NumCSE exercises
0. A learn-by-doing introduction to `C++`

A. Dabrowski, P. Bansal

September 21, 2017

Exercise 0.1. *A first* `C++` *primer.*

a) Implement a function `int recurFact(int n)` which returns the factorial of `n`. Use a **recursive** approach.

b) Implement **iteratively** the factorial in the function `int iterFact(int n)`.

c) How are integers stored/represented in memory? What about floating point numbers?

d) What is meant by **function overloading**?

e) For which `n`'s will your functions return an accurate value? What if we use `long` instead of `int`? What if we use `double`? (you can access the maximum `int` with `INT_MAX`, the maximum long with `LONG_MAX`, the maximum `double` with `DBL_MAX`; to use these macros include `climits` and `float.h`).

Exercise 0.2. *Passing arguments.*

a) What is the difference between passing an argument by value or by reference to a function?

b) What is the output of the following code?

```
#include <iostream>

void copyAddOne(int n) {
    n++;
}

void refAddOne(int &n) {
    n++;
}

int main () {
    int n = 1;

    copyAddOne(n);
    std::cout << n << std::endl;

    refAddOne(n);
    std::cout << n;
}
```

Exercise 0.3. *Pointer games.*

a) What is a **pointer**? How do the dereference operator `*` and the reference operator `&` work? If `p` is a pointer, what is `*(&p)`?

b) What value does the following function return?

```
int refAssignment() {
    int *p = new int;
    *p = 1;
    int &r = *p;
    r = 2;
    return *p;
}
```

c) How are arrays stored in memory? When an assignment `int x[2] = {1,2}` is run, what is `x`? What is `&(x[0])`?

d) What is meant by operator overloading?

e) What value does the following function return? Why?

```
int pointerShift() {
    int *p = new int[5];

    for (int i=0; i<5; i++)
        p[i] = i;

    p++;

    return p[2];
}
```

Exercise 0.4. (Long). *Linked lists, fractals, and plotting.*

a) We represent a polygonal curve in the plane as a sequence of points. We arrange the points in a singly linked list. For this purpose implement a class `Point` which stores the coordinates `double x`, `double y` of the current point and a reference `*next` to the next point of the curve (`*next` should be `Null` for the last point). Notice that with these conventions, the first point is enough to represent the whole polygonal curve.

b) Implement a function `double length()` in the class `Point` which returns the length of the curve starting at that point.

c) Implement a function `void plot()` which plots the curve. (You can use whatever plotting solution you prefer; we suggest to use `MathGL`, since its most simple functionalities will be used occasionally throughout the course).

d) Let Γ be the segment joining two points p_1, p_2 in the plane. Consider the following geometric construction:

- divide Γ into three segments of equal length;
- draw an equilateral triangle that has as base the middle segment and points to the left if we traverse Γ from p_1 to p_2;
- remove the line segment which is the base of the equilateral triangle.

After the construction we are left with a new polygonal curve made up of 4 different segments. (The effect of such a construction applied to the segment joining $(0, 0)$ and $(1, 0)$ is presented in Figure 1.) Given a segment represented by `Point p`, implement this procedure

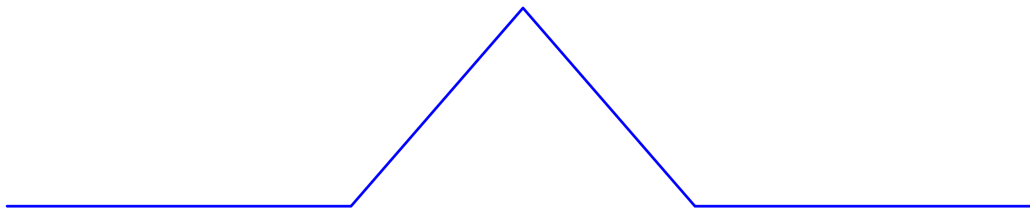

Figure 1: A single application of `fractalizeSegment()` to a segment.

in a function `void fractalizeSegment()`.

e) Consider the following iterative geometric construction:

- (Stage 0) Let Γ_0 be the segment joining the points $(0, 0), (1, 0)$;
- (Stage 1) Apply the procedure of point c) to obtain a new polygonal curve Γ_1 made of 4 segments;
- ...
- (Stage n) To each subsegment of Γ_{n-1}, apply again the procedure of point c) to obtain a new polygonal Γ_n made of 4^{n-1} segments.

(Stage 5 of the procedure starting from the segment joining $(0, 0)$ and $(1, 0)$ is presented in Figure 2.) Implement a single stage of this recursive construction in a function `void fractalize(Point &p)`.

f) Plot the length of the curve at the `n`th stage for `n` which varies from 1 to 12. Does it seem to converge to a finite value as `n` grows? Should it?

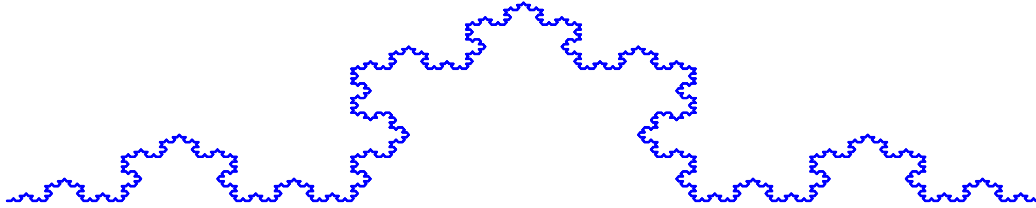

Figure 2: The fifth application of `fractalize()` to a segment.

Exercise 0.5. *A second* `C++` *primer.*

a) Implement **template** functions `iterFact` and `recurFact` which calculate the factorial, respectively iteratively and recursively. They should be able to handle any number as input (for example `int` or `long`, with the convention that the factorial of a `double` is the factorial of its integer part), and return an output of the same type as the input.

b) Implement a function `double funcTimer`, which takes as arguments a function `f`, a pointer input to an array of length `inputL`, and returns the time it took to run `f(x)` where `x` loops on the elements of `*input` (to time processes you can use `ctime` or, better, `chrono`[1]).

c) Implement a function `int* createRandNums`, which takes as input integers `N`, `k1` and `k2`, and returns a pointer to an array of `N` integers chosen randomly and uniformly from the interval [`k1`, `k2`] (to generate random numbers uniformly distributed you can use `rand`).

d) Calculate the elapsed time ratio between the recursive and the iterative implementation on a random array for different values of `N`, `k1`, `k2` (for instance you might try `N`=10^7, `k1`=5, `k2`=25). Is the iterative version faster or slower than the recursive one? Why? You might also try running your code changing the level of optimization (for example, with the `g++` compiler, by adding the flag `-O3`).

[1]for the documentation see `www.cppreference.com`.