Welcome!

COMP1511 18s1

Programming Fundamentals

COMP1511 18s1

— Lecture 15 —

malloc + Lists

Andrew Bennett

<andrew.bennett@unsw.edu.au>

Overview

after this lecture, you should be able to…

have a better understanding of **malloc**

be able to reason about dynamic memory management

have a basic understanding of **lists**

understand the fundamentals of **self-referential** and **linked** data structures

(**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.)

Admin

Don’t panic!

assignment 2

(if you haven’t started yet, start **ASAP**)

deadline extended to **Sunday 13th May**

assignment 1

tutor marking/feedback in progress

week 8 weekly test due tonight

don’t be scared!

don’t forget about **help sessions**!

see course website for details

malloc

allocates memory in “the heap”

memory “lives” *forever* until we **free** it

(or the program ends)

syntax:

```
malloc(number of bytes to allocate);
```

returns a **pointer** to the block of allocated memory

(i.e. the **address** of the memory, so we know how to find it!)

malloc – how many bytes?

```
malloc(number of bytes to allocate);
```

if we want **1000** ints, how many **bytes** is that?

1000?

how big is an int?

sizeof

we can find out the size of a type with **sizeof**

syntax:

```
sizeof(type);
// e.g. for an int:
sizeof(int);
```

```
printf("%ld", sizeof (char));    // 1
printf("%ld", sizeof (int));     // 4 commonly
printf("%ld", sizeof (double));  // 8 commonly
printf("%ld", sizeof (int[10])); // 40 commonly
printf("%ld", sizeof (int *));   // 4 or 8 commonly
printf("%ld", sizeof "hello");   // 6
```

malloc: syntax

```
// allocating enough memory for 1000 ints
malloc(1000 * sizeof(int));
```

remember, malloc returns the **address** of the memory it's allocated:

```
// allocating enough memory for 1000 ints
int *p = malloc(1000 * sizeof(int));
```

malloc – when things go wrong

what happens if the allocation **fails**?

```
int *p = malloc(1000 * sizeof(int));
if (p == NULL) {
    fprintf(stderr, "Error: couldn't allocate memory!\n");
    exit(1);
}
```

free

when we're done with the memory, we need to **free** it

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

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

Newton’s 3rd 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)

Putting it together

```
int *p;
p = malloc(100000000 * sizeof (int));
if (p == NULL) {
    printf("Error: array could not be allocated.\n");
    exit(1);
}

// we can now use the pointer
// ... lots of things to do

// free up the memory that was used
free(p);
```

let's look at something **new**…

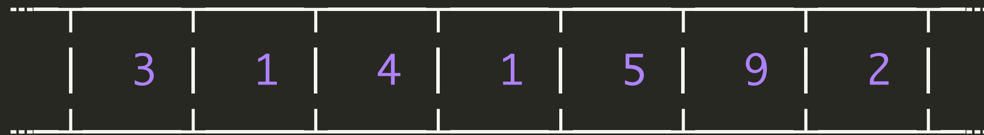
Remember Arrays?

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:



Varying the Length

what if we don't know how big our array needs to be in advance?

what if we need to **increase** the length?

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

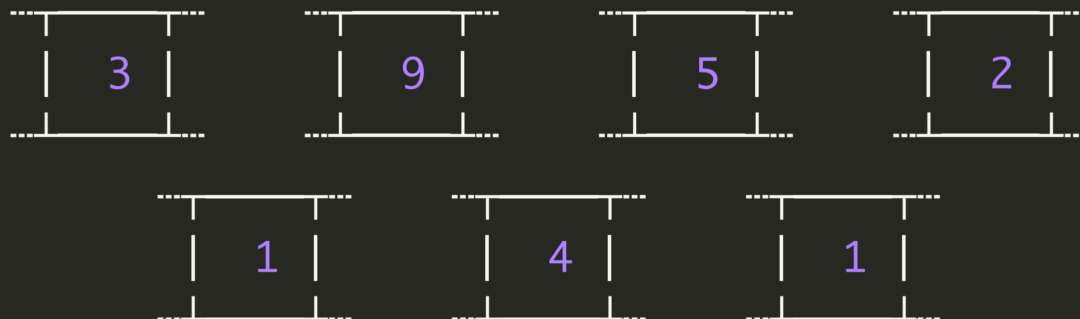
we need a new way to represent
collections of things

