PIC 10A 1C Week 10b Exercises. TA: Eric Kim. [Solutions]

1. Retail Therapy

a.

```
class ShopCart {
public:
    ShopCart(); // Initialize with empty cart
    ShopCart(const vector<string>& items); // Initialize with given items
    void add(const string& item); // Adds item to the cart
    size_t size() const; // Returns number of items in cart
    string get_item(size_t i) const; // returns item at index i, or empty string "" if
                                     // not valid index
private:
    vector<string> items;
};
```

A ShopCart is used to store items that we want to buy, ie at the grocery store, or on Amazon:

```
    ShopCart mycart;
    mycart.add("Scrubs Season 4");    mycart.add("Stuffed Dog");
    cout << mycart.size() << endl; // Outputs: 2
```

Define the class implementation that achieves the above desired behavior.

[Solution]

```
ShopCart::ShopCart() : items() {}
/* Note that the following could have worked too:
    ShopCart::ShopCart() {}
This works because "items" would be initialized by its default constructor. The
default constructor of the vector class creates an empty vector object, which
is exactly what we wanted. However, I think it's better style to use the
former, since it's more explicit and potentially less confusing.
*/
ShopCart::ShopCart(const vector<string>& items) : items(items) {}
/* Passing a vector into a vector initializer copies the contents of the first
vector into the second vector:
    vector<int> nums1 = {1, 2, 3};
    vector<int> nums2(nums1); // nums2 is: [1, 2, 3]
Alternately, if you did not know this, we could have done it the long way:
    ShopCart::ShopCart(const vector<string>& items) {
        for (string item : items) {
            this->items.push_back(item);
        }
    }
*/
void ShopCart::add(const string& item) {
    this->items.push_back(item);
}
size_t ShopCart::size() const {
    return this->items.size();
}
string ShopCart::get_item(size_t i) const {
```

```
    if ((i < 0) || (i >= this->items.size())) {
        return "";
    }
    return this->items[i];
}
```

b.

We want to be able to add the contents of one cart to another via the "+=" operator:

```
    ShopCart jdcart;    jdcart.add("Appletini");
    ShopCart janitorcart;    janitorcart.add("pager");
    jdcart += janitorcart;
    cout << jdcart.size(); // Outputs: 2, stores: ["Appletini","pager"]
    cout << janitorcart.size(); // Outputs: 1, stores: ["pager"]
```

Define the "+=" operator to implement the above desired behavior.

[Solution]

```
ShopCart& operator+=(ShopCart& left, const ShopCart& right) {
    for (size_t i = 0; i < right.size(); ++i) {
        left.add(right.get_item(i));
    }
    return left;
}
```

c.

Next, define the "<<" operator so that we can display carts in the following way:

```
    vector<string> items = {"knifewrench", "mop"};
    ShopCart janitorcart(items);
    cout << janitorcart << endl; // Displays: ShopCart(2, {"knifewrench",
"mop"})
```

In other words, the ShopCart should be displayed as:

```
    ShopCart(<nb. of items>, {"<item1>", "<item2>", ..., "<itemN>"})
```

Note: It's OK for your solution to have an extra space at the end of the list, ie:

```
    ShopCart(2, {"knifewrench", "mop" }); // note the space after "mop"
```

[Solution]

```
ostream& operator<<(ostream& os, const ShopCart& right) {
    os << "ShopCart(" << right.size() << ", {";
    for (size_t i = 0; i < right.size(); ++i) {
        os << "\"" << right.get_item(i) << "\", ";
    }
    os << "})";
    return os;
}
```

d.

