Compiler Design

Lecture 21: Object Oriented Features

Christophe Dubach
Winter 2021

Some content inspired from Dr. Michel Schinz's lecture on Advanced Compiler Construction, EPFL 2016

Timestamp: 2021/04/09 13:16:29

Object-Oriented features in Java

```
class A {
  int x;
  float foo() {...}
}

class B extends A {
  int y;
  float foo() {...}
  float bar() {...}
}

class Main {
  void f(A a, B b) {
    a.foo();
    b.x;
  }
}
```

How does the compiler supports object oriented features?

- Where is `b.x` in memory?
- Where is the implementationof `a.foo()`?

Object-Oriented Features

Object Layout

Method Dispatch

Object Layout: Single inheritance

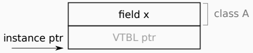
In a single-inheritage language, a class can only inherit from a single superclass.

For such languages, fields in an object can simply be laid out sequentially, starting from the ones from superclass.

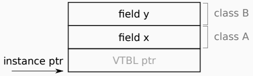
This means that a field declared in a class will always be in the same location, no matter what the *instance type* of the object of that class is.

```
class A {
  int x;
  float foo() {...}
}

class B extends A {
  int y;
  float foo() {...}
  float bar() {...}
}

class Main {
  void f(A a, B b) {
    a.foo();
    b.y;
    b.x
    a.x;
  }
}
```

Object layout for A



Object layout for B



Field x is at the same offset from the start of the object in both cases!

Assuming instance pointer in `$t0`:

Code for `b.y`:

```
lw $t1, 8($t0)
```

Code for `b.x`:

```
lw $t1, 4($t0)
```

Code for `a.x` (`a` can be instance of `A` or `B`):

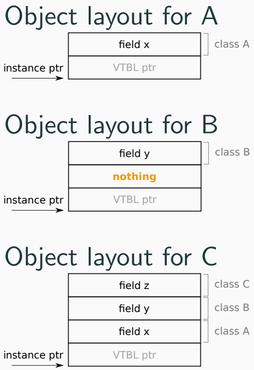
```
lw $t1, 4($t0)
```

Object Layout: Multiple inheritance

In the case of multiple-inheritage language, object layout becomes more complicated.

Unidirectional object layout

```
class A {
  int x;
}

class B {
  int y;
}

class C extends A&B{
  int z;
}
```



Wasted space!

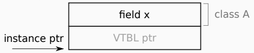
And requires to know, ahead of time, the entire class hierarchy.

Bidirectional object layout

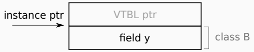
Idea: store fields above *and* below the object instance pointer.

```
class A {
  int x;
}

class B {
  int y;
}

class C extends A&B{
  int z;
}
```

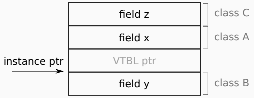
Object layout for A



Object layout for B



Object layout for C



No more wasted space!

However:

- Requires to know, ahead of time, the entire class hierarchy;
- Might not always be possible to avoid wasted space.

Accessor methods

In the context of multiple inheritance, we can take an alternative approach:

- lay out the fields from the class freely (in the most compact manner); and
- use getter and setter accessor methods and rely on the method dispatch mechanism.

The drawback: accessor methods are much slower than direct access to the fields.

Object layout summary

Problem is easy in the case of single-inheritance languages (*e.g.* Java).

The problem becomes more complex in the case of a multiple-inheritance languages (*e.g.* C++):

- trade-off between speed and space;
- might require access to the whole class hierarchy $\Rightarrow$ close-world;
- or, for instance, rely on accessor methods / dispatching.

Object-Oriented Features

Object Layout

Method Dispatch

Method Dispatch

Class methods

The problem: given a class and a method name (and its arguments), find the method's code to execute.

Trivially solved at compile time (static) with name analysis.

Instance methods

The problem: given an object instance and a method name (and its arguments), find the method's code to execute.

Not possible (in general) to solve at compile time in the presence of inheritance: the specific method's code to execute depends on the runtime type of the object!

```
class A {
  void foo() {print(a)}
}

class B extends A {
  void foo() {print(b)}
}

class Main {
  void f {
    A a  = new A();
    B b1 = new B();
    A b2 = new B();
    a.foo();  // prints a
    b1.foo(); // prints b
    b2.foo(); // prints b
  }
}
```

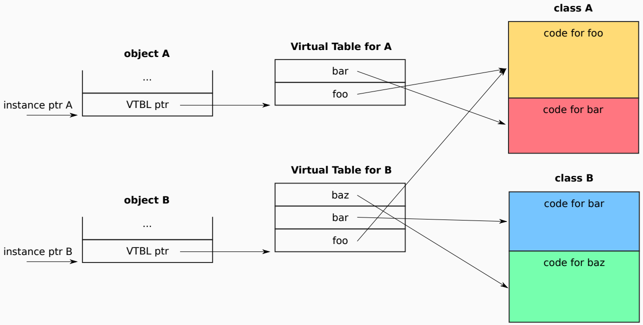
When calling `foo`, the runtime has to decide between the two implementations based on the instance type of the object.

This is generally what we refer to as dynamic dispatch.

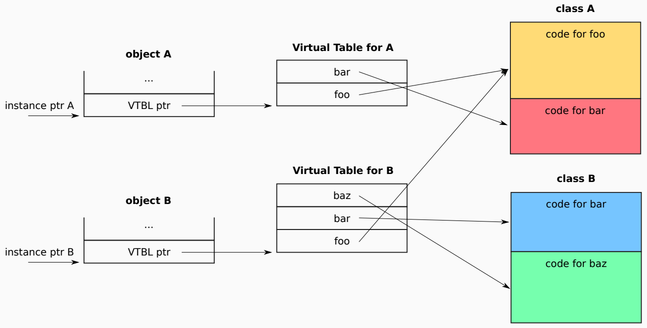
Dynamic dispatch with Virtual tables

```
class A {
 void foo(){print("foo_a")}
 void bar()(prinf("bar_a")}
}
```

```
class B extends A {
 void bar(){print("bar_b")}
 void baz(){print("baz_b")}
}
```



Inherited methods from the superclass are at the same fixed position in the virtuable table.



