COMP1511 - Programming Fundamentals

Term 2, 2019 - Lecture 8

What did we learn Yesterday?

Arrays

- Arrays don't change size
- 2D Arrays

Assignment 1 - CS Paint

- A simulation of a painting program
- Using a 2D array
- Reading in commands from standard input

What are we covering today?

Debugging

- What's a bug?
- How do we find them and remove them?

Functions

- We've seen our main function . . . now we introduce other functions
- We'll use both arrays and functions in a program

Debugging

It’s going to take up most of your time as a programmer!

- What is a bug?
- Different types of bugs
- How to find bugs

Debugging is the process of finding and removing software bugs

What is a Software Bug?

Errors in code are called “bugs”

- Something we have written (or not written) in our code
- Any kind of error that stops the program from running as intended

Two most common types of bugs

- Syntax Errors
- Logical Errors

Syntax Errors

C is a specific language with its own grammar

- **Syntax** - the precise use of a language within its rules
- C is very much more specific than most human languages

- Slight mistakes in the characters we use can result in different behaviour
- Some syntax errors are obvious and your compiler will find them
- Some are more devious and the error message will be the consequence of the bug, rather than the bug itself

Logical Errors

We can write a functional program that still doesn't solve our problem

- Logical errors can be syntactically correct
- But the program might not do what we intended!

Human error is real!

- Sometimes we read the problem specification wrongly
- Sometimes we forget the initial goal of the program
- Sometimes we solve the wrong problem
- Sometimes we forget how the program might be used

How do we find bugs?

Sometimes they find us . . .

- **Compilers** can catch some syntactical bugs
- We'll need to learn how to use compilers to correct our code
- Code Reviews and pair programming help for logical bugs
- **Testing** is always super important!
- Learning how to test is a very valuable skill

Using our Compiler to hunt Syntax Bugs

The Compiler can be trusted to understand the language better than us

- The simplest thing we can do is run dcc and see what happens

What to do when dcc gives you errors and warnings

- Always start with the first error
- Subsequent errors might just be a consequence of that first line
- An error is the result of an issue, not necessarily the cause
- At the very least, you will know a line and character where something has gone wrong

Solving Compiler Errors

Compiler Errors will usually point out a syntax bug for us

- Look for a line number and character (column) in the error message
- Sometimes knowing where it is is enough for you to solve it

- Read the error message and try to interpret it
- Remember that the error message is from a program that reads code
- It might not make sense initially!
- Sometimes it's an expectation of something that's missing
- Sometimes it's confusion based on something being incorrect syntax

Let's look at some code and fix some bugs

debugThis.c is a program with some bugs in it . . .

What errors did we find?

Just focusing on fixing compiler errors, let's read and fix some code

What did we discover? *(spoilers here . . . try debugging before reading this slide!)*

- Single = if statement.
  - = is and assignment of a value
  - == is a relational operator
- An extra bracket causes a lot of issues (and a very odd error message)
- Scanf not pointing at a variable

Testing

We'll often test parts of all of our code to make sure it's working

- Simple - Run the code
- Try different types of inputs to see different situations (autotest!)
- Try using inputs that are not what is expected

How do you know if the tests are succeeding?

- Use output to show information as the program runs
- Check different sections of the code to see where errors are

Simple Testing

Let's use a good process here that we can apply to all code testing

- Write your program to give you a lot of information
- Test with intention. It's valuable to test with specific goals
- Be able to find out what the code is doing at different points in the code
- Be able to separate different sections of code

Finding a needle in a haystack gets easier if you can split the haystack into smaller parts

Let’s try some information gathering

Some of the tricks we’ll use, continuing with our debugThis.c

- How is it meant to run?
- Decide on some ranges of inputs to test
- Modify the code to give useful information while it’s running

What did we test?

What techniques did we use?

