

OO:Under the hood

Steven R. Bagley

Previously...

- How objects are represented in memory
- How member variables are accessed
- Method Invocation

Today...

- Polymorphic method invocation
- Memory Management
- Object Creation and Deletion

Polymorphic Methods

- How does the computer know which method to call?
- Inheritance lets me replace a method's implementation on sub-classes
- Which get called even when the pointer type is for the base-class

Polymorphic invocation

- Parameter Passing is the same as for normal methods
- But which bit of code do we execute
- Is it known at compile time?
- Not always possible to trace which concrete type we are acting on

```
public interface IAnInterface
{
    public abstract void PrintSomething();
}

class AClass implements IAnInterface
{
    public void PrintSomething()
    {
        System.out.println("OBJ is easy!");
    }
}

class BClass implements IAnInterface
{
    public void PrintSomething()
    {
        System.out.println("OBJ is hard!");
    }
}
```

Look back at our example of Polymorphism from earlier

```
IAnInterface myInterfacePtr;  
AClass a = new AClass();  
  
myInterfacePtr = new BClass;  
myInterfacePtr = a;  
  
myInterfacePtr.PrintSomething();
```

In this fragment, we can tell the class of myInterfacePtr at compile-time

```
class SomeObj
{
    public void SomeMethod(IAnInterface i)
    {
        i.PrintSomething();
    }
}
```

```
IAnInterface myInterfacePtr;
AClass a = new AClass();
```

```
myInterfacePtr = new BClass;
myInterfacePtr = a;
```

```
SomeObj.SomeMethod(myInterfacePtr);
```

SomeMethod has no way of knowing what object will be implementing IAninterface or what its type will be

What type are you...

- Not possible to generate code that calls method directly
- Need a method of finding out what type the object is, so we can call the correct method
- Use a hidden type field?

Ask the object...

- At method invocation, we know:
 - Object's pointer
 - Name of Method we want to call
- Need to find out address of code for method
- Be nice if we could just ask the object

Method Lookup

- What if all objects implemented a method that could give you the address to call for a particular method
- Could call this method to get the address
- Then use that address to call the method

```
SomeObj *anObject;
```

```
anObject->DoStuff();
```

```
addr = anObject->GetAddressForMethod("DoStuff");  
CallMethodAt(addr);
```

So the method invocation at the top, would end up being expanded to the psuedo-code below

Problems...

- How do we call `GetAddressForMethod`
- Recursive problem
- Need a bootstrap
- Fortunately, implementation of this method will be identical for all objects
- Only the data that changes

Object Layout

- Think back to the layout of objects in memory
- Spare slot reserved at the start of the object in memory
- This is used to hold the *virtual function table*
- An array of addresses to the methods in memory

| offset | Contents |
|--------|----------|
| 0 | m_day |
| 4 | m_month |
| 8 | m_year |

Offset zero is used by the language to support polymorphism and inheritance, common across all languages but exact details vary (look at this in a later lecture)

| offset | Contents |
|--------|----------|
| 0 | |
| 4 | m_day |
| 8 | m_month |
| 12 | m_year |

Offset zero is used by the language to support polymorphism and inheritance, common across all languages but exact details vary (look at this in a later lecture)

| offset | Contents |
|--------|----------|
| 0 | |
| 4 | m_day |
| 8 | m_month |
| 12 | m_year |

← What goes here?

Offset zero is used by the language to support polymorphism and inheritance, common across all languages but exact details vary (look at this in a later lecture)

| offset | Contents |
|--------|---------------|
| 0 | <i>vtable</i> |
| 4 | m_day |
| 8 | m_month |
| 12 | m_year |

Offset zero is used by the language to support polymorphism and inheritance, common across all languages but exact details vary (look at this in a later lecture)

| offset | Contents |
|--------|-----------|
| 0 | |
| 4 | m_day |
| 8 | m_month |
| 12 | m_year |
| 16 | m_hours |
| 20 | m_minutes |
| 24 | m_seconds |

CDateAndTime

Our CDateAndTime object has an identical layout in memory as CDate for the shared portion
So CDate's methods will work correctly (won't know any difference)
CDateAndTime's methods can access the extra variables

| offset | Contents |
|--------|----------|
| 0 | |
| 4 | m_day |
| 8 | m_month |
| 12 | m_year |

CDate

| offset | Contents |
|--------|-----------|
| 0 | |
| 4 | m_day |
| 8 | m_month |
| 12 | m_year |
| 16 | m_hours |
| 20 | m_minutes |
| 24 | m_seconds |

