

# ENGG1003 - Friday Week 2

What does = *Really* do?  
More Flow Control

Brenton Schulz

University of Newcastle

March 11, 2019

# Eclipse Che Feedback

- ▶ Thanks to the 50-odd students who tested it!
- ▶ Turns out Che (specifically: Docker) has a limit of 30 *concurrent* users
- ▶ Two servers are now online:
  - ▶ `che1.vk2dds.net:8080`
  - ▶ `che2.vk2dds.net:8080`
  - ▶ Do we need a 3rd?
- ▶ They do *not* share data
- ▶ Use git to move projects between them
  - ▶ Documentation is in progress...
- ▶ Log in to one, I will use it during the lecture

# Who Should Pass This Subject?

- ▶ Everything is new, I will get things wrong, but:
- ▶ My *intention* is that if:
  - ▶ You attend (or watch) all lectures
  - ▶ You attend all labs
  - ▶ You read all course material
  - ▶ You attempt problems *in your own time*
  - ▶ You follow all instructions in lab notes
  - ▶ You chase up with demonstrators, me, textbooks, YouTube, etc any concepts you don't understand
- ▶ ...then you should confidently pass.
- ▶ tl;dr You should have to work hard to earn a fail

# What Even *is* a Pass?

From the course outline:

Satisfactory standard indicating an adequate knowledge and understanding of the relevant materials; demonstration of an adequate level of academic achievement; satisfactory development of skills; and achievement of all learning outcomes.

# What is a High Distinction?

Outstanding standard indicating comprehensive knowledge and understanding of the relevant materials; demonstration of an outstanding level of academic achievement; mastery of skills; and achievement of all assessment objective.

# Grade Expectations

- ▶ You get a “pass” if you learn enough that you won't *obviously fail* later courses
- ▶ You get a HD if you thoroughly understood *everything* and can recall and apply any relevant piece of course content to a problem, showing a sound level of engineering judgement in doing so
- ▶ More than 50% of the cohort being graded at D or HD level is cause for change

# Revision

- ▶ In mathematics, an *iterative* (or *recursive*) equation is written:

$$x_n = x_{n-1} + 1 \quad (1)$$

- ▶ In programming, the change with time is implicit with program execution when we write:

$$x = x + 1; \quad (2)$$

- ▶ The `=` operator is `assignment` and `overwrites` (destroys) the variable's previous value

# Fibonacci Sequence

- ▶ The *Fibonacci Sequence* is the list of numbers, starting with 0 and 1, where each number is the sum of the two which came before it
- ▶ ie: 0, 1, 1, 2, 3, 5, 8, 12, 21, ...
- ▶ This sequence has interesting properties, eg:
  - ▶ Its members appear in nature *for some reason*
  - ▶ The ratio of successive numbers converges towards the golden ratio:  $\phi \approx 1.618$
  - ▶ This has applications in art. eg: 16:10 screen ratio is 1.6 (16:9 can bite my shiny metal...)



# Fibonacci Sequence

- ▶ Mathematically, we can write this as a list of numbers,  $x_0, x_1, x_2, x_3, \dots, x_n$ , where:

$$x_n = x_{n-1} + x_{n-2} \quad (3)$$

and:

$$x_0 = 0 \quad (4)$$

$$x_1 = 1 \quad (5)$$

# Arithmetic Sequences in General

- ▶ Sequences seen in HSC mathematics are a subset of *constant-recursive sequences*
- ▶ *Linear* sequences have the form:

$$x_n = b_1x_{n-1} + b_2x_{n-2} + \dots + b_Nx_{n-N} \quad (6)$$

- ▶ Where  $b_1, b_2$ , etc are constant real numbers
- ▶ ie: Each number,  $x_n$ , is a *linear combination* of the  $N$  numbers before it
- ▶ The Fibonacci sequence is Equation (6) with  $b_1 = 1, b_2 = 1, N = 2, x_1 = 0$ , and  $x_2 = 1$ .

