Overview

• Some minor C++ points
• ADT Review
• Object-oriented Programming
• C++
  • Classes
  • Constructors
  • Destructors
  • More minor Points (if time)
Key Minor Points
const
Motivation

• A lot of bugs are rooted in unexpected state changes

• Something unexpectedly changes a variable’s value

• A “read-only” operation wasn’t read-only

• We would like a way to guarantee that state cannot change
Example

What is pointed to is constant | The pointer itself is constant

```c
void foo(const char* const s) {
    s[0] = 'a'; // disallowed
    s = NULL; // disallowed
}
```
References
Motivation

- Pointers allow us to indirectly refer to data, which is very powerful
- ...but it’s also very error-prone
- We want something in between
References

• These “reference” some other data directly

• References are indirect, but they behave as if they were direct

• Unlike pointers, references are not a distinct kind of data that lives in memory (more restricted)

• Trying to get the address of a reference gets the address of what it references
void swapPointers(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

void swapRef(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
struct point {
    int x;
    int y;
};

void swap(struct point& p) {
    int temp = p.x;
    p.x = p.y;
    p.y = temp;
}

int addedPoint(const struct point& p) {
    return p.x + p.y;
}
ADT Review

- What is the application level?
- What is the logical/abstract level?
- What is the implementation level?
Object-Oriented Programming (OOP)
Observation

• Life is filled with nouns (people, cars, table, projector...)

• These different nouns interact with each other (speak, accelerate, place object on, turn on)

• Often have a concept of internal state (thinking, speed, weight, bulb health)
Observation

- Code can often be modeled in the exact same way
Relationship to OOP

• Nouns - objects
• Creating an object - constructors
• Which interactions are possible - methods
• Performing an interaction - method calls
• Internal state - private state, or encapsulation
**Constructors**

```java
Car makeCar(int color);
```
Constructors
Car makeCar(int color);

Constructing an object
Car c = makeCar(GREEN);

\[\text{c: Car object}\]
Constructors
makeCar(int color);

Methods
void accelerateTo(double mph);
void brake();
double getSpeed();
Constructors
makeCar(int color);

Methods
void accelerateTo(double mph);
void brake();
double getSpeed();

Method call:
c.accelerateTo(15.5);
Constructors
makeCar(int color);

Methods
void accelerateTo(double mph);
void brake();
double getSpeed();

Private State:
double speed = 15.5;

c: Car object
Recall the Rectangle

- Width, height, finding area and perimeter
- What does the constructor look like?
- What sort of methods does it have?
- What kind of internal/private state does it have?
OOP for ADTs

- OOP is great for modeling ADTs. Why?
- Hint: why was C a suboptimal choice?
OOP for ADTs

• OOP is great for modeling ADTs. Why?
  • Methods good for defining interfaces (logical level)
  • Private state/encapsulation good for hiding implementation details
• C doesn’t always make encapsulation easy
OOP in C++
Objects

• In order to make an object, we first need to define a class

• A class behaves like a sort of template for making objects

• With the previous example, we would need a Car class
Classes

- Hold the constructors, methods, and private state of the objects we want to make
- To make an object, we call a class’ constructor to get an instance of a class
- Class instances are synonymous with objects
Car Class Example
Rectangle Class Example
Creating Class Instances

- Can be made either on the stack or the heap
  - On the stack: `Car c(speed);`
  - On the heap: `Car* c = new Car(speed);`
- Both of these examples call the same constructor
- For the heap, can free with: `delete c;`
Constructors in C++
Constructors

- C++ lets us define multiple constructors for a class
- Each can be used to make a class instance
Default Constructor
Default Constructor

- A constructor of special mention is the *nullary*, AKA *default* constructor
- Takes no arguments
- Used when creating an array of objects on the *stack*: `Car c[20];`
- Why?
Default Constructor

• A constructor of special mention is the *nullary*, AKA *default* constructor
  
• Takes no arguments

• Used when creating an array of objects on the stack: `Car c[20];`

• Why? - how would you pass the arguments if it weren’t this way?
Default Constructor

