COMP1511 revision lecture

Side A (11-1)

I Thirty Thousand Feet

II Memory, Pointers, Arrays, Lifetimes

III Strings and File I/O

IV Structured Data

Side B (2-4)

V Linked Lists

VI Stacks and Queues

VII Sorting and Searching

VIII Preparing for Exams

IX Debugging with GDB

This lecture cannot be played on old tin boxes,
no matter what they are fitted with.

COMP1511 **revision lecture**

these slides up on WebCMS3

Questions? Ask on Ed!

C Syntax

Curtis Millar

<c.millar@unsw.edu.au>

Overview

After this section, you should remember...

Variables

Arithmetic and boolean operators

Control Flow

Branching with `if`

Looping with `while`

Breaking code into *functions*

— 1.5 HD ReMix —

C Style

Why do I need good style?

Style isn't about making code simply **look good**.

Style is about writing code **effectively**.

Make your code **useful** and **understandable**.

Make **debugging** easier.

Make **growing** your code *easier*.

#1 rule for style.

Consistency

Consistent **indentation**.

Consistent **names**.

Variables

Should describe the **value** they store.

Should always be **honest**.

Usually **nouns** (`num_apples`) or **adjectives** for conditions (`empty`).

Functions

Should describe the **action** taken (as a **verb**, i.e. `count_elements`).

or

Should describe the **condition** being tested (`is_empty`).

Booleans

`FALSE` is **always** `0`.

`TRUE` can be anything else (usually `1`).

An Example!


```
#define FALSE 0
#define TRUE (!FALSE)
#define TRUE (1 == 1)

int correct = TRUE;

if (correct) {
    // is correct.
}

if (!correct) {
    // is not correct.
}
```

Comments

Always provide a **function comment**.

Describe how the function is used.

The rules for using the function.

Comments

Avoid **inline comments**.

If you need comments to explain your code,
your code isn’t clear enough.

Use **good style** to make sure your code is clear enough on its own.

Memory

Curtis Millar

<c.millar@unsw.edu.au>

Overview

After this section, you should have *memorized*...

Memory

Functions

Lifetimes

Pointers

Arrays

Memory

Everything lives in *memory*.

Memory is a bunch of **cells** in a long line.

Each cell has an **address**.

The first cell is `0x00000000`.

The second cell is `0x00000001`.

...

The last cell is `0xFFFFFFFF`.

Memory

Near the **start** of memory

(at the `0x00000000` end)

we have

our program **code**

then

the **heap**.

Memory

Near the **end** of memory

(at the `0xFFFFFFFF` end)

we have

the **stack**.

Not *a stack*.

The stack grows **backwards**

(*towards* `0x00000000`)

in memory.

The stack grows when we call **functions**.

Functions

Each **instance** of a function has its own **stack frame**.

The variables for a function live in its stack frame.

The stack frame is pushed onto *the* stack
when the function is **called**.

The stack frame is popped off of *the* stack
(and destroyed) when the function **returns**.

Functions

variables are *created* inside of functions.

They disappear when the function **returns**.

arguments are variables that have values **copied in** from outside the function.

The function can copy **one** value back out when it **returns**.

Functions - Pass By Copy

arguments are variables that have values **copied in** from outside the function.

changing their values **inside the function** **will not** change their values in the **calling function**

...

if we want to change the value of a variable in the calling function, we need its **address**

Pointers

allow us to **pass by reference**

*"I give you a reference to **where** this thing lives*
*rather than giving you your **own copy**"*

Arrays

an array is a contiguous sequence of values of the same type

e.g.

```
// creates 10 ints in the function's stack frame
int numbers[10];
```

`numbers` refers to the **address**
of the **first element** in the array

Arrays and Functions

when we pass an array into a function
we are passing the **address** of the **first element**

we have no way to distinguish this from
passing the **address** of a **single variable**

how do we tell which it is?

how do we know **how long** the array is?

...

we need to pass the **size** of the array into the function

Arrays and Functions

how do we know **how big** the array is?

```
void print_array(int *array) {
    int i = 0;
    while (i < ?????) {
        printf("%d ", array[i]);
        i++;
    }
}
```

we need to pass the **size** of the array into the function

```
void print_array(int *array, int length) {
    int i = 0;
    while (i < length) {
        printf("%d ", array[i]);
        i++;
    }
}
```

Returning Arrays From Functions

what happens if we create an array **inside a function**
and **return** that array?

```
int *make_array(void) {
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    return array;
}
```

remember, **`array`** is the address of the first element
in the array

the array lives inside the function's stack frame
so when the function returns… the array disappears