# Fibonacci Sequence

- ▶ **Task:** Write a C program which outputs the Fibonacci Sequence for all integers small enough to fit into an `int`.
- ▶ Lets break this into two problems:
  1. Calculate the Fibonacci Sequence
  2. Worry about the stop condition
- ▶ Always try to break programming problems down into small chunks
- ▶ Real-world problems are too difficult to complete “all in one go”

# Fibonacci Sequence

- ▶ How do we calculate the Fibonacci Sequence?
- ▶ Note that we need to keep track of *three* numbers:
  - ▶ The next number,  $x_n$
  - ▶ The previous two numbers,  $x_{n-1}$  and  $x_{n-2}$
- ▶ Lets also remember  $n$
- ▶ I will use these variable names:

```
1 int n;  
2 int xN;    // x N  
3 int xNm1;  // N minus 1  
4 int xNm2;  // N minus 2
```

# Fibonacci Sequence

- ▶ Aside: We might want to *remember* the whole sequence
- ▶ This would require all numbers to be stored as unique variables
- ▶ Declaring hundreds (or millions) of variables is impractical
- ▶ The concept of *arrays* will be introduced later to deal with this

# Fibonacci Sequence

- ▶ Lets sketch out what happens to these variables *by hand*
- ▶ Start at  $n=2$ , as that is the first unknown

n	xNm2	xNm1	xN
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5

- ▶ See the pattern? Numbers shift diagonally down to the left.

# Fibonacci Sequence

- ▶ Each time a new number is calculated, what happens to the variables?
- ▶ All 3 variables change, the order in which they change is **crucial**:

```
1 xN = xNm1 + xNm2; // Calculate next value
2 xNm2 = xNm1; // Move old values "down the chain"
3 xNm1 = xN;
```

- ▶ With = the old *lvalue* is lost
- ▶ Note that the *oldest* value is overwritten first
  - ▶ It is the one which is no longer needed

# L and R-Values

- ▶ Oops, I used new jargon...
- ▶ An *lvalue* is something that goes to the left of an = operator
- ▶ Likewise, the right side is an *rvalue*
- ▶ Not everything is a valid L- or R-value
- ▶ Lvalues have to be variables
- ▶ The following generate errors:
  - ▶ `4 = x;`
  - ▶ `rand() = 2;`



# Fibonacci Sequence

- Lets sketch some pseudocode to calculate the first 20 or so values:

```
BEGIN
  int xNm2 = 0
  int xNm1 = 1
  int xN
  int n = 2
  WHILE n < 20
    xN = xNm1 + xNm2
    PRINT xN
    n = n + 1
    xNm2 = xNm1
    xNm1 = xN
  ENDWHILE
END
```

# Fibonacci Sequence

...and convert it to C:

```
BEGIN
  int xNm2 = 0
  int xNm1 = 1
  int xN
  int n = 2
  WHILE n < 20
    xN = xNm1 + xNm2
    PRINT xN
    n = n + 1
    xNm2 = xNm1
    xNm1 = xN
  ENDWHILE
END
```

```
int main() {
  int xNm2 = 0;
  int xNm1 = 1;
  int xN;
  int n = 1;
  while(n < 20) {
    xN = xNm1 + xNm2;
    printf("%d\n", xN);
    n++;
    xNm2 = xNm1;
    xNm1 = xN;
  }
}
```

# Fibonacci Sequence

For everyone on Che, copy this link:  
`https://github.com/bschulznewy/  
fibonacci.git`  
and import into Che as I demonstrate

# Fibonacci Sequence

- ▶ Does the code work?
- ▶ Compare with: `https://www.wolframalpha.com/input/?i=first+20+fibonacci+sequence`
- ▶ What about the 2nd requirement in the original problem?
- ▶ How to tell if next value *exceeds* an `int`?

