COMP1511 - Programming Fundamentals

Week 5 - Lecture 9

What are we covering today?

Pointers recap

- Finishing the example from last week

Debugging

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

Characters

- Variables for letters

Recap - Pointers and Memory

What is a pointer?

- It's a variable that stores the address of another variable of a specific type
- We call them pointers because knowing something's address allows you to "point" at it

Why pointers?

- They allow us to pass around the address of a variable instead of the variable itself

Using Pointers

Pointers are like street addresses . . .

- We can create a pointer by declaring it with a ***** *(like writing down a street address)*
- If we have a variable *(like a house)* and we want to know its address, we use **&**

```
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
```

Using Pointers

If we want to look at the variable that a pointer "points at"

- We use the ***** on a pointer to access the variable it points at
- Using the address analogy, this is like navigating to the house at that address and looking inside the house

```
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
printf("The value of the variable at %p is %d", ip, *ip);
```

pointer self. just address

go to address find the value

Pointers in Functions

We'll often use pointers as input to functions

- Pointers give a function access to a variable that's in memory
- They also allow us to affect multiple variables instead of only having one output

```
void swap_nums(int *num1, int *num2) {
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Pointers and Arrays

These are very similar

- Arrays are sections of memory that contain multiple identical variables
- The variable of the array itself stores the memory address of the start of the array
- Pointers are also memory addresses
- This gives both pointers and arrays access to memory

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

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 is your teaching staff writing these tests!)
- 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

Characters

We've only used ints and doubles so far

- We have a new type called **char**
- Characters are what we think of as letters, like 'a', 'b', 'c' etc
- They can also represent numbers, like '0', '1','2' etc
- They are actually **8 bit** integers!
- We use them as characters, but they're actually encoded numbers
- ASCII (American Standard Code for Information Interchange)
- We will not be using **char** for individual characters, but we will in arrays

编码数字

ASCII and Characters as numbers

We make use of ASCII, but we don't need to know it

- ASCII specifically uses values 0-127 and encodes:
  - Upper and Lower case English letters
  - Digits 0-9
  - Punctuation symbols
  - Space and Newline
  - And more . . .
- It's not necessary to memorise ASCII, rather it's important to remember that characters can be treated like numbers sometimes

Characters in code

```c
#include <stdio.h>

int main (void) {
    // we're using an int to represent a single character
    int character;
    // we can assign a character value using single quotes
    character = 'a';
    // This int representing a character can be used as either
    // a character or a number
    printf("The letter %c has the ASCII value %d.\n", character,
character);
    return 0;
}
```

Note the use of %c in the printf will format the variable as a character

Helpful Functions

`getchar()` is a function that will read a character from input

- Reads a byte from standard input
- Usually returns an int between 0 and 255 (ASCII code of the byte it read)
- Can return a -1 to signify end of input, EOF (which is why we use an int, not a char)
- Sometimes **`getchar`** won't get its input until enter is pressed at the end of a line

`putchar()` is a function that will write a character to output

- Will act very similarly to `printf("%c", character);`

Use of getchar() and putchar()

```c
// using getchar() to read a single character from input
int inputChar;
printf("Please enter a character: ");
inputChar = getchar();
printf("The input %c has the ASCII value %d.\n", inputChar, inputChar);

// using putchar() to write a single character to output
putchar(inputChar);
```

scanf

printf

Invisible Characters

There are other ASCII codes for “characters” that can’t be seen

- Newline(`\n`) is a character
- Space is a character
- There’s also a special character, EOF (End of File) that signifies that there’s no more input
- EOF has been `#define`d in `stdio.h`, so we use it like a constant
- We can signal the end of input in a Linux terminal by using Ctrl-D

Working with multiple characters

We can read in multiple characters (including space and newline)

This code is worth trying out . . . you get to see that space and newline have ASCII codes!

```
// reading multiple characters in a loop
int readChar;
readChar = getchar();
while (readChar != EOF) {
    printf(
        "I read character: %c, with ASCII code: %d.\n",
        readChar, readChar
    );
    readChar = getchar();
}
```

More Character Functions

<ctype.h> is a useful library that works with characters

- `int isalpha(int c)` will say if the character is a letter
- `int isdigit(int c)` will say if it is a numeral
- `int islower(int c)` will say if a character is a lower case letter
- `int toUpper(int c)` will convert a character to upper case

- There are more! Look up `ctype.h` references or `man` pages for more information