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

Revisit: const correctness

```
int vecsum(vector<int>& myvec) {
    int out = 0.0;
    for (size_t i = 0; i < myvec.size(); ++i) {
        out += myvec[i];
    }
    return out;
}
const vector<int> nums = {1, 2, 3};
int s = vecsum(nums); // CompileError!
```

The above is a compiler error because the vecsum() function does not declare its input reference parameter "myvec" as const. In other words, when the compiler sees:

```
    int vecsum(vector<int>& myvec)
```

The compiler infers that the input vector may be modified within the vecsum() function. Thus, when it sees vecsum(nums), the compiler sees that we are passing something that shouldn't be modified (nums) to a function that may modify its input. Thus, the compiler throws up its arms, complains that something may go wrong (ie modify a const value), and issues a compile error. This compiler error happens even if the function doesn't modify the input vector!

The recommended way to fix this is:

```
    int vecsum(const vector<int>& myvec); // Make it const
```

The same idea holds for class member functions: if a method does not modify the object, then declare it const!

```
    class A {
    public:
        int get_x() const { return this->x; } // declared const method!
    ...
    };
```

Default Constructors

A default constructor is simply a no-argument constructor. If you define a class but don't define any constructors, then the compiler will create a no-argument constructor for you:

```
class A {
};
...
A mything; // Compiles! Calls no-arg constructor
```

What happens if the class has member variables?

```
class A {
public:
    int get_x() const { return this->myx; }
    string get_s() const { return this->mys; }
private:
    int myx;
    string mys;
};
...
```

```
A a; // What are myx, s?
```
Here, the values of **fundamental types** (int, double, char, bool, size_t), will not be initialized at all! Thus, they will have some garbage value:
```
    cout << "x: " << a.get_x(); // Outputs: x: -858993460
```

However, **objects** (ie string, vector, user-defined classes, non-fundamental types) will be initialized by their default constructor (ie no-argument constructor):
```
    cout << "s:" << a.get_s() << "."; // Outputs: s:.
```
Here, s is set to the empty string, since it's initialized as:
```
    string mys; // no-arg constructor is the empty string ""
```
If you define any constructor, then the compiler will no longer automatically define a no-argument constructor for you - you will have to define it yourself:
```
class A {
public:
    A(int x, string s) : myx(x), mys(s) {
    }
private:
    int myx; string mys;
};
...
A a(42, "hi"); // Call two-arg constructor
A a; // CompileError: no-arg constructor does not exist!
```
Thus, we would have to define our own no-arg constructor:
```
class A {
public:
    A(int x, string s) : myx(x), mys(s) {
    }
    A() : myx(0), mys("moo") {}
private:
    int myx; string mys;
};
```
In my opinion, it's always a good idea to explicitly define the no-argument constructor so that you can explicitly initialize member variables to reasonable starting values.
Finally - if you fail to initialize a member variable during the constructor, then the compiler will initialize it by the following rule:

If the member var is a fundamental type, leave it alone as **uninitialized**.

Else, initialize it via its **default (one-arg) constructor**.
```
class B {
public:
    B() : mychar('a') {}
private:
    char mychar;
};
class A {
public:
    A(int x, string s) : myx(x), mys(s) {} // Does not initialize b!
private:
    int myx; string mys; B b;
```

```
};
...
A a(42, "meow"); // b is initialized via its no-arg constructor
```

Operator Overloading

If you've ever wanted to define what it means to add two objects together, or check to see if one object is "greater" than another Book, then look no further!

operator+=

Suppose we have a Wallet class that stores an amount of money:

```
class Wallet {
public:
    Wallet(double amt) : amt(amt) {}
    double get_amt() const { return this->amt; }
    double set_amt(double a) { this->amt = a; }
private:
    double amt;
};
```

If we wanted to define the '+=' operator so that we can add the contents of one Wallet to another:

```
Wallet& operator+=(Wallet& left, const Wallet& right) {
    left.set_amt(left.get_amt() + right.get_amt();
    return left;
}
Wallet ericw(0.50); // grad student financing
Wallet billw(1e9); // 1 billion dollars
ericw += billw; // Eric now has 1 billion dollars, 50 cents!
```

To understand how this works, note that the "ericw += billw" is equivalent to:

```
    operator+=(ericw, billw); // Equiv: eric += billw;
```

To make sense of the return value of operator+=, observe the following:

```
    int i = 2, j = 3, k = 4;
    i += j += k;
```

We evaluate right-to-left: (j += k) is equiv to: operator+=(j,k), which performs j = j + k, and returns the (new) value of j, which is 7:

```
    i += (j += k); // Updates: j = 3+4 = 7. Returns: j
    => i += j; // Updates: i = 2+7 = 9. Returns: i
    => i;
```

