Practice test for midterm 3 – solutions

November 19, 2018

1 Classes

► Here is a pair of class definitions, and a pair of variable declarations:

```
class A {
  public:
    int x;
  private:
    float y;
};

class B {
  public:
    int y;
    A a;
  private:
    float x;
    A b;
};

A foo;
B bar;
```

Label each of the following expressions with OK if it is OK, or Error if it would cause an error:

a) `foo.x` OK

b) `bar.x` Error

c) `bar.a` OK

d) `foo.y` Error

e) `bar.b.x` Error

f) `bar.a.y` Error

► Add whatever *constructors* are necessary to the following class so that the code after it will be correct:

```
class A {
  public:

    // You can put whatever you want in the
    // definition of the constructor; it's the
    // parameter list that matters here.

    A(int v) {
        x = v;
    }

    A(float u, int v) {
        z = u;
        x = v;
    }

    A(string s, int u) {
        y = s;
        x = u;
    }

  private:
    int x;
    string y;
    float z;
};

A a1 = 12;
A a2{1.2, 10};
A a3{"Hello", 15};
```

► Here is a class with two member functions defined inside of it. Move the definitions of these member functions out of the class.

```
class dog {
  public:

    void feed();

    void walk();

  private:
    bool tired, fed;
};

void dog::feed() {
    if(!fed) {
        cout << "Dog is now fed.";
        fed = true;
    }
    else
        cout << "Dog is not tired.";
}

void dog::walk() {
    if(!tired) {
        cout << "Walkies";
        tired = true;
    }
    else
        cout << "Too tired to walk.";
}
```

► Complete the following class definition for a class that stores information about teachers by filling in the definitions of the member functions:

```
class teacher {
  public:

    void give_tenure() {
        has_tenure = true;
    }

    void assign_class(string c) {
        classes.push_back(c);
    }
    string get_name() {
        return name;
    }

  private:
    bool has_tenure = false;
    vector<string> classes;
    string name;
};
```

▶ Think about a class designed to represent a *color*. How would you represent a color? Would your representation support mixing colors together to get new colors? Sketch a class (data members and function declarations only) `color` and explain why you think it would work for this purpose (or explain what its limitations are).

There are all sorts of things you can do here. Storing the amount of the three (additive) primary colors, red, green, and blue, is common (e.g., as floats). You can also store the four subtractive primary colors: cyan, yellow, magenta, and black. You could represent colors as strings giving their names, so that mixing `"red"` with `"blue"` gives `"reddish-blue"`.

```
class color {
  public:
    float r,g,b;
};

color mix(color a, color b) {
    // Average
    return color{ (a.r + b.r)/2,
                  (a.g + b.g)/2,
                  (a.b + b.b)/2 };
}
```

2 Multi-file projects

▶ Suppose we want to split the following program into three files: `main.cpp`, `triangle.hpp` (containing declarations) and `triangle.cpp` (containing implementations). Circle the parts of the code that code into each file, and add anything else that would be needed to make the resulting project work.

```
// triangle.cpp:
#include <iostream>
#include <string>
#include "triangle.hpp" // Added
using namespace std;

void triangle::set_set(int s) {
    size = s;
}

void triangle::draw() {
    string t = "*";
    string s{size, ' '};

    for(int i = 1; i < size; ++i) {
        cout << s << t << s << endl;
        t += "**";
        s.pop_back();
    }
}

// triangle.hpp
#pragma once

class triangle {
  public:
    void set_size(int s);
    void draw();

  private:
    int size;
};

// main.cpp
#include "triangle.hpp" // Added

int main() {
    triangle t;
    t.set_size(10);
    t.draw();
    return 0;
}
```

▶ What are the commands you would use to manually compile the project in the previous problem?

```
g++ -c main.cpp
g++ -c triangle.cpp
g++ -o main main.cpp triangle.cpp
```

▶ Explain what the rules are for the order of object (`.o`) files in the final *link step*. If `A.cpp` uses definitions from `B.cpp`, where should `A.o` appear in the list of object files, relative to `B.o`?

If A.cpp uses B.cpp, then A.o should come before B.o.

▶ For each of the following, state whether it can/should appear in *source* files, *header* files, or both:

a) Function definitions: Source only

b) Function declarations: Both

c) `using namespace std;`: Preferably only in source

d) `#include<...>`: both

e) `#pragma once`: header

f) `int main()`: source

▶ Explain what problem header files are intended to solve; why do we need `.hpp` files at all?

Headers contain declarations and class definitions; in order to use a function or class, its declaration/definition must be visible. So rather than copy-paste everything we need into many different .cpp files, we can put it in one .hpp file and then `#include` it anywhere we need it.

3 Exceptions

▶ What is wrong with the following code? How would you fix it?

```
try {
    f();
}
catch(logic_error& e) {
    cout << "LE";
}
```

```
catch(length_error& e) {
    cout << "LenE";
}
catch(runtime_error& e) {
    cout << "RE";
}
catch(range_error& e) {

}
```

A `length_error` is a kind of `logic_error`, so the `logic_error` catch will catch both. Similarly for `runtime_error` and `range_error`. The fix is to rearrange them so that the more-specific exception types come first.

► The following function takes a vector of pairs of ints and divides the first element of each pair by the second. E.g., if the input vector was {4, 2, 9, 3, 12, 3} then the returned vector would be {2, 3, 4}. What kinds of errors could occur in this function? Add assertions to check for them.

```
#include <cassert>
vector<int> divide_by(vector<int> v) {

    assert(v.size() % 2 == ); // Even size

    vector<int> vout;



    for(int i = 0; i < v.size(); i += 2) {

        assert(v.at(i) != 0); // No divide by 0

        vout.push_back(v.at(i) / v.at(i+1));


    }


    return vout;
}
```

► For each of the standard expression types to the right, indicate what the following code would print if it were thrown from the function h

```
void h() {
    throw   // exception thrown here
}

void g() {
    try {
        h();
    }
    catch(domain_error& e) {
        cout << "DE in g";
    }
    catch(runtime_error& e) {
        cout << "RE in g";
    }
}

void main() {
    try {
        g();
        h();
    }
    catch(range_error& e) {
        cout << "RE in main";
    }
    catch(out_of_range& e) {
        cout << "OOR in main";
    }
    catch(logic_error& e) {
        cout << "LE in main";
    }
    catch(...) {
        cout << "Other in main";
    }
}
```

a)
```
domain_error
DE in g
LE in main
```

b)
```
range_error
RE in g
RE in main
```

c)
```
out_of_range
OOR in main
```

d)
```
length_error
LE in main
```

e)
```
system_error
RE in g
Other in main
```

f)
```
exception
Other in main
```

► Explain the difference between assertions and expressions. When would you use each?

Assertions are used to "check your work"; you add them to *assert* things that should always be true (and which, if false, would mean something has gone very wrong in your program). If an assertion fails, it ends the program immediately. Exceptions are used for situations which your program could potentially recover from.

► The following code uses *assertions* to check for problems. Convert it to using standard exceptions (and choose exception types that seem appropriate to you).

```
// Uses remainder hashing to compute the hash value of a string s.
// Take CSci 133 if you want to know more!
int hash(string s, int m) {
    if(s.empty())
        throw length_error; // Input string cannot be empty

    if(m ≤ 0)
        throw domain_error; // Size must be positive

    int h = 0;
    for(char c : s) {

        if(c ≤ 0)
            throw invalid_argument; // No non-ASCII characters

        if(256 * h + c ≤ h)
            throw overflow_error; // No numeric overflow

        h = (256 * h + c) % m;
    }

    return h;
}
```