COMP1511 - Programming Fundamentals

Term 1, 2019 - Lecture 20
Stream B

What did we cover in the last ten weeks?

What are we covering today?

What are we actually covering?

A recap of what we've covered in the course

- C language and programming constructs
- Programming Fundamentals

Assessment

- The exam
- The format
- How to prepare

Programming in C

COMP1511 C Language Topics in the order they were taught

- Input/Output
- Variables
- If statements
- While statements (looping)
- Arrays
- Functions
- Characters and Strings
- Pointers
- Structures
- Memory
- Linked Lists
- Abstract Data Types

C as a programming language

- A compiled language
- We use dcc as our compiler here, but there are others
  - clang
  - gcc
  - etc
- Compilers read code from the top to the bottom
- They translate it into executable machine code
- All C programs must have a main() function, which is their starting point
- Compilers can handle multiple file projects
- We compile C files while we #include H files

Input/Output

Scanf and Printf allow us to communicate with our user

- scanf reads from the standard input
- printf writes to standard input
- They both use pattern strings like `%d` and `%s` to format our data in a readable way

```
// ask the user for a number, then say it back to them
int number;
printf("Please enter a number: ");
scanf("%d", &number);
printf("You entered: %d", number);
```

Command Line Arguments

When we run a program, we can add words after the program name

- These extra words are given to the main function to use
- argc is an integer that is the total number of words (including the program name)
- argv is an array of strings that contain all the words

Command Line Arguments in use

```
int main (int argc, char *argv[]) {
    printf("The %d words were ", argc);
    int i = 0;
    while (i < argc) {
        printf("%s ", argv[i]);
        i++;
    }
}
```

When this code is run with: `"./args hello world"`

It produces this: `"The 3 words were ./args hello world"`

Variables

Variables

- Store information in memory
- Come in different types:
  - int, double, char, structs, arrays etc
- We can change the value of variables
- We can pass the value of variables to functions
- We can pass variables to functions via pointers

Constants

- #define allows us to set constant values that won't change in the program

Simple Variables Code

```
#define DOUGLAS 42

int main (void) {
    // Declaring a variable
    int answer;
    // Initialising the variable
    answer = 7;
    // Give the variable a different value
    answer = DOUGLAS;

    // we can also Declare and Initialise together
     int answerTwo = 88;
}
```

If statements

Questions and answers

- Conditional programming
- Evaluate an expression, running the code in the brackets
- Run the body inside the curly brackets if the expression is true (non-zero)

```
if (x < y) {
    // This section runs if x is less than y
}
// otherwise the code skips to here after the
// expression in the () equates to 0
```

While loops

Looping Code

- While loops allow us to run the same code multiple times
- We can stop them after a set number of times
- Or we can stop them after a certain condition is met

Loops are used for . . .

- Checking all the values in a data structure (array or linked list)
- Repeating a task until something specific changes
- and any other repetition we might need

While loop code

Very commonly used to loop through an array

```
int numbers[10] = {0};
int counter = 0;

// set array to the numbers 0-9 sequential
while (counter < 10) {
    // code in here will run 10 times
    numbers[counter] = counter;
    // increment the counter
    counter = counter + 1;
}
// When counter hits 10 and the loop's test fails
// the program will exit the loop
```

While loop code

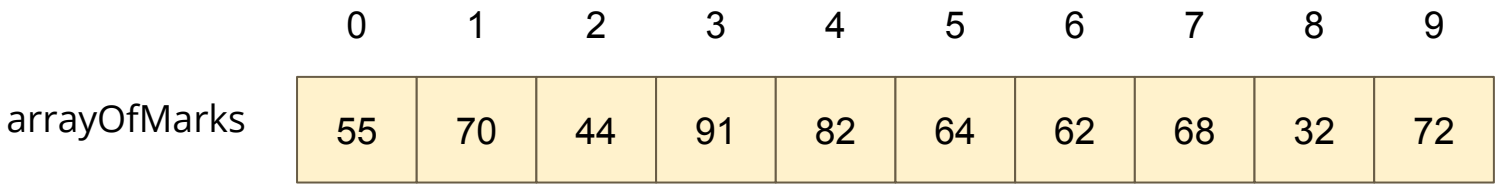
Assuming a Linked List, the code loops through it

```
struct node *loopNode = head;

while (loopNode != NULL) {
    // code in here will run until the loopNode pointer
    // moves off the end of the list

    // increment the node pointer
    loopNode = loopNode->next;
}
// When loopNode pointer is aiming off the end of the list
// the program will exit the loop
```

Arrays

Collections of variables of the same type

- We use these if we need multiple of the same type of variable
- The array size is decided when it is created and cannot change
- Array elements are collected together in memory
- Not accessible individually by name, but by index



