PIC 10A 1C Week 5a Notes. TA: Eric Kim

IO Manipulation

C++ offers several features to easily display text output in an organized (and even aesthetically pleasing) manner. This functionality is part of the iomanip standard library (#include <iomanip>).

Widths (setw) and Padding (setfill)

Code:
```
cout << "Age vs Weight"
    << endl;
cout << 6 << " " << 40
    << endl;
cout << 18 << " " << 120
    << endl;
cout << 35 << " " << 130
    << endl;
```

Output:
```
Age vs Weight
6 40
18 120
35 130
```

Code:
```
cout << "Age vs Weight" << endl;
cout << setfill('0');
cout << setw(3) << 6 << " " << setw(4) << 40
    << endl;
cout << setw(3) << 6 << " " << setw(4) << 40
    << endl;
cout << setw(3) << 18 << " " << setw(4)
    << 120 << endl;
cout << setw(3) << 35 << " " << setw(4)
    << 130 << endl;
```

Output:
```
Age vs Weight
006 0040
018 0120
035 0130
```

Take care that you pass a **character** to setfill(), rather than a string literal. For instance, don't do `setfill("0")`, instead do `setfill('0')`.

Note: setw ("set width") only modifies the next output, ie **short-term** change. This is why I had to repeat setw(3) and setw(4) for each output. On the other hand, setfill() modifies all future outputs, ie **long-term** changes.

setprecision, fixed/scientific

We can set the number of digits to display via setprecision(), ie doing "cout << setprecision(4)" will tell cout to only display (at most) four digits, rounding where necessary.

Code:
```
cout << setprecision(6);
cout << 5.12345678 << endl;
cout << setprecision(3);
cout << 5.12345678 << endl;
```

Output:
```
5.12934
5.13
```

Note: cout defaults to setprecision(6), ie display at most six digits.

You can explicitly tell cout to display numbers in scientific notation by using the `scientific` manipulator.
On the other hand, `fixed` is a manipulator to display numbers in decimal-point (ie fixed-point) notation. `fixed` will never display an exponent field.

Code:	**Output:**

`cout << fixed << setprecision(3);`
`cout << 51.9 << endl;`
`cout << scientific;`
`cout << 51.9 << endl;`
	`51.9`
	`5.190e+001`

<u>Side Note</u>: cout defaults to a "hybrid" mode halfway in between fixed and scientific. From the C++ documentation:
"On the default floating-point notation, the *precision field* specifies the maximum number of meaningful digits to display both before and after the decimal point, while in both the fixed and scientific notations, the *precision field* specifies exactly how many digits to display *after* the decimal point, even if they are trailing decimal zeros."

Code:	**Output:**

double a = 3.1415926534;
double b = 2006.0;
double c = 1.0e-10;
cout << setprecision(5);
 	
out << "default ('hybrid'):\n";
cout << a << '\n' << b << '\n' << c << '\n';
cout << '\n';
 	
cout << "fixed:\n" << fixed;
cout << a << '\n' << b << '\n' << c << '\n';
cout << '\n';
 	
cout << "scientific:\n" << scientific;
cout << a << '\n' << b << '\n' << c << '\n';
	default ('hybrid'):
	3.1416
	2006
	1e-010
	
	fixed:
	3.14159
	2006.00000
	0.00000
	
	scientific:
	3.14159e+000
	2.00600e+003
	1.00000e-010

Object Oriented Programming (OOP)

An extremely popular programming paradigm, object oriented programming has become one of the main programming paradigms since the mid 1990's. Whether you're a hobbyist or a full-time software engineer, you'll almost certainly work with OOP during your work.

Motivation: A Student Example

Suppose we were hired to write a program for the UCLA dining hall that kept track of student meal plan balances. To represent a single student, one could do the following:

```
    string std0_name = "Louis Reasoner";
    double std0_balance = 750.00;
```

To create a new student, we'd have to define a new set of variables:

```
    string std1_name = "Alyssa P. Hacker";
    double std1_balance = 1200.00;
```

This is a bit cumbersome, as we need to explicitly keep track of sets of variables. A better approach would be to represent a student as a **single entity** which internally keeps track of details such as: name, student ID, and balance:

```
    Student louis = Student("Louis Reasoner", 750.00);
    Student alyssa = Student("Alyssa P. Hacker", 1200.00);
    cout << louis.get_name() << endl; // displays: Louis Reasoner
    cout << alyssa.get_balance(); // displays: 1200.00
```

Class Interfaces

In OOP, a *class interface* is essentially an outline (or sketch) of a particular class. It typically has either no code (or very little code), and exists simply to sketch out the class skeleton. Typically, one will flesh out the class skeleton in a separate .cpp file. Here's a sample class interface for the Student class:

```
class Student {
private:
    string name;
    double balance;
public:
    Student(string name, double balance); // constructor
    string get_name(); // method that returns the name
    double get_balance(); // method that returns the balance
    void deposit(double amt); // method that adds money to student's
balance
    void withdraw(double amt); // method that removes money from balance
};
```

Constructors

A constructor is effectively a function that creates and initializes an object. For instance, to create a `Student` object, we pass in the name and balance so that the object knows its identity:

```
    Student louis("Louis", 750.00); // create Student object name and
balance
```

```
Student louis2 = Student("Louis", 750.00); // equivalent way
```
When you create an object, we call the new object an **instance** of the class. In the above, both `louis` and `alyssa` are **instances** of the `Student` class.

Member Variables and Functions

Classes contain both data (ex: name, balance) and behavior (ex: deposit, withdraw). In OOP terminology, we call the data **"member variables"**, and behavior **"member functions"** (or methods).

Function Signatures

When you see a function declaration such as:
```
string get_name(); // method that returns the name
```
This means that: the function get_name takes no input arguments, and returns a string:
```
Student velvet("Velvet", 101.00);
string s = velvet.get_name();
cout << s; // displays: Velvet. Alt: cout << velvet.get_name();
```
As another example, let's look at the Student::withdraw() member function:
```
void withdraw(double amt); // method that removes money from balance
```
Here, we see that withdraw takes a single input argument amt. Also, the "void" as the return type means that this function does **not** return anything:
```
Student morty("Morty", 9999.99);
cout << morty.get_balance() << endl; // displays: 9999.99
morty.withdraw(10);
cout << morty.get_balance(); // displays: 9989.99
cout << morty.withdraw(30); // CompileError: Can't cout nothing!
morty.withdraw(5) + 42; // CompileError: Can't add to nothing!
morty.withdraw(); // CompileError: Missing argument to withdraw!
```

Access Modifiers: `public` vs `private`

One can control what is allowed to access member variables/functions by declaring them as `public` or `private`.

Something declared **`public`** can be accessed from outside the class. Something declared **`private`** can only be accessed within the class definition. This will make more sense when we start filling in class definitions with code, but here's an example:
```
Student eric = Student("Eric", 0.85); // just enough for coffee!
eric.deposit(1.00); // Valid: deposit is public member function
cout << eric.get_balance(); // Valid: get_balance() is public member
function
cout << eric.balance; // Invalid: balance is private member variable
```