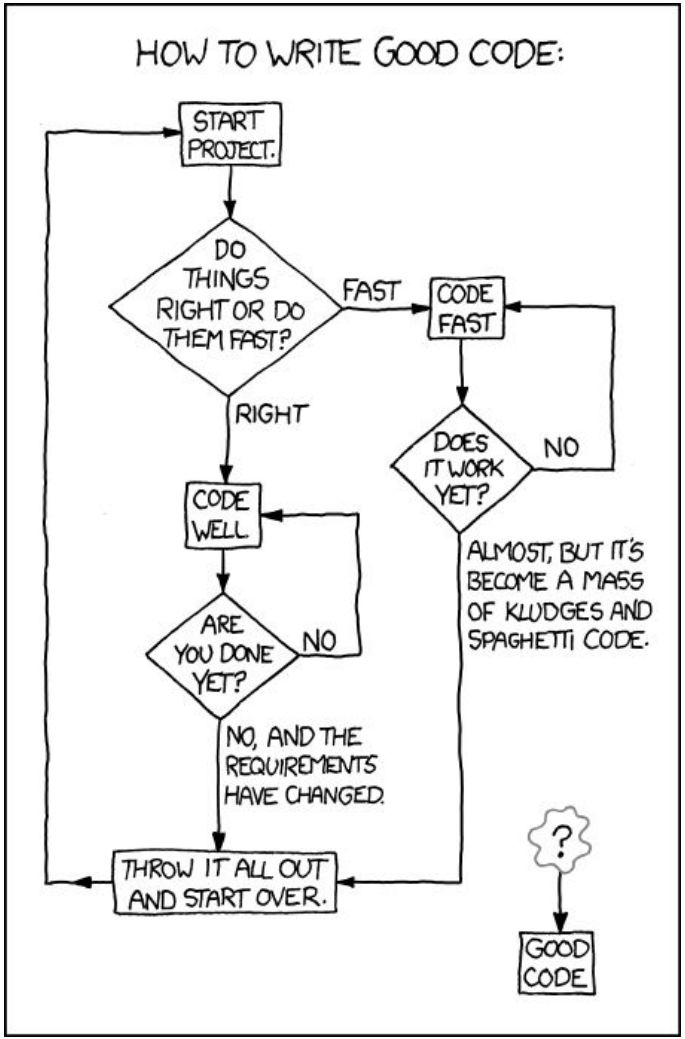
- Try different input ranges, including 0 and negative numbers
- Try outputting x and y values to make sure they're working
- Try outputting loop information so that we can see our structure

When we do good testing, we will be able to find our logical errors even if the code is syntactically correct

Break Time

Writing "Good" code is an ongoing journey

- It's never really going to be complete
- We're always going to be fixing bugs
- Getting comfortable with cleaning up your code and rewriting things is important!



https://xkcd.com/844/

Functions

Let's look at functions

- We've already been using some functions!
- **main** is a function
- **printf** and **scanf** are also functions

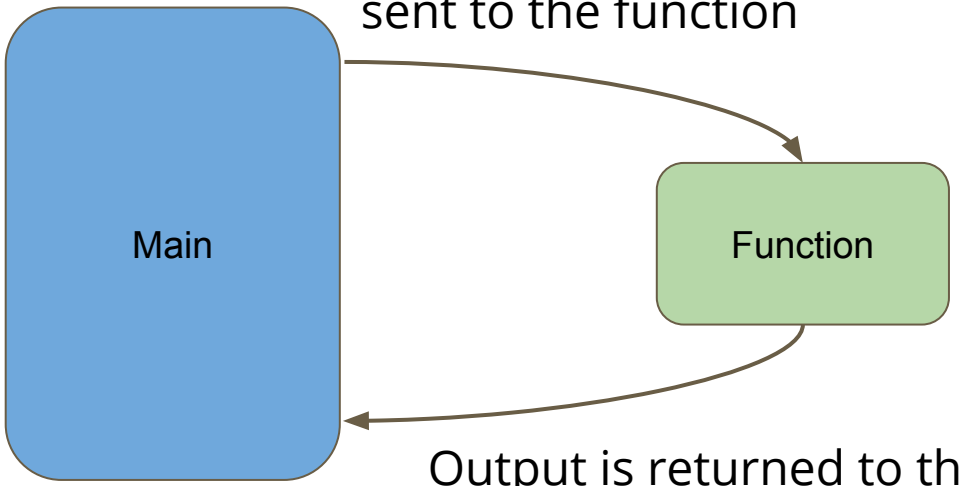
What is a function?

- A separate piece of code identified by a name
- It has inputs and an output
- If we "call" a function it will run the code in the function

Functions

How do they work?

Main calls the Function, asking it to run. Input is sent to the function



Function runs its code using the input

Output is returned to the Main. Main continues from where it was

Function Syntax

We write a function function with (in order left to right):

- An output (known as the function's type)
- A name
- Zero or more input(s) (also known as function parameters)
- A body of code in curly brackets

```
// a function that adds two numbers together
int add (int a, int b) {
    return a + b;
}
```

Return

An important keyword in a function

- Return will deliver the output of a function
- Return will also stop the function running and return to where it was called from

How is a function used?

