ENGG1003 - Friday Week 4

Functions
Static Variables
Commenting
Arrays (but probably not)

Brenton Schulz

University of Newcastle

March 22, 2019

Writing Functions - Example

- Lets view a few common errors

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

- 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("Library sqrtf(26): %.8f\n", sqrtf(26.0));
15 }
```

- Only useful in very small projects but common

Function Arguments

- Function arguments automatically 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);
```

Function Example

... Do it live in Che without preparation.

Future Brenton might regret this but Present Brenton don't care.

Static Vs Auto Variables

- Any "normal" variable declared within the function (including arguments) is lost on function exit
  - These are called *auto* variables
- By default, any declared variable is an auto variable
  - Their value is lost outside the block where they are declared

Static Vs Auto Variables

- Any "normal" variable declared within the function (including arguments) is lost on function exit
  - These are called *auto* variables
- By default, any declared variable is an auto variable
  - Their value is lost outside the block where they are declared
- Alternatively, `static` variables can be used

Static Vs Auto Variables

- Any "normal" variable declared within the function (including arguments) is lost on function exit
  - These are called *auto* variables
- By default, any declared variable is an auto variable
  - Their value is lost outside the block where they are declared
- Alternatively, `static` variables can be used
  - Their value is retained

Static Vs Auto Variables

- Any "normal" variable declared within the function (including arguments) is lost on function exit
  - These are called *auto* variables
- By default, any declared variable is an auto variable
  - Their value is lost outside the block where they are declared
- Alternatively, `static` variables can be used
  - Their value is retained
  - Their scope is still limited

Static Variables

- Example: the `rand()` function returns different random numbers each time it is called
  - How? Shouldn't everything be lost when the function returns?
  - Not always! The `rand()` function's "state" is kept by a `static` variable.

Static Variables

- Example: the `rand()` function returns different random numbers each time it is called
  - How? Shouldn't everything be lost when the function returns?
  - Not always! The `rand()` function's "state" is kept by a `static` variable.
- Variables are static if declared with the `static` keyword
- Declaration examples:

Static Variables

- Example: the `rand()` function returns different random numbers each time it is called
  - How? Shouldn't everything be lost when the function returns?
  - Not always! The `rand()` function's "state" is kept by a `static` variable.
- Variables are static if declared with the `static` keyword
- Declaration examples:
- `static int k = 0;`
- `float z = 0, static y = 0;`
- `static long bigNum = 2345235234432;`

Static Variable Example

- Example: Write a function, `counter()` which returns an integer equal to the number of times it has been called.

Static Variable Example

- Example: Write a function, `counter()` which returns an integer equal to the number of times it has been called.
- Function prototype: `int counter(void);`

Static Variable Example

- Example: Write a function, `counter()` which returns an integer equal to the number of times it has been called.
- Function prototype: `int counter(void);`
- Function definition:

```
1 int counter() {
2   static int count = 0;
3   return count++;
4 }
```

Static Variable Example

- The variable `count` is declared `static`
- The initialisation, `count = 0`, happens *once*
- The value of `count` is retained between function calls

```
1 int counter() {
2   static int count = 0;
3   return count++;
4 }
```

Static Variable Example

- Wait, why would you do this?

Static Variable Example

- Wait, why would you do this?
- The function can be called from *anywhere* in your code

Static Variable Example

- Wait, why would you do this?
- The function can be called from *anywhere* in your code
- A "counter" variable that did the same job would have to be "global" to be visible anywhere

Static Variable Example

- Wait, why would you do this?
- The function can be called from *anywhere* in your code
- A "counter" variable that did the same job would have to be "global" to be visible anywhere
  - For multiple reasons we try to avoid variables with global scope
    - Good discussion here

Static Variable Example

- Wait, why would you do this?
- The function can be called from *anywhere* in your code
- A "counter" variable that did the same job would have to be "global" to be visible anywhere
  - For multiple reasons we try to avoid variables with global scope
    - Good discussion here
  - There are *very good* reasons to use them in embedded systems, but not on a desktop PC or server

Static Variable Example

Wrapping the function in some test code:

```
1  #include <stdio.h>
2
3  int counter(void);
4
5  int main() {
6    for(int k = 0; k < 10; k++)
7      printf("counter(): %d\n", counter() );
8    return 0;
9  }
10
11 int counter(void) {
12   static int count = 0;
13   return count++;
14 }
```

Test Code?