Another way to break it down is to convert it to the functional operator+= notation:

```
    i += j += k;
    => operator+=(i, operator+=(j, k)); // Updates: j = 3+4 = 7
    => operator+=(i, j); // Updates: i = 2+7 = 9
    => i;
```

operator<

Further, we can define the comparison operator <, ie less than:

```
bool operator<(const Wallet& left, const Wallet& right) {
    return left.get_amt() < right.get_amt();
}
Wallet ericw(0.50);
Wallet billw(1e9);
cout << (ericw < billw); // Outputs: 1, since 0.50 < 1e9.
```

Note that you have to separately define the <=, >, >= as: operator<=, operator>, operator>=.

operator<<

Finally, you can customize how an object is printed when given to cout!

```
ostream& operator<<(ostream& os, const Wallet& w) {
    os << setprecision(2) << fixed << "Wallet($" << w.get_amt() << ")";
    return os;
}
Wallet ericw(12.50);
cout << ericw << endl; // Outputs: Wallet($12.50)
```

To understand how the above works, we first write it in functional form:

```
    cout << ericw << endl;
    => operator<<(operator<<(cout, ericw), endl);
```

Then, we evaluate starting inside, then working our way out:

=> operator<<(**operator<<(cout, ericw)**, endl); // Outputs: Wallet($12.50). Returns: cout

=> **operator<<(cout, endl)**; // Outputs: newline. Returns: cout

=> cout;

A few things to point out:

(1) cout is of type ostream, which explains the first argument type.

(2) Inside of operator<<, we return the ostream os by **reference**, rather than by **copy-value**. Indeed, this is a requirement: we are not allowed to ever make a copy of cout. If we had instead written:

```
ostream operator<<(ostream& os, const Wallet& w) {
    os << setprecision(2) << fixed << "Wallet($" << w.get_amt() << ")";
    return os;
}
```

This function would not compile, since here when we return the os object, we make a *copy* of it to the caller. However, since it is forbidden to make a copy of an ostream object, there is a compile error.

Exercises

1. Mean Girls

Consider the following function that computes the mean (average) of a vector of ints:

```
double vecmean(vector<int>& nums) {
   double mean = 0.0;
   for (size_t i = 0; i < nums.size(); ++i) {
       mean = mean + nums[i];
   }
   return mean / nums.size();
}
```

Which of the following lines have a compile error, if any?

```
    vector<int> v1 = {1, 2, 3};
    double mn1 = vecmean(v1);
    const vector<int> v2 = {2, 8};
    double mn2 = vecmean(v2);
    double mn3 = vecmean(&v1);
```

[Solution]

```
    const vector<int> v2 = {2, 8};
    double mn2 = vecmean(v2);
```

Compile Error! Since v2 is const, can't pass v2 to vecmean(), since vecmean does not declare nums as const.

```
    double mn3 = vecmean(&v1);
```

Compile Error! &v1 takes the address of v1, but vecmean expects a vector<int> object, not a vector<int> pointer! For vecmean(&v1) to work, vecmean would have to be defined as:

```
    double vecmean(vector<int>* nums) {
       double mean = 0.0;
       for (size_t i = 0; i < nums->size(); ++i) {
           mean = mean + (*nums)[i];
       }
       return mean / nums->size();
    }
```

2. Ashes to Ashes, Apples to Apples

```
class Apple {
public:
    Apple(int yum) : yum(yum) {}
private:
    int yum;
};
class Basket {
public:
    Basket(size_t size) : apples(vector<Apple>(size)) {}
private:
    vector<Apple> apples;
};
```

I claim that the above class definitions contains an error. What is it? How can you fix it?

[Solution]

When we create the vector<Apple>(size), we try to create a vector with "size" number of elements, each entry being default-initialized to an Apple object using its default (no argument) constructor. However, the Apple class does not have a no-argument constructor defined!

To fix it, we should explicitly define the no-arg constructor:

```
class Apple {
public:
    Apple(int yum) : yum(yum) {}
    Apple() : yum(1) {}
private:
    int yum;
};
```

3. cout-in' that Bass

```
class Bassist {
public:
    Bassist(int lvl) : funklvl(lvl) {}
    Bassist() : funklvl(1) {}
    int get_funklvl() const { return this->funklvl; }
private:
    int funklvl;
};
```

We want to overload the << operator for this class by adding a fun twist: rather than simply output a bassist's funk level, we want to output a title for each rank:

if funk level < 3, then the title is: "Understudy"
if 3 <= funk level < 7, then the title is: "InThePocket"
if funk level >= 7, then the title is: "GotThatSoul"

Example:

`Bassist louis(1);`
`cout << louis << endl;`
`cout << Bassist(6) << endl;`
`Bassist jamerson(10);`
`cout << jamerson << endl;`
	Output
	Understudy(1)
	InThePocket(6)
	GotThatSoul(10)

[Solution]

```
ostream& operator<<(ostream& os, const Bassist& b) {
    int funklvl = b.get_funklvl();
    if (funklvl < 3) {
        os << "Understudy(" << funklvl << ")";
    } else if ((3 <= funklvl) && (funklvl < 7)) {
        os << "InThePocket(" << funklvl << ")";
    } else if (funklvl >= 7) {
        os << "GotThatSoul(" << funklvl << ")";
    }
    return os;
}
```