Just how does OO work, anyway?

Steven R. Bagley

Science or Magic

- OO programming can appear as magic
- Issue incantations and things happen
- E.g. `new Object()` – Creates an object
- What's really going on?
- Let's lift the hood…

OO Fundamentals

- Objects have:
  - (Encapsulated) State
  - Identity
  - Operations (or an interface)
- But how does all this work?

Anatomy of an Object

- Objects exist in memory
  - A collection of bytes in a fixed pattern
  - Used to store the data in the object
  - I.e. the value of the instance variables
  - Has a fixed size (in bytes)
- How does the CPU access them?

Need to change gear, we tend to think of objects at a high-level representation
HAve to consider them today as the computer does
Size of an object is the sum of the size of its instance variables

```
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;
};
```

When methods are called they are implicitly told which object they are acting upon and so do not need to be stored within a class.

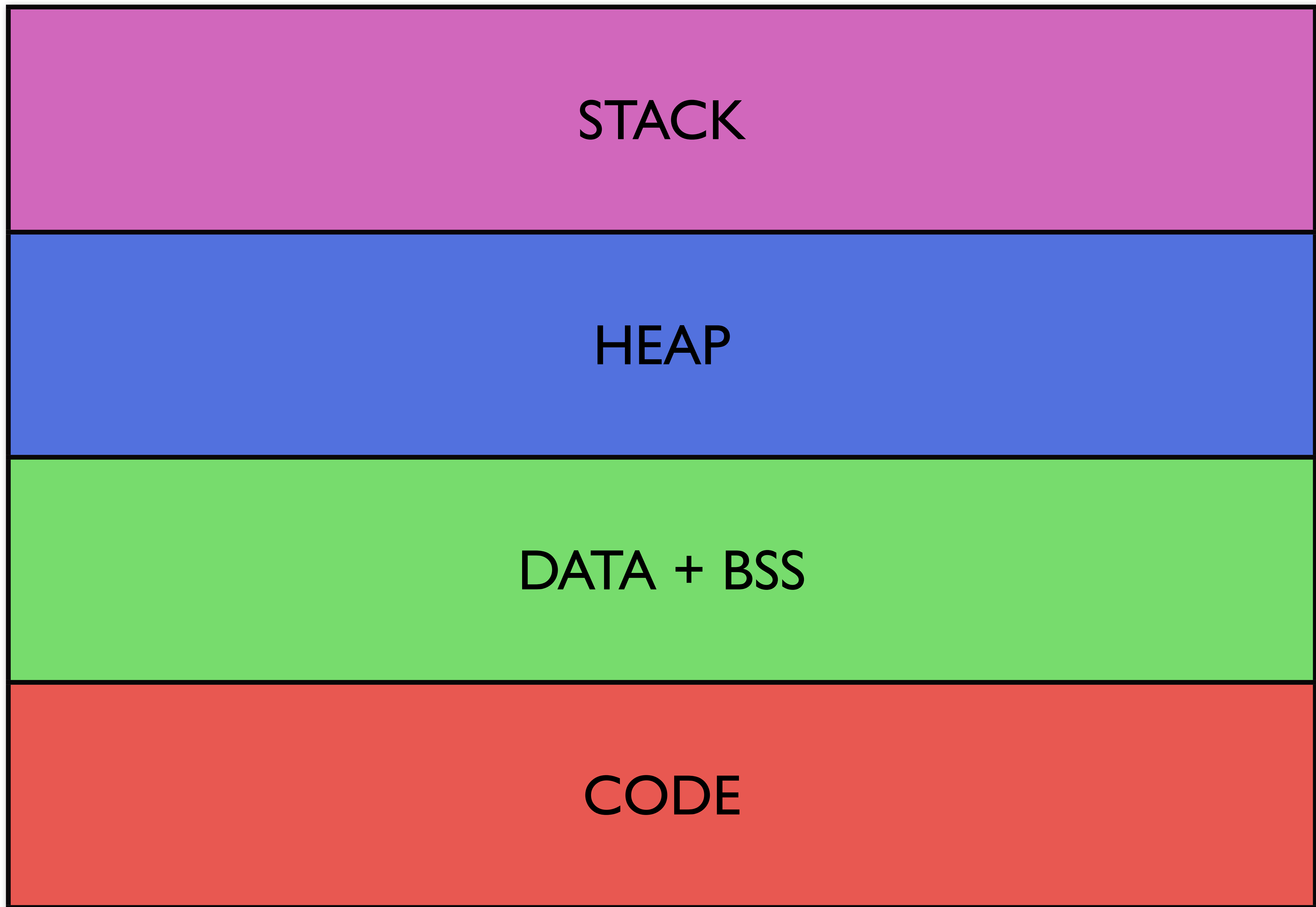
```cpp
class CDate
{
public:
  CDate(int day, int month, int year);

  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;
};
```