- "Test code" is a term I made up
- It means the minimum amount of code required to verify a function's behaviour
- Always test your functions *in isolation*!

Test Code?

- "Test code" is a term I made up
- It means the minimum amount of code required to verify a function's behaviour
- Always test your functions *in isolation*!
- If you write "too much" code before testing it will make debugging **much** harder

Test Code

- How much is “too much”?

Test Code

- How much is “too much”?
- Personally?

Test Code

- How much is “too much”?
- Personally?
- After 20 years of experience?

Test Code

- How much is “too much”?
- Personally?
- After 20 years of experience?
  - 1-5 lines

Test Code

- How much is "too much"?
- Personally?
- After 20 years of experience?
  - 1-5 lines
- Never underestimate:
  - How hard programming is
  - How easy it is to make mistakes
  - How *brutally catastrophic* bugs can be

Bug Case Study

Paraphrased from Wikipedia:

"The Therac-25 was a computer-controlled radiation therapy machine ... It was involved in at least six accidents ... in which patients were given massive overdoses of radiation. Because of concurrent programming errors, it sometimes gave its patients radiation doses that were hundreds of times greater than normal, resulting in death or serious injury."

Back to Functions...

- When should functions be used?

Back to Functions...

- When should functions be used?
- Well, what do they achieve?
  - *Much* easier to solve problems when they're broken down into sub-tasks
  - Reduce code line count and complexity (if they are called multiple times)
  - Allows code re-use between projects
  - *Much* easier to perform project management between multiple programmers
  - Bugs in a function are easier to fix than a bug in code which has been copy+pasted multiple times
  - ...the list goes on

When should functions be used?

- What about in an ENGG1003 context?

When should functions be used?

- What about in an ENGG1003 context?
  - Vague rule of thumb? No more 10-20 lines or so in one block.
  - Break a big problem into multiple sub-problems
    - Implement each as their own function

When should functions be used?

- What about in an ENGG1003 context?
  - Vague rule of thumb? No more 10-20 lines or so in one block.
  - Break a big problem into multiple sub-problems
    - Implement each as their own function
    - Yes, even if they are only called once

When should functions be used?

- What about in an ENGG1003 context?
  - Vague rule of thumb? No more 10-20 lines or so in one block.
  - Break a big problem into multiple sub-problems
    - Implement each as their own function
    - Yes, even if they are only called once
    - Do what you feel is most "readable"

When should functions be used?

- What about in an ENGG1003 context?
  - Vague rule of thumb? No more 10-20 lines or so in one block.
  - Break a big problem into multiple sub-problems
    - Implement each as their own function
    - Yes, even if they are only called once
    - Do what you feel is most "readable"
    - Your opinion here will change with experience, I will try to provide guidance

Functions and Comments

- Programming courses always tell you to comment your code
- But what is "good" commenting?
- Lets look at some examples:

Functions and Comments

- Programming courses always tell you to comment your code
- But what is "good" commenting?
- Lets look at some examples:
  - From the Linux kernel source

Functions and Comments

- Programming courses always tell you to comment your code
- But what is "good" commenting?
- Lets look at some examples:
  - From the Linux kernel source
  - From an embedded systems library

Functions and Comments

- Programming courses always tell you to comment your code
- But what is "good" commenting?
- Lets look at some examples:
  - From the Linux kernel source
  - From an embedded systems library
- Just a little different from each other, eh?
- Commenting is very application specific
- Commenting is very audience specific

Commenting in ENGG1003

- How many comments do we use in ENGG1003?
- On one hand: only comment what *you* need
- On the other: we need to assess your comments eventually...