# Fibonacci Sequence

- ▶ There are a few solutions:
  - ▶ Calculate  $xN$ , see if result *overflowed*
  - ▶ Do calculation as unsigned int (or long) and compare with INT\_MAX
    - ▶ INT\_MAX is defined in limits.h
    - ▶ #include <limits.h> if you want to use it
    - ▶ It includes the line:

```
1 #define INT_MAX 2147483647
```
    - ▶ We will learn about #define later
- ▶ In this context we can't run with a fixed iteration limit as that is “unknown” until the program is executed

# Fibonacci Sequence

- ▶ If overflow occurs when using `int` then the result of a calculation which *should* be positive will be negative
- ▶ Lets test for overflow with:

```
1 if (xNm1 + xNm2 > 0) {  
2     // Do an iteration  
3 }
```

# Fibonacci Sequence

```
int main() {  
    int xNm2 = 0, xNm1 = 1;  
    int xN;  
    int n = 1;  
    while(xNm1 + xNm2 > 0) {  
        xN = xNm1 + xNm2;  
        printf("%d\n", xN);  
        n++;  
        xNm2 = xNm1;  
        xNm1 = xN;  
    }  
}
```

- **NB:** If *optimisation* is enabled the calculation of  $xNm1 + xNm2$  will only occur once

# Fibonacci Sequence

Or, pre-testing overflow with unsigned int:

```
(unsigned int) xNm1 + (unsigned int) xNm2 <
2147483647u)
```

Lets try this one in Che...



# DO ... WHILE

- ▶ Same as WHILE except executes *at least once*
- ▶ The condition is tested at the end
- ▶ Loops repeats if condition is TRUE
- ▶ Pseudocode syntax:

```
DO
    stuff
WHILE condition
```

- ▶ C syntax:

```
1 do {
2     // do stuff
3 } while(condition);
```

# DO ... WHILE

## ► A toy example in C:

```
1 int main() {  
2     int x = 0;  
3     do {  
4         x = x - 1;  
5     } while (x > 0);  
6     return 0;  
7 }
```

# DO ... WHILE

## ► A slightly less toy example:

```
1 #include <stdio.h>
2 int main() {
3     int x;
4     do {
5         printf("Enter an integer: ");
6         scanf("%d", &x);
7         if(x%2==0)
8             printf("%d is even\n", x);
9         else
10            printf("%d is odd\n", x);
11    } while(x >= 0);
12    return 0;
13 }
```

# DO ... WHILE

- ▶ **NB:** The previous example had:

```
1     if(x%2==0)
2         printf("%d is even\n", x);
3     else
4         printf("%d is odd\n", x);
```

- ▶ The { } block is optional if *only one statement* is after an `if()`, `while()`, etc
- ▶ I omitted it to reduce line count so that the code would fit on the slide

# DO ... WHILE is Optional

- ▶ It is never *absolutely necessary*
- ▶ But sometimes it is easier or neater

while()

```
int x = 1;
while(x >= 0) {
    printf("Enter an integer: ");
    scanf("%d", &x);
    if(x%2==0)
        printf("%d is even\n", x);
    else
        printf("%d is odd\n", x);
}
```

do while();

```
int x; // Uninitialised
do {
    printf("Enter an integer: ");
    scanf("%d", &x);
    if(x%2==0)
        printf("%d is even\n", x);
    else
        printf("%d is odd\n", x);
} while(x >= 0);
```

# FOR Loops

- ▶ A FOR loop loops a given number of times
- ▶ Typically used when the number of loop repeats is known *before* entering the loop
  - ▶ Repeat count could be “hard coded” as a number
  - ▶ Could also be a variable
- ▶ Can be easier to read than WHILE
- ▶ Example pseudocode syntax:

```
FOR x = 1 to 10  
    Do something ten times  
ENDFOR
```

- ▶ The *loop variable* is automatically incremented

# FOR Loops in C

- ▶ The C FOR loop syntax is:

```
1 for( initial ; condition ; increment ) {  
2     // Loop block  
3 }
```