Array Code

```
int main (void) {
    // declare an array, all zeroes
    int marks[10] = {0};

    // set first element to 85
    marks[0] = 85;
    // access using an index variable
    int accessIndex = 3;
    marks[accessIndex] = 50;
    // copy one element over another
    marks[2] = marks[6];
    // cause an error by trying to access out of bounds
    marks[10] = 99;
```

Functions

Code that is written separately and is called by name

- Not written in the line by line flow
- A block of code that is given a name
- This code runs every time that name is "called" by other code
- Functions have input parameters and an output

Function Code

```c
// Function Declarations above the main or in a header file
int add (int a, int b);

int main (void) {
    int firstNumber = 4;
    int secondNumber = 6;
    int total = add(firstNumber, secondNumber);
    return 0;
}

// This function takes two integers and returns their sum
int add (int a, int b) {
    return a + b;
}
```

Characters and Strings

Used to represent letters and words

- char is an 8 bit integer that allows us to encode characters
- Uses ASCII encoding (but we don't need to know ASCII to use them)

- Strings are arrays of characters
- The array is usually declared larger than it needs to be
- The word inside is ended by a Null Terminator `'\0'`
- Using C library functions can make working with strings easier

Characters and Strings in code

```
// read user input
char input[MAX_LENGTH];
fgets(input, MAX_LENGTH, stdin);
printf("%s\n", input);

// print string vertically
int i = 0;
while (input[i] != '\0') {
    printf("%c\n", input[i]);
    i++;
}
```

Pointers

Variables that refer to other variables

- A pointer aims at memory (actually stores a memory address)
- That memory can be another variable already in the program
- It can also be allocated memory
- The pointer allows us to access another variable

- * dereferences the pointer (access the variable it's pointing at)
- & gives the address of a variable (like making a pointer to it)
- -> is used with structs to allow a pointer to access what's inside

Simple Pointers Code

```
int main (void) {
    int i = 100;
    // the pointer ip will aim at the integer i
    int *ip = &i;
    printf("The value of the variable at address %p is %d\n", ip, *ip);

    // this second print statement will show the same address
    // but a value one higher than the previous
    increment(ip);
    printf("The value of the variable at address %p is %d\n", ip, *ip);
}

void increment (int *i) {
    *i = *i + 1;
}
```

Structures

Custom built types made up of other types

- structs are declared before use
- They can contain any other types (including other structs and arrays)
- We use a . operator to access elements they contain
- If we have a pointer to a struct, we use -> to access elements

Structs in code

```
struct spaceship {
    char name[20];
    int engines;
    int wings;
};

int main (void) {
    struct spaceship xwing;
    xwing.name = "Red 5";
    xwing.engines = 4;
    xwing.wings = 4;

    struct spaceship *myShip = &xwing;

    // my ship takes a hit
    myShip->engines--;
    myShip->wings--;
}
```

Memory

Our programs are stored in the computer's memory while they run

- All our code will be in memory
- All our variables also
- Variables declared inside a set of curly braces will only last until those braces close (what goes on inside curly braces stays inside curly braces)
- If we want some memory to last longer than the function, we allocate it
- malloc() and free() allow us to allocate and free memory
- sizeof provides an exact size in bytes so malloc knows how much we need

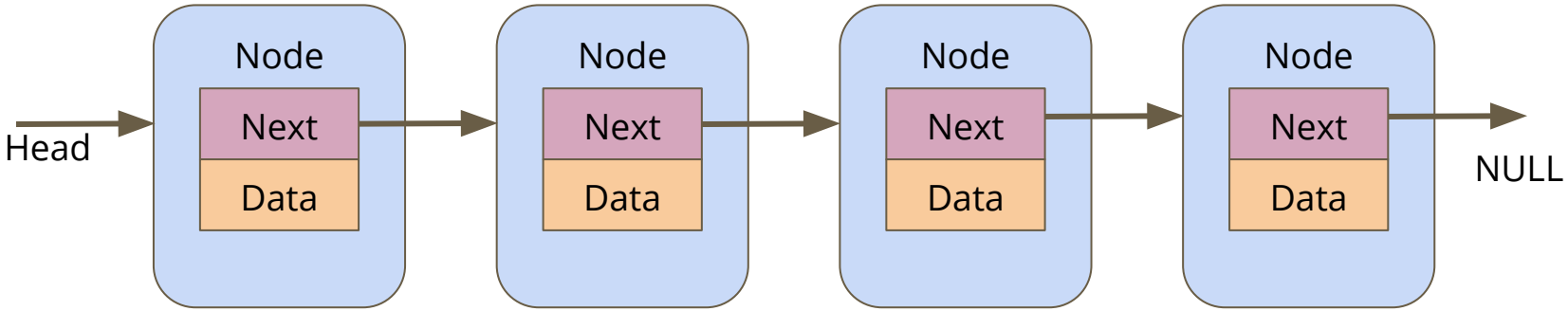
Memory code

```
struct spaceship {
    char name[20];
    int engines;
    int wings;
};

int main (void) {
    struct spaceship *myShip = malloc (sizeof (struct spaceship));
    myShip->name = "Millennium Falcon";
    myShip->engines = 1;
    myShip->wings = 0;

    // Lost my ship in a Sabacc game, free its memory
    free(myShip);
}
```

