Lesson 14: Classes

Ricardo Salazar, Pic 10A
(5.1) Classes

- Recall: a class is a data type 'on steroids'.
  - it comes with special 'helper' functions.
- An instance of a class is called an **object**.
- **string** is an example of a class
  ```java
  string myFriend = "Ramon";
  ```
  - The **insert** function is an example of a 'helper' function.
    ```java
    myFriend.insert(0,"Don ");
    ```
- Other examples include:
  - The **Time** and **Employee** classes;
  - The **Graphics** classes: **Point**, **Circle**, **Line**, **and** **Message**.
Why classes?

The idea is to model real world objects in a more intuitive way.

- Say that we want to create a phonebook app, to keep track of all of our friends.
  - Instead of having to keep track of different individual data types like
    - name (string n)
    - phone number (int p)
    - email address (string e)
    - twitter handle (string t)
  - We can toss them into one bigger better container, say friend.
  - We can also create helper functions to deal with the data.
(5.2) Interfaces

• A class declaration:

```cpp
class ClassName {
    public:
        constructor declarations
        member functions declarations
    private:
        data fields
};
```

- Public fields are accessible outside of the class.
  - E.g: the `get_y()` function in the `Point` class.
- Private fields are only known locally (encapsulated).
  - These are our secret rooms and helpers in our big ol' castle.

Did you notice the semi-colon?

These functions create the object

These are the 'helpers'. They can handle the object without breaking it.

Internal values and functions.
## Interfaces (examples: Employee)

<table>
<thead>
<tr>
<th>Class</th>
<th>Employee</th>
</tr>
</thead>
</table>

**Public:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee(string n, double s)</td>
<td>Constructs an employee with name n and salary s.</td>
</tr>
<tr>
<td>string get_name()</td>
<td>Returns the name of this employee.</td>
</tr>
<tr>
<td>double get_salary()</td>
<td>Returns the salary of this employee.</td>
</tr>
<tr>
<td>void set_salary(string s)</td>
<td>Sets the salary of this employee to s.</td>
</tr>
</tbody>
</table>

**Private:**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string name</td>
<td>The name of this employee.</td>
</tr>
<tr>
<td>double salary</td>
<td>The salary of this employee.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Accessor</th>
<th>Mutator</th>
</tr>
</thead>
</table>

To use this class we need to `#include "ccc_empl.h"`
Employee (cont)

The file ccc_empl.h contains (among other things) the following code:

class Employee {

public:
    Employee();
    Employee(string employee_name, double initial_salary);

    void set_salary(double new_salary);
    double get_salary() const;
    string get_name() const;

private:
    string name;
    double salary;
};

**Constructor functions**

**Accessor functions**

**Mutator function**

Data fields (hidden from the outside world). name and salary cannot be accessed directly. Only members of the class can handle them.
(5.3) Encapsulation

Placing variables or functions in the private section of a class is called **encapsulation**.

- It allows to free their names.
  - Variables in other parts of the program can also be called `name` or `salary` and they will not alter the data inside the class.
    ```
    int main(){
    Employee e = Employee("Ramon",2018);
    double salary = 8102; // <-- he still earns 2018
    }
    ```
  - To obtain the salary we need an accessor
    ```
    double aSalary = e.get_salary();
    ```
  - To change it, we need a mutator
    ```
    e.set_salary(8102); // now Ramon is happy!
    ```

- Most importantly…
  - it prevents outsiders --us included-- from misusing the class.
The **const** modifier

Did you notice the **const** after the declaration of the accessors?

class Employee {

public:
    Employee();
    Employee(string employee_name, double initial_salary);

    void set_salary(double new_salary);
    double get_salary() **const**;
    string get_name() **const**;

private:
    string name;
    double salary;
};

The reserved word **const** is optional…
but it is good practice to use it.

Think of it as a pledge.
These functions promise not to alter the object in any way.

It is another form of protection.
Always add the word **const** to your accessors.
The **Product** class

Say we are hired by 'Consumer Reports' to rate different products.

- **We want to implement a Product class with the fields**
  - name (**string**), the name of the product.
  - price (**double**), the price of the product.
  - score (**int**), how good it is, say on a 0 to 10 scale.

- **What member functions should we implement?**
  - create new product, (**constructor**)
  - read product's attributes, (**mutator**)
  - display product's attributes, (**accessor**)
  - compare two products, (**accessor**).
Product class (cont)

The declaration should look like this:

```cpp
class Product {
public:
    Product();
    void read();
    bool is_better_than(Product b) const;
    void print() const;
private:
    string name;
    double price;
    int score;
}; // <-- don't forget the semicolon
```

- Like a function declaration, it should be placed above `main()`.
(5.4) Member functions

- The general form of a member function definition is:
  
  ```
  returnType ClassName::functionName(parameters) const {
      -- STATEMENTS --
  }
  ```

- The `::` is called the *scope resolution operator*; it is there to make sure everyone knows `functionName` belongs to the class `ClassName`.

