Technical University of Denmark

Written examination, May 18, 2015

Course name: Programming in C++

Course number: 02393

Aids allowed: All written aids are permitted

Exam duration: 4 hours

Weighting: pass/fail

1. **Basic questions (10 points)**

   Answer to the following basic questions. Read the code *carefully*. Note that *code fragment* refers to a piece of code which is part of a proper program. This means that you can assume that a `main` function exists, that the necessary libraries have been imported, that some domainspace abbreviations have been declared, etc.

   (a) What is the output of the following code fragment? Why?

   ```
   int i = 0;
   int b = 3;
   if(i = 1){
       int i = 2;
       b = 7;
   }
   cout << "i=" << i << endl;
   cout << "b=" << b << endl;
   ```

   (b) What is the output of the following code fragment? Why?

   ```
   void change(int i){
       i = i + 3;
   }

   int main(){
       int i = 4;
       change(i);
       cout << "i=" << i << endl;
   }
   ```

   (c) Explain the difference between

   ```
   int a[10];
   ```

   and

   ```
   vector<int> a(10);
   ```

2. **Arrays and pointers (10 points)**

(a) Consider the following function to compute the sum of all numbers in an array `a` containing `n` integers:

```
int sum(int * a, int n){
    int result;

    for (int i=0; i<=n; i++){
        result = result + a[i];
    }

    return result;
}
```

Is this a correct implementation? I.e. will it produce the expected results, without compile- or run-time errors? If not, suggest a way to correct it.

(b) Consider the following variable declaration

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

and the following lines of code.

```
cout << a;
cout << *a;
cout << a[9];
cout << a[10];
cout << *(a) + 3;
cout << *(a+3);
```

For each line, (i) explain if the line will produce a compile-time error, (ii) explain if the line can produce a run-time error, and (iii) describe what the line will write in the standard output.

3. **Reversing a vector (10 points)**

   Implement a function for reversing a vector. The vector passed as argument has to be reversed, i.e. the function must not return a new vector. For example, if the vector `v` contains the elements `1, 2, 3` (in this order), after invoking `reverse(v)` it must contain the elements `3, 2, 1` (in this order).

   The header of the function must be

   ```
   template<typename T>
   void reverse(vector<T> & v);
   ```

   Note that the function must be parametric with respect to the type `T` of the elements of the vector.

4. **Inheritance (10 points)**

   Consider the following code fragment:

```
class Citizen {
public:
    void two(void) { cout << "2" << endl; };
    virtual void pie(void) { cout << "pie" << endl; };
};

class Nerd : public Citizen {
public:
    void two(void) { cout << "10" << endl; };
    void pie(void) { cout << "3.14159" << endl; };
};

int main(void){
    Nerd a;
    Citizen * b = &a;
    a.two();
    b->two();
    a.pie();
    b->pie();
    return 0;
}
```

   What is the output of the program?

5. **The Levenshtein distance (15 points)**

The Levenshtein distance between two sequence of characters $u = u_1, u_2, \ldots, u_k$ and $v = v_1, v_2, \ldots, v_l$ is defined by:

$$\mathrm{d}(u,v) = \begin{cases} |v| & \text{if } |u| = 0, \\ |u| & \text{if } |v| = 0, \\ \min \begin{cases} \mathrm{d}(u^1, v) + 1 \\ \mathrm{d}(u, v^1) + 1 \\ \mathrm{d}(u^1, v^1) + f(u_1, v_1) \end{cases} & \text{otherwise.} \end{cases}$$

where $|w|$ denotes the length of a sequence w; w^1 denotes the suffix $w_2, w_3, \ldots$ of a sequence $w = w_1, w_2, w_3, \ldots$; w_1 denotes the first element of a sequence $w = w_1, w_2, w_3, \ldots$; and $f(e, e')$ is 0 when $e = e'$ and 1 otherwise.

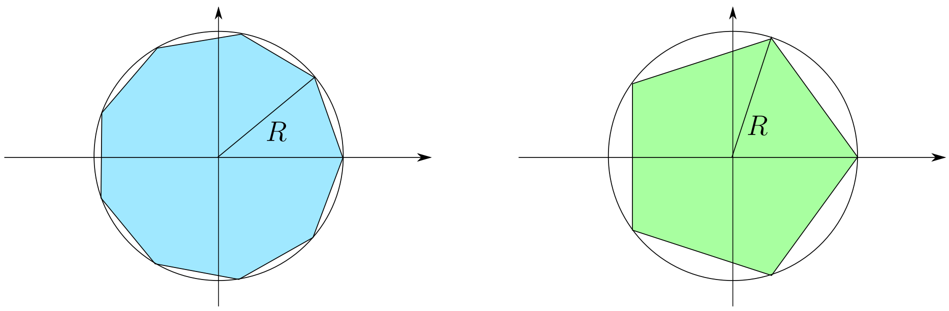
As an example you can easily check that the distance $d(\text{``}AB\text{''}, \text{``}B\text{''})$ between "AB" and "B" is 1 since:

$$d(\text{``}AB\text{''}, \text{``}B\text{''}) = min(d(\text{``}B\text{''}, \text{``}B\text{''}) + 1, d(\text{``}AB\text{''}, \text{``''}) + 1, d(\text{``}B\text{''}, \text{``''}) + 1) = min(1, 3, 2) = 1$$
$$d(\text{``}B\text{''}, \text{``}B\text{''}) = min(d(\text{``''}, \text{``}B\text{''}) + 1, d(\text{``}B\text{''}, \text{``''}) + 1, d(\text{``''}, \text{``''}) + 0) = min(2, 2, 0) = 0$$
$$d(\text{``}AB\text{''}, \text{``''}) = 2$$
$$d(\text{``}B\text{''}, \text{``''}) = 1$$
$$d(\text{``''}, \text{``}B\text{''}) = 1$$

Write a C++ implementation of this function. You can choose your favorite types or classes for the sequences of characters.