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 langauges
but exact details vary (look at this in a later lecture)

Date class in memory

- Our simple date class will take up 12 bytes of memory
- How does the program know where it is
- Need to consider the memory model of an application



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!

Position of Variables

- Global variables are always in a known position in the Data section
- Local variables are always in a known position on the stack
- Compiler knows position at compile time
- So can generate code to access them easily

Variable Position

- Memory is allocated from the heap in a first-come first-served fashion
- Object is constructed from memory allocated off the heap
- Object's position dependent on previously allocations
- Position unknown at compile time

Finding an Object

- Object's location is only known when we allocate memory for it
- If we store this value then we know where to access it
- Need to store it in a known location such as a variable
- Access object *in terms of* stored address

Indirection

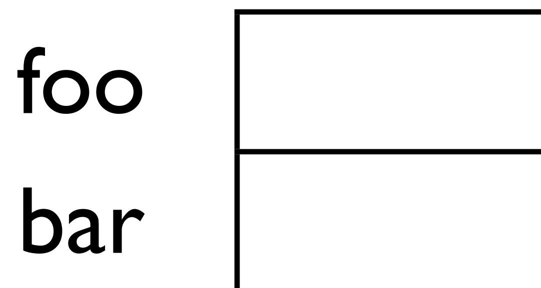
- This is called *Indirect access*
  - First, we look up in a known location where to find the object
  - Then we look up the object itself
- Referred to as a pointer or reference
- Really simple, think of an index in a book

Object Pointers

- Both C++ and Java refer to objects by pointers
- C++ is explicit (hence the * everywhere)
- Java tries to hide it (but it just comes up to bite you anyway)

```
Foo *foo = new Foo();
Foo *bar;

foo->SetFoo(42);
bar = foo;
foo->SetFoo(1963);

What is bar->GetFoo()?
```

C++




```
Foo foo = new Foo();
Foo bar;

foo.SetFoo(42);
bar = foo;
foo.SetFoo(1963);

What is bar.GetFoo()?
```

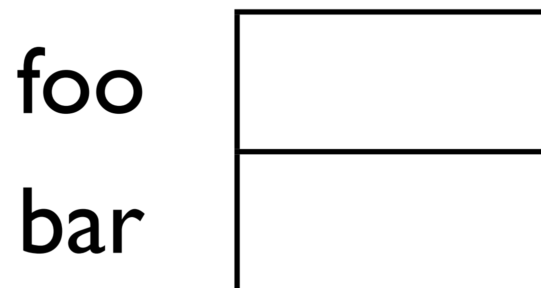
Java

Assume GetFoo returns the same value as passed into SetFoo
bar = foo just copies the pointer from foo into bar (it does not copy the object!)
so bar.GetFoo() returns 1963 since it points to the same object as Foo

```
Foo *foo = new Foo();
Foo *bar;

foo->SetFoo(42);
bar = foo;
foo->SetFoo(1963);

What is bar->GetFoo()?
```

```
Foo foo = new Foo();
Foo bar;

foo.SetFoo(42);
bar = foo;
foo.SetFoo(1963);

What is bar.GetFoo()?
```

