). collecting: … pointers?
). collecting: … pointers!

COMP1511 17s2

— Lecture 17 —

ADTs and Collections

Andrew Bennett

<andrew.bennett@unsw.edu.au>

review: abstract data types

collections of items

Don’t panic!

assignment 1 **reflection**

due Wed 20 Sep, 23:59:59 (Wednesday, week 9)

MandelbrArt

… The MandelbrArt Gallery

closes Wed 20 Sep, 23:59:59 (Wednesday, week 9)

vote on the most beautiful image soon!

Don’t panic!

next week (week “N1”) is

mid-semester break

no classes across the university!

Don’t panic!

the following week (week 10) is

quiet week for COMP1511

no lectures, tutorials, laboratories…

but the **week 10 prac exam** is still running

(help labs, consults are still running,

though on a modified timetable;

check the course website.)

Review: Making Allocations

allocating an array of known size?

use `calloc`, which takes

a **number of items**, and

the **size of each item**:

```
int *nums = calloc (17, sizeof (int));
// Don't forget to handle the error!
```

Review: Handling Allocation Errors

both `malloc` and `calloc` can go wrong!
when they do, they return `NULL`,
and it is **illegal** to dereference `NULL` (using `*` or `->`)

you **must** handle this error!
use either **err** or fprintf/exit

```
#include <err.h>



int *nums = calloc (17, sizeof (int));
if (nums == NULL) {
    err (EXIT_FAILURE, "couldn't allocate
memory");
}
```

Review: Handling Allocation Errors

both `malloc` and `calloc` can go wrong!

when they do, they return `NULL`,

and it is **illegal** to dereference `NULL` (using `*` or `->`)

you **must** handle this error!

use either err or **fprintf/exit**

```
#include <errno.h>
#include <string.h>

int *nums = calloc (17, sizeof (int));
if (nums == NULL) {
    fprintf (stderr, "couldn't allocate: %s",
strerror (errno));
    exit (EXIT_FAILURE);  // in `main`, you could
also return
}
```

Review: Abstract Data Types

implementation vs **interface**

interface: opaque values; details hidden from user

```
// in Adt.h
typedef struct _adt *Adt;
```

implementation: `struct` and function definitions

```
// in Adt.c
typedef struct _adt {
    // your implementation here!
} adt;
```

Review: Abstract Data Types

implementation vs **interface**

interface: opaque values; details hidden from user

```
// in Adt.h
Adt newAdt (void);
void destroyAdt (Adt a);
```

implementation: `struct` and function definitions

```
// in Adt.c
Adt newAdt (void) {
    adt *new = calloc (1, sizeof (adt));
    return new;
}

void destroyAdt (Adt a) {
    free (a);
}
```

COMP1511's Wide World of ADTs

a world of three parts:
the **interface** (`Complex.h`)
the **implementation** (`Complex.c`)
the **ADT users**

ADT users need not know the implementation,
only that it **complies to the interface**

ADT implementors need not know how they will be used
only that their users will **comply to the interface**

heads up!
match the file names
exactly in spelling and case
as the Style Guide (and
your ADT users)
expect your files to
conform to this interface

Implementing an ADT

when implementing an ADT,
you **must** implement **all** of
the structure and functions required

helper functions in your implementation
should be marked **static** to stop them 'leaking out'

structure definitions should match exactly;
you must define the concrete form of the structure

ADT implementations aren't "doing" anything
and **must not** have a `main` function...
ADT users are actually doing something,
and need a `main` function

Aside: `typedef` is commutative

```
typedef struct _adt *Adt;
typedef struct _adt { /* ... */ } adt;
```

now,

```
struct _adt *q;
adt *q;
Adt q;
```

all these variables have the same type

Assignment 2 is...

The Final Card-Down!

... thus named because the winner is

the first person to put their final card down

The Final Card-Down!

… will be in 3 parts:

tests for a game ADT
(spec released tomorrow)

implementing a game ADT
(opens in week 10)

writing a game player
(opens in week 11)

you should test before implementing!
otherwise you'll write *bad tests*…

The Final Card-Down!

… will be “due” on the last day of semester…
but there will be **competitions** where we:

run all your tests against all your implementations!

run all the AIs together to see whose AIs win the game!

submit early! submit often!
… even if you only have function stubs, submit it!
you **cannot** afford to wait until the last minute

The Final Card-Down!

working in **groups of four people**

... you must manage your own teams!

