ENGG1003 - Tuesday Week 4

Loose Ends
Functions

Brenton Schulz

University of Newcastle

March 18, 2019

Subscript Notation

▶ Last chance to learn that we use:

$$x_1, x_2, x_3, ..., x_n \tag{1}$$

and

$$x_n = x_{n-1} + x_{n-2} \tag{2}$$

notation because it is the simplest method that gets the point across.

Subscript Notation

▶ x_n means that x is "some number" and n is an *integer* value

▶ n implies *uniqueness* (ie: x_1 and x_2 can differ)

▶ n implies an *order* to the x's

▶ A formal mathematical statement of the above would be something like:

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

▶ $\mathbb{R}$ 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:

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

Subscript Notation

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

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

▶ If you instead wrote:
"Calculate a sequence of numbers, $a, b, c, d, ...$"
how would you write the equation?

Considering Dropping?

- ▶ HECS census is Fri 22nd
- ▶ Before you drop:
  - ▶ Talk to me
  - ▶ Are you *legitimately* unprepared or experiencing "imposter syndrome"?
    - ▶ It is surprisingly common
  - ▶ Most of you have to pass eventually
  - ▶ There are some legitimate reasons
- ▶ Ignore unsolicited advice from demonstrators
  - ▶ Seriously, this isn't their job

FOR Loops in C

▶ The C FOR loop syntax is:

```c
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 flow control early
▶ The flow-control mechanism for this is a BREAK statement
▶ If executed, execution jumps outside the current `if()`, `while()`, `for()`, etc
▶ BREAKs *typically* go inside an IF

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
```

break Statements

- ▶ The example is mildly pointless
  - ▶ In C, the $|tmp| < 1e - 6$ condition can go in the for() statement. In pseudocode it *sort of* can't.
- ▶ It is there to illustrate what break does, not explain how to use it
- ▶ As the "experienced engineer' you choose when to use it

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

switch() - case:

▶ Sometimes you want to code something like:

```
1 if(x == 0) {
2   // stuff
3 } else if(x == 1) {
4   // stuff
5 } else if(x == 2) {
6   // stuff
7 } ...etc
```

▶ This is difficult to read and gets unwieldy. Fast.

`switch() - case:`

▶ Instead, C has:

```
1 switch(expression) {
2    case constant:
3        break;
4    case constant:
5        break;
6    default:
7 }
```

▶ The *expression* is anything which evaluates to a number

▶ The *constant*s are either literals or variables declared as `const` (covered later)

`switch()` - `case`: Example

```
1 int x=1, y=2;
2
3 switch(x==y) { // Evaluates to 0 or 1
4   case 0:
5       printf("x and y differ\n");
6       break;
7   case 1:
8       printf("x and y are equal\n");
9       break;
10  default:
11      printf("Something went very wrong\n");
12 }
```

▶ The `default:` case happens if the expression doesn't match any other option

switch() - case: Example

▶ If the break; is omitted execution continues
  line by line - run this in Che

```c
#include<stdio.h>
int main() {
  int x = 2;
  switch(x) {
    case 1: printf("x is 1\n");
    case 2: printf("x is 2\n");
    case 3: printf("x is 3\n");
    default: printf("x is not 1, 2, or 3\n");
  }
  return 0;
}
```

switch() - case: Limits

▶ Because the case statements only accept *constants* there are some limitations

▶ Example, this doesn't translate well:

```
1 if(x < 0) {
2   // stuff
3 } else if (x == 0) {
4   // stuff
5 } else if (x > 0) {
6   // stuff
7 }
```

