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

A. Dabrowski, P. Bansal

October 1, 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.

Solution:

```
int recurFact(int n) {
    // recursive implementation of the factorial
    if (n==0) {
        return 1;
    }
    return n * recurFact(n-1);
}
```

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

Solution:

```
int iterFact(int n) {
    // iterative implementation of the factorial
    int out = 1;
    while (n>0) {
        out *= n;
        n--;
    }
    return out;
}
```

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

Solution: We always assume to use the `g++` compiler. Integers of the type `int` are stored in binary format in 32 contiguous bits[1]. Floating point numbers of the type `double` are stored in 64 bits, the first one for the sign, the following 11 for the exponent, the last 52 for the mantissa[2].

[1] more specifically, `gcc` implements *two's complement* representation (see for instance `softwareengineering.stackexchange.com/a/239039` for the most common types of memory representation of integers).

[2] for further information on floating point numbers and arithmetic we suggest the article *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, David Goldberg, ACM Computing Surveys, March 1991.

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

Solution: When multiple functions share the same name but have different signatures. An example are the functions `int iterFact(int n)` and `long iterFact(long n)`. When the function `iterFact` is run from the code, the compiler decides which definition to use based on the input type.

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`).

Solution: The accuracy of the results for `int` and `long` is determined by the maximum number which we can represent with the considered type. A simple code which determines in the variable `i-1` the maximum number whose factorial can be represented with an `int` is the following.

```
int i = 1;
double x = INT_MAX;
while (x >= 1) {
    x /= ++i;
}
```

The accuracy for `double` decreases as we consider larger numbers, until the result overflows.

Exercise 0.2. *Passing arguments.*

a) What is the difference between passing an argument by value or by reference to a function?
Solution: When an argument is passed by value its content is copied; when it is passed by reference only the memory address of its content is copied.[3] In the code of point b), `n` is passed by value to `copyAddOne` and it is passed by reference to `refAddOne`.

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;
}
```

Solution:

```
1
2
```

[3]Disclaimer: this is a nice and quick way to visualize what is happening, but it might not correspond to what the compiled code actually does at the assembler level. For further details, see for instance `http://www.cs.mcgill.ca/~cs573/fall2002/notes/lec273/lecture15/15_4.htm` for different possible implementations of paramater passing.

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)`?

Solution: A pointer `p` stores the address of a memory location. `*p` returns the content of the memory address to which `p` points. When `&` is applied to any variable, it returns the memory address of where the variable resides. Therefore `*(&p)` = `p`.

b) What value does the following function return?

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

Solution: `2`. The assignement `int &r = *p` just means: `r` is a new name for `*p` (it is but a reference to the same object).

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

Solution: An array is stored as contiguous cells in memory. After the assignment, `x` is an array of two `int`. If we try to print the value of `x` however, we simply get the address of the first element of the array, which is `&(x[0])`. Therefore we can think, as a first approximation, of an array and of a pointer to its first element as the same thing[4].

d) What is meant by operator overloading?

Solution: It is a special case of function overloading, when the function is an operator like `+`, `-`, `=`, ... For example, when `p` is a pointer to an array of `int` which starts at byte `0x172ac20` (the `0x` suffix indicates hexadecimal notation), then `p++` will point to `0x172ac24`, the position of the next `int` in the array (recall that an `int` is stored in 4 bytes $= 4 \cdot 8$ bits, thus the addition of 4 to the hexadecimal address). Instead, when `p` is for example a pointer to an array of `double` which starts at byte `0x172ac20`, then `p++` will point to `0x172ac28`.

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];
}
```

Solution: `3`. The reason is explained in the example of point d).

[4]as a second approximation however, we remark that the compiler can distinguish between an array and a pointer (for example, you can recover the size of an array with the function `sizeof`). See for instance `stackoverflow.com/questions/3959705` for a more thorough discussion.

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.

