ENGG1003 - Friday Week 2

Fixing the Mistakes I Made on Tuesday

Brenton Schulz

University of Newcastle

March 6, 2019

Rick is likely presenting this lecture.

My voice was destroyed by being over-enthusiastic while presenting 6 hours of material Monday and Tuesday. Oops.

Example: Testing Factors

Task: Write a C program which reads two integers from the user and tests if the second is a factor of the first.

Example: Testing Factors

Lets break the problem down into pseudocode:

```
BEGIN
  integer x
  integer y
  READ x from the user
  READ y from the user
  Test if x is a factor of y
  Tell the user the result
END
```

Can every line of pseudocode be turned into C?
How are we going to test if one number is a factor
of another?

Example: Testing Factors

Lets make an attempt at factor testing, perhaps start with:

Definition: *Factors* are numbers we can multiply together to make another number. Eg: 2 and 3 are factors of 6 as $2 \times 3 = 6$.

Is this definition useful for this problem? Is it easy to turn this fact into C code?

Example: Testing Factors

No, not really. C can't easily do "can I find another integer that multiplies with x to make y". That instruction is not *executable*.

Lets try again:

An integer, x, is a factor of a number, y, if the integer evaluation of $y \div x$ has no remainder.

Can *this* become C code?

Example: Testing Factors

YES! We can use the modulus operator, %, to test
if a division has a remainder with the code:

```
1 if( (x % y) == 0) {
2   // x is a factor of y
3 }
```

Modulus Example - Factor Testing

With this fact, lets tweak the pseudocode:

```
BEGIN
  integer x
  integer y
  READ x from the user
  READ y from the user
  IF (x % y) == 0
    PRINT x is a factor of y
  ELSE
    PRINT x is NOT a factor of y
  ENDIF
END
```

Modulus Example - Factor Testing

...and convert each line to C:

($printf()$; output changed to fit on slide)

Pseudocode	C Code
| ```
BEGIN
  integer x
  integer y
  READ x from the user
  READ y from the user
  IF (x % y) == 0
    PRINT y is a factor
  ELSE
    PRINT y isn't a factor
  ENDIF
END
``` | ```
int main() {
  int x;
  int y;
  scanf("%d", &x);
  scanf("%d", &y);
  if( (x % y) == 0) {
    printf("%d is a factor\n", y);
  } else {
    printf("%d isn't factor\n", y);
  }
}
``` |

Modulus Example - Code with Prompt

```c
#include <stdio.h>
int main() {
  int x, y;
  printf("Enter an integer: ");
  scanf("%d", &x);
  printf("Enter another integer: ");
  scanf("%d", &y);
  if(x % y == 0) { // ie: if the remainder is zero
    printf("%d is a factor of %d\n", y, x);
  } else {
    printf("%d is NOT a factor of %d\n", y, x);
  }
  return 0;
}
```

Listing 1: factorTest.c

Factor Testing Discussion

- ▶ Is this code *robust*?
- ▶ Can the user enter numbers which make the code produce the wrong number?
- ▶ What happens if $y > x$?
  - ▶ It might be fine, it might not. Have a think about it and do some testing in the lab
- ▶ Get in the habit of testing code, both with "expected" input and "weird" input
- ▶ What happens if you enter letters instead of numbers?

Modulus Example 2 - Factorisation

▶ We now know how to test if a number is a factor of another

▶ What about a full factorisation?

Task: Write a C program which reads an integer from the user and outputs all of its factors.

Modulus Example 2 - Factorisation

- ▶ How does factorisation happen, anyway?
    - ▶ Normally? In your head. "Dream up" the answer.
    - ▶ On a computer? We need to *brute force* it.
        - ▶ ie: Simply test every integer which might work
    - ▶ All factors of a number, k are in the range [1,k]
        - ▶ 1 and k are always factors, so explicitly testing them is optional
    - ▶ They are also all integers
    - ▶ Thankfully, this is a finite number of tests
- ▶ Faster algorithms *might* exist
    - ▶ You would need to consult number theory literature
    - ▶ This is beyond my (Brenton's) knowledge

Modulus Example 2 - Factorisation

▶ How can we test lots of numbers?
  ▶ The program needs to count
  ▶ The input is unknown, so we need a loop
  ▶ ie: we can't "hard code" counting when we don't know when to stop!

Modulus Example - Factorisation

Lets write some pseudocode for the factorisation problem. We start with something really "high level":

```
BEGIN
   Integer x
   READ a value for x from the user
   Calculate x's factors
   PRINT x's factors
END
```

Modulus Example - Factorisation

Now lets *imply* a loop, but not explicitly write it:

```
BEGIN
  Integer x
  READ a value for x from the user
  Test if all integers from 2 to x-1 is a factor of
      x
  PRINT x's factors
END
```

Modulus Example - Factorisation

```c
#include <stdio.h>
int main() {
  int input, x;
  printf("Enter an integer to factorise: ");
  scanf("%d", &input);
  x = input;
  while(x > 0) {
    if(input % x == 0) // ie: if the remainder is
    zero
      printf("%d is a factor of %d\n", x, input);
    x--;
  }
  return 0;
}
```

Listing 2: factors.c

DO ... WHILE

▶ Same as WHILE except executes *at least once*
▶ The condition is tested at the end
▶ Loops repeats if condition is TRUE
▶ Syntax:

```
DO
  stuff
WHILE condition
```

FOR Loops

- ▶ A FOR loop executes a given number of times
- ▶ Used when the number of loop repeats is known *before* entering the loop
  - ▶ Repeat count could be "hard coded" as a number
  - ▶ Could also be a variable
- ▶ Can be easier to read than WHILE
- ▶ Example pseudocode syntax:

```
FOR x = 1 to 10
  Do something ten times
ENDFOR
```

- ▶ The *loop variable* is automatically incremented

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 to control their execution

◀ □ ▶ ◀ 🗗 ▶ ◀ 💈 ▶ ◀ 💈 ▶   💈   ✑ ۹ ୯

Loop `continue` Statements

▶ Previously we saw loop `break` statements
  ▶ Caused loop to exit
▶ A `continue` causes execution to jump back to the loop start
▶ The *condition* is tested before reentry

FOR Example 1

▶ Two equivalent ways to implement the $\cos()$ series from before are:

NB: $|\mathrm{tmp}|$ 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
```

FOR Example 2 - 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*)

GOTO

- ▶ There exists a GOTO flow control mechanism
  - ▶ Sometimes also called a *branch*
- ▶ It "jumps" from one line to a different line
- ▶ 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 *can* use GOTO in ELEC1710

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 3: `increment.c`

Pre/post-inc/decrements have many applications, more details in coming weeks.

Binary Nomenclature

▶ The 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

Variable Scope

#define Constants

`for(;;)` Loops