Compiler Construction
Lent Term 2014
Lecture 10 (of 16)

- Assorted topics
  - Universal polymorphism
  - Tuples
  - Objects

Timothy G. Griffin
tgg22@cam.ac.uk
Computer Laboratory
University of Cambridge

A peek at universal polymorphism

```
map : (‘a -> ‘b) -> ‘a list -> ‘b list

fun map f [] = []
  | map f (a::rest) = (f a) :: (map f rest)
```

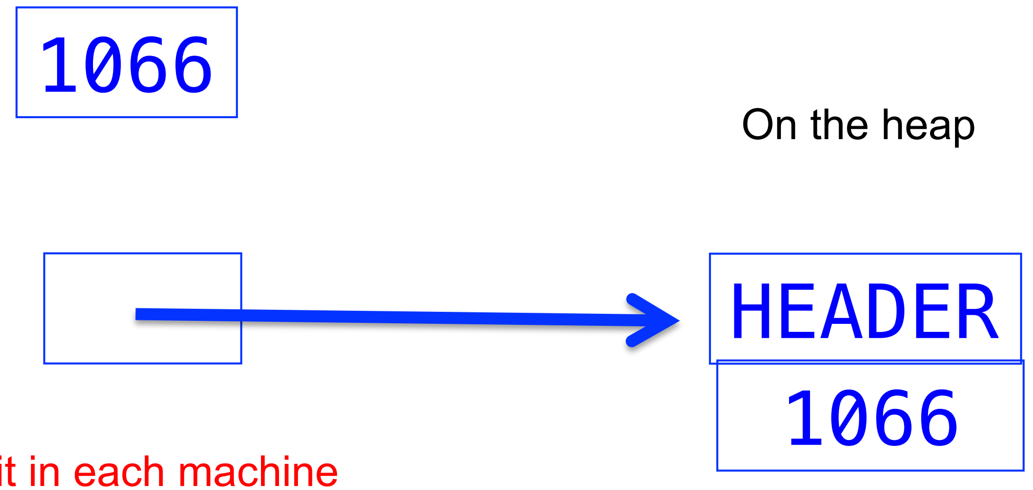
The code generated for `map` must work
for any times `‘a` and `‘b`.

So it seems that all values of any type must
be represented by objects of the same size.

Boxing and Unboxing

An unboxed integer :

A boxed integer :



Many ML compilers use a single bit in each machine word to distinguish boxed from unboxed values. This is why mosml has 31 (or 63) bit integers.

It is better to work with unboxed values than with boxed values.

Compilers for ML-like languages must expend a good deal of effort trying to find good optimizations for boxed/unboxed choices.

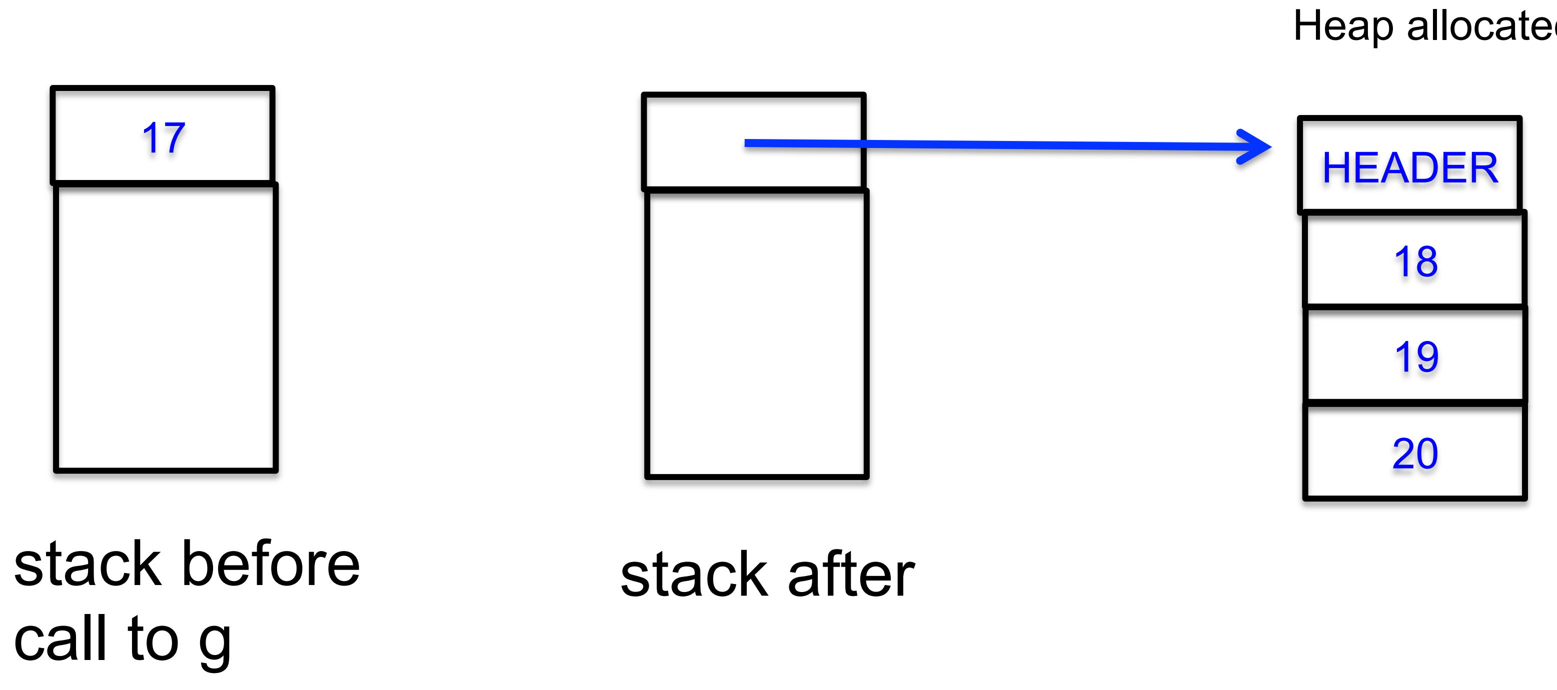
See Appel.

