COMP1511 18s1 — Lecture 13

Intensity, Temporality, Complexity

~~Andrew Bennett~~
~~<andrew.bennett@unsw.edu.au>~~

Jashank Jeremy
<jashank.jeremy@unsw.edu.au>

Admin

Don't panic!

- **assignment 2** is here!
  - discussion: coming up shortly
- **Weekly test #4** ... due *tonight* 23:59:59 AEST
- **week 8** (next week) is **quiet week**!
  - no lectures! no tutorials! no labs!
  - ... help sessions still running

Overview

After this lecture, you should be able to...

- use composite data types as a part of a software system,
- use and reason about lifetimes, scope, and dynamic memory,

(**note**: you shouldn't be able to do all of these *immediately* after watching this lecture. however, this lecture should (hopefully!) give you the foundations you need to develop these skills. remember: programming is like learning any other language, it takes consistent and regular practice.)

Assignment 2: Intensity!

specification

10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

referee: `1511 intensity_referee`
plays your code against Lulu, Morgan, Amy

More C

`break`, `continue`

Keywords that allow us to change
control flow in our program.

Usually a bad idea...
Style Guide
... § Avoid These C Features

for

```
init;
while (cond) {
    body;
    step;
}
```

```
for (init; cond; step) {
    body;
}
```

Is terseness always better?

Lifetimes and Scope

Stack Frames and Lifetimes

in a stack frame...
previous frame, return address,
parameters, return values,
local variables

values on the stack will only live
as long as the stack frame does...

lifetimes of stack variables are
bounded by the stack frame.

Review: Staying Alive

Pass a reference up:
Push the value down.
Take a reference *lower* on the stack,
pass it up the stack to called functions

Globals! Statics!
absolutely goddamn' not.

Ask for memory elsewhere
Manage your own dynamic allocations
using `malloc`, `calloc`, `free`

`malloc`, `calloc`, `free`

three functions for managing heap allocations

`malloc`: make an allocation

```
void *malloc (size_t nBytes);
```

request `nBytes` of uninitialised memory;
return a reference to that, or `NULL` if it goes wrong.

`calloc`: contiguous allocation

```
void *calloc (size_t nItems, size_t itemSize);
```

request `nItems * itemSize` bytes of memory, initialised to zero;
return a reference to that, or `NULL` if it goes wrong.

`free`: release allocation

```
void free (void *obj);
```

release memory associated with a reference.
must be the same reference we got when allocating!

we can get allocated references back from functions;
they will explicitly say what is needed to `free` them.

Newton's Third Law of Memory Management

"For every `malloc`, there is an equal and opposite `free`."

Why?
Memory is a finite resource.
Leaking memory is bad practice,
especially in long-lived programs.
(see, e.g., Chrome)

Aside: Things Go Wrong

Wouldn't it be nice
if everything worked perfectly,
all the time?

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

int *xs = calloc (10, sizeof (int));
if (xs == NULL) {
    err (1, "couldn't allocate");
}
```

```
jashank@emeralfel:~$ ./remember
remember: couldn't allocate: Out of memory
```

Aside: Casting

C has *static* types:
data must be of the declared type.

C has *weak* types:
you can turn one type into another type,
using a *type cast*.
(You should never actually do this.)

Some C references (e.g., older textbooks, the Internet)
will make you do an explicit type-cast;
this is discouraged by our style guide
(and isn't needed anyway):

```
int *xs = (int *) calloc (4, sizeof (int));
// is equivalent to
int *xs = calloc (4, sizeof (int));
```

A Complex Composition

struct

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

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

Aside: typedef

```
//          refer to this type
//       v~~~~~~~~~~~~~~~~~~
typedef existing_type_name new_type_name;
//                         ^~~~~~~~~~~~~
//          with a better name!
```

Why?

create meaning with better names

hide details of implementation

(... save typing)

Aside: `struct` tags

A unique name in the space of `struct` names.
Only meaningful associated with `struct` keyword.

```
//      v~~
struct tag {
    field_type name;
};

struct tag instance;
```

Complex Numbers

$$z = x + iy$$

two pieces of related data!

A complex structure

```
typedef struct _complex {
    double real;
    double imag;
} complex;
```