CS153: Compilers
Lecture 11: Compiling Objects

Stephen Chong
https://www.seas.harvard.edu/courses/cs153
Announcements

- Project 3 due today
- Project 4 out
  - Due Thursday Oct 25 (16 days)
- Project 5 released on Thursday
Today

• Object Oriented programming
  • What is it
  • Dynamic dispatch
  • Code generation for methods and method calls
• Fields
• Creating objects
• Extensions
• Type system
What Is Object-Oriented Programming?

• Programming based on concept of objects, which are data plus code
• OOP can be an effective approach to writing large systems
  • Objects naturally model entities
  • OO languages typically support
    • information hiding (aka encapsulation) to support modularity
    • inheritance to support code reuse
• Several families of OO languages:
  • Prototype-based (e.g. Javascript, Lua)
  • Class-based (e.g. C++, Java, C#)
• We focus on the compilation of class-based OO languages
Brief Incomplete History of OO

• (Early 60’s) Key concepts emerge in various languages/programs: sketchpad (Sutherland), SIMSCRIPT (Hoare), and probably many others.

• (1967) Simula 67 (Dahl, Nygaard) crystalizes many ideas (class, object, subclass, dispatch) into a coherent OO language

• (1972) Smalltalk (Kay) introduces the concept of object-oriented programming (you should try Squeak!)

• (1978) Modula-2 (Wirth)

• (1985) Eiffel (Meyer)

• (1990’s) OO programming becomes mainstream: C++, Java, C#, …
What’s the difference between a class and an object?
A class is a blueprint for objects
Class typically contains
- Declared fields / instance variables
  - Values may differ from object to object
  - Usually mutable
- Methods
  - Shared by all objects of a class
  - Inherited from superclasses
  - Usually immutable

Methods can be overridden, fields (typically) can not
• Every Vehicle is an Object
• Every Car is a Vehicle, every Truck is a Vehicle
• Every Vehicle (and thus every Car and Truck) have a position field and a move method
• Every Car also has a passengers field and an await method

```java
class Vehicle extends Object {
    int position = 0;
    void move(int x) { this.position += x; }
}

class Car extends Vehicle {
    int passengers = 0;
    void await(Vehicle v) {
        if (v.position < this.position) {
            v.move(this.position - v.position);
        } else { this.move(10); }
    }
}

class Truck extends Vehicle {
    void move(int x) {
        if (x < 55) this.position += x;
    }
}
```
**Example Java Code**

```java
class Vehicle extends Object {
    int position = 0;
    void move(int x) { this.position += x; }
}

class Car extends Vehicle {
    int passengers = 0;
    void await(Vehicle v) {
        if (v.position < this.position) {
            v.move(this.position - v.position);
        } else { this.move(10); }
    }
}
class Truck extends Vehicle {
    void move(int x) { if (x < 55) this.position += x; }
}
```

- A **Car** can be used anywhere a **Vehicle** is expected (because a **Car** is a **Vehicle**!
- **Class Truck** **overrides** the **move** method of **Vehicle**
  - Invoking method `o.move(i)` will invoke **Truck**’s `move` method if `o`’s class at run time is **Truck**
Code Generation for Objects

• Methods
  • How do we generate method body code?
  • How do we invoke methods (dispatching)
  • Challenge: handling inheritance

• Fields
  • Memory layout
  • Alignment
  • Challenge: handling inheritance
Need for Dynamic Dispatch

• Methods look like functions. Can they be treated the same?
• Consider the following Java code

```java
interface Point {
    int getX();
    float norm();
}

class ColoredPoint implements Point {
    ...
    float norm() { return sqrt(x*x+y*y); }
}

class 3DPoint implements Point {
    ...
    float norm() { return sqrt(x*x+y*y+z*z); }
}

Point p = foo();
p.norm();
```
Need for Dynamic Dispatch

• Methods look like functions. Can they be treated the same?
• Consider the following Java code

```java
interface Point { int getx(); float norm(); }

class ColoredPoint implements Point {
    ...
    float norm() { return sqrt(x*x+y*y); }
}

class 3DPoint implements Point {
    ...
    float norm() { return sqrt(x*x+y*y+z*z); }
}

Point p = foo();
p.norm();
```

If `p` is object of class ColoredPoint, should execute ColoredPoint.norm()

If `p` is object of class 3DPoint, should execute 3DPoint.norm()

At run time could be either case!
Dynamic Dispatch Solution

• So we need some way at run time to figure out which code to invoke

• Solution: dispatch table (aka virtual method table, vtable)
  • Each class has table (array) of function pointers
  • Each method of class is at a known index of table
What Offset Into the VTable?

