create a new empty queue, and
to release all resources of a queue

Queue ADTs

<queue.h>

```
typedef struct queue *Queue;

/** Create a new, empty Queue. */
Queue queue_new (void q);

/** Destroy a Queue, releasing its resources. */
void queue_drop (Queue q);

/** Add an item to the front of a Queue. */
void queue_en (Queue q, Item it);

/** Remove an item from the end of a Queue. */
Item queue_de (Queue q);

/** Get the number of items in a Queue. */
size_t queue_size (Queue q);
```

Queue ADTs

An Implementation using Linked Lists

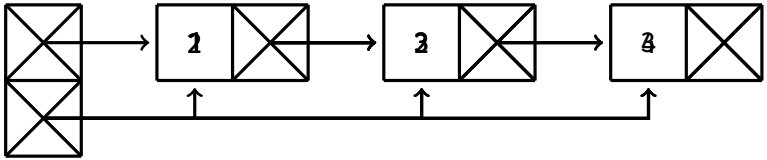
We need to add and remove items from opposite ends now!
We woulde either add or remove from the tail.
Can we do this efficiently? What do we need?

- If we only have a pointer to the head, **no**!
  We'd need to traverse the list to the tail every time.
- If we have a pointer to both head *and* tail,
  we don't have to traverse, and *adding* is efficient.
  (But not removing … why?)

Queue ADTs

An Implementation using Linked Lists

Add nodes to the end; take nodes from the front.

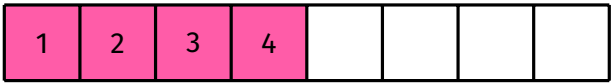


NEW $\quad$ ENQ (1) $\quad$ ENQ (2) $\quad$ ENQ (3) $\quad$ DEQ $\Rightarrow 1$ $\quad$ ENQ (4)

Queue ADTs

An Implementation using Arrays

- Allocate an array with a maximum number of elements
  - … some predefined fixed size
  - … dynamically grown/shrunk using *realloc(3)*
- Maintain an index for the front and back of the queue
- Maintain a counter of the number of items



NEW $\quad$ ENQ (1) $\quad$ ENQ (2) $\quad$ ENQ (3) $\quad$ DEQ $\Rightarrow 1$ $\quad$ ENQ (4)

Analysis and Testing

Analysis of Software

In COMP1911/1917/1511/1921,
the focus was on building software
(with unit testing for ‘quality control’)

In COMP2521, we focus more on analysis.
… which implies we have something to analyse.

Analysis of Software

Empirical vs Theoretical

Lots of the analysis we will do is
empirical, executing and measuring, or
theoretical, proving and deriving.

(We'll only be using proof-by-hand-waving…
COMP2111, COMP3141, COMP3153, COMP4141, COMP4161
go into formal methods in *much* more depth!)

Analysis of Software

What makes software ‘good’?

correctness	returns expected result for all valid inputs
robustness	behaves ‘sensibly’ for non-valid inputs
efficiency	returns results reasonably quickly (even for large inputs)
clarity	clear code, easy to maintain/modify
consistency	interface is clear and consistent (API or GUI)

In this course, we’re interested in correctness and efficiency.

A Moment of Robustness

Postel's robustness principle:

Be conservative in what you do;
be liberal in what you accept from others

"defensive" programming

Analysing Effectiveness

We have two ways to determine effectiveness:

- empirical: **testing**, via program execution
  - devise a comprehensive set of test cases
  - compare actual results to expected results
- theoretical: **proof** of program correctness
  - define pre-conditions and post-conditions
  - establish that code maps from pre- to post-
  - (very loosely, Hoare logic)

Empirical vs Theoretical

For example: finding the maximum value in an unsorted array:

```
max = a[0];
for (i = 1; i < N; i++)
    if (a[i] > max) max = a[i];
```

What test cases should we use?

- max value is first, last, middle, ...
- values are positive, negative, mixed, same, ...