C++



Java

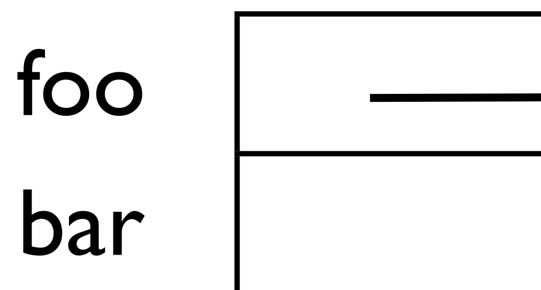
Assume GetFoo returns the same value as passed into SetFoo
bar = foo just copies the pointer from foo into bar (it does not copy the object!)
so bar.GetFoo() returns 1963 since it points to the same object as Foo

```
Foo *foo = new Foo();
Foo *bar;

foo->SetFoo(42);
bar = foo;
foo->SetFoo(1963);

What is bar->GetFoo()?
```

C++

```
Foo foo = new Foo();
Foo bar;

foo.SetFoo(42);
bar = foo;
foo.SetFoo(1963);

What is bar.GetFoo()?
```

Java



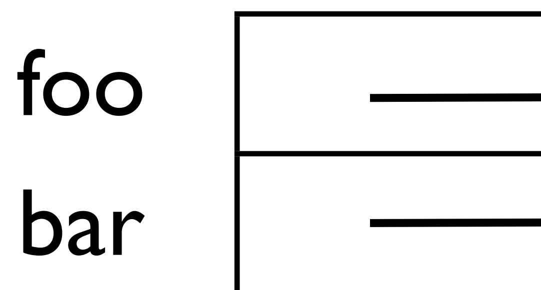
Assume GetFoo returns the same value as passed into SetFoo
bar = foo just copies the pointer from foo into bar (it does not copy the object!)
so bar.GetFoo() returns 1963 since it points to the same object as Foo

```
Foo *foo = new Foo();
Foo *bar;

foo->SetFoo(42);
bar = foo;
foo->SetFoo(1963);

What is bar->GetFoo()?
```

C++

```
Foo foo = new Foo();
Foo bar;

foo.SetFoo(42);
bar = foo;
foo.SetFoo(1963);

What is bar.GetFoo()?
```

Java



Assume GetFoo returns the same value as passed into SetFoo
bar = foo just copies the pointer from foo into bar (it does not copy the object!)
so bar.GetFoo() returns 1963 since it points to the same object as Foo

Identity

- This also gives an object's identity
- Since each object is a block of memory
- Then address of that block can be used to distinguish one object from another
- Think of it like Houses, often look identical but they have different addresses

```
CDate *a;
CDate *b;

a = new CDate(23, 11, 1963);
b = new CDate(25, 12, 2008);
```

The two objects are distinguishable because they are different blocks of memory and so have different addresses

Load up Visual Studio and try it?

Encapsulation

- Encapsulation is handled purely by the compiler
- If you can get access to raw memory, its possible to modify an object's variables at runtime
- This is usually good enough

How does OO work?

- Easy to implement State and Identity in a non-OO language
- Very common programming technique even today
- See Win32, MacOS X Carbon, Acrobat API etc. for examples

Encapsulated State

- Most languages provide support for abstract data types
  e.g. `struct` in C
- Can pass pointers to these ADTs about
- Provide procedures to alter the structure
- Client code calls these procedures
- It's all *manual*…

```
struct date
{
    int day;
    int month;
    int year;
};

date *CreateDate(int day, int month, int year)
{
  date *n = malloc(sizeof(date));
  n->day = day;
  n->month = month;
  n->year = year;
  return n;
}

date *a = CreateDate(23,11,1963);
printf(DateToString(a));
```

THIS IS NOT OO CODE!

OO Variable access

- Easy to see how non-OO code can access variables
- Procedures passed address of structure
- Variables accessed by offset from this address
- OO code never passes the method the object's pointer
- It just appears to magically work