Linked Lists

Structs for nodes that contain pointers to the same struct

- Nodes can point to each other in a chain to form a linked list
- Convenient because:
  - They're not a fixed size (can grow or shrink)
  - Elements can be inserted or removed easily anywhere in the list
- The nodes may be in separate parts of memory



Linked Lists in code

```
struct pokenode {
    struct pokenode *next;
    int id;
};

int main (void) {
    struct pokenode *head = NULL;
    head = addNode(1, head);
    head = addNode(5, head);
}

// Add a node to the start of a list and return the new head
struct pokenode *addNode(int newID, struct pokenode *list) {
    struct pokenode *newNode = malloc(sizeof(struct pokenode));
    newNode->id = newID;
    newNode->next = list;
    return newNode;
}
```

Abstract Data Types

Separating Declared Functionality from the Implementation

- Functionality declared in a Header File
- Implementation in a C file
- This allows us to hide the Implementation
- It protects the raw data from incorrect access
- It also simplifies the interface when we just use provided functions

Abstract Data Types Header code

```
// ship type hides the struct that it is
// implemented as
typedef struct shipInternals *ship;

// functions to create and destroy ships
ship shipCreate(char* name);
void shipFree(ship);

// set off on a voyage of discovery
ship voyage(ship, int years);
```

Abstract Data Types Implementation

```
// ship type hides the struct that it is implemented as
struct shipInternals {
    char name[MAX_NAME_LENGTH];
};

ship shipCreate(char* name) {
    ship newShip = malloc(sizeof (struct shipInternals));
}
void shipFree(ship) {
    free(ship);
}
// set off on a voyage of discovery
ship voyage(ship, int years) {
    int discoveries = 0, yearsPast = 0;
    while(yearsPast < years) {
        discoveries++;
    }
}
```

Abstract Data Types Main

- Including the Header allows us access to the functions
- The main doesn't know how they're implemented
- We can just trust that the functions do what they say

```
#include "ship.h"

int main (void) {
    ship myShip = newShip("Enterprise");
    myShip = voyage(myShip, 5);
}
```

Programming is much more than just code

COMP1511 Programming Skills Topics in the order they were taught

- History of Computing
- Problem Solving
- Code Style
- Code Reviews
- Debugging
- Theory of a Computer
- Professionalism

Problem Solving

Approach Problems with a plan!

- Big problems are usually collections of small problems
- Find ways to break things down into parts
- Complete the ones you can do easily
- Test things in parts before moving on to other parts

Code Style

Half the code is for machines, the other half for humans

- Remember . . . readability == efficiency
- Also super important for working in teams
- It's much easier to isolate problems in code that you fully understand
- It's much easier to get help if someone can skim read your code and understand it
- It's much easier to modify code if it's written to a good style

Code Reviews

No one has to work without help

- If we read each other's code . . .
- We learn more
- We help each other
- We see new ways of approaching things
- We are able to teach (which is a great way to cement knowledge)

Debugging

The removal of bugs (errors)

- Syntax errors are code language errors
- Logical errors are the code not doing what we intend

- The first step is always: Get more information!
- Once you know exactly what your program is doing around a bug, it's easier to fix it
- Separate things into their parts to isolate where an error is
- Always try to remember what your intentions are for your code rather than getting bogged down

Professionalism

There's so much more to computing than code

- What's the most important thing for a Software Professional?
- It's not coding!
- It's caring about what you do and the people around you!
- Even in terms of pure productivity, it's going to get more work done long term than being good at programming
- If you care about your work, you will be fulfilled by it
- If you care about your coworkers you'll teach and learn from them and you'll all grow into a great team

Break Time

Human memory is based on active recall

- You can store something in your long term memory by reminding yourself of it repeatedly
- Active recall means using, not just reading
- Link your memory to things you already know (use examples in your revision code that are things you know well)
- Get some exercise! Active blood flow, even just a bit of walking, helps the brain

What's in the Exam?

