CS1010 Laboratory 02

Basic Debugging Procedures, Exercise 1

Zhang Puyu

Group BD04

September 5, 2024

Plan of the Day

Exercise 0 Review: General Remarks

- Make sure you **follow the question's specifications exactly**! So for example:
  1. If the question says "read two integers", don't declare two `doubles`.
  2. If the question asks you to compute the area of **one rectangle**, don't compute the cuboid's surface area.
  3. If the question says that `compute_bmi` takes in a height **in meters**, don't pass in the height in centimetres.
- Aim for the **minimum** possible number of operations in your code even if the improvement seems insignificant (this trains your ability to think for the optimal case). Example:
- `2 * (a + b + c)`, not `2 * a + 2 * b + 2 * c`. $(5 \to 3)$

Common Issues

- **Ones**: you do not need to do `(n - last) / 10`. Instead, do `n / 10`.
- It's advised to remove all boilerplate comments, i.e., `// TODO` etc. to make your code cleaner.
- Remember to do **explicit type casting** wherever applicable!
- Best practice to catch all syntax warnings: run `:w` first before you `:q`. If you do `:wq`, you won't be able to see the syntax warnings reported by Vim!
- Don't over-use parentheses: they are a good tool to make the order of arithmetic evaluations clearer, but **too many** parentheses reduce readability! E.g. things like `(x * x) + (y * y)` or `return (x + y);` are unnecessary.
- https://nus-cs1010.github.io/2425-s1/guides/style.html: coding conventions in CS1010.
- https://nus-cs1010.github.io/2425-s1/guides/c-in-cs1010.html: disallowed syntax in CS1010.

Some (Interesting) Extensions from Exercise 0

- `Ones.c`: what if we were to allow the input to be **non-positive** integers (i.e., given -11 as the input, we want to output 1 and -1)? Will your code still work?
- `Quadratic.c`: all test cases given guarantee that the discriminant is non-negative, but what if we want our code to work for **any quadratic equation**?
- `Cuboid.c`: what if I want to generalise this for any *n*-**dimensional "cuboid"** (a fun question to think is how you would define such a shape mathematically)? Can I build from the current algorithm and produce a new program that calculates the length of the diagonal for any *n*-**dimensional cuboid**?

Debugging: Analyse Your Code

- Name your variables **descriptively** so that your code is easier to read.
- E.g. to store the total number of students in our lab group, `num_of_students` is a much better name than `x`.
- Translate your code into human language and try to figure out what goes on during execution.

An Example (use `cp ~cs1010/lab02/bmi.c .` to copy it over)

Think: how to make the following code more readable?

```c
#include "cs1010.h"
double compute_bmi(double x, double y) {
    return x / (y * y);
}

int main() {
    double x = cs1010_read_double();
    double y = cs1010_read_double();

    cs1010_println_double(compute_bmi(x, y));
}
```

An Example

Re-write variables using more descriptive names:

```
#include "cs1010.h"
double compute_bmi(double mass, double height) {
    return mass / (height * height);
}

int main() {
    double mass = cs1010_read_double();
    double height = cs1010_read_double();

    cs1010_println_double(compute_bmi(mass, height));
}
```

An Example

Alternatively, you may consider using short-hand if you are familiar with naming conventions in maths and physics:

```
#include "cs1010.h"
double compute_bmi(double m, double h) {
    return m / (h * h);
}

int main() {
    double m = cs1010_read_double();
    double h = cs1010_read_double();

    cs1010_println_double(compute_bmi(m, h));
}
```

With Printing

- We may want to **keep track** of certain variables to see whether their values update in the way we expect.
- The easiest way to do so is to **print out** the variables before and after they are used by a function.
- You may also want to consider print out tooltip texts to help you read the console's output.

An Example


```c
#include "cs1010.h"
double compute_bmi(double m, double h) {
    cs1010_print("Get the BMI, where the height is ");
    cs1010_print_double(h);
    cs1010_println(" m");
    return m / (h * h);
}

int main() {
    double m = cs1010_read_double();
    double h = cs1010_read_double();

    cs1010_print("Height: ");
    cs1010_print_double(h);
    cs1010_println(" cm");

    cs1010_println_double(compute_bmi(m, h));
}
```

Note: you have to **delete/comment** the debugging code to pass the test cases.