• What happens?

```cpp
class Foo {
    private:
    int x;
};
...
Foo f;
```
Default Constructor

• All OK - the compiler generates a default constructor for you

class Foo {
    private:
    int x;
};
...
Foo f;
Default Constructor

• What happens?

class Foo {
    private:
    int x;
};
...
Foo f; f.x;
Default Constructor

- Undefined - `f.x` can be set, but not accessed

```cpp
class Foo {
    private:
        int x;
};
...
Foo f; f.x;
```
Default Constructor

• What happens?

class Foo {
  public:
    Foo(int y);
  private:
    int x;
};
...
Foo f;
Default Constructor

• Compile-time error: compiler cannot generate a default constructor

```cpp
class Foo {
    public:
        Foo(int y);
    private:
        int x;
};
...
Foo f;
```
Copy Constructor
Copy Constructor

• Used in contexts where we need to copy an object
• Declarations with initialization
• Function calls

    Car(const Car& other);
Copy Constructor

• What happens?

```cpp
class Foo {
  public:
    Foo(int y);
  private:
    int x;
};
...
Foo a(1);
Foo b = a; // copy constructor
```
Copy Constructor

- All ok - the compiler generates a default copy constructor that copies everything

```cpp
class Foo {
public:
    Foo(int y);
private:
    int x;
};
...
Foo a(1);
Foo b = a;  // copy constructor
```
Default Copy Constructor

- Caveat: the copy performed is a shallow copy
Default Copy Constructor

- Caveat: the copy performed is a shallow copy
Default Copy Constructor

- If you want a *deep copy*, you must do it yourself with your own copy constructor.
Default Copy Constructor

- If you want a deep copy, you must do it yourself with your own copy constructor.
Destructors
Destructors

• Optionally, you can define a destructor for a class: `Car::~Car() {}`

• Destructors are called during deallocation
  • When is this for something on the stack?
  • When is this for something on the heap?
Destructors

- Optionally, you can define a destructor for a class:

```
Car::~Car() {}
```

- Destructors are called during deallocation
  - When is this for something on the stack?
    - Return from scope that introduced it
  - When is this for something on the heap?
    - When `delete` is called on it
Destructors

- Useful for objects which dynamically allocate memory internally
- Why?
Destructors

• Useful for objects which dynamically allocate memory internally

• Why? - Allows for memory to be deallocated in synchronization with the object being deallocated
More Minor Points
#include
#include

- No longer correct to put `.h` after the filename for **system-provided** files
- Still expected for your own files

```cpp
// provided by system:
#include <iostream>

// provided by you:
#include "myfile.h"
```
Namespaces
Motivation

- Every name (variable, function, `struct`) in C lives in the some distinct `namespace`
- Means we cannot define two variables with the same name at the same scope
- Global variable `pain`
Namespaces

• A way for the programmer to define custom namespaces

• In this class, you won’t be defining your own, but you will be using existing ones

• Most notable: std for the standard library
Namespaces

• Need to fully specify the name of something

• For example, `endl` is defined in namespace `std`, so to use it we must say:
  • `std::endl`
Namespaces

• Repeatedly typing out the namespace can be annoying, so we can instead say:
  • using std::endl;
  • ...and then later simply say endl everywhere we would have said std::endl
Namespaces

• Sometimes we want everything from a namespace. For that, we can say:
  • `using namespace std;`

• ...to put everything in the std namespace in scope (no more need to prepend `std::` to everything)
Terminal I/O
Terminal I/O

- Terminal input and output are modeled as *streams* that can be read from and written to
  - `cin`: input stream
  - `cout`: output stream
  - `cerr`: error stream (often synonymous with the output stream)
Reading and Writing

- Can be done using `>>` and `<<`, respectively

```cpp
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;
    cout << "Saw: " << x << endl;
    return 0;
}
```