Tutorial 2: loops and conditionals

MATH10017

Fall 2022

Contents

Instructions:

- The text contains code snippets illustrating some ideas, or giving complete programs. Implement these in working programs by adding headers (include statements) and a `main()` function.
- Problems for you to practice are labelled **Exercise**.
- Test the output of code by running it in `main`, adding print statements if necessary.
- Make sure all code compiles!
- You do not have to submit your work.

1 Printing ASCII codes

Computers store characters (letters, numbers, punctuation, even emoji) using *character encoding*, which assign a code to each character. An old example is *Morse code*, which assigns three "dots" and "dashes" to letters. If you think of "dot" as 0, and "dash" as 1, Morse code stores letters as a three digit binary number.

The basic character set used by C is encoded using *ASCII code*, which assigns a seven digit binary number to each character. This means there are 128 characters, labelled 0 to 127. For instance, the ASCII code of A is 65.

In C, *single quotes* gives you the ASCII code for a character, so 'A' is the integer 65. Here is a full table of ASCII codes: `https://www.asciitable.com`.

The following code prints out the uppercase letters and their ASCII codes:

```
for(int i = 0; i < 26; i=i+1){
  printf("The ASCII code for %c is %d.\n", 'A'+i, 'A'+i);
 }
```

Exercise:

- Create a file called "ascii.c", and put the previous code in `main()`.
- Write a function that prints the lowercase letters and their ASCII codes.

2 Loops with conditionals

The following problem is sometimes used in coding interviews to filter out people who don't know the basics. Even if you can code, it's a simple program to test your thought process.

Exercise: In a file called fizzbuzz.c, write a loop for n from 1 to 45 (including 45) that prints:

- "fizz" if n is divisible by 3,
- "buzz" if n is divisible by 5,
- "fizzbuzz" if n is divisible by both 3 and 5 (i.e. divisible by 15),
- n otherwise.

Hints:

- Use if, else if, else.

- Check if a number is divisible by 15 before checking the other divisibility conditions.

- To check if a number a is divisible by a number b, use the condition $a\%b == 0$, which checks if the remainder of a divided by b is zero.

Further questions/ideas (if you want, do these after you've finished everything else):

- Can you write a function to check divisibility? (Return 1 for true, 0 for false.)

- Make the program print "Foo" if a number is divisible by 7, "FizzFoo" if it is divisible by 21, "BuzzFoo" if it is divisible by 35, and "FizzBuzzFoo" if it is divisible by 105?

- **Advanced**: can you use three `if` statements to implement FizzBuzz? (Hint: declare `int FLAG`. Before the conditional statements set `FLAG = 0`, then if one of the divisibility conditions is true, set `FLAG` to 1, and only print the number if `FLAG` is 0.) Now can you do "FizzBuzzFoo" with four `if` statements?

3 Summing

Summation notation is a mathematical notation for expressing sums compactly. For instance,

$$1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2$$

can be written as

$$\sum_{n=1}^{10} n^2.$$

The symbol Σ is sigma, the Greek letter for "S"; it stands for "summation". The variable n is the *index*; for this sum, the value of n starts at 1, and is incremented by 1 until it reaches 10. The expression n^2 after the summation sign is the term that is added to the total for each value of n.

In general, if f is a function of an integer variable, we have

$$\sum_{k=a}^{b} f(k) = f(a) + f(a+1) + \cdots + f(b).$$

We can convert summation notation into C code by using *for loops*.

```
int total = 0;
for(int k = 1; k <= 10; k=k+1){
  total = total + k * k;
 }
```

Exercise: Put the previous code and the next two problems in a file called "sums.c". Print the results in `main()`.

1. Write a loop that computes the sum
$$\sum_{i=1}^{100} i = 1 + 2 + 3 + \cdots + 99 + 100.$$
2. Write a loop that computes the sum of all even numbers from 1 to 100 (including 100). (Hint: replace $k = k + 1$ with $k = k + 2$.)