Method Invocation

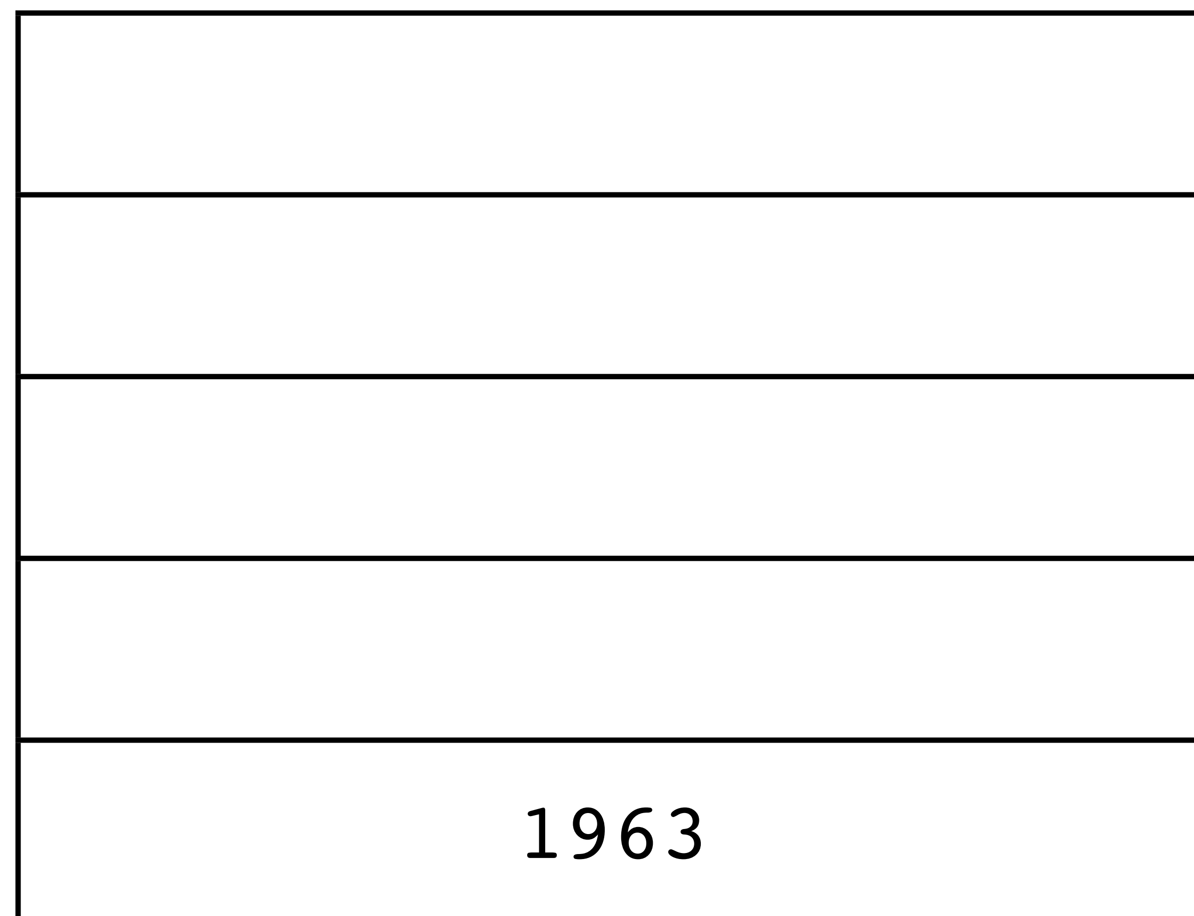
- Method or Procedure calls involve two things
  - Executing a separate block of code
  - Passing Parameters to this block of code
- In the case of OO-code, the pointer to the object is also passed over