- `const` is optional, add it if `functionName` is an accessor.

- This should be the first line of our member functions:
  
  ```
  Product::Product() {
  void Product::read() {
  bool Product::is_better_than(Product b) const {
  void Product::print() const {
  ```
The `Product::read()` function

- `Product::read()` requests a name, price and a score, then stores them inside the object's data fields.

```cpp
void Product::read() {
    cout << "Please enter the model name: ";
    getline(cin, name);
    cout << "Please enter the price: ";
    cin >> price;
    cout << "Please enter the score: ";
    cin >> score;
    string remainder;
    getline(cin, remainder);
}
```

- We did not declare the variables `name`, `price` nor `score`!!!
- They are available to `Product::read()`. Why???

- Your turn: implement `Product::print()`.
Other member functions

● `product::is_better_than(Product b)` compares two products based on the ratio `score/price`.

● A typical call would be:

```cpp
if ( a.is_better_than(b) ) cout << "buy a!";
```

● Here is the implementation:

```cpp
bool Product::is_better_than(Product b) const {
    if ( b.price == 0 ) return false;
    if ( price == 0 ) return true;
    return score/price > b.score/b.price;
}
```

● Notice:
  ○ Member functions should be 'certified entities'. (avoid division by 0)
  ○ `is_better_than` is a member function of the `Product` class and it can access the fields `score` and `price` of `Product` b.
  ○ If the code is outside of the class, say in `main()`, it would not work.
(5.5) Default constructors

- There is one member function left to define:
  - the constructor `Product`.
- In the class declaration, `Product()` was the only function with no return value… not even `void`.
- This only happens for constructors!
  - The only thing this constructor does is "create" the object.
  - Here we assign default values to `name`, `price` and `score`.

```cpp
Product::Product() {
    price = 1;
    score = 0;
    name = "No item";
}
```

- To create an object use the statement `Product newProduct;`
The Product class (summary)

- To create a product with default values call the constructor

```java
Product newProduct = Product();
/* Product newProduct; // also works */
```

- To set the internal values use the `read()` function

```java
newProduct.read();
```

- To display the internal values use the `print()` function.

```java
newProduct.print();
```

- To compare two products call `is_better_than`

```java
newProduct.is_better_than(Product()); // returns false
```
Non-default constructors

- What if we want to create a product and set the values as soon as it is created?

- This can be achieved with another constructor:

  ```cpp
  Product::Product(string n, double p, int s) {
    name = n;
    price = p;
    score = s;
  }
  ```

- Examples of calls to this constructor:

  ```cpp
  Product("Chavo del 8 (DVD)", 14.99, 10);
  Product("Learn C++ with Chapulin Colorado", 9.99, 8);
  Product("El Chavo animado (comic)", 5.59, 6);
  ```
Multiple constructors

- Notice that we have two functions with the same name:
  - `Product()`, and
  - `Product(string, double, int)`
- This does not produce a compile error.
- When a product is created the function that matches the parameters is called.
  - Turns out the compiler is smart!
  
  E.g:
  ```java
  Product product1 = Product();
  Product product2 = Product("Prod. 2", 10.50, 4);
  ```
- It would also work if we had a third constructor as long as the parameter types are different.
The **Product class** (example)

- Let us write some code that:
  - repeatedly reads products, and
  - reports the best one.

```cpp
int main()
{
  string response, dummy;
  Product next, best;

  do{
    next.read();
    if ( next.is_better_than(best) )
      best = next;
    cout << "More products (y/n)? ";
    cin >> response; getline(cin, dummy);
  }while(response == "y");

  cout << "The best product is: ";
  best.print();
  return 0;
}
```

The general outline:
```
#include <library1>
#include <library2>
...
using namespace std;

global constants

class declaration {
  public:
    ...
  private:
    ...
};

class member functions

function declarations

program functions

main routine
```
The **Rectangle** class (exercise)

- The graphics library `ccc_win.h` has classes for `Point`, `Circle`, `Line` and `Message`.
- Let’s build a **Rectangle** class.
- A rectangle can be specified by the *top left corner* point, the *height*, and the *width*.

- We should have member functions which *construct*, *draw*, and *move* the rectangle.
- Help me write this class…