CDateAndTime

Our CDateAndTime object has an identical layout in memory as CDate for the shared portion
 So CDate's methods will work correctly (won't know any difference)
 CDateAndTime's methods can access the extra variables

vtables

- An array of addresses of methods
- Convert method names into array indexes
- Need a standardized method
- Take the order methods are defined
 - First method is index 0
 - Second method is index 1 etc

```
class CDate
{
public:
    CDate(int day, int month, int year);

    int Day();
    int Month();
    int Year();
    void SetDate(int day, int month,
                int year);

    char* ToString();

private:
    int m_day;
    int m_month;
    int m_year;
};
```

Methods placed in vtable in order so, Constructor is index 0,
Day is index 1 etc etc

vtables and Classes

- Only need one vtable per class
- All objects of a class support the same operations
- vtable can be shared between objects

Method Invocation

- Method Invocation becomes:
 - Place parameters on stack
 - Look up method's address in object's vtable
 - Jump to that address

Inheritance

- Inheritance becomes easy to implement
- New methods are added on at the end of the array
- Methods that override pre-existing methods replace the existing entries

Calling the Superclass

- What about calling `super()`?
- How does it know what to call if the old value is replaced in the vtable?
- A Class only has one superclass,
- Implementation is known at compile-time
- Correct address compiled directly into generated code

Restrictions

- Seen how we can call methods polymorphically
- No restriction on what the method does or how it works
- This can lead to problems...
- E.g. do you index collections from zero or from one?

The Object Lifetime

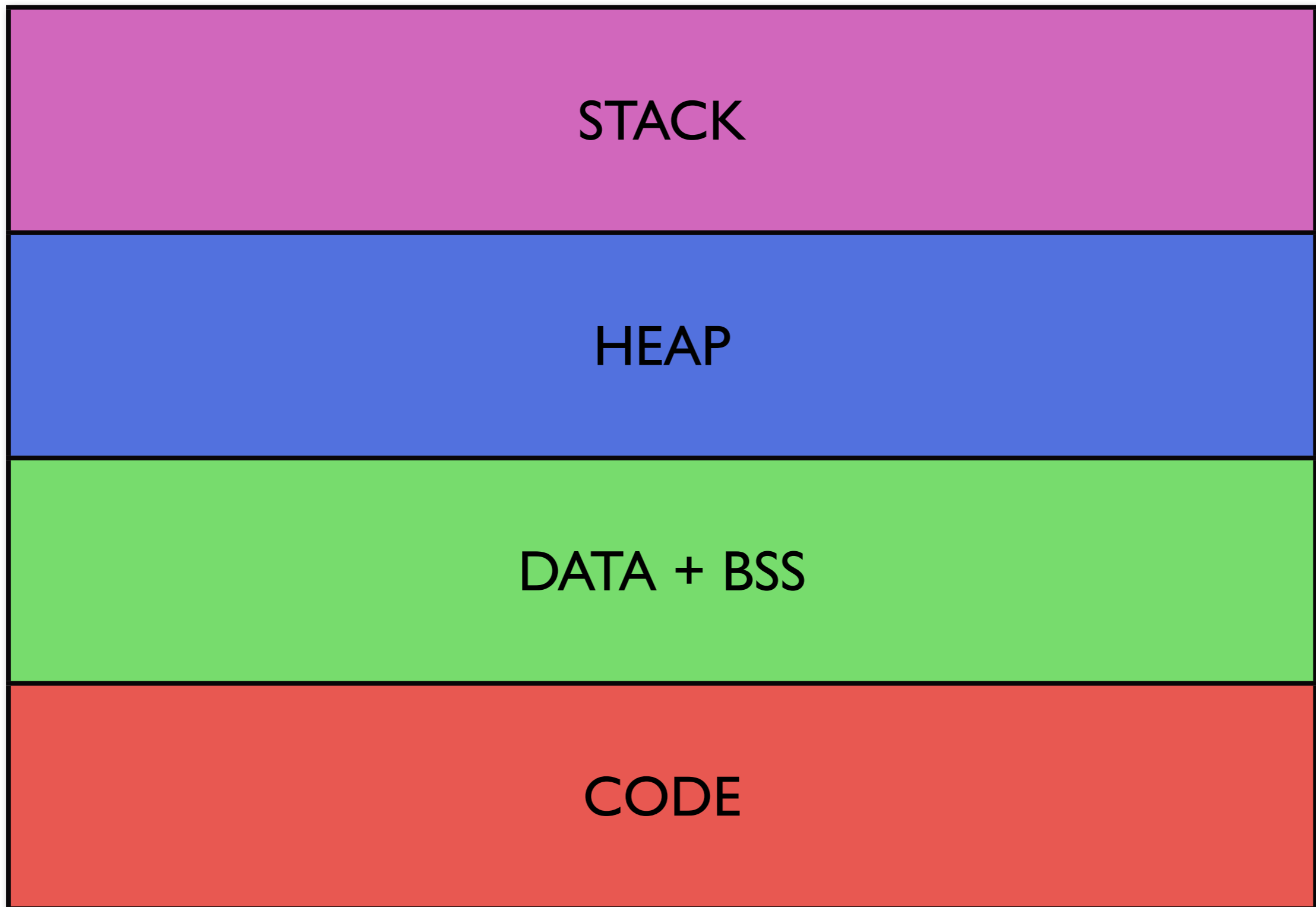
Life of a Typical Object

- Object gets created (by `new`)
- Sometime later it is destroyed
- Destroyed when no longer needed
 - How do we know when it is no longer needed?
 - Who is responsible for clearing it up?

