## 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; // shadowing
  //  age = age;
  //}
  Student(const string& name, const unsigned int age) : name(name), age(age) {}
  //Student() { name = "John Doe"; age = 30; }
  Student() : name("John Doe"), age(30) {}
  string get_name() const { return name; }
  unsigned int get_age() const { return age; }
  void set_age(unsigned int newAge) ~~const~~ { age = newAge; }
private:
  string name;  unsigned int age;
};
```

**[Solution]**

```
(1) Constructors should use member initialization lists, ie:
    Student(const string& name, unsigned int age) : name(name), age(age) {}
    Student() : name("John Doe"), age(30) {}
(2) string parameters should be passed by reference, not by value (copy)!
(3) Also: in original two-argument constructor, the body didn't actually set the
Student's name/age member variables! Local variable shadowing.
(4) get_name(), get_age() should be declared const, ie:
    string get_name() const { return name; }
    unsigned int get_age() const { return age; }
(5) Member variables "name", "age" should be declared private.
```

## 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
```

**[Solution]**

```
void swap1(int& x, int& y) {
    int xtmp = x;
    x = y;    y = xtmp;
}
```

**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
```

**[Solution]**

```
void swap2(int* x, int* y) {
    int xtmp = *x;
    *x = *y;    *y = xtmp;
}
```

**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;
```

**[Solution]**

```
Output is:
    x: 3 y: 5
It doesn't actually swap the values! swap3() only swaps its local variables, since x
and y are passed by value (copies), not by reference/pointer.
```

## 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;` | **Output:** |
|---|---|

| | |
|---|---|
| `cout << "Pirate age: " << p.get_age() << endl;`<br>`birthday(p);`<br>`cout << "Pirate age: " << p.get_age() << endl;`<br>`cout << "Ninja age: " << n.get_age() << endl;`<br>`birthday(n);`<br>`cout << "Ninja age: " << n.get_age() << endl;` | `Pirate age: 30`<br>`Pirate age: 31`<br>`Ninja age: 35`<br>`Ninja age: 36` |

**[Solution]**

```
template <typename T>
void birthday(T thing) {
    thing.set_age(thing.get_age() + 1);
}
// If I didn't have templates, then I'd have to do something like this using
// function overloading:
void birthday(Pirate thing) {
    thing.set_age(thing.get_age() + 1);
}
void birthday(Ninja thing) {
    thing.set_age(thing.get_age() + 1);
}
```

## 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:

```
    T 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.

**[Solution]**

The compiler complains that the function template doesn't match, since maxvalue expects both inputs to be of the same type, but here, we are passing two different types: a double, and an int.

One fix: explicitly call the double-version of maxvalue:

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

Second fix: cast the int to a double:

```
double bigger = maxvalue(static_cast<double>(3), 4.4);
```

## 5. Who needs default constructors anyways?

| | |
|---|---|
| `class A {`<br>`public:`<br>`    A(int x) : myx(x) {}`<br>`private:` | `class B {`<br>`private:`<br>`    A mya;`<br>`};` |

| | |
|---|---|
| `    int myx;`<br>`};` | |

In my code, I have the following line:

```
B b;
```

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

**[Solution]**

Note that class A does not have a default constructor, since we defined a one-argument constructor.
When we create the "B b" object, since "A mya" is a member variable of the B class, we default-initialize mya by calling its default-constructor. However, the A class does not have a default constructor, so the compiler complains.

## 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;
};
```

**[Solution]**

The Cat constructor is trying to modify a const member variable (myID). Instead, it should set it in its member initialization list, ie:

```
Cat(int str) : myStr(str), myID(rand()) {}
```

## 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.

**[Solution]**

Nope! vecsquare makes a *copy* of the input vector, and modifies only the local copy. The simplest fix is to pass the input vector to vecsquare by reference:

```
void vecsquare(vector<int>& vec) { // Only change: &
  for (size_t i = 0; i < vec.size(); ++i) {
    vec[i] = vec[i]*vec[i];
  }
}
```

## 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]
```

**[Solution]**

Here, we iteratively swap each element. I am very careful to stop at the "halfway" point, otherwise the code would start undo-ing its previously-swapped changes! For instance, for a vector with even-length, ie [1, 3, 2, 0], we should stop at (vec.size()/2) = 2. For a vector of odd-length, ie [4, 2, 1], we should stop at (vec.size()/2) = 1.

```
void vec_reverse(vector<int>& vec) {
    for (size_t i = 0; i < (vec.size()/2); ++i) {
        /* indswap: index to swap with */
        size_t indswap = vec.size() - i - 1;
        int tmp = vec[i];
        vec[i] = vec[indswap];
        vec[indswap] = tmp;
    }
}
```

## 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.

**[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());
}
```

## 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:

| | |
|---|---|
| "+++##"<br>Becomes: "+3#2" | "++++#++"<br>Becomes: "+4#1+2" |
| "+#"<br>Becomes: "+1#1" | "+++++"<br>Becomes: "+5" |
| ""<br>Becomes: "" | "#++#####+++"<br>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.

**[Solution]**

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    cout << "Enter a string: ";
    string s;
    getline(cin,s);
    if (s.size() == 0)
        return 0; // Terminate early.
    size_t i = 0, cnt = 0;
    bool isplus = (s[0] == '+'); // initialize
    while (i < s.size()) {
        if (isplus && (s[i] == '+')) {
            cnt = cnt + 1;
        } else if (isplus && (s[i] == '#')) {
            /* Switch! '+' -> '#' */
            isplus = false;
            cout << "+" << cnt;
            cnt = 1; // We've seen 1 '#'
        } else if (!isplus && (s[i] == '#')) {
            cnt = cnt + 1;
        } else if (!isplus && (s[i] == '+')) {
            /* Switch! '#' -> '+' */
            isplus = true;
```

```
            cout << "#" << cnt;
            cnt = 1; // We've seen 1 '+'
        }
        i = i + 1;
    }
    /* Output final run */
    if (isplus)
        cout << "+" << cnt;
    else
        cout << "#" << cnt;
    return 0;
}
```

Here's an alternate solution (suggested by a student) where each iteration of the for-loop explicitly looks ahead to determine if we are switching from '+' to '#' (or vice-versa). To avoid indexing out of bounds, we add a padding character 'X' to the end of the input string, and take care to only iterate from 0 to (s.size()-1). In my opinion, this code is easier to understand:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s;
    /* Ask user for input string */
    cout << "Enter a string: ";
    cin >> s;
    if (s.size() == 0)
        return 0; // if input is empty, program terminates early
    /* plus, hash keep track of nb of '+','#' seen during current run */
    unsigned int plus = 0, hash = 0;
    /* Hack: add padding char 'X' at end to avoid indexing s out of bounds */
    string spad = s + "X";
    /* Perform runlength encoding of s */
    for (size_t i = 0; i < (spad.size()-1); ++i) {
        if (spad[i] == '+') {
            ++plus;
            if (spad[i + 1] != '+') {
                // prints out the + value if the following value is not a +
                cout << "+" << plus;
                plus = 0; // resets the plus value
            }
        } else if (spad[i] == '#') {
            ++hash;
            if (spad[i + 1] != '#') {
                // prints out the # value if the following value is not a #
                cout << "#" << hash;
                hash = 0; // resets the hash value
            }
```

```
        }
    }
    cin.ignore();
    cin.get();
    return 0;
}
```