... you should arrange time to meet regularly

... everyone should contribute to every part

Stacks

Last **I**n, **F**irst **O**ut (LIFO)

we can only
add, see, or remove
(“push”, “peek”, or “pop”)
the **top-most** element

The Stack ADT, Again


```
typedef struct _stack *Stack;
```

```
Stack newStack (void);
void destroyStack (Stack s);
void stackPush (Stack s, char elt);
char stackTop (Stack s);
char stackPop (Stack s);
```

how can we implement this?

with a **fixed-size array**

(as we saw on Monday)

Varying the Length

what if our stacks aren't all of known maximum length?

what if we need to have arrays of dynamic length?

in C, we have dynamic allocation…
but that's still (effectively) fixed in length

we need a new way to represent
collections of things

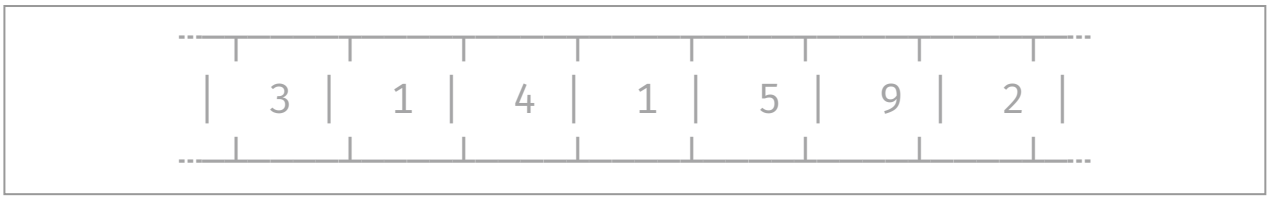
Continuity

arrays: a *contiguous* block of memory
(a continuous series of boxes in memory)

each item has a known location –
it's right after the previous item

```
int array[7] = { 3, 1, 4, 1, 5, 9, 2 };
```

gives us a sequence of elements in memory:

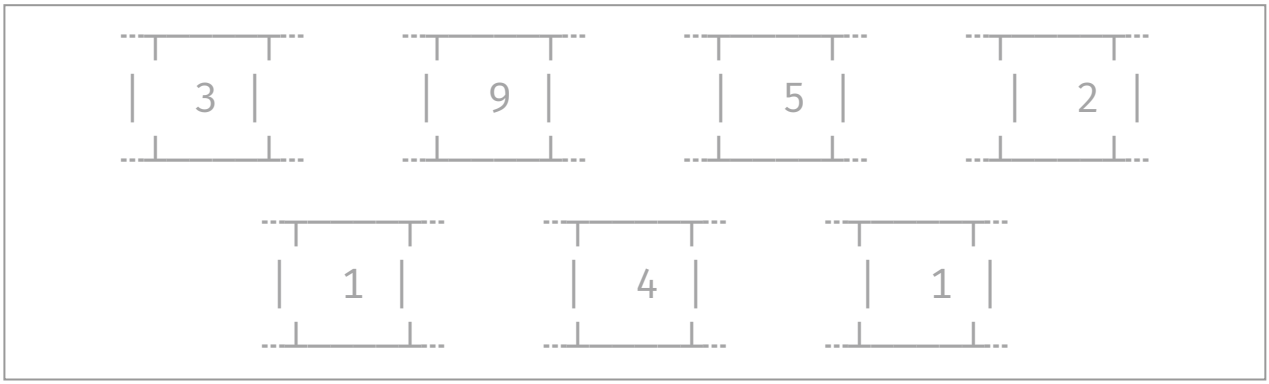


Discontinuity

We could have several pointers to separate allocations:

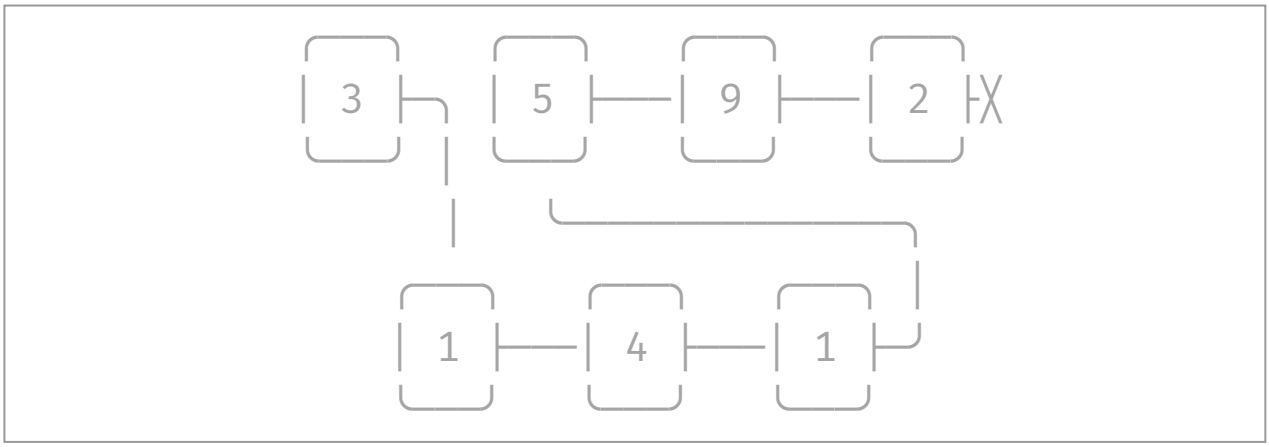
```
int *a = calloc (1, sizeof (int));
*a = 3;
int *b = calloc (1, sizeof (int));
*b = 1;
int *c = calloc (1, sizeof (int));
*c = 4;
// ... and so on
```

we'd have to hold on to all those pointers...
and we don't have a nice way to do that.