Basic Stuff

- The `||` operator in C is an **inclusive OR** — `p || q` means **at least one** of `p` and `q` is true, rather than **exactly one** of `p` and `q` is true (there's a drastic difference between the two). **If you are interested, you can search for "exclusive or" or "symmetric difference".**
- The negation of AND is the OR of the negations; the negation of OR is the AND of the negations (look up De Morgan's Laws).
- The above is concisely written as:

```
!(p && q) == (!p) || (!q),
!(p || q) == (!p) && (!q).
```

Good Practices in Using Logical Expressions

- Always use parentheses when nesting OR and AND, i.e., you should write `p && (q || r)` or `(p && q) || r`, rather than `p && q || r`.
- **Think:** does `p || q` differ from `q || p`?
  - In general, they are equivalent.
  - However, if `p` is very **likely to be true** while `q` is very **likely to be false**, writing `p || q` is preferred (why?).
- **Think:** does `p && q` differ from `q && p`?
  - In general, they are equivalent.
  - However, if `p` is very **likely to be false** while `q` is very **likely to be true**, writing `p && q` is preferred (why?).
- The above two techniques are collectively known as **short-circuiting**.

Exploit Complementation

A positive integer n is called **Lucky** if any of the following is satisfied:

- its last digit is 7;
- its last digit is **not** 7 **but** itself is divisible by 7;
- it is an odd number **and** an even number when the last digit of it gets removed;
- **neither** of its last two digits is 7 **but** they add up to be 7.

What is a condition to check if n is **Unlucky**?

Date

- Just to make the question statement more mathematically rigorous: you are supposed to judge if the three dates are listed in **strictly increasing** order.
- What does it mean when we say "a date comes after another date"?
- If it's not so easy to compare three dates, what if we are just comparing two?
- Should we check for the month or the day first? Or, does it make no difference?
- Hint: What are the comparisons to do if you are arranging 3 integers from the smallest to the largest?

Pressure

- Logically, how are systolic and diastolic pressures connected to determine the blood pressure status?
- What is the most brute-force way to solve this?
- It seems that we can divide this into four cases and check them one by one using if-else blocks.
- BUT CAN WE DO BETTER?
- Hint: Divide into (pre-high, high) and (pre-low, low) first. Within each group you can divide into two categories again. This ensures that you reach the answer with at most 2 conditionals for any kind of input.

Wishfully, We All Know What Wishful Thinking Is...

Quick math check:

Mathematical Induction (adapted ver.)

Suppose we want to find the value of $P(n)$ for any non-negative integer n, we only need to do two things:

1. Find $P(0)$.
2. Find a way to get $P(k+1)$ based on the value of $P(k)$.

The so-called "Wishful Thinking" is just the reverse of the above!

Wishfully, We All Know What Wishful Thinking Is...

Wishful Thinking

Suppose we already know:

1. How to compute $P(k+1)$ using $P(k)$;
2. The value of $P(0)$, i.e., "the base case" (or **terminating condition**).

Now we want to find $P(n)$, and so we can just find $P(n-1)$ first. To find $P(n-1)$, we just need to find $P(n-2)$ first... the cycle goes on until we reach $P(0)$ where the recursion terminates.

Examples:

- $x^y = x^{y-1} * x$ with $x^0 = 1$.
- $\sum_{i=1}^{n} aq^i = aq^n + \sum_{i=1}^{n-1}$ with $aq^0 = a$.

GCD (Interesting)

- We **claim** that $\gcd(a, b) = \gcd(b, a \,\%\, b)$. (in fact we can prove this!), so what is a **sufficient condition** for us to determine the value of $\gcd(a, b)$?
- But how do we find $\gcd(b, a \,\%\, b)$ then? (Answer: No need to care because the same procedure will be applied to these new parameters when we recurse.)
- If this process goes on and on, when do we know that the algorithm needs to halt? (Answer: the second argument becomes 0.)
- For your information: this is the Euclidean Algorithm, a famous algorithm in *number theory* devised in around 300BC. If you wonder why such an algorithm always reaches the same base case, read this proof and Euclid's Division Lemma.

General Framework for Recursive Algorithms


```
f(parameters):
    if parameters match the base case:
        terminate and return the trivial result f(base case)
    else:
        find previous_step_result = f(previous step parameters)
        perform the transition:
            previous_step_result -> current_step_result
        terminate and return current_step_result
```

Try to use this framework to complete an example program for computing x^y by filling in the blanks!

```
f(parameters) {
    if (check for base case) {
        // Terminate and return base case
    }
    // Find the previous step
    // Try to reach the current step from the previous step's re
    // Return the final answer
}
```