- Want to make sure that every object of class `B` has same layout of dispatch table
  - Even if object is actually a subclass of `B`!

```java
class A {
    void foo() {
    }
}

class B extends A {
    void bar() {
    }
    void baz() {
    }
}

class C extends B {
    void foo() {
    }
    void baz() {
    }
    void quux() {
    }
}
```

- List methods in order
- Ensures that a dispatch table for class `C` also looks like a dispatch table for class `B`, and for class `A`
Dispatch Tables

Dispatch table for class A

&.foo

Dispatch table for class B

&A.foo
&B.bar
&B.baz

Dispatch table for class C

&C.foo
&B.bar
&C.baz
&C.quux

• Dispatch table for class C looks like a dispatch table for class B
  • i.e., address for method foo is at index 0 (offset 0 bytes)
    address for method bar is at index 1 (offset 4 bytes)
    address for method baz is at index 2 (offset 8 bytes)

• And it looks like a dispatch table for class A
  • i.e., address for method foo is at index 0
Generating Code for Methods

• For method declarations
  • Methods have implicit argument, the **receiver object** (i.e., this, self)
  • In essence, method `bar` declared in class `B`
    ```
    class B {
        void bar(int x) { ... }
    }
    ```
    is translated like a function
    ```
    void bar(B this, int x)
    ```
• For method call `o.bar(54)`
  • Essentially: `void (*f)(obj *,int);`
    ```
    f = o->class_ptr->vtable[offset for bar]
    f(o, 54);
    ```
  • i.e., use vtable to get pointer to appropriate function, invoke it with receiver and arguments
Fields

• Same basic idea for fields as for methods!

```java
class 2DPoint implements Point {
    int x;
    int y;
    ...
}

class 3DPoint implements Point {
    int z;
    ...
}
```

• Representation of object of class 3DPoint has space to store fields of 3DPoint and superclasses

Object `o` of class 3DPoint
- `class_ptr`
- `2DPoint.x`
- `2DPoint.y`
- `3DPoint.z`
Generating Code for Field Accesses

• To access field $x.f$
  • $x$ will be represented as pointer to object
  • Need to know (static) type of $x$
  • $x.f$ refers to memory location at appropriate offset from base of object $x$
• E.g., reading $o.y$ would translate to dereferencing address $o+(offset \ for \ y)$
Creating Objects

- `new C` creates a new object of class `C`
  - Creates record big enough to hold a `C` object
  - Initializes instance variables
  - Evaluates to pointer to newly created object
Extensions...

• Multiple inheritance
  • Typically use multiple vtables (one for each base class) and switch between them based on the static type
  • Other approaches possible

• Separate compilation
  • Don’t know how many fields/method in superclass! (Superclass could be recompiled after subclass)
  • Resolve offsets at link or load time
Extensions...

- Prototype based OO languages
  - Similar approach, but vtable belongs with object (no classes!)
  - Objects are created by cloning other objects
  - Many objects will have the same vtable: can share them, with copy-on-write

- Runtime type check: `o instanceof C`
  - Each object contains pointer to its class, so can figure out at runtime if a `o`'s class is a subclass of `C`
  - But how to efficiently store inheritance information in runtime representation of classes?
• Visibility
  • To support encapsulation, some OO languages provide visibility restrictions on fields and methods
  • Java has *private*, *protected*, *public* (and some more)
    • private members accessible only to implementation of class
    • public members accessible by any code
    • protected members accessible only to implementation of class and subclasses

• Subclassing vs inheritance
  • Somewhat conflated in Java
  • Inheritance: reuse code from another class;
    Subclassing: every object of *subclass* is a superclass *object*
  • C++ has visibility restrictions on inheritance
Subclassing vs subtyping
- Not the same!
- No contravariance in argument type in Java methods

Overriding vs overloading
- Given $\text{C.m}(T_1, T_2, \ldots, T_n)$ and $\text{D.m}(S_1, S_2, \ldots, S_m)$ where C is subclass of D,
  $\text{C.m}$ overrides $\text{D.m}$ only if $T_1, T_2, \ldots, T_n = S_1, S_2, \ldots, S_m$
  - Otherwise, $\text{D.m}$ just overloads the method name $m$...

Null values
- In Java type C for class C is analogous to C option in ML
  - Since any object value can be null
  - ...