Method Invocation

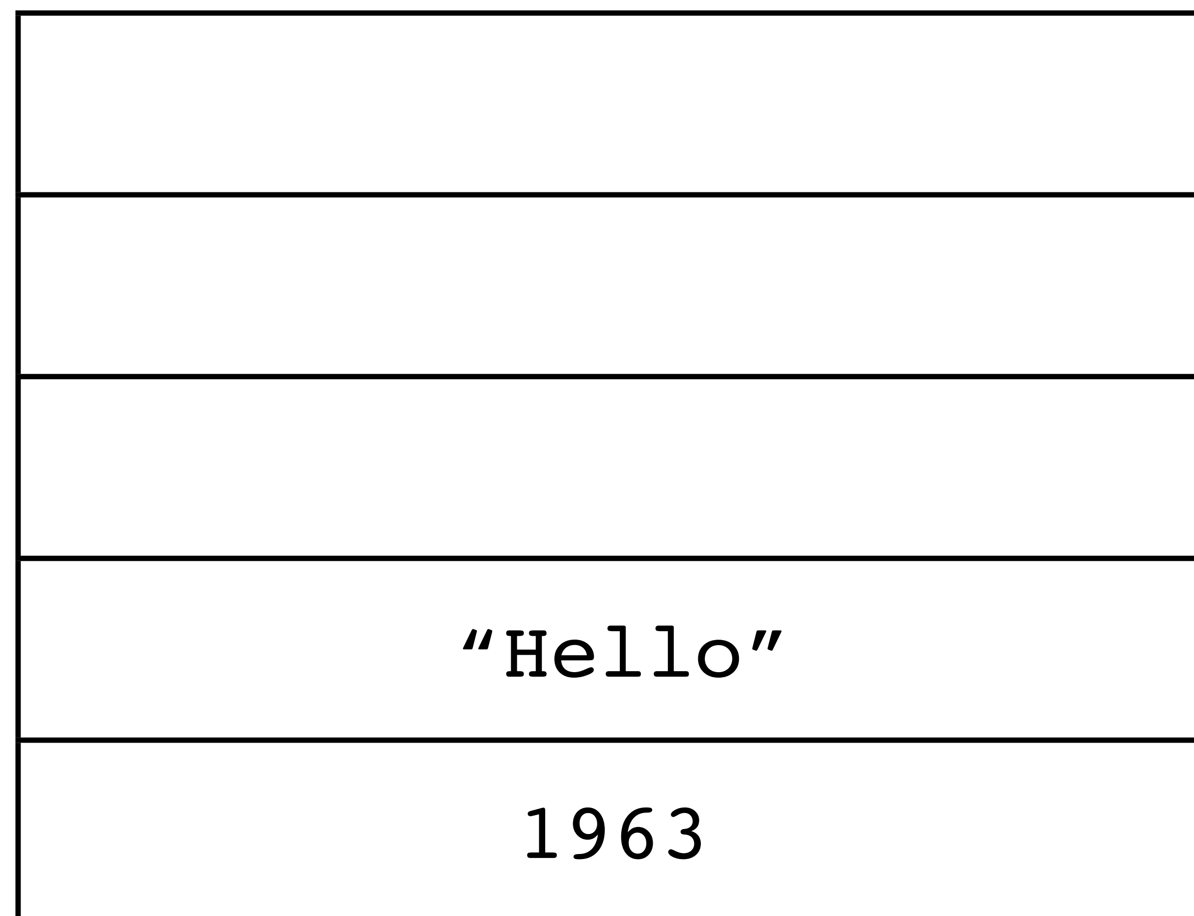
- Parameters are passed over on the stack
- Exact method depends on language and/or compiler settings
- Traditional C fashion is to push values on the stack from right to left
- Then jump to the address of the code

```
obj->Foo(anObj, 42, “Hello”, 1963);
```