The Exam

10th May in one of two 3 hour sessions

- You will receive an email this Wednesday to select a session
- If you are allocated a session and have a serious reason why you can't attend that session, email cs1511@cse.unsw.edu.au after allocations

- Completed on a lab computer under exam conditions
- No external materials allowed
- Your Week 10 labs will show you what the exam environment is like

The Exam Format

The following details might change, but only slightly

- 30 minutes of **theory** questions
- Around 15 theory questions
- During the first 35 minutes, you will not have access to a code editor or compiler
- 2.5 hours of **practical** questions
- Around 8 practical questions
- Practical questions will involve actual programming

Exams - Marc's tips

How to survive an exam

- Bring a pen(s) possibly with multiple colours to draw diagrams
- Bring a (transparent) water bottle . . . dehydration affects your brain
- Eat a decent meal before the exam. Blood sugar also affects your brain, especially in a stressful situation
- Remember the C Reference Sheet is available in the exam

I'd say *"chill out, this isn't a big deal"* but no one will believe me

Theory Questions

Quick Questions, mostly in the same format:

- Here's some code
- It compiles like this
- Here's the command to run it
- What is the output?

- These questions will be about whether you understand core coding concepts and the C programming language
- Your answers will either be multiple choice or short answers

Theory Questions - Marc's tips

How to maximise marks in a high speed theory test

- Read through them all fast before answering
- Skim quickly and answer the ones you definitely know
- Then go back to the ones that take some time to think about
- Don't get stuck . . . If something is going to take you some serious time to work out, then move on
- Prioritise your time! Get the easy marks, then spend time on the ones you're reasonably sure of. If you're not sure of something then don't let it eat your time!

Practical Questions

Less questions, more time

- Two types of hurdle questions
  - Array Hurdle
  - Linked List Hurdle
- Questions are similar to the Weekly Tests
- Some will have provided code as frameworks
- Each question will need to be written, compiled and tested
- You will have access to an autotest (but it's just a test!)
- There will be no specific style marking, so you don't need to explain your code in comments

Practical Questions - Marc's tips

Solving Problems under pressure

- Read all the questions before starting
- Pick the easy ones as you read. Most likely the earlier questions
- Make sure you're covering both hurdles
- Don't rush! A couple of minutes thinking and writing a diagram might be much faster than smashing out code that doesn't answer the question
- Remember your lab exercises! Debugging and testing will be important here

Questions 1-2

Basic C Programming - similar to Weekly Test question 1

- Create C programs
- Use variables (ints and doubles)
- scanf and printf
- if statements and loops
- Read command line arguments (possibly convert to ints and doubles)
- Use arrays of ints/doubles

Example Question 1

Read values, perform some kind of computation on the input

Eg: Read two numbers from a user. Print out a line for every even number between them. That line should show the number and its square

```
% ./squares
Enter lower: 12
Enter upper: 17
14 196
16 256
```

Example Question 2

Perform some computation from command line arguments

Eg: Your program will be given 1 or more command line arguments which you can assume are all integers. Calculate the sum of their squares and print it.

```
% ./sumSquares 5 4 3
50
```

Questions 3-4

More advanced C - similar to Weekly Test question 2

- Use fgetc and fgets to read characters and lines
- Read until end-of-input using scanf, fgets, fgetc
- Use string arrays
- Work with strings
- Work with linked lists

Questions 5-6

Even Harder C - similar to Weekly Test question 3

- Use malloc() and free()
- Manipulate linked lists (adding and removing items etc)
- Manipulate strings

Questions 7+

Challenge Questions for people chasing HDs

- Everything taught in the course might be in these questions
- Think Challenge Exercises, but not the outright silly ones

What to study

A little preparation goes a long way

- The basics are important!
- A basic knowledge of all topics is better than an extreme level of knowledge in just one
- Know how to use both an array and a linked list (the hurdles)
- Try some revision questions from the Tutorials or Labs while putting yourself under a stressful time limit
- The revision exercises on the course webpage are also very useful

How important are different topics?

Important

- Variables, If, Looping, Functions, Arrays, Linked Lists, Characters and Strings

Things that you might need to understand the important topics

- Pointers, Structs, Memory

Stretch Goals

- Abstract Data Types
- Multi-file programs will not be tested in the exam

Exam Marking

Most of the marking will be automated

- Make sure your input/output format matches the specification
- All answers will also be hand marked
- Minor errors, like a typo in an otherwise correct solution, will only result in a very small loss of marks
- Results should be ready by approximately the 20th May

Special Consideration and Supplementary Exam

- If you attend the exam, it's an indication that you are well enough to sit the exam
- If you are not well enough to sit the exam, apply for Special Consideration and do not attend the exam
- If you become sick during the exam, ask the exam supervisor for assistance and talk to the Lecturer(s).
- A supplementary exam will be held between the 27th and 31st May. If you think you will need to sit this exam, make sure you are available.

So, you're programming now…

Where do we go from here?

- There's so much you can do now
- But there's also so much to learn
- Computer Science has more to offer than any one person can learn in a lifetime
- It will never leave you without something you could discover
- It's up to you to decide what you want from it and how much of your life you want to commit to it
- Enjoy yourselves, keep working as hard as you can and I hope to bask in your future glory

What did we learn this term?