Lecture 4: Classes

PIC 10B
Todd Wittman

Pop Quiz, Hotshot

- Ex What does this print? What's the runtime?
  ```cpp
  for (int i=1; i <= N; i++) {
    for (int j=1; j <= i; j++)
      cout << j << " ";
    cout << "\n";
  }
  ```
  Using formula we saw last class, takes $O(N^2)$ time.

- Ex What's the runtime of this code?
  ```cpp
  for (int i=1; i<=N && i<=100; i++)
    cout << i << " ";
  ```
  If $N<100$, runs for $N$ (linear) iterations. If $N>100$, runs for 100 (constant) iterations. Since Big O is for large values of $N$, the runtime is $O(1)$. 
Sec 5.1: Classes

- Recall that a class is a suped-up data type equipped with special member functions.
- We call an instance of a class an object.
- The string class is an example of a class.
  ```
  string my_string = "vader";
  ```
- The insert function is an example of a member function.
  ```
  my_string.insert(3, "yoda"); // Use period after object name.
  ```
- This week we're going to gradually build up two example classes.
  - The Product Class (from Ch. 6 of your book)
  - The IntMatrix Class (for HW2)

Sec 5.2: Declaring Classes

- Declaring a class is a lot like declaring a function.
  ```
  class Class_name {
      public:
          constructor declarations
          member function declarations
      private:
          data fields
  }
  ```
- Public fields are accessible outside the class.
- Private fields are only accessible to the class itself, much like local variables.
- Member function types: constructor, accessor, mutators.
- Add the const modifier to accessors.

The Product Class

- Suppose we’re cataloging and comparing different products, e.g. Star Wars toys.
- It would be helpful to store the various items for sale in some standard class.
- What data should the product store?
  - Product name (string)
  - Product price (double)
  - Product score on 0-10 scale (int)
- What functions would we like to have?
  - Create new product (constructor).
  - Read in a new product’s details (mutator).
  - Compare two products (accessor).
  - Print out a product’s details (accessor).
- Which of our functions should get the \texttt{const} modifier?

The Product Class Declaration

```cpp
class Product {
    public:
        Product();
        void read();
        bool is_better_than(Product b) const;
        void print() const;
    private:
        string name;
        double price;
        int score;
};
```

- Place in your program above main() or in own file.
Sec 5.4: Member Functions

- We've declared the class, but we haven't defined any of its functions yet.
- The general form of a member function definition is:
  ```cpp
  return_type ClassName::function_name (parameters) const {
    ** STATEMENTS **
  }
  ```
- We add the `const` modifier at the end if it's an accessor. Should match the `consts` used in the declaration.
- By adding `ClassName::` before the function name, we make sure that the program knows this is a member function for that class.

Product's print function

- Every member function has the private variables ready to go (name, price, score).
- Print the values of a product.
  ```cpp
  void Product::print( ) const {
    cout << "Name: " << name << "\n";
    cout << "Price: " << price << "\n";
    cout << "Score: " << score << "\n\n";
    return;
  }
  ```
- We call this function from main (or wherever) with something like:
  ```cpp
  Product my_product;
  my_product.print( );
  ```
Product's read function

- Read in the name, price, and score of a product.
  ```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);
  }
  ```
- Note we do not declare the variables name, price, and score. They're already declared by the class declaration.
- What does the variable remainder do?
- Why do we use getline once and cin twice?

Product's is_better_than function

- To find the “bang for the buck” ratio, compute score/price.
- Return true if the current product has better ratio than a passed product b. Be sure to avoid division by zero!
  ```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;
  }
  ```
- We can access the private variable of another product.
  Explict Object's Price: b.price   Implict Object's Price: price
- To call this function for Products a & b, use something like:
  ```cpp
  if ( a.is_better_than(b) ) cout<<"Buy a!";
  ```
- Note how we access the different variables. To get the implicit price, just say “price”. To get price of b, use “b.price”.
- Recall that non-member functions can't get to price directly.
Sec 5.5: The Default Constructor

- Last class we created a default constructor for the Product class.
  
  ```
  Product::Product () {
    price = 1;
    score = 0;
    name = "No item";
  }
  ```

- Recall the constructor is just the name of the class and has no return value.
- This function is called to set up a uninitialized Product with dummy values.
  ```
  Product my_product;
  ```

- Then we have to use read( ) to set the values.
- A better constructor would read in values when we create it.