▶ (x<0), (x==0), and (x>0) are all 0 or 1

▶ Can't *easily* translate this into three unique constants

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 }
```

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 adopted 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.

Functions

- ▶ A *function* is a block of code which can be *called* multiple times, from multiple places
- ▶ They are used when you want the same block of code to execute in many places throughout your code
- ▶ A function requires:
  - ▶ A name
  - ▶ (optional) A *return value*
  - ▶ (optional) One or more *arguments*

Functions in Mathematics

▶ In mathematics you saw functions written as:

$$y = f(x)$$

▶ Here, the function is called f, takes an argument of x and returns a value which is given to y

▶ C and pure mathematics have these general ideas in common

Functions in Mathematics

▶ In mathematics you saw functions written as:

$$y = f(x)$$

▶ Here, the function is called f, takes an argument of x and returns a value which is given to y

▶ C and pure mathematics have these general ideas in common

▶ The similarities stop there

Function Examples

► So far, some of you have used *library functions*

► These are functions which are pre-existing within the compiler (and its libraries)

► I have shown you:
  ► `scanf();`
  ► `printf();`
  ► `rand();`

Function Syntax

- ▶ Writing `rand();` in you code is *calling* the function
- ▶ The program execution "jumps" into the function's code, executes it, then jumps back
- ▶ Function call syntax is:

  `name([arguments])`

- ▶ Not all functions take arguments
- ▶ The function can "turn into" its *return value*

Function Examples

▶ Example 1:

```
1 x = rand();
```

  ▶ `rand` is the function name
  ▶ It returns a "random" integer
  ▶ The return value is allocated to `x`
  ▶ It doesn't take an argument

Function Examples

▶ Example 1:

```
1 x = rand();
```

  ▶ `rand` is the function name
  ▶ It returns a "random" integer
  ▶ The return value is allocated to `x`
  ▶ It doesn't take an argument

▶ Example 2:

```
1 y = sqrtf(x);
```

  ▶ `sqrtf` is the function name
  ▶ `x` is the argument
  ▶ It returns the square root of `x`
  ▶ The return value is allocated to `y`

Functions

▶ Function arguments and return values have pre-defined data types

Functions

- ▶ Function arguments and return values have pre-defined data types
- ▶ Example from documentation
  - ▶ `int rand(void);`
    - ▶ The return value is an `int`
    - ▶ The argument is type `void`
    - ▶ This just means there aren't any

Functions

- ▶ Function arguments and return values have pre-defined data types
- ▶ Example from documentation
  - ▶ `int rand(void);`
    - ▶ The return value is an `int`
    - ▶ The argument is type `void`
    - ▶ This just means there aren't any
  - ▶ `float sqrtf(float x);`
    - ▶ The return value is a `float`
    - ▶ The argument is a `float`
    - ▶ Argument is called `x` in documentation but you can pass it any `float`

Return Values (an Engineer's View)

- ▶ The function's *return value* is the number a function gets "replaced with" in a line of code

Return Values (an Engineer's View)

- ▶ The function's *return value* is the number a function gets "replaced with" in a line of code
- ▶ Function return values, variables, and literals can all be used in the same places:
    - ▶ In arithmetic
    - ▶ In conditions
    - ▶ As arguments to other functions

Return Values (an Engineer's View)

▶ The function's *return value* is the number a function gets "replaced with" in a line of code

▶ Function return values, variables, and literals can all be used in the same places:
  ▶ In arithmetic
  ▶ In conditions
  ▶ As arguments to other functions

▶ The C standard is *very* specific about what return values are but I will be informal for now
  ▶ Technically, for example, an expression like $x=y+5.0$; also has a "return value" equal to the value allocated to x

Return Values

- ▶ I use functions from `math.h` in these examples, we'll cover them in a few mins

Return Values

▶ I use functions from `math.h` in these
  examples, we'll cover them in a few mins

▶ The following are all valid:

  ▶ `x = rand();`
  ▶ `printf("%f\n", sin(y));`
  ▶ `if( (rand()%6) < 2)`
  ▶ `while( sin(x) < 0 )`

Return Values

- ▶ I use functions from `math.h` in these examples, we'll cover them in a few mins
- ▶ The following are all valid:
  - ▶ `x = rand();`
  - ▶ `printf("%f\n", sin(y));`
  - ▶ `if( (rand()%6) < 2)`
  - ▶ `while( sin(x) < 0 )`
  - ▶ This next one is complicated...

Return Values

- ▶ I use functions from `math.h` in these examples, we'll cover them in a few mins
- ▶ The following are all valid:
  - ▶ `x = rand();`
  - ▶ `printf("%f\n", sin(y));`
  - ▶ `if( (rand()%6) < 2)`
  - ▶ `while( sin(x) < 0 )`
  - ▶ This next one is complicated...
  - ▶ `x = sin((double)rand());`

Return Values

▶ I use functions from `math.h` in these examples, we'll cover them in a few mins
▶ The following are all valid:
  ▶ `x = rand();`
  ▶ `printf("%f\n", sin(y));`
  ▶ `if( (rand()%6) < 2)`
  ▶ `while( sin(x) < 0 )`
  ▶ This next one is complicated...
  ▶ `x = sin((double)rand());`
    ▶ Generates a random integer, casts to `double`, uses that number as an argument to the `sin()` function

Using Functions

- ▶ (semi-revision)
- ▶ Before you use a function you must:
    - ▶ Read the documentation
    - ▶ `#include` the correct header file
    - ▶ Add the correct library to the compiler options
        - ▶ In Che I've done this for the maths library
        - ▶ `stdio` and `stdlib` are always there
    - ▶ Be aware of the data types
        - ▶ Do you need any type casting?
        - ▶ Are you using the correct function?

Maths Functions

▶ Since some of you have already used them, lets learn about the maths library...

▶ It includes functions for:
  ▶ Trigonometry
  ▶ Exponentials (base e) & logarithms (base e, 10, 2)
  ▶ Exponents (`pow()`;)
  ▶ Rounding (`floor()`; & `ceil()`;)
  ▶ Floating point modulus (`fmod()`;)
  ▶ Square roots
  ▶ ...etc

Maths Functions

- ▶ There are typically *different* functions for `float` and `double`
- ▶ This can have a huge speed impact
- ▶ Use the right ones!
- ▶ `float` maths functions typically end in 'f'
    - ▶ `cosf();`
    - ▶ `sqrtf();`
    - ▶ `atanf();`
    - ▶ ...etc
- ▶ `double` maths functions **don't**
    - ▶ `cos();`
    - ▶ `log();`

Maths Functions

▶ Math functions are written by mathematicians
  ▶ All angles are in radians

Maths Functions

▶ Math functions are written by mathematicians
  ▶ All angles are in radians
  ▶ `log();` is $\log_e$
    ▶ `log10();` is $\log_{10}$
    ▶ `log2();` is $\log_2$

Maths Functions

- Math functions are written by mathematicians
    - All angles are in radians
    - `log();` is $\log_e$
        - `log10();` is $\log_{10}$
        - `log2();` is $\log_2$
    - Inverse trig functions are called "arcus functions"
        - $\sin^{-1}$ is `asin();`
        - $\cos^{-1}$ is `acos();`
        - $\tan^{-1}$ is `atan();`

Maths Functions

- ▶ Math functions are written by mathematicians
  - ▶ All angles are in radians
  - ▶ `log();` is $\log_e$
    - ▶ `log10();` is $\log_{10}$
    - ▶ `log2();` is $\log_2$
  - ▶ Inverse trig functions are called "arcus functions"
    - ▶ $\sin^{-1}$ is `asin();`
    - ▶ $\cos^{-1}$ is `acos();`
    - ▶ $\tan^{-1}$ is `atan();`
  - ▶ The "4 quadrant" arctan function is `atan2();`
    - ▶ `atan(x);` returns $[-\pi/2, \pi/2]$
    - ▶ `atan2(x, y);` returns $[-\pi, \pi]$ depending on the quadrant of the point `x, y`
    - ▶ Very useful for polar to Cartesian coordinate transforms (probably beyond 1st semester 1st year)

Example - Quadratic Equation

Write a C program which uses the standard library function `sqrtf();` as part of the calculations required to produce solutions to a quadratic equation:

$$ax^2 + bx + c = 0 \tag{5}$$

using

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{6}$$

...do it live in Che

Example - `scanf();`'s Return Value

Read the `scanf();` <u>documentation</u> and observe that it returns an `int`. What does that `int` represent? Write some test code and experiment with its behaviour.

Demonstrate it live in Che .

Writing Functions

▶ What about writing your own functions?

Writing Functions

▶ What about writing your own functions?
▶ Do the following:
   1. Choose a name

Writing Functions

▶ What about writing your own functions?
▶ Do the following:
  1. Choose a name
  2. Decide on the function arguments

Writing Functions

▶ What about writing your own functions?
▶ Do the following:
  1. Choose a name
  2. Decide on the function arguments
  3. Decide on the return value

Writing Functions

▶ What about writing your own functions?

▶ Do the following:

1. Choose a name
2. Decide on the function arguments
3. Decide on the return value
4. Write a *function prototype*
   ▶ Write it at the top of your code [or in a header file]

Writing Functions

▶ What about writing your own functions?

▶ Do the following:

1. Choose a name
2. Decide on the function arguments
3. Decide on the return value
4. Write a *function prototype*
   ▶ Write it at the top of your code [or in a header file]
5. Somewhere below `main()` (or in another `.c` file) write the function *definition*

Writing Functions

▶ What about writing your own functions?
▶ Do the following:
  1. Choose a name
  2. Decide on the function arguments
  3. Decide on the return value
  4. Write a *function prototype*
     ▶ Write it at the top of your code [or in a header file]
  5. Somewhere below `main()` (or in another `.c` file)
     write the function *definition*
▶ For now just keep everything in one file
  ▶ Unless you study ahead. I won't stop you.

Function Prototypes

▶ Huh? What's a function prototype?

Function Prototypes

- ▶ Huh? What's a function prototype?
- ▶ Before a function is *called* the compiler needs to know:
  - ▶ Its name
  - ▶ Its argument's data type(s)
  - ▶ Its return data type

Function Prototypes

▶ Huh? What's a function prototype?
▶ Before a function is *called* the compiler needs to know:
  ▶ Its name
  ▶ Its argument's data type(s)
  ▶ Its return data type
▶ A function prototype documents these things for the compiler

Function Prototypes

▶ The function prototype syntax is:

```
1 [return data type] function_name(arguments);
```

Function Prototypes

▶ The function prototype syntax is:

```
[return data type] function_name(arguments);
```

▶ The `arguments` section is a comma separated list with the following syntax:

```
(datatype name, datatype name, ...)
```

▶ Examples:
  ▶ `float sqrtf(float x);`
  ▶ `int rand(void);`
  ▶ `double log(double x);`
  ▶ `double atan2(double x, double y);`

Void

- ▶ If either the arguments or return value aren't required declare them as `void`
- ▶ This is an explicit way of saying "this item doesn't exist"

Function Prototypes

- ▶ The function prototype must be *before* the function's first use
- ▶ For "small" projects: above `main()`
- ▶ For "big" projects: in their own *header file*
  - ▶ We'll cover this later
- ▶ Don't leave the prototype's arguments blank
  - ▶ The compiler won't complain but it is a deprecated language feature

Function Definitions

▶ The function prototype tells the compiler how the function interacts with other code

▶ The function definition is the *actual code* that gets executed when the function is called

```
1 int add(int a, int b); // Prototype
2
3 main() {
4   // do stuff
5 }
6
7 int add(int a, int b) { // Definition
8   return a + b;
9 }
```

Function Prototypes Vs Definitions

▶ For the time being:
  ▶ The prototype goes *above* `main()`
    ▶ It is 1 line and ends with a semicolon ;
  ▶ The definition goes *below* `main()`
    ▶ It is the prototype repeated followed by a { } block

Writing Functions - Example

- ▶ Lets implement the Week 2 `sqrt` algorithm as a function
- ▶ ...Then compare with `sqrtf();`
- ▶ Keep it simple: fixed iteration count `n=10`

Writing Functions - Example

▶ In mathematics, calculate $\sqrt{k}$ by iterating:

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{k}{x_{n-1}}\right)$$

$$x_0 \neq 0$$

▶ In a code snippet:

```
1  // Calculate sqrt(k)
2  float k = 26; // Test value, sqrt(26)=5.0990
3  float xn = x/2.0; // x0 = x/2 because why not?
4  int n;
5  for(n = 0; n < 10; n++) {
6    xn = 0.5*(xn + k/xn);
7  }
```

Writing Functions - Example

- ▶ Lets make some design decisions:
  - ▶ Name: `mySqrt();`
  - ▶ Argument: `float k`
  - ▶ Return Value: `float`
- ▶ The function prototype is therefore:

```
1 float mySqrt(float k);
```

Writing Functions - Example

▶ Place the function prototype before `main()`:

```
1 #include <stdio.h>
2
3 float mySqrt(float k);
4
5 int main() {
6   // Do stuff
7 }
```

Writing Functions - Example

▶ Write the function definition below `main()`

```c
#include <stdio.h>
float mySqrt(float k);
int main() {
  printf("sqrt(26) = %f\n", mySqrt(26.0));
}

float mySqrt(float k) {
  int n;
  float xn = k/2.0;
  for(n = 0; n < 10; n++)
    xn = 0.5*(xn + k/xn);
  return xn;
}
```

Writing Functions - Example

▶ Lets view a few common errors

```c
1 #include <stdio.h>
2 float mySqrt(float k);
3 int main() {
4   printf("%f\n", mySqrt(26));
5 }
```

▶ Results in:

```
/tmp/ccT6mLDi.o: In function 'main':
/projects/voidTest/hello.c:4: undefined
    reference to 'mySqrt'
collect2: error: ld returned 1 exit status
```

Writing Functions - Example

▶ Likewise, forgetting the prototype:

```
1 #include <stdio.h>
2 int main() {
3   printf("%f\n", mySqrt(26));
4 }
```

▶ Results in (cut down):

```
hello.c: In function 'main':
hello.c:4:17: warning: implicit declaration of
    function 'mySqrt'
  printf("%f\n", mySqrt(26));
/projects/voidTest/hello.c:4: undefined
    reference to 'mySqrt'
```

Function Compiler Errors

- ▶ "implicit declaration of..."
  - ▶ The function prototype is missing
- ▶ "undefined reference to..."
  - ▶ The function definition is missing

Function Definition Placement

▶ The following *works* but isn't recommended:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  float mySqrt(float k) {
5    int n;
6    float xn = k/2.0;
7    for(n = 0; n < 10; n++)
8      xn = 0.5*(xn + k/xn);
9    return xn;
10 }
11
12 int main() {
13   printf("sqrt(26) = %.8f\n", mySqrt(26.0));
14   printf("Libarsy sqrtf(26): %.8f\n", sqrtf(26.0));
15 }
```

▶ Only useful in very small projects but common

Function Arguments

▶ Function arguments become variables inside the function

```
1 float mySqrt(float k) {   // k is an argument
2   int n;
3   float xn = k/2.0;   //k used here
4   for(n = 0; n < 10; n++)
5     xn = 0.5*(xn + k/xn);  // and here
6   return xn;
7 }
```

▶ Don't declare them as variables!

Function Arguments

- ▶ By default, arguments are "passed by value"
- ▶ The function gets *copies*
- ▶ Modifying them in a function doesn't change the original variable
  - ▶ No, not even if they have the same name
- ▶ The argument variables are discarded on function return
- ▶ The return value is the *only thing* that goes back

Function Return Values

▶ Return values can only be one number
▶ How can we write a function which modifies (or returns) multiple things?

Function Return Values

- ▶ Return values can only be one number
- ▶ How can we write a function which modifies (or returns) multiple things?
- ▶ Trigger warning....

Function Return Values

- ▶ Return values can only be one number
- ▶ How can we write a function which modifies (or returns) multiple things?
- ▶ Trigger warning....
- ▶ Pointers!

Function Return Values

▶ Return values can only be one number
▶ How can we write a function which modifies (or returns) multiple things?
▶ Trigger warning....
▶ Pointers!
▶ We'll learn how to use pointers in Week 6(ish)
▶ For now, just learn to live with the single return value

Function Example

Write a C function, `isPrime()`, which takes an `int` as an argument and returns 1 if it is prime and zero otherwise

- ▶ Name: `isPrime`
- ▶ Argument(s): `(int x)`
- ▶ Return Value: `int`

Function Example

Write a C function, isPrime(), which takes an int as an argument and returns 1 if it is prime and zero otherwise

- ▶ Name: isPrime
- ▶ Argument(s): (int x)
- ▶ Return Value: int
- ▶ Function prototype:
  int isPrime(int x);