Define the "<" operator so that we can compare carts based on the total number of items in the cart:

```
    ShopCart dr_reid;
    dr_reid.add("coffee"); dr_reid.add("clipboard");
    ShopCart turkleton;
```

```
    turkleton.add("pancake");
    if (turkleton < dr_reid)
        cout << "Turk has fewer items than Dr. Reid";
```

[Solution]

```
bool operator<(const ShopCart& left, const ShopCart& right) {
    return (left.size() < right.size());
}
```

2. string2double2string

Suppose I have a vector of strings that contain decimal values:

```
    vector<string> v1 = {"2.0", "1.32", "2.44", "4.2"};
```

Write a function that doubles each of these values, but keeps each value as a string:

```
    funny_double(v1); // v1 is now: ["4.0", "2.64", "4.88", "8.4"]
```

Hint: You'll want to use istringstream and ostringstream.

[Solution]

```
void funny_double(vector<string>& v) {
    for (size_t i = 0; i < v.size(); ++i ) {
        istringstream iss(v[i]); // populate iss with string-number
        double d;
        iss >> d; // Convert string to double!
        ostringstream oss;
        oss << 2*d; // Convert 2*d into a string, via oss
        v[i] = oss.str(); // str() returns oss as a string
    }
}
```

3. I'm in your base, overloading your mans.

a.

Consider the following code. What is the expected output? If there is an error, explain why.

```
string foo(int a, int b) {
    if (a < b)
        return "foo";
    return "bar";
}
string foo(int a, double b) {
    if (a < b)
        return "baz";
    return "garply";
}
cout << foo(2, 1) << endl;
cout << foo(2, 1/2) << endl;
cout << foo(2, 1.0) << endl;
```

Output:
[Solution]
bar
bar
garply

b.

Suppose I added the following function to my file:

```
int foo(int a, double b) {
    if (a < b)
        return 42;
    return 0;
}
```

My code no longer compiles correctly. Why?

[Solution]

The compiler will error because there are two foo functions with the same *input* parameters. Recall that only **input parameters** matter for function overloading - even if the return types differ (ie string vs int), we still must ensure that the input parameters differ.

4. We all make mistakes

Louis Reasoner wants to write the following program:

(1) Creates a file "mynums.txt", and writes the first 5 even numbers to the file, each on a separate line.

(2) Reads the file we just created ("mynums.txt"), but outputs the square of each number.

The below code attempts to accomplish the spec:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    /* (1) Write the first 3 even numbers to: mynums.txt */
    ofstream out("mynums.txt");
    for (int i = 0; i < 3; ++i) {
        out << i*2 << endl;
    }
    out.close();
    /* (2) Read in the numbers from mynums.txt, and output the square of each
number */
    ifstream myfile("mynums.txt");
    int num;
    while (!myfile.eof()) {
        myfile >> num;
        cout << num*num << endl;
    }
    cout << "Done!" << endl;    return 0;
}
```

However, the output is strange:

```
0
4
16
16
Done!
```

Why did the last number get output twice?

Hint: Consider that myfile's buffer looks like: "0\n2\n4\n". How would cin/ifstream process this?

Hint 2: After the while loop ends, myfile is in a failure state:

```
cout << "isfail? " << myfile.fail() << endl; // displays: "isfail? 1"
```

[Solution]

The while-loop actually does 4 iterations, not 3! Let's display the contents of myfile's buffer during/after each iteration. myfile starts off as: "0\n2\n4\n". Then:

Iteration 1: Display: 0*0 = 0
After Iteration 1: "\n2\n4\n"
Iteration 2: Display: 2*2 = 4
After Iteration 2: "\n4\n"
Iteration 3: Display: 4*4 = 16
After Iteration 3: "\n"
Iteration 4: myfile fails! Can't do: myfile >> num, since "\n" is not a valid integer! However, we still do the next line ("cout << num*num << endl"), and since num was still 4 from the previous iteration, it outputs 16 again.

If this is confusing, try reviewing how cin (which is similar to an ifstream) works (slide 9 of my week3b notes):
http://eric-kim.net/teaching/pic10a_page/slides/pic10a_1c_week3b.pdf

b.

Can you write a fixed version of the while loop in (2)?

[Solution]

The simplest fix is to simply check for the failure state:

```
while (!myfile.eof()) {
    myfile >> num;
    if (myfile.fail() == 1) {
        break;
    }
    cout << num*num << endl;
}
```