Assuming variable p declared with type `A`, code for `p.bar()`:

```
# assuming p stored in $t0
lw   $t1, 0($t0) // get virtual table pointer
lw   $t2, 4($t1) // get address of code for subroutine bar
jalr $t2         // jump&link to subroutine
```

Depending on the *instance type of p*, the corresponding `bar` method will be called.

Accessing fields from an instance method

Consider the following example:

```
class A {
  int i;
  void inc() { i = i+1 }
}
```

How does the implementation of `inc` reach the instance variable `i`?

In fact, the code above looks more like this:

```
class A {
  int i;
  void inc() { this.i = this.i+1 }
}
```

Okay, but where do we get the reference `this` from?

Easy, it is passed as an argument to the instance method:

```
void inc(A this) { this.i = this.i+1 }
```

So when you write:

```
A p = ...;
p.inc();
```

what is really happening being the scence is that you virtually dispatch to the implemention of `inc` passing `p` as the first argument:

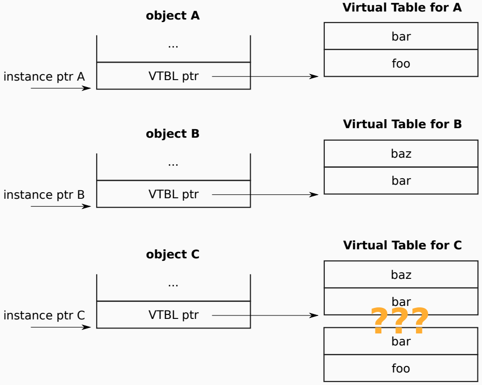
```
A p = ...;
p.inc(p);
```

Multiple inheritance

What have been shown above works for single-inheritance.

However, in the presence of multiple-inheritance, problems start arising again as we cannot guarantee that the methods are always at the same fixed position in the virtual table:

```
class A {
 void bar(){...}
 void foo(){...}
}
class B {
 void bar(){...}
 void baz(){...}
}
class C extends A&B{
  void foo(){...}
}
```

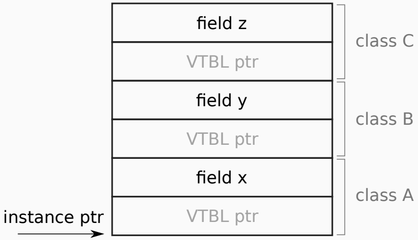


Back to square one!

Luckily solutions exist, based on the idea of embeddeding layout of superclasses into the subclass:

```
class A {
  int x;
}
class B {
  int y;
}
class C extends A&B{
  int z;
}
```

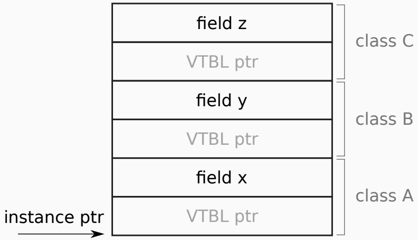
Layout for C:



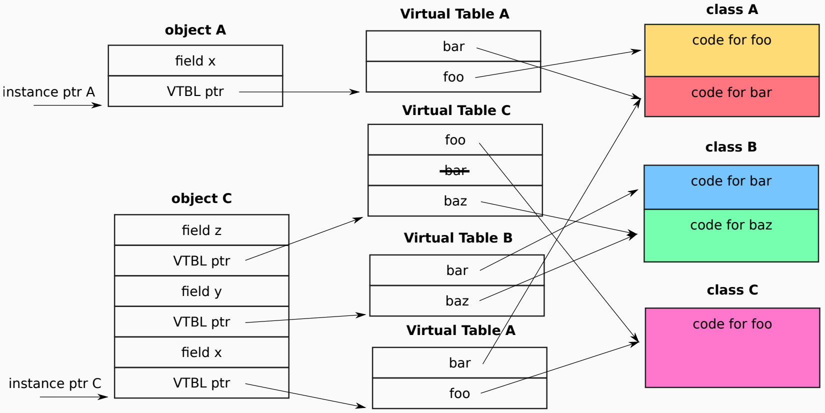
There is one virtual table for each super class in the object and the virtual table is chosen based on the static type of the instance `ptr`.

```
class A {
 int x;
 void bar(){...}
 void foo(){...}
}
class B {
 int y;
 void bar(){...}
 void baz(){...}
}
class C extends A&B{
 int z;
 void foo(){...}
}
```

Layout for C:



Layout and virtual tables for object A and C:



Key property: instance methods of given class are always in the same location in the corresponding virtual table.

Given the following code:

```
C c  = new C();
A ac = new C();
A a  = new A();
c.foo();
ac.bar();
a.bar();
c.bar(); <-- compile time error!
```

To make this work, we add an offset to the instance pointer based on the static type which will bring us to the right table.

This is not the end of the story, but we won't cover more in this lecture. Additional techniques exist to make this efficient:

- Trampoline (used commonly in C++ implementations);
- Row displacement tables;
- Inline caching (great when using a Just-In-Time (*JIT*) compiler).

Summary:

- Single-inheritance languages are easy to implement
  - Layout the fields sequentially in the object;
  - Use a single virtual table to perform dynamic dispatch.
- Multiple-inheritance brings some challenges but solutions exist
  - Embedded layout;
  - Together with Trampoline, row displacement tables or inline caching.