Commenting in ENGG1003

- How many comments do we use in ENGG1003?
- On one hand: only comment what *you* need
- On the other: we need to assess your comments eventually...
- And the assessment needs to minimise demonstrator judgement...

Commenting in ENGG1003

- How many comments do we use in ENGG1003?
- On one hand: only comment what *you* need
- On the other: we need to assess your comments eventually...
- And the assessment needs to minimise demonstrator judgement...
- Maybe I create different strict rules for different assignments? Similar to ENGG1500 report rules.

Arrays

- Anyway, new topic!
- So far: all variables have been a *single* number
- What do you do if you need a million of them?

Arrays

- Anyway, new topic!
- So far: all variables have been a *single* number
- What do you do if you need a million of them?
- Declare a million variables?

Arrays

- Anyway, new topic!
- So far: all variables have been a *single* number
- What do you do if you need a million of them?
- Declare a million variables?
- Cry?

Arrays

- Anyway, new topic!
- So far: all variables have been a *single* number
- What do you do if you need a million of them?
- Declare a million variables?
- Cry?
- Use an *array*!

Arrays

- Anyway, new topic!
- So far: all variables have been a *single* number
- What do you do if you need a million of them?
- Declare a million variables?
- Cry?
- Use an *array*!
  - Maybe still cry...at first.

Arrays

- Anyway, new topic!
- So far: all variables have been a *single* number
- What do you do if you need a million of them?
- Declare a million variables?
- Cry?
- Use an *array*!
  - Maybe still cry...at first.
- An *array* is a collection of variables of the same data type

Arrays

- Remember the mathematics notation:

$$x_0, x_1, x_2, x_3, \ldots$$

- We used it for a single variable, x, changing with time
  - The "old" values of x were discarded

Arrays

- Remember the mathematics notation:

$$x_0, x_1, x_2, x_3, \ldots$$

- We used it for a single variable, x, changing with time
  - The "old" values of x were discarded
- An array allows us to store *all* the values of x_n in memory
- The variable name, x, and the "index", n, are both needed to access a particular value

Arrays

- In C, an array declaration **needs** three things:
  - The data type
  - A name
  - The number of *elements*
- Optionally, the array can also be initialised
- The syntax for an array of length N is:
  data type `name[N]`
- Examples:
  - `int list[20];`
  - `char name[200], c; //array and var`
  - `double data[100000];`

Arrays

- The length **must be known at compile time**
- This won't cause a compile error but will give you **BIG PROBLEMS**:

```
1 int x;
2 scanf("%d", &x);
3 int array[x];
```

Arrays

- The length **must be known at compile time**
- This won't cause a compile error but will give you **BIG PROBLEMS**:

```
1 int x;
2 scanf("%d", &x);
3 int array[x];
```

  - The size of `array` is not known at compile time

Arrays

- The length **must be known at compile time**
- This won't cause a compile error but will give you **BIG PROBLEMS**:

```
1 int x;
2 scanf("%d", &x);
3 int array[x];
```

  - The size of `array` is not known at compile time
  - The compiler doesn't know how big it is

Arrays

- The length **must be known at compile time**
- This won't cause a compile error but will give you **BIG PROBLEMS**:

```
1 int x;
2 scanf("%d", &x);
3 int array[x];
```

  - The size of `array` is not known at compile time
  - The compiler doesn't know how big it is
  - If `x` is large enough your program will access memory the operating system has not allowed it to

Arrays

- The length **must be known at compile time**
- This won't cause a compile error but will give you **BIG PROBLEMS**:

```
1 int x;
2 scanf("%d", &x);
3 int array[x];
```

  - The size of `array` is not known at compile time
  - The compiler doesn't know how big it is
  - If `x` is large enough your program will access memory the operating system has not allowed it to
  - This will cause segmentation faults (Linux/macOS) or illegal operations (Windows)