Given the existence of the function . . .

- We can use a function by calling it by name
- And providing it with input(s) of the correct type(s)

```
// using the add function
int main (void) {
    int firstNumber = 4;
    int secondNumber = 6;
    int total;

    total = add(firstNumber, secondNumber);
    return 0;
}
```

Compilers and Functions

How does our main know what our function is?

- A compiler will process our code, line by line, from top to bottom
- If it has seen something before, it will know its name

```
// An example using variables
int main (void) {
    // declaring a variable means it's usable later
    int number = 1;

    // this next section won't work because the compiler
    // doesn't know about otherNumber before it's used
    int total = number + otherNumber;
    int otherNumber = 5;
}
```

Functions and Declaration

We need to declare a function before it can be used

```
// a function can be declared without being fully
// written (defined) until later
int add (int a, int b);

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

// the function is defined here
int add (int a, int b) {
    return a + b;
}
```

Void Functions

We can also run functions that return no output

- We can use a void function if we don't need anything back from it
- The return keyword will be used without a value in a void function

```
// a function of type "void"
// It will not give anything back to whatever function
// called it, but it might still be of use to us
void add (int a, int b) {
    int total = a + b;
    printf("The total is %d", total);
    return;
}
```

Let’s play a modified game of Snap

Snap is a simple card game

- We play cards one after the other
- If we play a card that’s the same as the previous, we call out "Snap!"

Our “Snap!” program will be a little different

- We’re going to write a program that takes input numbers one at a time
- It will then call out "Snap!" if it’s seen your input before

What features will we start with

- A constant that tells us how many turns are in the game
- A loop that runs once per game turn
- Some useful messages to our user so they know what's happening
- Store the information the user gives us in an array

Here's one I prepared earlier . . .

Starting Code

```c
#define NUM_TURNS 10

int main (void) {
    int prevNums[NUM_TURNS] = {0};

    printf("Welcome to my game of Snap!\n");
    printf("You can type in any number until %d turns are over.\n", NUM_TURNS);
    printf("If I've seen that number before, I will say SNAP!\n");

    // the main game loop. Each turn of the game is one iteration
    int i = 0;
    while (i < NUM_TURNS) {
        printf("Please type in a number: ");
        int input;
        scanf("%d", &input);
        prevNums[i] = input;
        i++;
    }
```

What will we add to this?

Checking if a number is in an array

- Loop through the array
- Check the number we want against the number in the array
- If it’s true, then we know it’s in the array

Let's write this in a function

What does it need to tell us?

- Output type

What does it need to know to be able to tell us this?

- Input Parameters

What does it do?

- Name

NumberCheck Function

A function for saying Snap! if a number has been found in an array

```
void numberCheck(int prevNumbers[], int number) {
    int i = 0;
    // print "Snap!" if the number has been seen before
    while (i < NUM_TURNS) {
        if (number == prevNumbers[i]) {
            printf("Snap!\n");
        }
        i++;
    }
    return;
}
```

Using this Function in our Snap Program

We need to do the following:

- Declare the function before our main
- Add the function code to our C file
- Call the function in our main

Adding a Function

```
void numberCheck(int prevNumbers[], int number);

int main (void) {
    // main function not shown on this slide
}

void numberCheck(int prevNumbers[], int number) {
    int i = 0;
    // print "Snap!" if the number has been seen before
    while (i < NUM_TURNS) {
        if (number == prevNumbers[i]) {
            printf("Snap!\n");
        }
        i++;
    }
    return;
}
```

Testing!

What kind of tests can we use to check this code?

- A few different numbers
- Pairs of the same numbers to see if it snaps
- All of the same number
- The number 0?

Even if we're free of syntax errors, we might still have logical errors!

How do we fix these errors?

Identifying bugs is good . . . fixing them is even better

- See if you can fix the duplicate snaps issue
- Also the false positives with the number 0
- There's a nifty solution that will be able to solve both of these at once!
- It's all about stopping the loop early . . .

What did we learn today?

Debugging

- Different types of bugs (software errors)
- Syntax and Logical Errors
- Using testing to find bugs

Functions

- Separate code we can write and call
- Has a type (it’s output)
- Has input parameters