Comp 11 Lectures

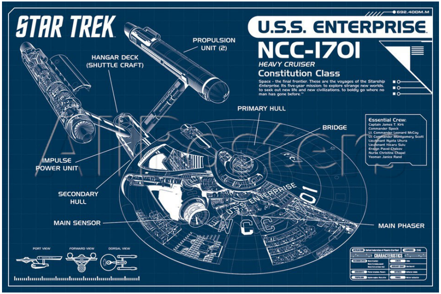
Mike Shah

Tufts University

July 12, 2017

Object Oriented Programming 1

- A blue print provides a description of something we want to build.
- We can build many different instances of whatever the blue print is. They have the same functionality, but are still unique entities in the world.



Lecture

Alan Kay

Figure 1: Alan Kay is an American computer scientist known for doing lots of work on object-oriented programming. He worked at Xerox PARC on Graphical User Interfaces we use today, as well as other companies like Atari.

Structs

- Today we are going to learn about object-oriented programming.
- I have a secret though, we have already been using objects!
- In C++, when we create structs, we are creating our own custom data type (or otherwise known as an object).
- The struct serves as a blueprint, for which we can create several objects for.

OOP Programming

- Why do we care about Object-Oriented Programming (OOP)?
- One main reason, is reusability! We can reuse code.
- OOP techniques allow us to build modular code, and think of our code as puzzle pieces to solving a bigger problem (just like FUNctions).
- OOP is a convenient way to reason about building software that reflects the real world.
- The other is encapsulation, we are able to expose only parts of code that we need.

class

- Along with the **struct** we have another C++ keyword–**class**
- A **class** is almost the exact same as a struct.
- An important difference is however, that we can hide(i.e. encapsulate) data within it that we do not want to expose to the user.
- A **struct** on the other hand, by default makes all information available. We can explicitly do this with the C++ keyword **public:**

The trouble with struct

```
#include <iostream>
struct myIntVector{
public: // By default this is implicit in a struct
  int* storage;
    int size;
    myIntVector(int _size){ // A constructor
      storage = new int[_size];
        for(int i =0; i < _size; i++){
          storage[i] = i;
        }
        size = _size;   // save size;
    }
    ~myIntVector(){delete[] storage;}
  // print method
  void print(){
      for(int i =0; i < size; i++){
          std::cout << storage[i] << "\n";
        }
    }
};

int main(){
  myIntVector a(20);
    a.print();

  return 0;
}
```

Listing 1: A struct with all public members

What if the user makes some changes?

```
#include <iostream>
struct myIntVector{
public: // By default this is implicit in a struct
  int* storage;
    int size;
    myIntVector(int _size){ // A constructor
      storage = new int[_size];
        for(int i =0; i < _size; i++){
          storage[i] = i;
        }
        size = _size;   // save size;
    }
    ~myIntVector(){delete[] storage;}
  // print method
  void print(){
      for(int i =0; i < size; i++){
          std::cout << storage[i] << "\n";
        }
    }
};

int main(){
  myIntVector a(20);
    a.size = 5000;    // hehe -- I am a malicious user!!!!
    a.print();

  return 0;
}
```

Listing 2: A user being a little malicious

What if the user makes some changes?

- Well, in the previous example, I would argue the user is being a little malicious.
- But we want to protect users from doing this!
- So we can instead use **private:** to set everything that follows the word private, to be hidden.

Protecting the user by good design

```
#include <iostream>
struct myIntVector{
private:
  int size; // Now size is hidden, cannot be accessed anywhere!
public:
  int* storage;
    myIntVector(int _size){
      storage = new int[_size];
        for(int i =0; i < _size; i++){
          storage[i] = i;
        }
        size = _size;
    }
    ~myIntVector(){delete[] storage;}
  // print method
  void print(){
      for(int i =0; i < size; i++){
          std::cout << storage[i] << "\n";
        }
    }
};

int main(){
  myIntVector a(20);
    // a.size = 5000;   // This line would not compile!
    a.print();

  return 0;
}
```

Listing 3: A struct with private members

class vs struct

- Functionally, the exact same
- The only difference is that a class makes all members private by default.
- A struct on the other hand makes all members(i.e. member variables, member functions, constructors, destructors, etc.) public.

Constructor and Destructor

- We briefly learned about Constructors and Destructors, and in fact those have to be publicly accessible.
- Lets take a look at our previous code, implemented as a class.

class implementation

```cpp
#include <iostream>
class myIntVector{
  // Hide items we do not want users to have access too
    int size;
    int* storage;
public:
    // Constructor needs to be public!
    myIntVector(int _size){
      storage = new int[size];
        for(int i =0; i < size; i++){
          storage[i] = i;
        }
        size = _size;
    }
    // Define your destructor here
    ~myIntVector() {delete[] storage;}
  // print method
  void print(){
      for(int i =0; i < size; i++){
          std::cout << storage[i] << "\n";
        }
    }
};

int main(){
  myIntVector a(20);
    // a.size = 5000;   // This line would not compile!
    a.print();

  return 0;
}
```

Listing 4: Data encapsulation with a class

Introducing the Third Amigo!

