ENGG1003 - Friday Week 3

More Sequence Examples
Maybe More Flow Control

Brenton Schulz

University of Newcastle

March 15, 2019

Assessment Task Rules

...Jump to rules PDF

Easy(ish) Assessment Task Example

Write a C program which generates a sequence of numbers:

$$x_1, x_2, x_3, \ldots$$

with the iterative equation:

$$x_n = 3x_{n-1} + 2x_{n-2}$$

and initial conditions:

$$x_1 = 3,\ x_2 = 1$$

The program should exit after printing (x_8 or an $x_n > 100$).

Easy(ish) Assessment Task Example

The program's output format is:

```
n x<newline>
```

For the values given, the output is:

```
1 3.000000
2 1.000000
3 9.000000
4 29.000000
5 105.000000
```

Easy(ish) Assessment Task Example

- What do we need to do?
  - Set up variables
  - Give some initial values
  - Implement the equation
  - Print the initial values
  - Write a `while()` loop
  - Get the exit condition correct
  - Print results
  - Wrap the whole thing in `main()`

Set up variables

Question didn't specify, but lets assume `float`

```
float xn, xnm1, xnm2;
int n;
```

Give some initial values

Question gave us:

$$x_1 = 3,\ x_2 = 1$$

Be careful with `xnm1` and `xnm2`, where are we starting?

```
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3
```

Implement the equation

$$x_n = 3x_{n-1} + 2x_{n-2}$$

```
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3

xn = 3.0*xnm1 + 2*xnm2;
```

That calculates x_3, but how does the program "advance in time"?

Implement the equation

Shift all the variables "forward in time" with:

```
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3

xn = 3.0*xnm1 + 2*xnm2;
xnm2 = xnm1;
xnm1 = xn;
```

Print the initial values

```c
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3

// x1 and x2 given so just hard code n
printf("1 %f\n", xnm2);
printf("2 %f\n", xnm1);

xn = 3.0*xnm1 + 2*xnm2;
xnm2 = xnm1;
xnm1 = xn;
```

Write a `while()` loop

We need to calculate x_n more than once, so:

```
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3

// x1 and x2 given so just hard code n
printf("1 %f\n", xnm2);
printf("2 %f\n", xnm1);

while( /* something */ ) {
  xn = 3.0*xnm1 + 2*xnm2;
  xnm2 = xnm1;
  xnm1 = xn;
}
```

Get the exit condition correct

The value of n goes from 1 to 8, and xn must remain below 100:

```
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3
// x1 and x2 given so just hard code n
printf("1 %f\n", xnm2);
printf("2 %f\n", xnm1);
while( (n <= 8) && (xn < 100) ) {
  xn = 3.0*xnm1 + 2*xnm2;
  xnm2 = xnm1;
  xnm1 = xn;
  n++;
}
```

Print results

```
float xn, xnm1 = 1, xnm2 = 3;
int n = 3; // The first unknown is x for n=3
// x1 and x2 given so just hard code n
printf("1 %f\n", xnm2);
printf("2 %f\n", xnm1);
while( (n <= 8) && (xn < 100) ) {
  xn = 3.0*xnm1 + 2*xnm2;
  xnm2 = xnm1;
  xnm1 = xn;
  n++;
  printf("%d %f\n", n, xn);
}
```

Wrap the whole thing in `main()`

```
#include <stdio.h>
main() {
  float xn, xnm1 = 1, xnm2 = 3;
  int n = 3; // The first unknown is x for n=3
  // x1 and x2 given so just hard code n
  printf("1 %f\n", xnm2);
  printf("2 %f\n", xnm1);
  while( (n <= 8) && (xn < 100) ) {
    xn = 3.0*xnm1 + 2*xnm2;
    xnm2 = xnm1;
    xnm1 = xn;
    n++;
    printf("%d %f\n", n, xn);
  }
}
```

Is the solution optimal?

- Some marks are allocated to reducing variable count
- It tests your understanding of how the = operation works
- Lets look at the maths:

```c
xn = 3.0*xnm1 + 2*xnm2;
xnm2 = xnm1;
xnm1 = xn;
n++;
printf("%d %f\n", n, xn);
```

- Do we need *all* those variables?