6. **Regular Polygons (15 points)**

   We want to implement a class to represent 2D regular polygons. In a regular polygon, all the sides are of equal length. For example, in the below figure, we have regular polygons with $N = 9$ sides (left) and $N = 5$ sides (right), respectively. R is the radius of the circumscribed circle, i.e. the distance from the center of the polygon to each of the vertices of the polygon.



   We define a templated `RegularPolygon` class as follows:

```
struct point {
	double x,y;
};

template<int N>
class RegularPolygon {
private:
	vector<point> vertices;
	double radius;
public:
	RegularPolygon(double radius);
	double area();
	double perimeter();
	void print_vertices();
};
```

   The template argument `N` is the number of sides of the polygon. Provide an implementation of the public interface of the class. Some details on the methods:

   - `RegularPolygon(double radius)`: constructs a polygon. The `radius` parameter, is the radius of the circumscribed circle of the polygon (see the figure above). The constructor must initialize the `vertices` and `radius` member variables with the vertices of the polygon and the radius respectively. The vertex (x_i, y_i) can be computed as

$$(\texttt{radius} * \cos(\frac{2 * \pi * i}{\texttt{N}}), \texttt{radius} * \sin(\frac{2 * \pi * i}{\texttt{N}}))$$

- `double area()`: returns the area of the polygon, i.e.

$$\frac{\texttt{N} * \texttt{radius}^2}{2} * \sin(\frac{2 * \pi}{\texttt{N}})$$

- `double perimeter()`: returns the perimeter of the polygon, i.e.

$$2 * \texttt{N} * \texttt{radius} * \sin(\frac{\pi}{\texttt{N}})$$

- `void print_vertices()`: prints all the vertices of the polygon on `std::cout` (see the output later for the specific format).

In your implementation you can assume to have a library of mathematical functions at hand (e.g. providing functions `sin` and `cos`).

An example usage of the class is as follows:

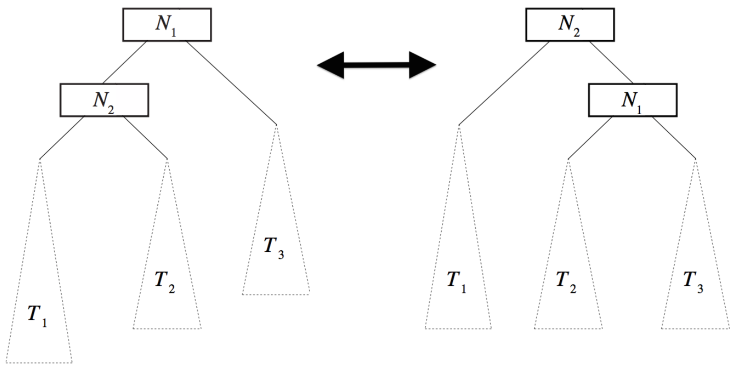
```
const int sides = 4;
RegularPolygon<sides> square(sqrt(2));
square.print_vertices();
cout << "Area:␣" << square.area() << endl;
cout << "Perimeter:␣" << square.perimeter() << endl;
```

That gives the output:

```
Point 1:  (1.41421, 0)
Point 2:  (0, 1.41421)
Point 3:  (-1.41421, 0)
Point 4:  (0, -1.41421)
Area:  4
Perimeter:  8
```

7. **Rotating a tree (15 points)**

The structure of a tree can be re-arranged using *rotations*. The following figure illustrates how they work. A *right-rotation* transforms the tree in the left of the figure into the tree in the right of the figure. Vice versa, a *left-rotation* transforms the right tree into the left one.



Consider the following type for the nodes of a binary tree:

```
struct Node {
    T * key;
    Node * father;
    Node * left;
    Node * right;
}
```

where the `left` and `right` attributes are used to point to the root nodes of the left- and right sub-trees, respectively; `father` points to the father of a node; and `key` points to an object of type `T` that records the data associated to a node. For example, in the tree on the left of the figure, N_1.`left` points to N_2 and N_2.`father` points to N_1.

You are asked to write a function implementing a right rotation. The function must have the following header:

```
Node * rightRotate(Node * x);
```

Parameter `x` is a pointer to the root of the tree to be rotated. The function must return a pointer to the new root of the tree being rotated. For example, if you consider the right rotation example of the figure, an invocation to `rightRotation(&`N_1`)` should return a pointer to N_2.

8. **Analysis of a function (15 points)**

   Consider the following function `f`:

```
unsigned int * f(unsigned int * a, unsigned int n, unsigned int m){
    int i, j;
    unsigned int c[m+1];
    unsigned int * b;

    b = new unsigned int[n];

    for(i = 0; i <= m; i++)
        c[i] = 0;

    for(j = 0; j < n; j++)
        c[a[j]] = c[a[j]] + 1;

    for(i = 1; i <= m; i++)
        c[i] = c[i] + c[i-1];

    for(j = n-1; j >= 0; j--){
        b[c[a[j]]-1] = a[j];
        c[a[j]] = c[a[j]] - 1;
    }

    return b;

}
```

   Answer to the following questions:

   (a) Compute `f(a,4,3)`, where `a` is an array of integers declared as `int a[4] = {2, 3, 2, 1}`.

   (b) Can you guess what the algorithm computes?

   (c) Determine the complexity of the function. Use the big-O notation and justify your answer. Hint: count the number of operations in the body of each loop. Count then how many times a loop is iterated. Finally, sum up the *cost* of the four loops.