Defining Objects

Matthew Mussomele

Objects

Objects are a form of data abstraction.

They take complicated ideas and processes hide the messy stuff, leaving the user exposed only to a simple exterior.

The human body is a form of data abstraction.

We only need to do simple things like eat and sleep, but really, there are very complicated things going on inside that are hidden from us.

Why Use Objects?

Like I said before, objects make our lives easier by abstracting complicated details away from us.

Functions do this too, but functions hide away only one thing per function.

Objects have the potential to hide many complicated things using functions.

An object is like the structs we talked about before, but objects can contain both variables and functions.

Defining Objects

Every object in C++ must be defined.

A function definition looks like this:

```
class ExampleClass {
public:
    ExampleClass();
    ~ExampleClass();

};
```

This is a very simple example of a class.

It really doesn't do anything.

Object Names

The name of an object definition (called a class) is important for those that are using it to understand what it's meant for.

Lets give our new object a better name first so we can better understand what it's for.

```
class Dog {
public:
    Dog();
    ~Dog();

};
```

*structors

You may have noticed two things inside our function definition.

Those were the constructor and the destructor for the object. These do exactly what they sound like they do.

The constructor is responsible for creating and initializing all the variables belonging to the object.

The destructor is responsible for freeing up memory and other resources used by the object when it's done its job.

```
class Dog {
public:
    Dog();
    ~Dog();

};
```

Constructor

Destructor

Fields

Without fields (variables, functions, etc), objects are pretty useless.

There isn't even anything the constructor or destructor can really do yet.

Lets add a few fields to our Dog class to make it more interesting.

```
class Dog {
public:
    Dog();
    ~Dog();

    std::string name;
    std::string breed;

    void bark();

};
```

Fields

You'll notice a few new things in the class definition.

They are the name and breed variables along with the bark function.

Now our Dog has the potential to do some dog-like things.

```
class Dog {
public:
    Dog();
    ~Dog();

    std::string name;
    std::string breed;

    void bark();

};
```

Field Access

You may also have noticed the “public” keyword on the last slide.

That was an access specifier.

It told C++ that the fields that fall under that specifier are available for use by everyone.

Field Access

C++ has three access levels for object fields.

These are public, private and protected.

Public means that anyone can access a given field.

Private means that no one but the object itself can access the field.

Protected means that the object and its subclasses (more on these later) can access the field.

Constructor Arguments

In order to do things, objects usually need some outside information.

In our case, we'll get that through the constructor, so it will need some parameters.

```
class Dog {
public:
    Dog(std::string const& _name, std::string const& _breed);
    ~Dog();

    std::string name;
    std::string breed;

    void bark();

};
```

Object Headers

The only thing we've done so far is declare the things that our object has.

This seems very much like a header file, doesn't it?

Well, objects tend to be defined in a header file and them implemented in a source file, just like the functions we've been writing so far.

```
#ifndef EXAMPLE_DOG_H_
#define EXAMPLE_DOG_H_

#include <string>

class Dog {
public:
    Dog(std::string const& _name, std::string const& _breed);
    ~Dog();

    std::string name;
    std::string breed;

    void bark();

};

#endif /* EXAMPLE_DOG_H_ */
```

To the right is the full file for Dog.

Implementing Objects

Like stated on the last slide, objects are often implemented in their own cpp source files.

Once the object header defines it, implementing the object is usually simple syntax-wise.

You simply define functions on the global level like usual, but preface them with a scope operator saying what class they belong to.

On the next slide we'll see the skeleton for our Dog class's cpp file.

Implementing Objects

```cpp
#include <string>
#include "Dog.h"

Dog::Dog(std::string const& _name, std::string const& _breed) {
    /* IMPLEMENTATION HERE */
}

Dog::~Dog() {
    /* IMPLEMENTATION HERE */
}

void Dog::bark() {
    /* IMPLEMENTATION HERE */
}
```

Note the scope resolution operators

Referencing Variables

To reference an object's variables, you can do a number of things.

Below are two ways to reference an object's variables inside the functions it owns.

```
printf("%s\n", this->name);
printf("%s\n", name);
```

The first uses the 'this' keyword, which is a special pointer that points at the current object.

The second uses the fact that there is only one variable called 'name' owned by this object. We would have to use the first method if 'name' were overridden in the local scope.

Member Functions

An objects functions can be declared and written just like any other function.

The one caveat is that the name must be preceded by a scope operator labeling which object the function belongs to.

The exception to this is when you actually implement the function in the header file; then it knows where it belongs.

Implementing Dog

Now that we know how to declare objects and reference their variables internally, lets finish up our implementation of Dog.

The first thing we should do is write the constructor.

We're going to use something called an initializer list.

Initialization lists allow us to quickly set variables to new values with simple syntax.

We simple do `<variable>(<new_value>)` for all of our wanted variables and C++ takes care of the rest.

The Constructor

```
Dog::Dog(std::string const& _name, std::string const& _breed) : name(_name), breed(_breed) { }
```

Scope/Name

Parameters

Initializer List

The Destructor

Since our object doesn't use any pointers, we can just leave our destructor empty.

C++ has a nice compiler, so any non-pointer variables are automatically erased when they fall out of scope or are deleted.

But imagine that we were using pointers. You would put some form of the code below in your destructor to free up the space you were using.

```
Class::~Class() {
    delete pointer_0;
    delete pointer_1;
    ...
}
```

The Bark Function

The Bark function can be implemented just like any other, but it's going to use the variables that Dog owns.

```
void Dog::bark() {
    printf("Roof! Roof! I'm %s, a %s!\n", name.c_str(), breed.c_str());
}
```

Note that we didn't need 'name' and 'breed' to be passed in.

Because they are members of the Dog class, it already knew where to find them.

Class variables are an exception to the common scope rules.

The Whole Thing

Alright! We just made our first object in C++!

On the next 2 slides we'll look at the whole header and source files to get an idea of what a simple object looks like.

The Header

```
#ifndef EXAMPLE_DOG_H_
#define EXAMPLE_DOG_H_

#include <string>

class Dog {
public:
    Dog(std::string const& _name, std::string const& _breed);
    ~Dog();

    std::string name;
    std::string breed;

    void bark();

};

#endif /* EXAMPLE_DOG_H_ */
```

The Source

```cpp
#include <string>
#include "Dog.h"

Dog::Dog(std::string const& _name, std::string const& _breed) : name(_name), breed(_breed) { }

Dog::~Dog() { }

void Dog::bark() {
    printf("Roof! Roof! I'm %s, a %s!\n", name.c_str(), breed.c_str());
}
```