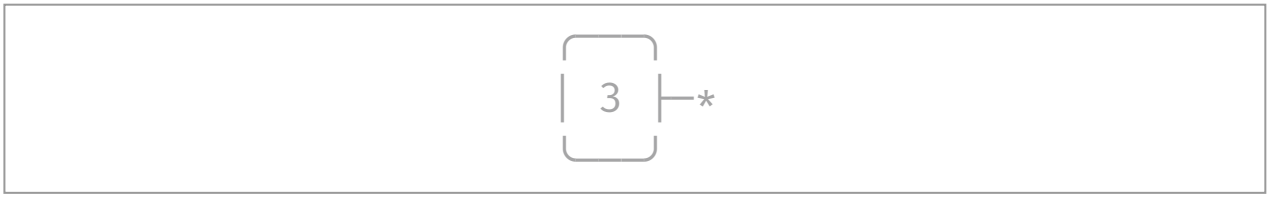
... what if each item knows
where the next one is?

Continual Discontinuity



Continual Discontinuity

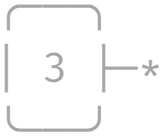
a sequence of **nodes**,
each with a **value** and a **next**



Continual Discontinuity


```
typedef struct _node *Node;
typedef struct _node {
    int value;
    Node next;
} node;
```

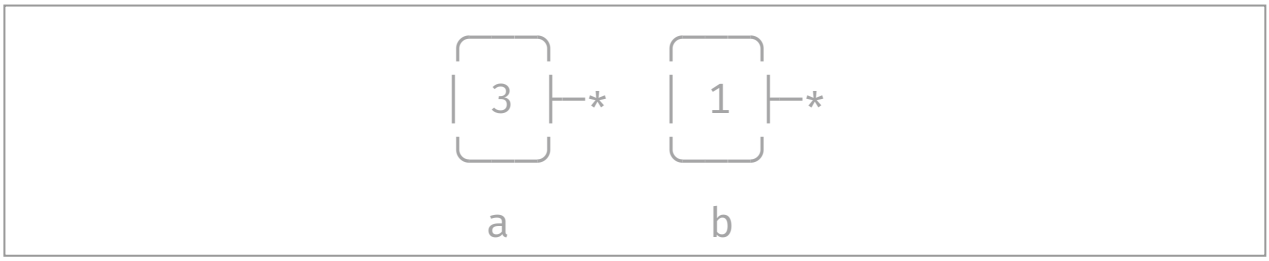
One Fish...



a

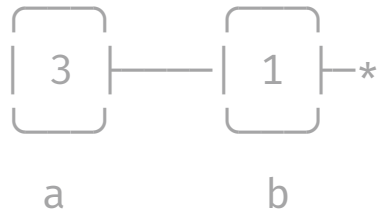
```
node a = { 3 };
```

Two Fish...




```
node a = { 3 };
node b = { 1 };
```

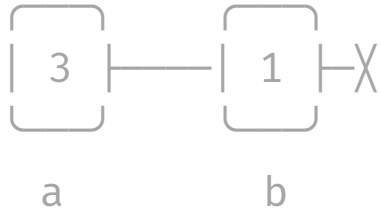
Red Fish...




```
node a = { 3 };
node b = { 1 };

a.next = &b;
```

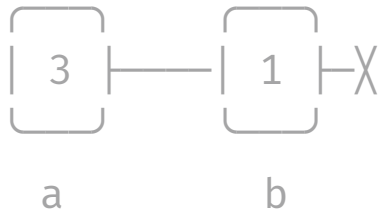
Blue Fish!




```
node a = { 3 };
node b = { 1 };

a.next = &b;
b.next = NULL;
```

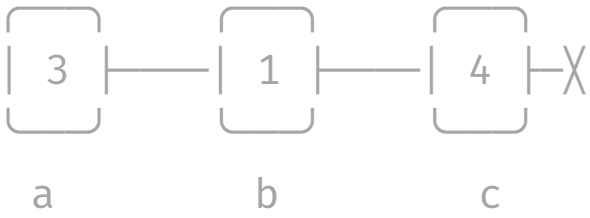
… pointers?




```
node a = { 3 };
node b = { 1 };

a.next = &b;
*(a.next).next = NULL;
```

… pointers!




```
node a = { 3 };
node b = { 1 };
node c = { 4 };
a.next = &b;
a.next->next = &c;
a.next->next->next = &d;
```