The Heap

if we want to **create** an array inside a function
that will **still exist** when the function **returns**,
we need to put it somewhere else

where?

on the **heap**!

we do this using **`malloc`**

```
int *make_array(void) {
    int *array = malloc(10 * sizeof(int));
    // set the values somehow
    return array;
}
```

The Heap - Malloc

malloc gives us a reference to where the memory is stored

because this memory is stored on the heap
it will still exist once the function returns

...

we need to tell malloc how many bytes we want

malloc(**number of things** * **size of each thing**)

The Heap - Malloc - Free

when we’re finished using the memory, we need to **free** it

```
int *array = malloc(...);

...

free(array);
```

— III —

Strings and File I/O

Curtis Millar

<c.millar@unsw.edu.au>

User Interaction

Interact with users by

reading from **stdin**

and

writing to **stdout**.

Displaying Text

We can use `printf`!.

Display **numbers** with `"%d"` and `"%f"`.

Display **strings** with `"%s"`.

Reading Numbers


```
int integer;
scanf("%d", &integer);

double decimal;
scanf("%lf", &decimal);
```

Reading Many Numbers

scanf returns the number of items **successfully read**.

```
int numbers[5] = {0};
int num_read = scanf(
    "%d %d %d %d %d",
    &numbers[0],
    &numbers[1],
    &numbers[2],
    &numbers[3],
    &numbers[4]
);
assert(num_read == 5);
```

Reading Many Numbers

We can use a loop to read until the end of input.

```c
int numbers[5] = {0};

int i = 0;
while (scanf("%d", &numbers[i]) == 1 && i < 5) {
    i++;
}
```

Reading Text

We can use `fgets` to read text from a file into an array.

```
char text[BUFFER_SIZE] = "";
fgets(text, BUFFER_SIZE, stdin);

// in a loop
while (fgets(text, BUFFER_SIZE, stdin) != NULL) {
    // Do something
}
```

Processing Characters


```c
int c = getchar();
while (c != EOF) {
    // Do something
    c = getchar();
}
```

Redirecting Input and Output


```
$ ./my_program < input.txt
$ ./my_program > output.txt
$ ./my_program < input.txt > output.txt
```

Files

Open a file and start **reading** at the beginning.

```
FILE *file = fopen("filename.txt", "r");
```

Open a file and start **writing** at the beginning

(removing everything that is already there).

```
FILE *file = fopen("filename.txt", "w");
```

Open a file and start **appending** to the end,

after everything that is already there.

```
FILE *file = fopen("filename.txt", "a");
```

What To Use When Reading...

if you want to...

read numbers

... from the terminal: scanf

... from a file: fscanf

with %d (int) or %lf (double)

read text

... from the terminal: fgets

... from a file: fgets

read characters

... from the terminal: getchar

... from a file: fgetc

What To Use When Writing...

if you want to...

write numbers

... to the terminal: printf

... to a file: fprintf

with %d (int) or %lf (double)

write text

... to the terminal: printf

... to a file: fprintf

with %s

write characters

... to the terminal: putchar

... to a file: fputc

How To Read All Input

getchar: EOF

```
while ((ch = getchar()) != EOF)
```

fgets: NULL

```
while (fgets(array, SIZE, stdin) != NULL)
```

scanf: num items

```
while (scanf("%d", &num) == 1)
while (scanf("%d %d", &num1, &num2) == 2)
```

— V —

Linked Lists

node Nodes

impl Implementation

o^o Abstractions

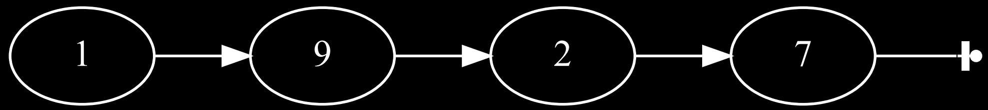
map Comprehensions

wrap Wrapped Linked Lists

dll Doubly-Linked Lists

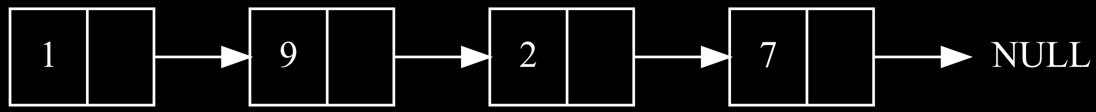
cll Circularly-Linked Lists

node Nodes



a simple list

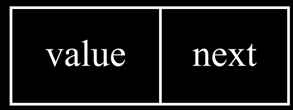
node Structure



listing lazily to the left

"this value, and all the other values"

impl Implementing



* * *

```
struct node {
    Item value;
    struct node *next;
}
```

`value` can represent arbitrarily complex structures:
a single integer! arrays of data! other linked lists!

impl Why?