- ▶ Where:

- ▶ `initial` is a statement executed *once*
- ▶ `condition` is a statement executed and tested *after* every loop iteration
- ▶ `increment` is a statement executed *after* every loop iteration, but *before* the `condition` is tested

# FOR Loops in C

```
1 for( x = 0 ; x < 10 ; x++ ) {  
2     printf("%d\n", x);  
3 }
```

- ▶ Run this code
- ▶ Observe that:
  - ▶ 0 is printed
  - ▶ 10 is **not** printed
  - ▶ x increments automatically



# FOR Example 1 - Factorials

- ▶ Use FOR to count from 2 to our input number
- ▶ Keep a running product as we go

```
BEGIN
  INPUT x
  result = 1
  FOR k = 2 TO x
    result = result * k
  ENDFOR
END
```

- ▶ Is this algorithm robust? What happens if:
  - ▶  $x = -1$
  - ▶  $x = 1$
  - ▶  $x = 0$  (**NB:**  $0! = 1$  because *maths*)

# BREAK Statements

- ▶ Sometimes you want to exit a loop *before* the condition is re-tested
- ▶ The flow-control mechanism for this is a BREAK statement
- ▶ If executed, the loop quits
- ▶ BREAKs typically go inside an IF to control their execution

## FOR Example 2

- ▶ Two equivalent ways to implement the  $\cos()$  series from before are:

**NB:**  $|tmp|$  means “absolute value of tmp”.

```
BEGIN
  INPUT x
  sum = 0
  FOR k = 0 to 10
    tmp =  $\frac{(-1)^k x^{2k}}{(2k)!}$ 
    sum = sum + tmp
    IF |tmp| < 1e-6
      BREAK
    ENDIF
  ENDWHILE
END
```

```
BEGIN
  INPUT x
  tmp = 1
  k = 0
  sum = 0
  WHILE (k<10) AND (|tmp|>1e-6)
    tmp =  $\frac{(-1)^k x^{2k}}{(2k)!}$ 
    sum = sum + tmp
    k = k + 1
  ENDWHILE
END
```

# FOR Loops in C (Advanced)

- ▶ `for()` syntax allows multiple items in the initial / condition / increment sections
- ▶ Separate statements with commas
- ▶ eg:

```
1 int x, y=10;  
2 for( x = 0 ; x < 10 ; x++, y++ ) {  
3     printf("x: %d y: %d\n", x, y);  
4 }
```

- ▶ This increments both `x` and `y` but only `x` is used in the condition and `y` is not initialised

# Loop continue Statements

- ▶ A `continue` causes execution to jump back to the loop start
- ▶ The *condition* is tested before reentry
- ▶ eg, run this in the Che debugger:

```
1 int x;  
2 for(x = 0; x < 10; x++) {  
3     if(x%2 == 0)  
4         continue;  
5     printf("%d is odd\n");  
6 }
```

- ▶ (Not the best example but gets the point across)

# break and continue

- ▶ Some programmers claim that `break` and `continue` are “naughty”
- ▶ Well, yes, but actually no
- ▶ They can make your code needlessly complicated
- ▶ They might make is simpler
- ▶ It is up to you to judge
- ▶ As engineers you shouldn't follow strict rules
- ▶ Always try to choose the best tool for the job

# GOTO

- ▶ There exists a GOTO flow control mechanism
  - ▶ Sometimes also called a *branch*
- ▶ It “jumps” from one line to a different line
  - ▶ An ability some consider to be unnatural
- ▶ It exists for a purpose
- ▶ That purpose does not (typically) exist when writing C code
  - ▶ C *supports* a `goto` statement
  - ▶ It results in “spaghetti code” which is hard to read
  - ▶ Don't use it in ENGG1003
- ▶ You *must* use branch instructions in ELEC1710

# Loose End Increment Example

```
1 #include <stdio.h>
2 int main() {
3     int x = 0;
4     int y = 0;
5     int z = 0;
6     y = ++x + 10;
7     printf("Pre-increment: %d\n", y);
8     y = z++ + 10;
9     printf("Post-increment: %d\n", y);
10    return 0;
11 }
```

Listing 1: increment.c

Pre/post-inc/decrements have many applications,  
more details in coming weeks.



