CONSTRUCTORS & DESTRUCTORS & HEADER FILES

Class Review

- What is encapsulation?
- In C++, how is a struct different than a class?
- How do you declare member functions in a class? How do you define them?
- What's the syntax for calling those methods?
- What happens when we mark a method as public? A member variable? How about private?

Const Review

What are the different uses of const in this code snippet?

```
const int PI = 3.14159;

class whatever
{
public:
    int getAlpha( const int& bob ) const;

private:
    int stuff;
};
```

Question

- So if a variable is declared private (like bupkis and foo)...
- Then can outside code - like main() - initialize it?
- If not, how does it ever get initialized?

```
class Data
{
public:
    int getAlpha();

private:
    int alpha;
    int beta;
};
```

Constructors

- This kind of initialization happens through a **constructor**
- A constructor is a special class method that is run when the object is first instantiated
- Purpose of a constructor: to initialize the object, setup any dynamic memory, etc.

Constructors

constructors have:

The *same name* as the class (aka Data)

no return type

```
class Data
{
public:
    Data();
    Data( int a, int b );

    int getAlpha();

private:
    int alpha;
    int beta;
};
```

Note that constructors can be overloaded too -
You can have as many constructors as you need, as long as each one has a unique signature

Constructors

- A constructor with no parameters is called a *default constructor*. That lets you do this:

```
Data d;
```

- The other constructor allows you to do this:

```
Data d(4,5);
```

```
class Data
{
public:
    Data();
    Data( int a, int b );

    int getAlpha();

private:
    int alpha;
    int beta;
};
```

Default Constructors

- You aren't required to define *any* constructors (we didn't in the last class!)
- If you don't define any constructors, C++ will define an empty constructor for you - it doesn't actually do anything
- Once you define *any* constructor then C++ stops giving you the empty one for free

```
class BZisaFoo
{
public:
    BZisaFoo( int a );
};
```

```
// this will not compile
BZisaFoo correct;
```

Default Parameters

- Constructors can have default parameters too
- Like any other C++ function, you have to make sure that constructors aren't ambiguous!

```
class Circle
{
public:
    Circle ();
    Circle( float radius = 1.0 );
};
```

which constructor would this use?

```
Circle c;
```

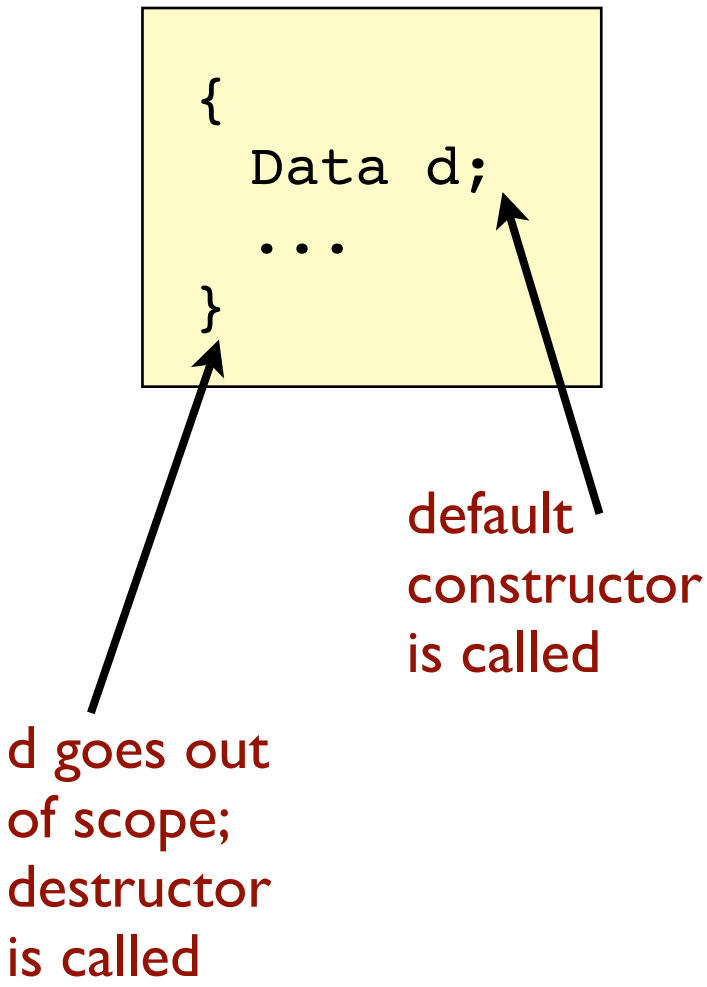
Destructors

- Constructors are called when an object is created...
- A **destructor** is called when the object is deleted.
- A destructor has no return value, and is named after the class, but with a tilde (~) at the beginning.

```
class speaker
{
public:
    speaker();
    ~speaker();
};
```

To Summarize...

