COMP1511 17s2

— Lecture 14 —

Towards ADTs

Andrew Bennett

<andrew.bennett@unsw.edu.au>

review: allocation, `struct`
concrete vs abstract types
Abstract Data Types

Don’t panic!

assignment 1 out now!
due Sun 17 Sep, 23:59:59 (Sunday, week 8)

Practice Prac Exam debrief

congratulations!
200 people got full marks!
wk07_prac mark available now

read and follow the instructions!
people showed late…
people showed up to their timetabled room?!
people *left their corralling room*!!!
people wrote functions that `scanf`d? `getchar`d?!

Review: `structs`

```
typedef struct _type-name {
    type member;
    [...]
} type-name;
```

a way to group together related data of differing types
we refer to the individual pieces of data
as fields or members

Review: Lifetimes

values on the stack will only live
as long as the stack frame does
we can say a variable has a lifetime,
bounded by the stack frame.

[[demo: lifetimes.c]]

Review: `struct` lifetimes

we usually want a `struct` to outlive a function
… how do we do that?

there's the "systems programming way":
pass a pointer to the `struct` down.

```
struct student s;
initStudent (&s);
```

[[demo: structLifetimes.c]]

… becomes messy when you need to refactor.
relatively concrete, relatively explicit.
there has to be a better way!

Review: Dynamic Allocation

values on the stack only last
as long as the stack frame does…
so returning a value that lives within a stack frame
is illegal

how do we get around this?
by putting the value somewhere else: the heap

we have calloc and free
which let us allocate and release
space on the heap

[[demo: calloc.c]]

Review: Dynamic Allocation

we have calloc and free
which let us allocate and release
space on the heap

Newton's third law of memory management:
for every allocation,
there must be an equal and opposite free.

Review: Dynamic Allocation of `structs`


```
#include <stdlib.h>

complex *c = calloc (1, sizeof (complex));
```

Aside: Handling Allocation Errors

using err, fprintf/exit, assert

```
#include <err.h>
#include <stdlib.h>
#include <sysexits.h>
```

```
complex *c = calloc (1, sizeof (complex));
if (c == NULL) {
    err (EX_OSERR, "couldn't allocate memory");
}
```

Aside: Handling Allocation Errors

using err, fprintf/exit, assert

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
complex *c = calloc (1, sizeof (complex));
if (c == NULL) {
    fprintf (stderr,
        "couldn't allocate memory: %s",
        strerror (errno));
    exit (1);
}
```

Aside: Handling Allocation Errors

using err, fprintf/exit, assert

NO!

`assert` is not an error handling mechanism

semantically, `assert` states an *invariant*,
only used while developing/debugging code

Concrete vs Abstract


```
typedef struct _complex {
    double re;
    double im;
} complex;
```

a type is...
concrete
if a user of that type has knowledge of how it works

a type is...
abstract
if a user has no knowledge of how it works

An Aside: USBs

works… anywhere!

Naming Types


```
typedef struct _complex { /* ... */ } complex;
//      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
typedef struct _complex *Complex;
//      ^~~~~~~~~~~~~~~~~
```

pointers to structures have
UpperCamelCaseNames

Concrete vs Abstract


```
typedef struct _complex {
    double re;
    double im;
} complex;
```

a concrete type is “right here”:
if you can see the type, you can use it

```
complex c;
c.re = 1.0;
c.im = 1 / 2.0;
```

Concrete vs Abstract

you cannot change the insides of the type
without breaking current software:
we couldn't, for example, easily switch to:

```
typedef struct _complex {
    double mod;
    double arg;
} complex;
```

Abstraction

our old friend, abstraction

use functions to retrieve
the real part, the imaginary part,
the modulus, the argument

doesn't really matter
how the implementation works…
only that the interface is correct.

Hiding Structures


```
typedef struct _complex *Complex;
```

we can now refer to `Complex`,
without knowing what's in
`struct _complex`...

we cannot stab it
but it can move around the system
as an opaque value.

ADTs
Abstract Data Types

separating the implementation from the interface

ADT Jargon

interface
the header file

implementation
the .c file with functions defined

consumer, user
other .c files that use the functions

constructor
makes a new instance of the ADT

destructor
destroys an instance of the ADT

getter, setter
retrieve or change a value in an instance