Well, actually just one more!

So far we have learned two:

1. Constructor – called when we create an instance of an object.
2. Destructor – called when an instance of an object leaves scope or is deleted.

Copy Constructor

So far we have learned two:

1. Constructor – called when we create an instance of an object.
2. Destructor – called when an instance of an object leaves scope or is deleted.
3. **Copy Constructor** – called when we use assignment to make a copy of an instance of an object.

The three amigos[1] have been hiding!

- C++ has been creating default constructor, destructor, and copy constructors for us this whole time!
- Our structs have usually consisted of simple primitive types, such as int, char, float, etc. So this is easy for the compiler to do.
- But now that we are creating our own objects, and have learned about pointers, we will need to create our own.

[1] Three Amigos the movie has a 6.4 out of 10 rating on IMDB

Copy Constructor - Shallow Copy

- Lets manually create the copy constructor for an object that the compiler would create for us for free.
- The types of copy constructor the compiler can make for us, only consist of doing a shallow copy, that is, only with primitive types.

Copy Constructor - Shallow Copy Example

```cpp
#include <iostream>
class Student{
public:
  int age;
  Student(){ age = 0; } // Constructor
  ~Student() { }  // Destructor
    // Copy Constructor
    Student(const Student &otherStudent) {
      // Change this students age to the other one
        // which we pass by reference, but also const so the
        // 'otherStudent' does not get modified accidently
      age = otherStudent.age;
    }
};

int main(){
  Student s1; s1.age = 10;
    // Because we have a copy constructor we can do this
    Student s2 = s1;
    std::cout << "Student2 is: " << s2.age << "\n";
  return 0;
}
```

Listing 5: Shallow Copy example

Copy Constructor - Deep Copy

- Now the case where we need to make our own copy constructor, is when we deal with memory allocation.
- Why might that be the case?

Copy Constructor - Deep Copy Motivation

- Now the case where we need to make our own copy constructor, is when we deal with memory allocation.
- This is because when we allocate memory, we are pointing to addresses. In order for a pointer to point to a unique location, we need to allocate new memory.
- Let's see an example.

Copy Constructor - Deep Copy Wrong Example

```
#include <iostream>
class dynamicArray{
public:
  int* numbers;
  dynamicArray(){
      numbers = new int[10];
        for(int i =0; i < 10; ++i){ numbers[i] = i; }
    }
  ~dynamicArray() { delete[] numbers; }
    // Copy Constructor
    dynamicArray(const dynamicArray &otherArray) {
      numbers = otherArray.numbers;
    }
};

int main(){
  dynamicArray d1,d2; // Create both dynamicArray's
    // Try to perform a copy
    d2 = d1;
  d1.numbers[0] = 127;
    d2.numbers[0] = 555;
    // Ooops, we did not do a deep copy!
    std::cout << "d1.numbers[0] " << d1.numbers[0] << "\n";
    std::cout << "d2.numbers[0] " << d2.numbers[0] << "\n";

  return 0;
}
```

Listing 6: Deep Copy wrong example

Copy Constructor - Deep Copy Correct Example

```cpp
#include <iostream>
class dynamicArray{
public:
    int* numbers;
    dynamicArray(){
        numbers = new int[10];
        for(int i =0; i < 10; ++i){ numbers[i] = i; }
    }
    ~dynamicArray() { delete[] numbers; }
    // Copy Constructor
    dynamicArray(const dynamicArray &otherArray) {
        // Allocate our memory
        // Our other constructor is not called--only this one!
        numbers = new int[10];
        // Copy each item one at a time
        for(int i =0; i < 10; i++){
            numbers[i] = otherArray.numbers[i];
        }
    }
};
int main(){
    dynamicArray d1;
    dynamicArray d2=d1; // Perform a deep copy
    d1.numbers[0] = 127;
    d2.numbers[0] = 555;
    // Different values means a successful deep copy
    std::cout << "d1.numbers[0] " << d1.numbers[0] << "\n";
    std::cout << "d2.numbers[0] " << d2.numbers[0] << "\n";
    return 0;
}
```

Listing 7: Deep Copy correct example

Review of today

- class and struct have different access levels in order to encapsulate data. This often helps avoid the user from breaking data structures or using them incorrectly.
- Constructors and Destructor's matter, and we are going to use them from now on when the compilers default is not good enough.
- Copy Constructors have two types of copy depending on the data awe use.

Next time!

- Operator overloading, so we can do a deep copy by just using the equal sign.
- Breaking our programs into different files (separating the interface from the implementation)
- function overloading within classes

In-Class Activity

Midterm review

Activity Discussion

Review of what we learned

- (At least) Two students
- Tell me each 1 thing you learned or found interesting in lecture.

5-10 minute break

To the lab!

Lab: `http://www.mshah.io/comp/11/labs/lab10/lab.pdf` [2]

[2] You should have gotten an e-mail and hopefully setup an account at `https://www.eecs.tufts.edu/~accounts` prior to today. If not–no worries, we'll take care of it during lab!