“self-referential” data structure:
points to the same *type* of structure

items aren’t guaranteed to be adjacent in memory

reordering is ‘easy’ pointer-shuffling,
not ‘hard’ value moving

grow and shrink to fit a collection,
instead of having fixed pre-allocations

items can be added or removed in any order

impl Creating Nodes

Usually dynamically allocated:

```
struct node *list = calloc (1, sizeof (struct node));
```

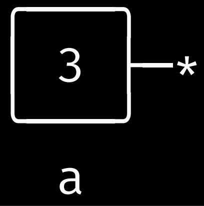
* * *

```
struct node *list = calloc (1, sizeof *list);
```

* * *

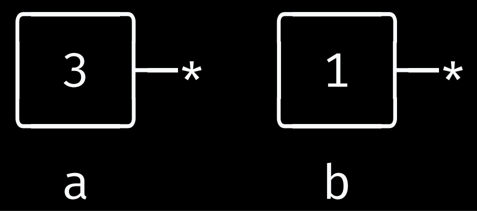
```
struct node *list = malloc (sizeof *list);
```

One Fish...



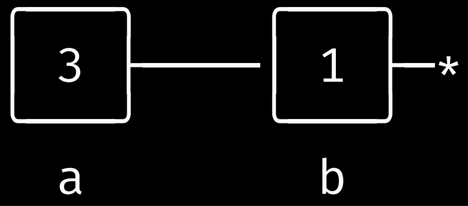

```
struct node a = { 3 };
```

Two Fish...




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

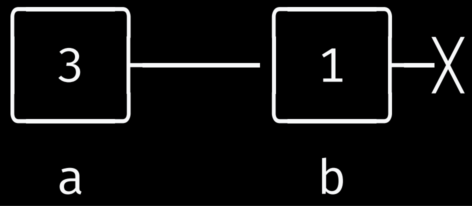
Red Fish...




```
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;
```

impl Linking Nodes

```
struct node *n1 = malloc (sizeof *n1);
struct node *n2 = malloc (sizeof *n2);
struct node *n3 = malloc (sizeof *n3);
struct node *n4 = malloc (sizeof *n4);

n1->next = n2;
n2->next = n3;
n3->next = n4;
n4->next = NULL;
```

impl One-Argument list_new

```
struct node *list_new  (Item value);
struct node *list_new2 (Item value, struct node *next);
```

* * *

```
struct node *list                  = list_new (1);
list->next                         = list_new (9);
list->next->next                   = list_new (2);
list->next->next->next             = list_new (7);
list->next->next->next->next = NULL;
```

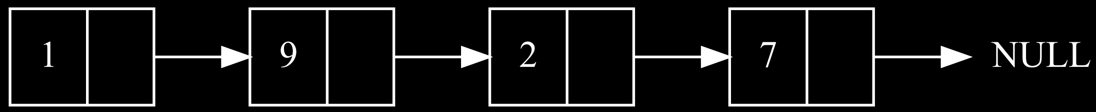
impl Two-Argument list_new

```
struct node *list_new  (Item value);
struct node *list_new2 (Item value, struct node *next);
```

* * *

```
struct node *list =
    list_new2 (1,
      list_new2 (9,
        list_new2 (2,
          list_new2 (7, NULL))));
```

impl Traversing



here's one we prepared earlier

```
struct node *curr = list;
while (curr != NULL) {
    // ...
    curr = curr->next;
}
```

("travel across or through;

move back and forth or sideways")

impl Destroying Nodes (I)


```
free (list);
```

… what’s wrong with this?

impl Destroying Nodes (II)


```
free (list);
free (list->next);
free (list->next->next);
free (list->next->next->next);
```

Newton’s third law of memory management:
for every allocation, there is an equal and opposite free

… what’s wrong with this?

impl Destroying Nodes (III)


```
struct node *curr = head;
struct node *next;
while (curr != NULL) {
    next = curr->next;
    free (curr);
    curr = next;
}
```

A delicate dance... because

use after free is illegal

(if you do it,

I climb out of your screen

and set your hair on fire)

(*dcc* yells at you if you do!)

o^o Abstractions

you should ***ALWAYS*** build abstractions to make LLs easier

doing pointer-y evils all the time is terrible, no good, very bad.

list_new creates a new list

list_insert_head prepends a value to the list

list_insert_tail appends a value to the list

list_remove_head removes the first value of the list

list_remove_tail removes the last value of the list

list_is_empty tells you if the list is empty

list_delete destroys the whole list

(These are very handy functions!)

