

ENGG1003 - Friday Week 3

More Sequence Examples
Maybe More Flow Control

Brenton Schulz

University of Newcastle

March 15, 2019

Subscript Notation

- ▶ Last chance to learn that we use:

$$x_1, x_2, x_3, \dots, x_n \quad (1)$$

and

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

notation because it is the simplest method that works.

Subscript Notation

- ▶ x_n means that x is “some number” and n is an *integer* value that uniquely identifies x_n from other x 's
- ▶ n also implies an *order* to the x 's
- ▶ A formal mathematical statement of the above would be something like:

$$x_n : x \in \mathbb{R} \text{ and } n \in \mathbb{Z} \quad (3)$$

- ▶ \mathbb{R} is the set of real numbers
- ▶ \mathbb{Z} is the set of all integers

Subscript Notation

- ▶ Without this notation it is *really* hard to write things like:

$$x_n = x_{n-1} + x_{n-2} \quad (4)$$

- ▶ If you instead wrote:
“Calculate a sequence of numbers, a, b, c, d, \dots ”
how would you write the equation?

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 *before* 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
- ▶ It adds an extra condition on loop exit placed at any point in the loop

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 expressions in the `initial` / `condition` / `increment` sections
- ▶ Separate expressions 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

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 it 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

- ▶ A data type's 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.