- In this case: yes

```
xn = 3.0*xnm1 + 2*xnm2;
xnm2 = xnm1;
xnm1 = xn;
n++;
printf("%d %f\n", n, xn);
```

- We can't overwrite `xnm1` before shifting it into `xnm2`
- Result must be stored in `xn` first

Another Isolated Example

- What if the equation was:

$$x_n = 0.2x_{n-1}$$

- This will *work*:

```
xn = 0.2*xnm1
xnm1 = xn;
```

- But because we never need `xnm1` elsewhere this is more optimal:

```
xn = 0.2*xn;
```

- Marks (above a pass) may be allocated to variable optimisation

Hard Assessment Task Example

Write a C program which generates two sequences of numbers:

$$x_0, x_1, x_2, \ldots$$

$$y_0, y_1, y_2, \ldots$$

with the coupled iterative equations:

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

and initial conditions:

$$x_0 = 5$$
$$y_0 = 0$$

Hard Assessment Task Example

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

- Lets have an attempt at implementing the equations
- We need *at least* two variables:
  - `float xn`
  - `float yn`
- Lets also use two "previous" variables:
  - `float xnm1`
  - `float ynm1`

Hard Assessment Task Example

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

- Our calculation code can then be:

```
xn = 0.6*xnm1 + 0.2*ynm1;
yn = 0.1*xnm1 + 0.9*ynm1;
xnm1 = xn;
ynm1 = yn;
```

- **Question:** Do we need all these variables?

Hard Assessment Task Example

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

- **Counter-question:** What is wrong with this?

```
xn = 0.6*xn + 0.2*yn;
yn = 0.1*xn + 0.9*yn;
```

Hard Assessment Task Example

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

- **Counter-question:** What is wrong with this?

```
xn = 0.6*xn + 0.2*yn;
yn = 0.1*xn + 0.9*yn;
```

- Why doesn't mathematics convert into code?

Hard Assessment Task Example

- Mathematics is *instant*

Hard Assessment Task Example

- Mathematics is *instant*
- Code is evaluated line by line

Hard Assessment Task Example

- Mathematics is *instant*
- Code is evaluated line by line
- Variables can *change* between lines, resulting in the wrong equation being implemented
- The previous slide was *actually* doing:

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_n + 0.9y_{n-1}$$

```
xn = 0.6*xn + 0.2*yn;
yn = 0.1*xn + 0.9*yn;
```

Hard Assessment Task Example

- Observe the correct subscripts:

$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

- In the 2nd equation we need x_{n-1} but the first equation would destroy that value
- We *must* use an extra variable to store x_{n-1} for y_n to be calculated correctly

Hard Assessment Task Example

- Aside: You may see coupled equations vaguely like this in signals and systems theory

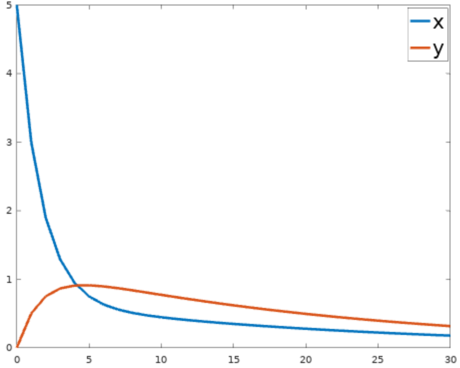
$$x_n = 0.6x_{n-1} + 0.2y_{n-1}$$
$$y_n = 0.1x_{n-1} + 0.9y_{n-1}$$

- Lets write C code with the minimum variables:

```
xtmp = x; // store xn before we lose it
x = 0.6*x + 0.2*y;  // Original xn value lost
y = 0.1*xtmp + 0.9*y; // stored xn used, yn
```

- ...And implement in Che

Results



Aside: results data was pulled from Che using SSH. Advanced students will appreciate this feature in later weeks.

FOR Loops in C

- The C FOR loop syntax is:

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

- 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

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

- 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 = (-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 = (-1)^k x^(2k) / (2k)!
    sum = sum + tmp
    k = k + 1
  ENDWHILE
END
```

where in both listings $\text{tmp} = \frac{(-1)^k x^{2k}}{(2k)!}$

FOR Loops in C (Advanced)

- `for()` syntax allows multiple expressions in the `inital` / `condition` /`increment` sections
- Separate expressions with commas
- eg:

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

- 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:

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

- (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

```c
#include <stdio.h>
int main() {
  int x = 0;
  int y = 0;
  int z = 0;
  y = ++x + 10;
  printf("Pre-increment: %d\n", y);
  y = z++ + 10;
  printf("Post-increment: %d\n", y);
  return 0;
}
```

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:

```c
int main() {
  int x;
  {
    printf("%d\n", x);
  }
  return 0;
}
```

- 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:

```c
#include <stdio.h>
int main() {
  int x = 2;
  if(x == 2) {
    int k;
    k = 2*x;
  }
  printf("%d\n", k);
  return 0;
}
```

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.