4 The Collatz Conjecture

Consider the following process:

- take a positive integer n
- if n is even, replace n by $n/2$
- if n is odd, replace n by $3n + 1$
- repeat.

The Collatz Conjecture posits that this process will always end at 1, no matter what value of n we start with.

We can use a while loop to print out the steps of this process:

```
void collatz(int n)
{
  printf("%d\n", n);

  while(n > 1){
    if(n % 2 == 0){
      n = n/2;
    }
    else{
```

```
      n = 3*n+1;
    }
    print("%d\n", n);
  }
}
```

The Collatz Conjecture says that this while loop will always stop in a finite amount of time.

> **Exercise:** Put the previous code in a file called collatz.c.
>
> - run the `collatz` function for a few choices of n
> - create a function called `collatz_steps` that returns the number of steps taken to reach 1, starting at an integer n. (Do not print anything. The return type should be `int`, and you need to return an `int` called `count`, which you increase by 1 for each step.)
> - print out the number of steps for each integer n from 1 to 50.

5 Newton's method

Newton's method is an algorithm for approximating the roots of a function.

For instance, to find an approximation of $\sqrt{2}$ we can use Newton's method to find a root of $f(x) = x^2 - 2$. In this case, the algorithm is:

1. Start with a guess x_0 for the root.
2. Repeat until the approximation is "good enough": $x_{i+1} = x_i/2 + 1/x_i$.

So if our initial guess is $x_0 = 2$, we have:

$$\begin{aligned} x_1 &= \frac{x_0}{2} + \frac{1}{x_0} = 1 + \frac{1}{2} = \frac{3}{2} = 1.5 \\ x_2 &= \frac{x_1}{2} + \frac{1}{x_1} = \frac{3}{4} + \frac{2}{3} = \frac{17}{12} = 1.416666\ldots \end{aligned}$$

So after two steps, we already have an answer that is good to two decimal places, since $\sqrt{2} = 1.41421\ldots$.

Exercise: In a file called newton.c, implement the following code to find the square root of a postive number a.

- Write a function `double sqrt_iter(double x, double a)` that takes x and returns $x/2 + a/2x$.
- Write a function `double my_sqrt(double a)` that:
  - Declares a double variable `x` and sets it equal to `a`.
  - Uses a loop to do five iterations of `sqrt_iter` (updating `x` to `sqrt_iter(x, a)` at each step of the loop).
  - Returns the final value of x.

The algorithm we used to find square roots does a fixed number of iterations. This does not guarantee that the approximation we find will be good. We can use a while loop to run the algorithm until the successive approximations do not change by more than a fixed amount.

```
#include <math.h> // we need this to use fabs below

// absolute error between two numbers x and y
double abs_err(double x, double y)
{
  return fabs(x - y); // fabs returns the absolute value of a float
}

// compute one iteration of Newton's method for approximating the square root of a
double sqrt_iter(double x, double a); // define later, or add your code here

/*
 * Newton's algorithm for the square root of a.
 *
 * If a > 1, the approximation is good to 6 decimal places.
 */
double my_sqrt2(double a)
{
  double err = 1E-6; // error of 0.000001

  double x0 = a;
  double x1 = sqrt_iter(x0, a);

  while(abs_err(x1, x0) > err){
```

```
    x0 = x1;
    x1 = sqrt_iter(x0, a);
  }
  return x1;
}
```

This algorithm works well if the square root is fairly large. If a is very small, then the absolute error will automatically be very small. For small values of a you can use the *relative error* $|x - y|/|x|$ instead of the absolute error. Try finding the square root of `2E-10` using absolute error (you will need to use `%.15f` to see 5 significant digits). The answer is $\sqrt{2}/10^5 = 0.0000141421\ldots$, but you might find `my_sqrt2` is less accurate. Does using relative error fix this?