o^o Implementing

[[demo: list/list.h]]

[[demo: list/list.c]]

map Comprehension


```
int list_sum (struct node *n) {
    int sum = 0;
    struct node *curr = n;
    while (curr != NULL) {
        count += curr->value;
        curr = curr->next;
    }
    return sum;
}
```

map Recursion (II)

"do something here, and with the rest of the list"

linked lists are particularly amenable to recursion

```
int list_sum (struct node *n) {
    int sum;
    if (n == NULL) {
        sum = 0;
    } else {
        sum = n->value + list_sum (n->next);
    }
    return sum;
}
```

Recursion (II)


```
int list_sum (struct node *n) {
    if (n == NULL) {
        return 0;
    }
    return n->value + list_sum (n->next);
}
```

one of the few places where an early `return` is probably okay

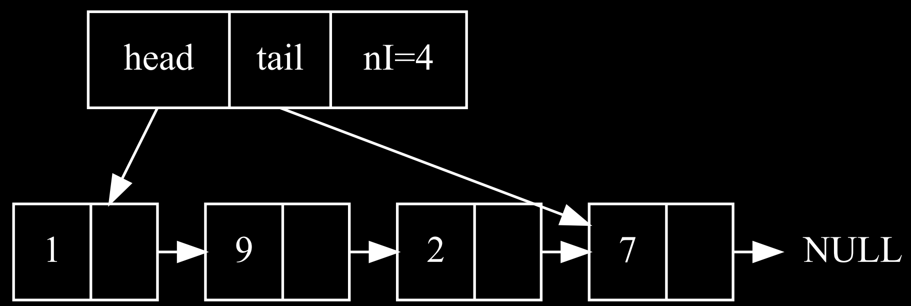
Wrapping (I)

Sometimes, we wrap it.

this lets us...

easily move the head

have constant-time operations



what a head

wrap struct list

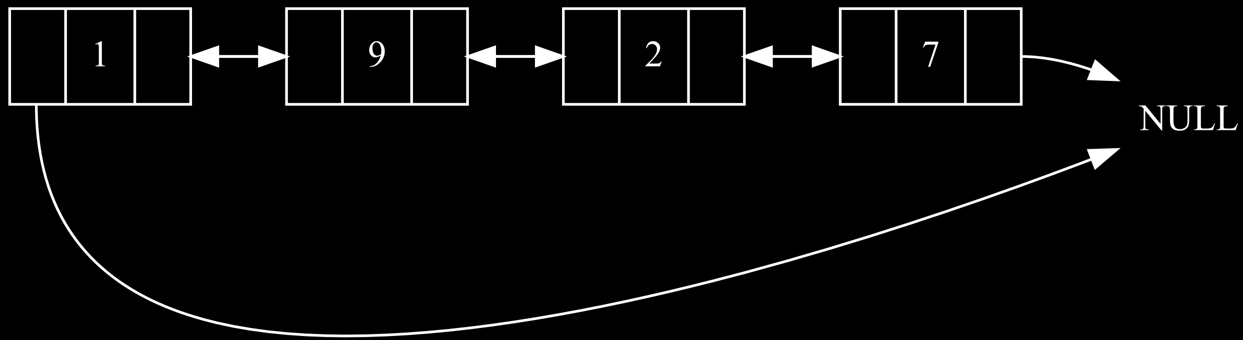
```
struct list {
    struct node *head;          // or first
    struct node *tail;          // or last
    int n_items;                // or length
};
```

`head` ⇒ first item; easy head insertion

`tail` ⇒ last item; easier tail insertion… why?

`n_items` ⇒ item count; easier length… why?

dll Doubly-Linked Lists

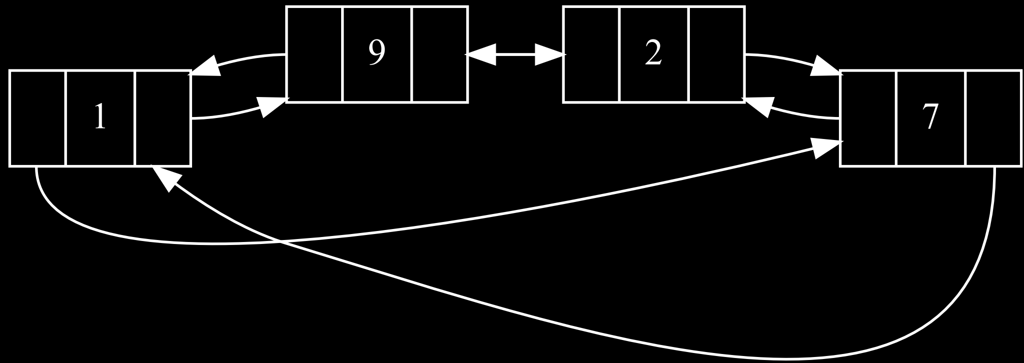


both sides together

traverse in both directions!