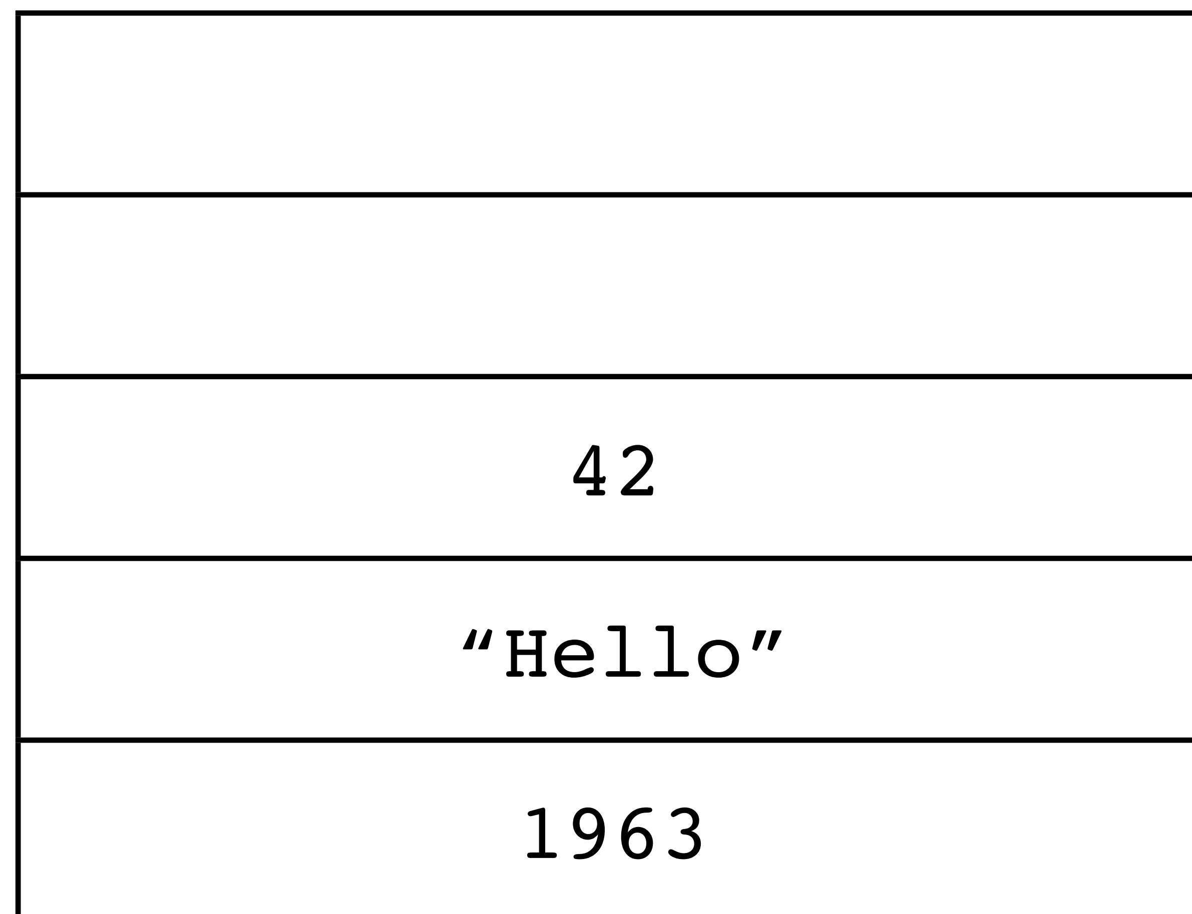
```
obj->Foo(anObj, 42, “Hello”, 1963);
```