Solution:

```
class Point {
    public:
    double x,y;
    Point *next;

    Point (double x, double y) {
        this->x = x;
        this->y = y;
        this->next = NULL;
    }
};
```

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

Solution:

```
    static double dist(Point &a, Point &b) {
        // returns euclidean distance between two points
        double dx = a.x - b.x;
        double dy = a.y - b.y;
        return sqrt(dx*dx + dy*dy);
    }

    double length() {
        // returns euclidean length of polygonal curve
        Point *iter = this;
        double len = 0;
        while (iter->next != NULL) {
            len += dist(*iter, *iter->next);
            iter=iter->next;
        }
        return len;
    }
```

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).

Solution:

```
    void toArrays(double* xcoords, double* ycoords, int len) {
        // writes in *xcoords and *ycoords the coordinates of
        // the first len points of the curve
        Point *iter = this;
        for (int i=0; i<len; i++) {
            xcoords[i] = iter->x;
            ycoords[i] = iter->y;
            iter = iter->next;
        }
    }

    int countPoints() {
        // returns number of points until end of curve
        // (current point included in the count)
        int counter = 1;
        Point *iter = this->next;
        while (iter != NULL) {
            counter++;
            iter=iter->next;
        }
        return counter;
    }

    void plot(const char *name) {
        // saves plot of the curve in the file *name
        int len = this->countPoints();
        double* xcoords = new double[len];
        double* ycoords = new double[len];
        this->toArrays(xcoords, ycoords, len);

        mglData datx, daty;
        datx.Link(xcoords, len);
        daty.Link(ycoords, len);
        mglGraph gr;
        gr.SetRanges(0,1,0,.5);
        gr.Plot(datx, daty, "0");
        mglPoint pleftdown(0,0), prightup(1,.5);
        gr.SetCutBox(pleftdown, prightup);
        gr.WriteFrame(name);
    }
```

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

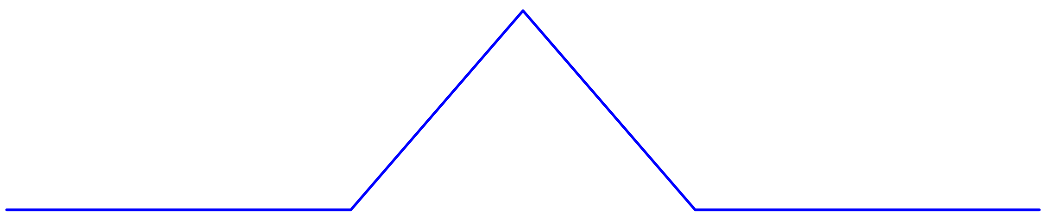

Figure 1: A single application of `fractalizeSegment()` to a segment.

presented in Figure 1.) Given a segment represented by `Point p`, implement this procedure in a function `void fractalizeSegment()`.

Solution:

```
void fractalizeSegment() {
    // constructs a triangular bump on the first segment

    Point *n = this->next;
    // calculation of coordinates of the points used in the construction
    double xonethird = (2*this->x + n->x)/3;
    double yonethird = (2*this->y + n->y)/3;
    double xmid = 1./2*(this->x + n->x);
    double ymid = 1./2*(this->y + n->y);
    double xtwothird = (this->x + 2*n->x)/3;
    double ytwothird = (this->y + 2*n->y)/3;
    double yperp = xtwothird - xonethird;
    double xperp = - ytwothird + yonethird;

    // creation of points and pointer referencing
    Point *p1 = new Point(xonethird, yonethird);
    Point *p2 = new Point(xmid + xperp*sqrt(3)/2, ymid + yperp*sqrt(3)/2);
    Point *p3 = new Point(xtwothird, ytwothird);
    p3->next = n;
    this->next = p1;
    p1->next = p2;
    p2->next = p3;
}
```

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)`.

Solution:

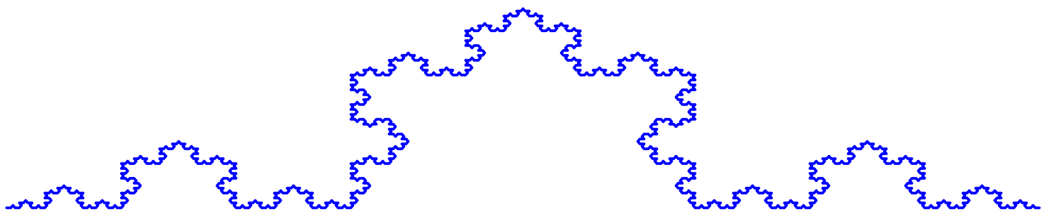

Figure 2: The fifth application of `fractalize()` to a segment.

```
void fractalize() {
    // repeats fractalizeSegment on every subsegment of a iteratively
    Point *iter1 = this;
    Point *iter2 =this->next;
    while (iter2 != NULL) {
        iter1->fractalizeSegment();
        iter1 = iter2;
        iter2 = iter1->next;
    }
}
```

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?

