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;
   ```

   The output is

   ```
   i=1
   b=7
   ```

   The key point is that the condition of the `if` is an assignment and not a comparison. Identifying such side effect and paying attention to the scope of `i` is also a fair enough answer.

   (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;
   }
   ```

   The output is

   ```
   i=4
   ```

Since `i` is passed by value, not by reference.

(c) Explain the difference between

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

and

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

In the first case `a` is a statically allocated array of 10 integers.

In the second case `a` is a statically allocated object of class `vector<int>`, namely a vector of integers. The object is created using one of the constructors, namely the one that allows one to specify the number of initial elements in the vector, i.e. 10 in this case.

In both cases `a` is statically allocated and will "die" when its scope ends. All memory allocated for `a` (possibly dinamically, in the case of the vector) will be deallocated. This includes the vector: the destructor of vector objects will be invoked and will take care of deallocating memory all memory allocated during `a`'s life (for the 10 initial elements or further elements).

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;
       result = 0;
       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.

   The program is not a correct implementation. It compiles and may not rise any run-time error (it depends of how the run-time deals with indexing arrays out of bounds).

   A possible correction is marked in red in the code. Another issue is a potential arithmetic overflow since the sum may exceed the maximum value of type `int`. This may be mitigated by replacing the return type to a type with a larger domain, like `double`.

   (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; The address of a
   cout << *a; 1
   cout << a[9]; 10
   cout << a[10]; undefined
   cout << *(a) + 3; 4
   cout << *(a+3); 4
   ```

   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.

There is no compile-time error. There may be run-time errors, for instance when accessing `a[10]` which is out of bounds.

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.

   We solved a similar exercise in the assignments. A simple solution is this one:

   ```
   template <typename T>
   void reverseR(vector<T> & v){

       T temp;

       for(int i = 0; i < v.size()/2; i++){
           temp = v[i];
           v[i] = v[v.size-(i+1)];
           v[v.size-(i+1)] = temp;
       }

   }
   ```

   Other solutions, for instance using recursion, are possible.

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?

   The answer is

```
10        // static dispatch
2         // static dispatch
3.14159   // static dispatch
3.14159   // dynamic dispatch
```

   The key point was in observing that method two is not dynamically dispatched since it is not declared as `virtual` in class `Citizen`.

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.

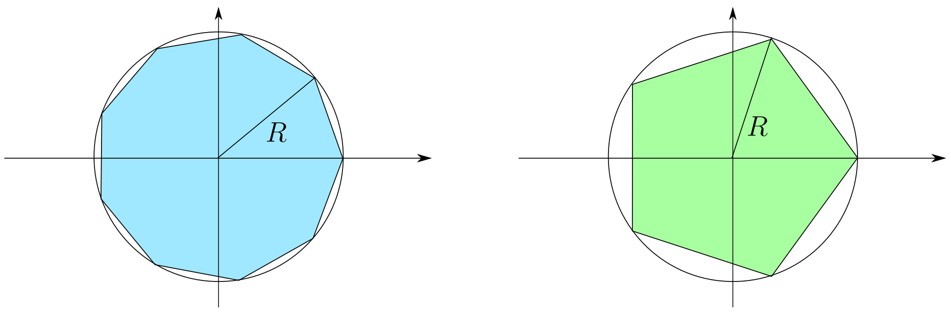
   Here is a simple implementation, using strings.

```
unsigned int d(string u, string v){

    if (u.size()==0) return v.size();
    if (v.size()==0) return u.size();

    string u1 = u.substr(1,u.size()-1);
    string v1 = v.substr(1,v.size()-1);

    return min(min(d(u1,v)+1,d(u,v1))+1,d(u1,v1)+(u[0]==v[0]?0:1));

}
```

   Similar solutions can be obtained using other data structures for the sequences, like vectors 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:

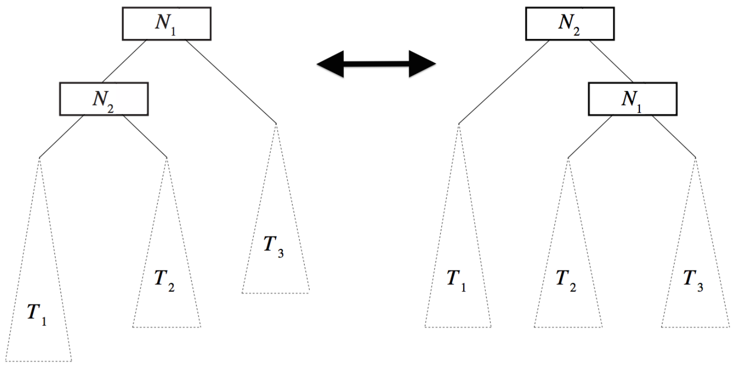
```
Point 1:  (1.41421, 0)
Point 2:  (0, 1.41421)
Point 3:  (-1.41421, 0)
Point 4:  (0, -1.41421)
Area:  4
Perimeter:  8
```

Here is an example implementation. The key point was in getting the constructor right, i.e. properly initializing the members `radius` and `vertices`.

```
template<int N>
RegularPolygon<N>::RegularPolygon(double radius) {
       this->radius=radius;
       points.resize(N);
       double angle = 2.0 * M_PI / N;
       for (int i = 0; i < N; i++){
              double currentAngle = angle * i;
              vec2 point = { cos(currentAngle) * radius, sin(currentAngle) * radius };
              points[i] = point;
       }
}

template<int N>
double RegularPolygon<N>::area(){
       return N * radius * radius * 0.5 * sin(2.0 * M_PI / N);
}

template<int N>
double RegularPolygon<N>::perimeter(){
       return 2 * radius * N * sin(M_PI / N);
}


template<int N>
void RegularPolygon<N>::print_points(){
       for (int i = 0; i < N; i++){
              cout << "Point␣" << i << ":␣(" << vertices[i].x
                     << ",␣" << vertices[i].y << ")" << endl;
       }
}
```

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.

Here is a possible solution. Note that we change the name of the parameter to make it easier to understand.

```
Node * rightRotate(Node * N1){

    Node * N2 = N1 -> left;

    // Re-arrange left/right pointers
    N1->left = N2->right;
    N2->right = N1;

    // Re-arrange father pointers
    N2->father = N1->father;
    N1->father = N2;

    // Update eventual father
    if(N2->father == nullptr){
        // Check if N1 was left or right son
        if(N2->father->left == N1){
            N2->father->left = N2;
        } else {
            N2->father->right = N2;
        }
    }

}
```

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}`.

   The array {`1, 2, 2, 3`} is returned.

   (b) Can you guess what the algorithm computes?

   The algorithm called is *counting sort* and sorts an array of n integers between 0 and m.

   (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.

The run-time complexity is $O(n + m)$, that is linear in the size n of the array and the size $m + 1$ of the domain of the numbers in the array (assuming domain $0..m$).

This can be seen as follows. Loops 1 and 3 iterate $O(m)$ times. Each iteration consists in checking the loop condition and executing the body. The loop condition just requires constant time (compare elements, increment, etc.). Likewise the bodies perform just a constant number of operations (assignments, access to array elements, etc.).

Similarly, loops 2 and 4 iterate $O(n)$ times. Each iteration takes constant time (similarly as above).