Discontinuity

We could have several pointers to separate allocations:

```
int *a = malloc (1 * sizeof (int));
*a = 3;
int *b = malloc (1 * sizeof (int));
*b = 1;
int *c = malloc (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.

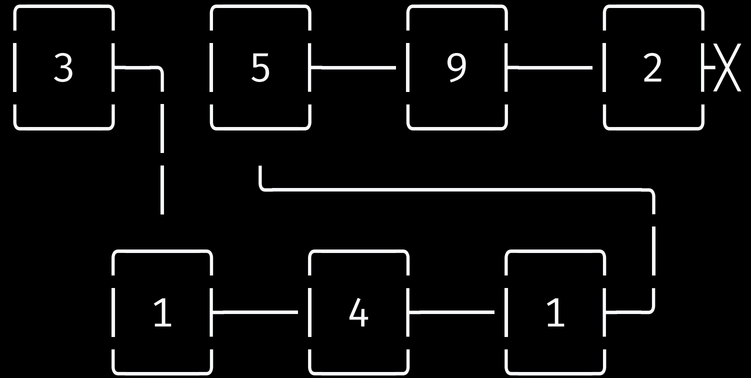


… what if each item knows

where the **next** one is?

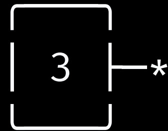
introducing: **lists**

Continual Discontinuity



Continual Discontinuity

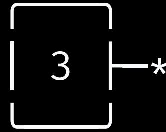
a sequence of **nodes**,
each with a **value** and a **next**



Continual Discontinuity

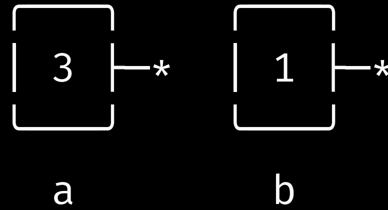
```
struct node {
    struct node *next;
    int          data;
};
```

One Fish...



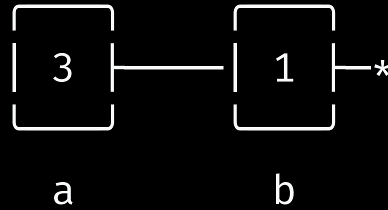

```
struct node a = { 3 };
```

Two Fish...




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

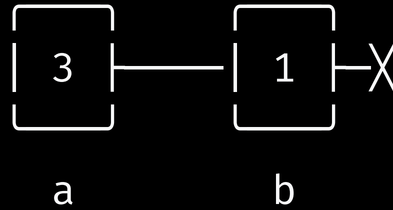
Red Fish...




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

a.next = &b;
```

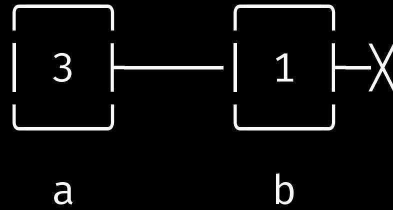
Blue Fish!




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

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

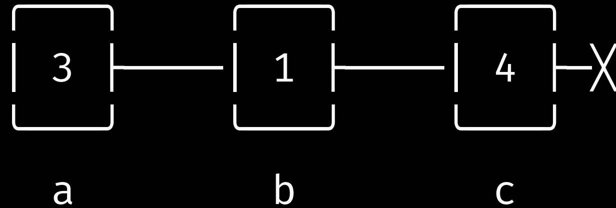
… pointers?




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

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

... pointers!




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