ENGG1003 - Friday Week 4

Functions
Static Variables

Brenton Schulz

University of Newcastle

March 20, 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:

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

- 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:

```c
#include <stdio.h>
#include <math.h>

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;
}

int main() {
  printf("sqrt(26) = %.8f\n", mySqrt(26.0));
  printf("Library sqrtf(26): %.8f\n", sqrtf(26.0));
}
```

- Only useful in very small projects but common

Function Arguments

- Function arguments automatically become variables inside the function

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

- 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;`
- `static float z = 0, y = 0;`
- `static long bigNum = 234523523443 2;`

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:

```c
int counter() {
  static int count = 0;
  return count++;
}
```

Static Variable Example

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

```c
int counter() {
  static int count = 0;
  return count++;
}
```

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 would doing 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 would doing 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 would doing 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:

```c
#include <stdio.h>

int counter(void);

int main() {
  for(int k = 0; k < 10; k++)
    printf("counter(): %d\n", counter() );
  return 0;
}

int counter(void) {
  static int count = 0;
  return count++;
}
```

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

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 (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+past'ed 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.
  - Segment 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.
  - Segment a big problem into multiple sub-problems
    - Implement each as their own function
    - Yes, even if they are only called once