PIC 10A 1C. Week 9B Problems. TA: Eric Kim

1. Class Act

Given this course's rules, determine what's wrong with this class definition, and how to fix it:

```
class Student {
public:
  Student(string name, unsigned int age) {
    name = name;
    age = age;
  }
  Student() { name = "John Doe"; age = 30; }

  string get_name() { return name; }
  unsigned int get_age() { return age; }
  void set_age(unsigned int newAge) const { age = newAge; }
  string name;  unsigned int age;
};
```

2. Swpa This!

2.a. Define the swap1 function, which given two integers, swaps them in the following way:

```
int x = 3, y = 5;
swap1(x, y);
cout << "x: " << x << " y: " << y; // x: 5 y: 3
```

2.b. Define the swap2 function, which given two integers, swaps them in the following way:

```
int x = 3, y = 5;
swap2(&x, &y);
cout << "x: " << x << " y: " << y; // x: 5 y: 3
```

2.c. Suppose we have the following swap3 function:

```
void swap3(int x, int y) {
  int xtmp = x;
  x = y;
  y = xtmp;
}
```

What is the output of the following code?

```
  int x = 3, y = 5;
  swap3(x, y);
  cout << "x: " << x << " y: " << y;
```

3. Pirates vs Ninjas

Consider the following classes:

```
class Pirate {
private:
  unsigned int age;
public:
  Pirate() : age(30) {}
  unsigned int get_age() const { return age; }
  void set_age(unsigned int age) { this->age = age; }
};
class Ninja {
private:
  unsigned int age;
public:
  Ninja() : age(35) {}
  unsigned int get_age() const { return age; }
  void set_age(unsigned int age) { this->age = age; }
};
```

Write a single function template called `birthday()` that given either a Pirate or a Ninja, increments their age by one, for example:

```
Pirate p;  Ninja n;
cout << "Pirate age: " << p.get_age() << endl;
birthday(p);
cout << "Pirate age: " << p.get_age() << endl;
cout << "Ninja age: " << n.get_age() << endl;
birthday(n);
cout << "Ninja age: " << n.get_age() << endl;
```

Output:

```
Pirate age: 30
Pirate age: 31
Ninja age: 35
Ninja age: 36
```

4. Templates for Success

Consider the following template function:

```
template <typename T>
T maxvalue(const T& a, const T& b) {
  return (a > b) ? a : b;
}
```

Louis Reasoner is working on a project, and has the following line in his code:

```
    double bigger = maxvalue(3, 4.4);
```

There is a compiler error, complaining about this line. What is wrong? List two different ways to fix the line.

5. Who needs default constructors anyways?

```
class A {
public:
    A(int x) : myx(x) {}
private:
    int myx;
};
```

```
class B {
private:
    A mya;
};
```

In my code, I have the following line:

```
    B b;
```

However, my code has a compile error. What is the issue?

6. Error: Cat does not compile.

The following class has a compiler error. What is the issue, and how can we fix it?

```
class Cat {
public:
    Cat(int str) : myStr(str) {
        myID = rand();
    }
private:
    int myStr;
    const int myID;
};
```

7. Hip to be square

Consider the following function vecsquare:

```
/**
 * Squares the integers of a vector. Modifies (mutates) the vector.
 * @param vec The input vector of integers.
 * @return void
 **/
void vecsquare(vector<int> vec) {
  for (size_t i = 0; i < vec.size(); ++i) {
    vec[i] = vec[i]*vec[i];
  }
}
```

The desired usage is:

```
    vector<int> v = {2, 3, 4};
    vecsquare(v); // v is now: [4, 9, 16]
```

Does the function vecsquare behave correctly? If not, describe why not, and suggest a simple fix.

8. Flip-floppin'

Write a function vec_reverse that, given a vector of ints, reverses the order:

```
    vector<int> myints = {8, 6, 7, 5, 3, 0, 9};
    vec_reverse(myints); // myints is now: [9, 0, 3, 5, 7, 6, 8]
```

9. 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.

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.

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"
```

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";
```

10. #justpic10Athings*

We would like to write a program that, given a string of '+' and '#', computes a more compact version of the original string. For instance, here are some expected outputs:

`"+++##"` `Becomes: "+3#2"`	`"++++#++"` `Becomes: "+4#1+2"`
`"+#"` `Becomes: "+1#1"`	`"+++++"` `Becomes: "+5"`
`""` `Becomes: ""`	`"#++########++++"` `Becomes: "#1+2#3+4"`

Write a program that, given such a user-inputted string, outputs its compressed version:

```
Enter a string: ++###
+2#3
```

Aside: This is a form of run-length encoding, a technique used to compress data into a smaller (yet equivalent) form. For instance, when you compress a file to a .zip file, the compression program is likely using this principle to achieve a much smaller file size!

As you can imagine, some types of data are more amenable to compression that others. A file with lots of long runs, ie "+++++++++", compress well, whereas files with only short runs, ie "+#+#+#+", compress poorly.