# Binary Nomenclature

- ▶ The value range is a result of the underlying binary storage mechanism
- ▶ A single binary digit is called a *bit*
- ▶ There are 8 bits in a *byte*
- ▶ In programming we use the “power of two” definitions of kB, MB, etc:
  - ▶ 1 kilobyte is  $2^{10} = 1024$  bytes
  - ▶ 1 Megabyte is  $2^{20} = 1048576$  bytes
  - ▶ 1 Gigabyte is  $2^{30} = 1073741824$  bytes
  - ▶ (Advanced) These numbers look better in hex: 0x3FF, 0xFFFFF, etc.

# Binary Nomenclature

- ▶ Observe that kilobyte, Megabyte, Gigabyte, etc use scientific prefixes
- ▶ These *normally* mean a power of 10:
  - ▶ kilo- =  $10^3$
  - ▶ Mega- =  $10^6$
  - ▶ Giga- =  $10^9$
  - ▶ ...etc (see the inside cover of a physics text)
- ▶ Computer science stole these terms and re-defined them

# Binary Nomenclature

- ▶ This has made some people *illogically angry*
- ▶ Instead, we can use a more modern standard:
  - ▶  $2^{10}$  bytes = 1 kibiByte (KiB)
  - ▶  $2^{20}$  bytes = 1 Mebibyte (MiB)
  - ▶  $2^{30}$  bytes = 1 Gibibyte (GiB)
  - ▶ ...etc
- ▶ Generally speaking, KB (etc) implies:
  - ▶ powers of two to *engineers*
  - ▶ powers of ten to *marketing*
    - ▶ The number is smaller
    - ▶ Hard drive manufacturers, ISPs, etc like this

# Unambiguous Integer Data Types

- ▶ Because the standard `int` and `long` data types don't have fixed size unambiguous types exist
- ▶ Under OnlineGDB (ie: Linux with `gcc`) these are defined in `stdint.h` (`#include` it)
- ▶ You will see them used commonly in embedded systems programming (eg: Arduino code)
- ▶ The types are:
  - ▶ `int8_t`
  - ▶ `uint8_t`
  - ▶ `int16_t`
  - ▶ `...etc`

# Code Blocks in C

- ▶ Semi-revision:
- ▶ The curly braces `{ }` encompass a *block*
- ▶ You have used these with `if()` and `while()`
- ▶ They define the set of lines executed inside the `if()` or `while()`

# Code Blocks in C

- ▶ You can place blocks anywhere you like
- ▶ Nothing wrong with:

```
1 int main() {  
2     int x;  
3     {  
4         printf("%d\n", x);  
5     }  
6     return 0;  
7 }
```

- ▶ This just places the `printf()` ; inside a block
- ▶ It doesn't do anything useful, but...

# Variable Scope

- ▶ A variable's "existence" is limited to the block where it is declared
  - ▶ Plus any blocks within that one
- ▶ Example this code won't compile:

```
1 #include <stdio.h>
2 int main() {
3     int x = 2;
4     if(x == 2) {
5         int k;
6         k = 2*x;
7     }
8     printf("%d\n", k);
9     return 0;
10 }
```

# Variable Scope

- ▶ Note that `k` was declared inside the `if()`
- ▶ That means that it no longer exists when the `if()` has finished
- ▶ This generates a compiler error
- ▶ It frees up some RAM
- ▶ It also lets the variable's name be reused elsewhere
  - ▶ This can be *really* confusing. Be careful.



Oh, end of the lecture already? Lets go read the lab notes...