Solution:

```
int main() {
    // Initialization
    Point p1(0,0);
    Point p2(1,0);
    p1.next = &p2;

    int nStages = 12;
    double *lengths = new double[nStages];

    // fractalizes the segment and calculates the lengths
    for (int k=0; k<nStages; k++) {
        p1.fractalize();
        lengths[k] = p1.length();
    }

    // plots length of curve as a function of stage
    mglData data;
    data.Link(lengths, nStages);
    mglGraph gr;
    gr.SetRanges(0,nStages,0,nStages*3);
    gr.Plot(data);
    gr.Axis();
    gr.Label('x',"Stage");
    gr.Label('y',"Length");
    gr.WriteFrame("lengthsPlot.png");
}
```

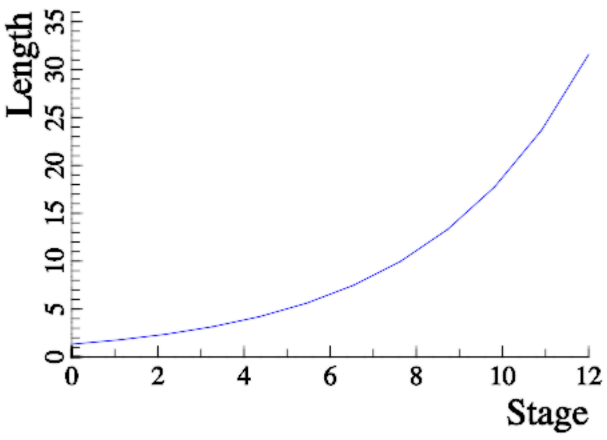

Figure 3: Length vs stage.

The length at the nth step is given by the length at the $n-1$ step plus the length of 4^{n-1} segments of length 3^n. Therefore the length at the nth step can be rewritten as

$$1 + \frac{1}{3}\sum_{k=0}^{n}\left(\frac{4}{3}\right)^k,$$

which goes to infinity as n goes to infinity (since the generic term in the previous sum is always larger than 1).

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.

Solution:

```
template<typename T>
T recFact(T n) {
    if (n<=0) return 1;
    return n * recFact(n-1);
}

template<typename T>
T iterFact(T n) {
    T out = 1;
    while (n>0) {
        out *= n;
        n--;
    }
    return out;
}
```

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`[5]).

Solution:

```
template<typename T>
double funcTimer (T (f)(T), int* input, int inputL) {
    // returns the time the function f took to run on
    // all elements of input.
    auto start = std::chrono::system_clock::now();
    for (int i=0; i<inputL; i++) {
        f(input[i]);
    }
    auto end = std::chrono::system_clock::now();
    auto elapsed = end - start;
    return elapsed.count();
}
```

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`).

Solution:

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

```
int* createRandArray(int N, int k1, int k2) {
    // returns N integers chosen randomly from [k1,k2]
    int* randNums = new int[N];
    srand(time(NULL));
    for (int i=0; i<N; i++) {
        randNums[i] = k1 + rand() % (1 + k2 - k1);
    }
    return randNums;
}
```

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`).

Solution:

```
int main ()
{
    int N = 10000000, k1 = 5, k2 = 25;
    int *randNums = createRandArray(N, k1, k2);
    double timeRecFact = funcTimer(recFact<long>, randNums, N);
    double timeIterFact = funcTimer(iterFact<long>, randNums, N);
    cout << timeRecFact / timeIterFact;
}
```

Without any optimization, the recursive version is slower, since additional allocation and access on the stack memory are required for recursion. However, when higher optimization is requested, the compiler is able to recognize that the calls in the factorial can be rewritten iteratively, and it does so.