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

Object Oriented Programming (OOP)

A 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 s1_name = "Louis Reasoner";
double s1_balance = 750.00;
```

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

```
string s2_name = "Alyssa P. Hacker";
double s2_balance = 1200.00;
```

This is 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** that keeps track of their name and balance:

```
Student s1 = Student("Louis Reasoner", 750.00);
Student s2 = Student("Alyssa P. Hacker", 1200.00);
cout << s1.get_name() << endl; // displays: Louis Reasoner
cout << s2.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(const string& name, double balance); // constructor
    string get_name() const; // method that returns the name
    double get_balance() const; // 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 s1("Louis", 750.00); // create Student object name and balance
```

When you create an object, we call the new object an **instance** of the class. In the above, both `s1` and `s2` are **instances** of the `Student` class.

Member Variables and Functions

Classes contain both data (eg name, balance) and behavior (eg 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: the function `get_name` takes no input arguments/parameters, and returns a string object:

```
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. A return type of "`void`" means that the 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; // Compiler Error: balance is a private member variable
```

In PIC10A: Member variables should always be `private`, and member functions should be `public`!

Include Guards

When working with multiple header files, one must be careful to avoid accidentally defining the same variable/function/class multiple times, as doing so is a compiler error. Consider the following example, where I have `main.cpp`, and two header files (`myheader1.h`, `myheader2.h`):

```
// main.cpp
#include <iostream>
#include "myheader1.h"
#include "myheader2.h"
using namespace std;
int main() {
    cout << "Bye." << endl;
    return 0;
}
```

```
// myheader1.h
const double PI = 3.14; // pi
```

```
// myheader2.h
#include "myheader1.h"
const double E = 2.718; // Euler's number
```

The above code will fail to compile, and the compiler will complain that the global variable `PI` is defined multiple times. Recall that the preprocessor expands out the `#include "myheader1.h"` with the contents of the header file. In other words, the preprocessor first transforms `main.cpp` to something like:

```
// main.cpp
#include <iostream>  // I'll leave this unexpanded for clarity
/* BEGIN myheader1.h */
const double PI = 3.14; // pi
/* END myheader1.h */
/* BEGIN myheader2.h */
/* BEGIN myheader1.h */
const double PI = 3.14; // pi
/* END myheader1.h */
const double E = 2.718; // Euler's number
/* END myheader2.h */
using namespace std;
int main() {
    cout << "Bye." << endl;
    return 0;
}
```

When the compiler sees this code, it complains that `PI` is defined multiple times! This is because `myheader2.h` includes `myheader1.h`, so `myheader1.h` ends up being included twice.
To avoid this problem, we can use include guards:

```
// main.cpp
#include <iostream>
#include "myheader1.h"
#include "myheader2.h"
using namespace std;
int main() {
    cout << "Bye." << endl;
    return 0;
}
```

```
// myheader1.h
#ifndef MYHEADER1_H
#define MYHEADER1_H
const double PI = 3.14; // pi
#endif
```

```
// myheader2.h
#ifndef MYHEADER2_H
#define MYHEADER2_H
#include "myheader1.h"
const double E = 2.718; // Euler's number
#endif
```

`#ifndef` stands for "if not defined". In myheader1.h, the preprocessor will first check to see if the preprocessor variable `MYHEADER1_H` is defined.

If `MYHEADER1_H` is already defined, then it will skip whatever is between the `#ifndef` and `#endif`.

If `MYHEADER1_H` is not defined, then it include the code between the `#ifndef` and `#endif`.

Include guards are a nice way to only include a file once. The preprocessor variable `MYHEADER1_H` is like a "tripwire" variable - if it is defined, then we know that myheader1.h has already been included. For illustration, here is the main.cpp expanded out by the preprocessor when we use include guards:

```
// main.cpp
#include <iostream>    // I leave this unexpanded for clarity
/* BEGIN myheader1.h */
const double PI = 3.14; // pi
/* END myheader1.h */
/* BEGIN myheader2.h */
const double E = 2.718; // Euler's number
/* END myheader2.h */
using namespace std;
int main() {
    cout << "Bye." << endl;
    return 0;
}
```

Note: The preprocessor variables MYHEADER1_H, MYHEADER2_H don't have to be the name of the file(s). For instance, I could have also done: `#ifndef MEOW`, `#define MEOW`, etc.

if/else statements

If statements let us run code only if certain conditions are met. The common three forms are:

```
if (x == 3) {
  cout << "x is 3!" << endl;
}
```

```
if (x < 3) {
  cout << "A" << endl;
} else {
  cout << "x is > 3.\n";
}
```

```
if (x < 3) {
  cout << "A" << endl;
} else if (x > 3) {
  cout << "B" << endl;
} else {
  cout << "x is 3." << endl;
}
```

In the first, the body of the if statement is run only if x is 3. If x is not 3, then nothing happens.
In the second, if x is less than 3, then we output A. Otherwise, we run the "else" body, which outputs "x is > 3.".
The third example is an extension of the second example, where we have added another condition.
Note that curly braces are optional if the body of the if/else is only a single line, eg the following are equivalent:

```
if (x == 3)
    cout << "x is 3!" << endl;
```

```
if (x == 3) {
    cout << "x is 3!" << endl;
}
```

However, if the body is longer than 1 line, then you must have the curly braces:

```
int foo = 10;
cout << "Hi.";
if (foo == 2) {
    cout << "A";
    cout << "B";
}
Outputs: Hi.
```

```
int foo = 10;
cout << "Hi.";
if (foo == 2)
    cout << "A";
    cout << "B";
Outputs: Hi.B
```

Finally, be careful to not confuse assignment with equality comparisons:

`int x = 2;`
`if (x == 3) {`
`  cout << "Hi! ";`
`}`
`cout << "x is: " << x;`
Outputs: x is: 2
	`int x = 2;`
	`if (x = 3) {`
	`  cout << "Hi! ";`
	`}`
	`cout << "x is: " << x;`
	Outputs: Hi! x isn't 3: 3

In the right example, the `(x = 3)` first assigns x to be 3, then returns x as a reference. Thus, "`if (x = 3)`" undergoes the following transformations:

(1) Assign x = 3.
(2) if stmt becomes: if (3)
(3) Since 3 is a true value, we enter the body of the if-stmt.

Finally, the general form of the if statement is:

```
if (CONDITION1) {
    BODY1
} else if (CONDITION2) {
    BODY2
...
} else {
    BODYELSE
}
```

Where each BODY can be multiple lines, you can have as many "else if" as you like, and the "else" is optional.