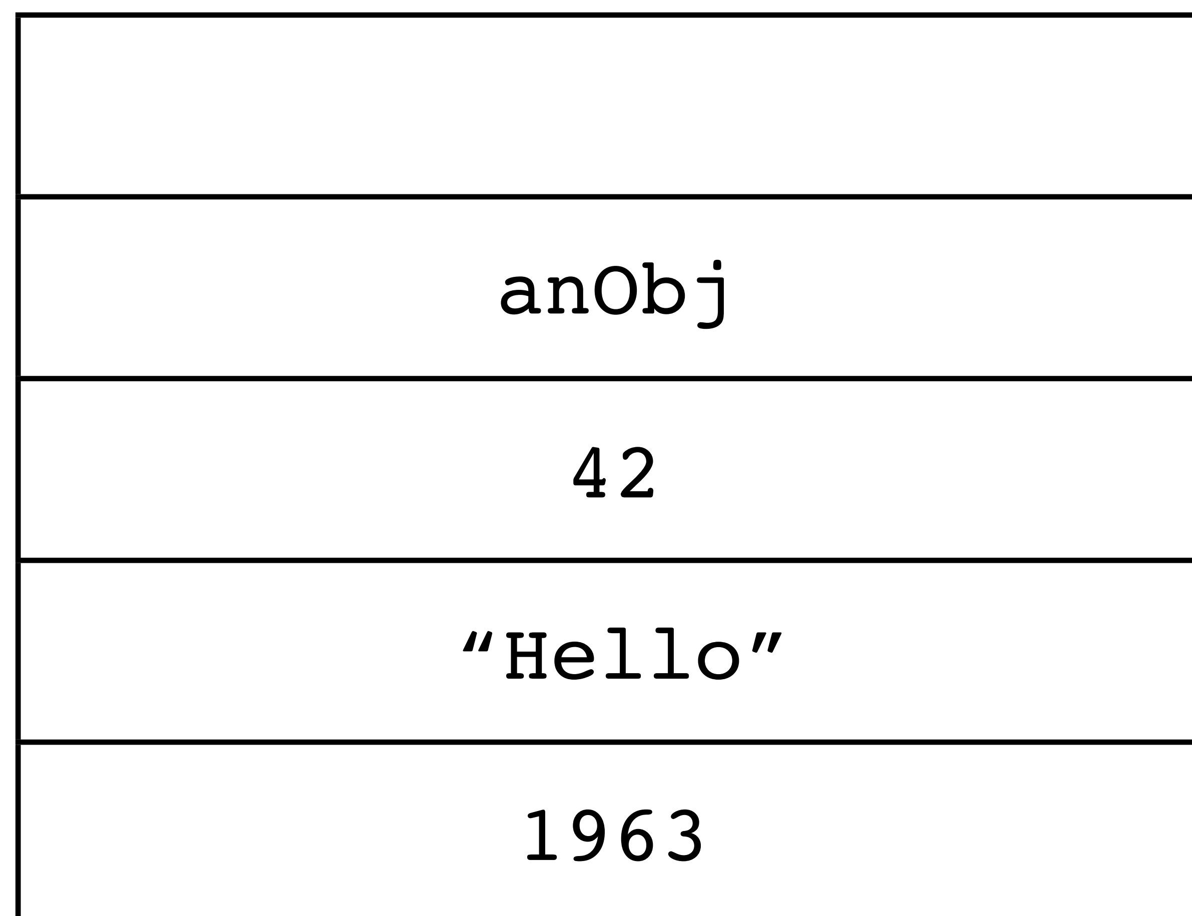
```
obj->Foo(anObj, 42, “Hello”, 1963);
```



```
obj->Foo(anObj, 42, “Hello”, 1963);
```



```
obj->Foo(anObj, 42, “Hello”, 1963);
```



```
obj->Foo(anObj, 42, “Hello”, 1963);
```

obj
anObj
42
“Hello”
1963

Method Invocation

- So Object-Oriented languages cheat
- They silently pass the base-pointer of the object over to the method
- The compiler can use this pointer to access variables by adding the relevant offset
- Same as in C, but done for you behind the scenes

Which method?

- For classes that don’t involve inheritance or polymorphism
- The address of the method to call is known at compile time
- For polymorphism, it isn’t known until runtime which object is called

Inheritance

- Inheritance is a way of extending a class
- Take an existing class and add new features to it
- New variables, new methods
- But can still call and access methods and variables on the old methods
- How does this work?

Inherited layout

- Sub-classes are laid out in memory such that the variables they inherit are at identical offsets in memory
- This means that the base-class methods can still access them when passed the new object's address

```
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;
};
```

```
class CDateAndTime : public CDate
{
public:
 CDate(int day, int month, int year, int
hour, int minutes, int seconds);

 int Hour();
 int Minutes();
 int Seconds();
 void SetTime(int hour, int minutes,
              int seconds);

private:
 int m_hour;
 int m_minutes;
 int m_seconds;
};
```

Here we extend CDAte to also store Times as a new object CDateAndTime

offset	Contents
0	m_day
4	m_month
8	m_year
12	m_hours
16	m_minutes
20	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	m_day
4	m_month
8	m_year

CDate

offset	Contents
0	m_day
4	m_month
8	m_year
12	m_hours
16	m_minutes
20	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

Method Swapping

- `CDate`'s `Day()` method can be passed an object of `CDate` or `CDateAndTime`
- Correct functionality is assured
- Because the variables for `CDate` are in the same place in both objects

Time Access

- `CDateAndTime`'s `Hour()` method can be passed an object of `CDateAndTime`
- Correct functionality is assured
- Because the variables for `CDateAndTime` are in the stored after those for `CDate`