Sec 5.6: Constructor With Parameters

- In C++ we can overload a function, so that we can have multiple versions of the same function.
- The compiler will look up the function with corresponding parameters.
- So in addition to our default constructor Product( ), we could have the following constructor.
  ```
  Product :: Product (string new_name, double new_price, int new_score) {
    name = new_name;
    price = new_price;
    score = new_score;
  }
  ```

- Note the parameters are NOT named name, price, score.
- We can then create a product with the call:
  ```
  Product my_product ("Millenium Falcon", 29.99, 10);
  ```
Constructor With Parameters

- There is an alternative syntax for the constructor with parameters (p. 248).
  ```cpp
  ClassName :: ClassName (parameters)
  : field1(expression1), field2(expression2), ... {
    **STATEMENTS**
  }
  ```
- It sets the value of the private variable field1 to whatever is inside the parentheses, usually a parameter.
- So we could rewrite our constructor as:
  ```cpp
  Product :: Product (string new_name, double new_price,
   int new_score) : name(new_name), price(new_price),
   score(new_score) {
  }
  ```
- You don't have to use this syntax. Personally I don't care for it.
- But it's necessary if you don't have a default constructor.

Sec 15.3: The Copy Constructor

- A common programming style is to copy one object into another through the parameters:
  ```cpp
  ClassName newObject (oldObject);
  ```
- Even if you don't like this constructor, it's useful because the compiler automatically looks for this constructor for assignment.
  ```cpp
  ClassName newObject = oldObject;
  ```
- It's also implicitly called when you need a temporary object, like in a calculation.
  ```cpp
  cout << object1 + object2;
  ```
- The copy constructor just needs to copy the data from the passed object.
  ```cpp
  ClassName::ClassName (const ClassName& right) {
    **COPYING STATEMENTS**
  }
  ```
- Note we pass the parameter by reference (&). Why?
- We also add the word const to the parameter. Why?
The Copy Constructor

- The copy constructor for Product is:
  
  ```cpp
  Product::Product (const Product& right) {
      name = right.name;
      price = right.price;
      score = right.score;
  }
  ```

- Now both of the following statements work.
  ```cpp
  Product new_product (old_product);
  Product product1 = product2;
  ```

- The copy constructor automatically defines the `=` operation to copy whatever's on the right side.
- This automatic `=` doesn't work with dynamic arrays.

Sec 5.9: Separate Compilation

- Put the main routine and its non-member functions in the main source cpp file.
- Put Class declarations in Class.h (or whatever class name is).
- Put Class member function definitions in Class.cpp (same name).

```
main.cpp  (e.g. bw2.cpp)
#include "Class.h"

void fun() {
    ...
}

int main() {
    ...
}
```

```
Class.h
#ifndef CLASS_H
#define CLASS_H

class Class {
    ...
};
#endif
```

```
Class.cpp
#include "Class.h"
Class::Class () {
    ...
}

void Class::fun () {
    ...
}
```

*Note both main.cpp and Class.cpp #include the Class header file.*
Using Header Files

- The source file includes the header file at the top:
  
  ```
  #include "filename.h"
  ```

- The basic header file looks like:

  ```
  ifndef FILENAME_H
  define FILENAME_H
  include <libraries>
      **YOUR CLASSES HERE**
  endif
  ```

- The `ifndef` statements prevent multiple inclusion.
- So even if this header file is included by more than one program, it will still only be compiled once.

Ch. 7: Review of Pointers

- We create a pointer to block of given type with the declaration:

  ```
  type* pointer_name;
  ```

- A pointer stores a memory address, accessed with &.

  ```
  double x = 3.14;        double* p = &x;
  ```

- Dereferencing accesses the pointee’s value directly with *.

  ```
  *p = -42.3;
  ```

- Setting a pointer equal to array name, it points to 1st element.

  ```
  string* s = heroes;  //s points to 1st string Luke
  ```

- Adding to the pointer moves the pointer through the array.

  ```
  s = s+2;    //Now s points to 3rd element Han
  ```
Memory Management

- When our program runs, all the variables created are stored in the memory heap.

<table>
<thead>
<tr>
<th>b=4</th>
<th>x = 23</th>
<th>x2='A'</th>
<th>card1=&quot;2 of Hearts&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>y=3.1</td>
<td>k=&quot;Han&quot;</td>
<td>str=&quot;Hi Chewie!&quot;</td>
<td></td>
</tr>
</tbody>
</table>

- Pointers give us a way to control the heap, so that the heap doesn't get full.

- The new function allocates a memory block.
  ```
  int* p = new int;
  *p = 42;
  ```

- The delete function frees up that block so it can store other stuff (recycling). Does not delete the pointer.
  ```
  delete p; //Frees up the block with 42.
  ```

- We should now make p point to nothing: `p = NULL;`

Dynamic Arrays

- A *dynamic array* is an array whose size does not need to be a constant.

- We create dynamic arrays through pointers.
  ```
  int N;
  cout << "Enter size of list: ";
  cin >> N;
  int *list;
  list = new int[N]; //Allocates N int blocks.
  ```

- We can access elements just like an array.
  ```
  list[0] = 42;
  ```

- When we're done, we can delete the entire array using the delete with blank brackets after it.
  ```
  delete[] list;
  ```