Object Creation

- Object creation is
 - Allocating the memory for the object
 - Initializing it (by calling the constructor)
- The `new` operator — returns pointer to created object
- Some OO languages make the tasks explicit

Objective-C -- common on the mac. First you ask the class to allocate some memory, then you call the constructor on that proto-'object'...



Typical Memory layout

Explain typical memory layout of a program

Code -- machine code

Data + BSS -- global variables, (plus other bits and pieces such as predefined strings)

Stack -- local variables (grows downward -- top of stack below bottom!)

Heap -- This is where objects live!

Allocating memory

- Need to know how big the object is
- Allocate some of the heap for the object
- Mark the memory as used
- Library/OS routines to handle this

Manages a list of free memory and a list of used memory

When memory is allocated a section is removed from the free memory (a block of free memory is split in two if it is too big) and pushed into the list of used memory. when it is finished with it is moved out of the used pile into the free memory

Initializing memory

- The job of the Constructor – `new` calls the constructor
- Values of instance variables will be whatever was in memory before
- Likely to be complete nonsense
- Constructor there ensures it isn't nonsense

Object Access

- Pointer to the created object stored in a variable for later use
- Program can access object until either pointer goes out of scope (local variables)
- Or it is overwritten (with a pointer to another object)

```
void Method()  
{  
    Foo *foo = new Foo();  
    foo->DoStuff();  
}
```

```
Foo *foo = new Foo();  
  
foo->DoStuff();  
foo = new Foo();  
foo->DoStuff();
```

In first example, object no longer accessible after end of Method() because the pointer is local to method

In second example, original object no longer accessible because the pointer now points at a new and very different Foo Object;

Object Access

- Object still exists, even if we do not have a pointer to it
- Continues to take up memory
- As program executes, more and more objects created
- Eventually run out of memory
- This is a problem!

Memory worries

- Memory is BIG, but finite
- Especially on embedded systems (mobile phones, iPods etc)
- Destroy unused objects
- So Memory can be reused for new objects

Objects of Destruction

- Both C++ and Java destroy objects
- In C++, the programmer is responsible
- In Java, it is done automatically by the JVM

Object Deletion

- In C++, we destroy objects with `delete`
- `delete pFoo;`
Delete the object pointed to by `pFoo`
- Warning! Doesn't wipe the memory or clear the pointer
- Deletion is the inverse of construction
- Tell Memory Manager that memory is 'free'

Creation vs Destruction

- Object creation is
 - Allocate memory
 - Call constructor
- Object Destruction is
 - Call Destructor
 - Deallocate memory

Destructors

- Destructor called when an object destroyed
- Its job is to ensure that any objects the object has references too are also destroyed (remember you are in total control)
- Close any resources the object might use

Destructors in C++

- C++ specifies destructors as a method called `~ClassName()`
- Just as constructor is the same as the class name

```
class CSomeClass
{
public:
    CSomeClass(); // This is a constructor
    ~CSomeClass(); // this is a destructor
}
```

Object Ownership

- Who deletes an object?
- An object must be deleted when it is no longer used
- But how can a piece of code know that the object isn't being used

Simple Case

```
void AMethod()
{
    Foo *foo = new Foo();

    foo->SetFoo(42);
    foo->DoStuff();
    foo->SetFoo(1963);
    foo->DoMoreStuff();

    delete foo;
}
```


Easy to say when we delete here -- Code uses object, then when it is finished with deletes it
Can we just say, what creates an object is responsible for destroying it
No -- what happens if we give the pointer to something else

Problem Case

```
void AMethod(Collection *aCollection)
{
    Foo *foo = new Foo();

    foo->SetFoo(42);
    aCollection->Add(foo);

    delete foo;
}
```



We have some sort of Collection representation that we add the object too

At this point, we have too things pointing to the object. Our local variable foo, and the collection. If we follow, the creator deletes it paradigm then it is deleted on the next line. The collection now points to dead memory

Premature Deletion

- `AMethod` finishes and `foo` is deleted
- Object's memory is marked as unused
- Other bits of the code continue to use it and things work
- Eventually, memory will be reused
- And then it will break...