- A *constructor* is a special function thatis called when an object is *created*
- A *destructor* is a special function that is called when an object is *destroyed*
  - when the object is manually deleted (via delete)
  - or, when the object goes out of scope



Questions

- Should a constructor / destructor be const? Should they accept const parameters?
- What's wrong with the following snippet of code:

```
class Circle
{
    int Circle();
    int Circle( float radius );
};
```

```
#include <iostream>
using namespace std;

class printer
{
public:
    printer()
    {
        cout << "CREATE"
             << endl;
    }

    ~printer()
    {
        cout << "DESTROY"
             << endl;
    }
};

int main()
{
    printer a[5];
    return 0;
}
```

Quizlet

- Is this valid code?
- What's wrong with it?
- What would the output be if it worked properly?

Copying Classes

```
class Square
{
public:
    Square();
    Square( int, int, int, int );
    int area();
private:
    int x, y, w, h;
};
```

Let's say we have an instance of the Square class:

```
Square ted;
```

And we want to copy all its data into a new Square instance that we're creating. Can we do this?

```
Square bill( ted );
```

Sure we can!

- C++ automatically defines a **copy constructor** for each class.
- That copy constructor copies each element of the class individually, by *value*, into the new class

```
class String
{
public:
    String( char* s );
private:

    // dynamically allocated
    char* str;
};
```

- This is often fine, but not always
- Why would we not want to do this with this String class?

Copy Constructors

- We can also define our own copy constructor. It looks like this:

```
class String
{
public:
    String( char* s );
    String( const String& s );
private:

    // dynamically allocated
    char* str;
};
```

The copy constructor accepts a *const reference*.

In this class the copy constructor would allocate memory before copying.

- This copy constructor replaces the default C++ one.

Using Multiple Files

- Most programs have too much code to fit in a single source file
- So how do we separate code into multiple source files?
- We can do this because there's usually a difference between *declaration* and *definition*

Header Files

- We use header files to contain *declarations* of stuff: classes and functions, mainly
- *Definitions* can go in a separate source file
- Any source file that includes the header file can use anything declared in that header
- Each source file is compiled into a separate binary "object file"; they all get linked together in a final linking stage

Example! Example!

Note that when we're #including header files we've made, instead of "standard" ones, we use quotes in our include statement instead of <> brackets

func.h

```
void func();
```

main.cpp

```
#include "func.h"

int main()
{
    func();
    return 0;
}
```

func.cpp

```
#include <stdio.h>
#include "func.h"

void func()
{
    printf( "hi!\n" );
}
```

Example! Example!

func.h

```
void func();

int variable;
```

What happens if we add a variable in the header?

main.cpp

```
#include "func.h"

int main()
{
    func();
    return 0;
}
```

func.cpp

```
#include <stdio.h>
#include "func.h"

void func()
{
    printf( "hi!\n" );
}
```

Problems

- Each C++ program can only have *one* object of each name.
- If a header declares a variable, and the header gets included multiple times...
- Linker errors!

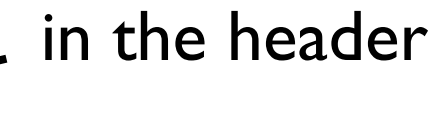
```
/usr/bin/ld: multiple definitions of symbol _variable
/var/tmp//ccSMaERA.o definition of _variable in section (__DATA,__common)
/var/tmp//ccIC3xNa.o definition of _variable in section (__DATA,__common)
collect2: ld returned 1 exit status
```

Solution

- The solution to this particular error is a new keyword: **extern**

```
void func();

extern int variable;
```



- This tells the compiler about the variable (name, type, etc) but that it will *actually* be declared later
- So in a source file somewhere, you need to declare the variable "for real"

```
int variable;
```

structs.h

```
#ifndef _STRUCTS_H_
#define _STRUCTS_H_

struct foo
{
};

#endif
```

- or -

structs.h

```
#pragma once

struct foo
{
};
```

Other Header Stuff

- In the "C is dumb" category...
- Sometimes you'll see stuff like this in a header file to make sure that the header only gets included once
- If a header is included more than once, the compiler will complain that "foo" is defined more than once

Code!

- Let's write a simple dynamic array class (not like one you'd ever write)
  - constructor/destructor
  - private pointer variable
  - member get/set functions
  - member length function
  - copy constructor