swapping & deletion becomes harder...

cll Circular Linked Lists



circle!

the linked list turns, and the nodes come and pass,
leaving pointers that become invalidated...
pointers become freed, and even the list is long forgotten
when the node that gave it birth comes again...

with apologies to Robert Jordan

don’t lose track of the “beginning” of the list

— VI —

Stacks and Queues

Curtis Millar

<c.millar@unsw.edu.au>

Abstract Data Types

Define the **interface** for interacting with the data type.

Hide the **implementation** for the data type.

Interface vs. Implementation


```
#ifndef DATA_TYPE_H
#define DATA_TYPE_H
```

Describe the **interface** in the **DataType.h** file.

Always need a way to **create** and **destroy** the ADT.

```
#endif /* DATA_TYPE_H */
```

Interface vs. Implementation


```c
#include "DataType.h"
```

Define the **implementation** in the **DataType.c** file.

Stacks

Stacks are **first in, last out**.

The **first** value we insert into a stack is the **last** value we remove.

Inserting is **pushing** onto a stack.

Removing is **popping** off of the stack.

Stacks

Could implement with a fixed size **array**.

Stacks

Could implement with a **linked list**.

Queues

Stacks are **first in, first out**.

The **first** value we insert into a queue is the **first** value we remove.

Inserting is **joining** (or enqueueing) a queue.

Removing is **leaving** (or dequeueing) a queue.

Queues

Could implement with a fixed size **array**.

Queues

Could implement with a **linked list**.

— VII —
Sorting and Searching

sort Ordering Things

search Finding Things

stdlib `stdlib.h` for fun and profit

Bubble Sort

best $O(n)$, average $O(n^2)$, worst $O(n^2)$

probably stable, probably adaptive

https://youtu.be/Cq7SMsQBEUw

sort Quick Sort

max: best $O(n)$, average $O(n \log n)$, worst $O(n^2)$
med3: best $O(n)$, average $O(n \log n)$, worst $O(n \log n)$
rand: best $O(n)$, average $O(n \log n)$, worst $O(n \log n)$
maybe stable, maybe adaptive

https://youtu.be/8hEyhs3OV1w

stdlib qsort for fun and profit


```
void qsort (
    void *base,      // bottom of the array
    size_t nmemb,    // number of items
    size_t size,     // size of each item
    int (*compar)(const void *, const void *)
                     // function comparing two items
);
```

[[demo: sose/ord.c]]

Finding Things

$$\{3, 8, 7, 4, 2, 9, 1, 5, 10, 6\}$$

how many questions to find a value in this space?

[[demo: sose/lsearch.c]]

Finding Things

$$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

how many questions to find a value in this space?

[[demo: sose/bsearch.c]]

stdlib bsearch for fun and profit


```
void *bsearch (
    const void *key,     // search key
    const void *base,    // bottom of the array
    size_t nmemb,        // number of items
    size_t size,         // size of each item
    int (*compar)(const void *, const void *)
                         // function comparing two items
);
```

[[demo: sose/monthnum.c]]

— VIII —

Preparing for Exams

Tips and tricks for the exam

Coding Under Pressure

tip #1: sit on your hands

don't just jump straight in and start coding

(or your code will be a tangled mess)

first, **read the question**

think about what it is you're setting out to do

think about how you need to approach the question

draw diagrams

(we'll give you paper to write on)

Coding Under Pressure

tip #2: read all of the questions first

start with the questions you find easy

start with the easy **hurdle questions**

(arrays and linked lists – these will be clearly marked)

don't spend all of your time on the harder problems
without doing the easier ones first

Revision

tip #3: practice writing code

do the **revision exercises**

(linked on course website)

do the **extra tute questions** – there are lots of them!

(at the bottom of every tutorial page)

Skeleton Exam

tip #4: read the skeleton exam

we give you the **actual** exam paper
before the exam

(with the content of the questions removed)

read this so you know what to expect

(how many questions, what type they are, how many marks)

Don't Panic

tip #5: if there's a problem,

tell the invigilator

if the autotests are broken

if you don't understand what a question means

if your keyboard/mouse don't work properly

if you can't concentrate because the door is beeping

if the monitor is so bright it's giving you a headache

...

tell the invigilator!

The End

good luck with the exam!