What are our pre- and post-conditions?

- pre: $\forall \texttt{j} \in [0 \cdots N-1], \text{defined}(\texttt{a[j]})$
- post: $\forall \texttt{j} \in [0 \cdots N-1], \texttt{max} \geq \texttt{a[j]}$

Empirical vs Analytical

Testing increases our confidence in correctness ...
better chosen test cases $\Rightarrow$ higher confidence
more thorough test cases $\Rightarrow$ higher confidence
...but cannot, in general, guarantee it!

Verification guarantees correctness:
any valid input will give a correct result,
but there's gaps; *e.g.*, how are invalid inputs treated?
(unless invalid input classes are included in pre-/post-conditions)

Dijkstra on Testing

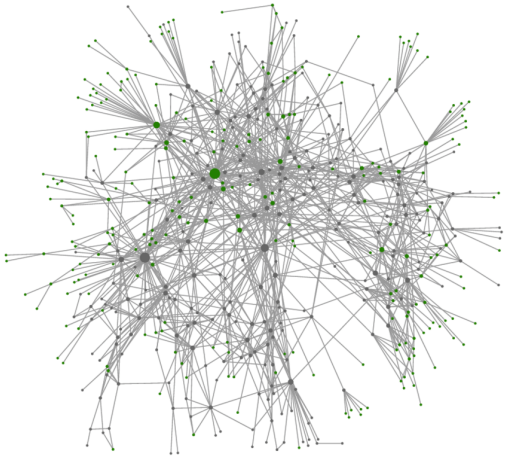
"Program testing can be used
to show the presence of bugs,
but never to show their absence!"

— from *Notes on Structured Programming*
by E. W. Dijkstra (EWD 249), April 1970

Software Verification

‘seL4: Formal Verification of an OS Kernel’, 2009
G. Klein, K. Elphinstone, G. Heiser *et al*;
UNSW/NICTA (now Data61 at CSIRO)

~9 kLoC C … ~55 kLoP, ~11 py



Testing Approaches

The “Big Bang” approach

The “Big Bang” approach:

- you write the entire program!
- then you design and maybe even run some test cases!

This is terrible!

Testing Approaches

Test-Driven Development

Test-Driven Development (TDD), or "test-first":

- write the tests for a function,
- then, write the function,
- then, test the function!
- integrate that with other tested functions.
- rinse and repeat until you have constructed and tested an entire program

Testing Approaches

Test-Driven Development

Regression testing:

- Keep a comprehensive test suite!
- Always run all your tests; don't throw tests away!
- Re-run all your tests after changing your system!

Testing Approaches

Create, Mutate, Inspect [1]

Every test should follow a simple pattern:

create
set up a well-known environment

mutate
make *one* well-known change

inspect
check the results

[1] I'm sure there's a better name for this.

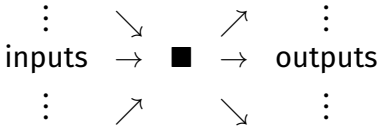
White-Box and Black-Box Testing

Black-Box Testing

Black-box testing

tests code from the outside...

- checks specified behaviour
- expected input to expected output
- uses *only* the interface!
  ... implementation-agnostic



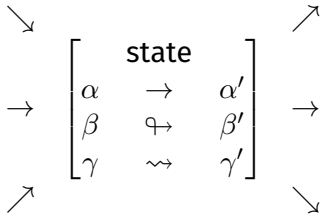
White-Box and Black-Box Testing

White-Box Testing

White-box testing

tests code from the inside...

- checks code structure and structure consistency
- checks internal functions
- tests rely on a particular implementation



Testing with *assert(3)*

Useful while developing, testing, debugging…
but *not* in production code!

assert(3) aborts the program;
emits error message useful to a programmer,
but not to the user of the application.
(e.g., those *gedit* errors)

Use *exception handlers* in production code
to terminate gracefully with a sensible error message