Similar terminology is used in Java for putting a value in a container class (boxing) and taking it out (unboxing)

For example, put an int into the Integer container class.

Tuples (in ML-like, L3-like languages)

```
g: int -> int * int * int

fun g x = (x+1, x+2, x+3)

   . . . (g 17) . . .
```

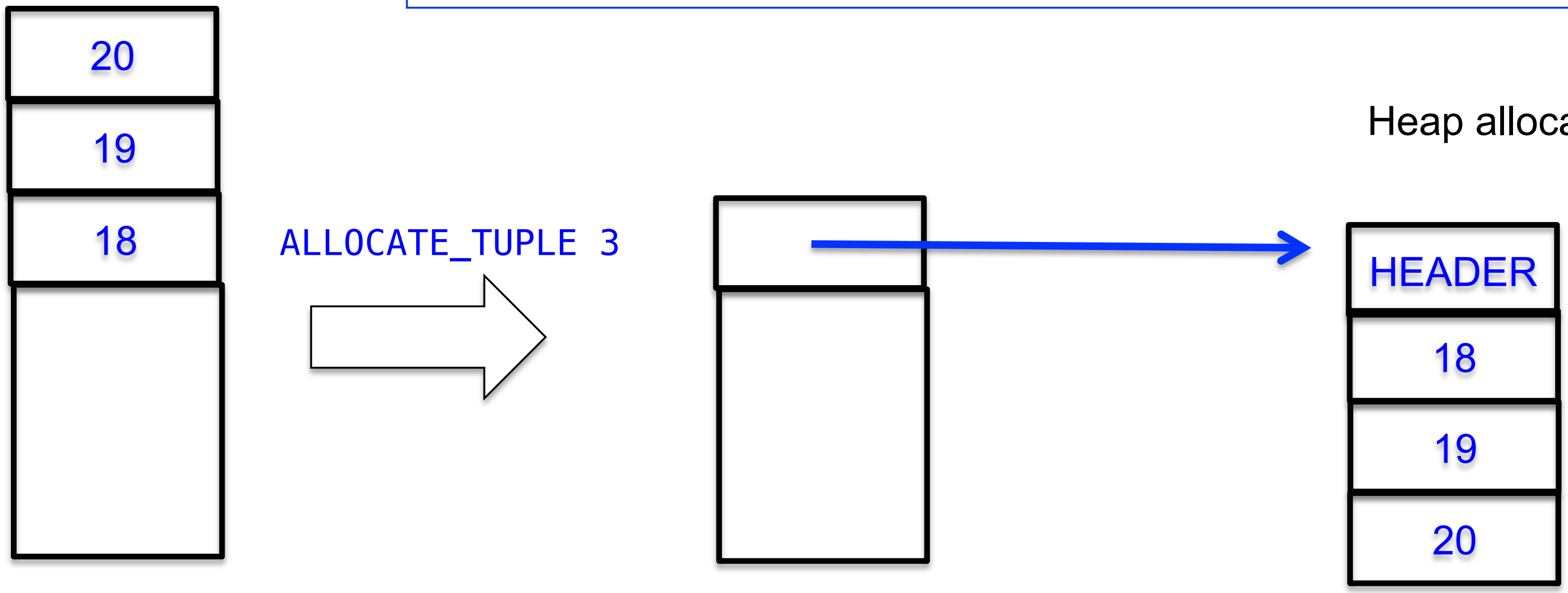


On a stack-oriented machine

```
fun g x = (x+1, x+1, x+3)
```

```
fun g x =
   let val y1 = x+1
       val y2 = x+2
       val y3 = x+3
   in return (ALLOCATE_TUPLE 3) end
```

Some IR

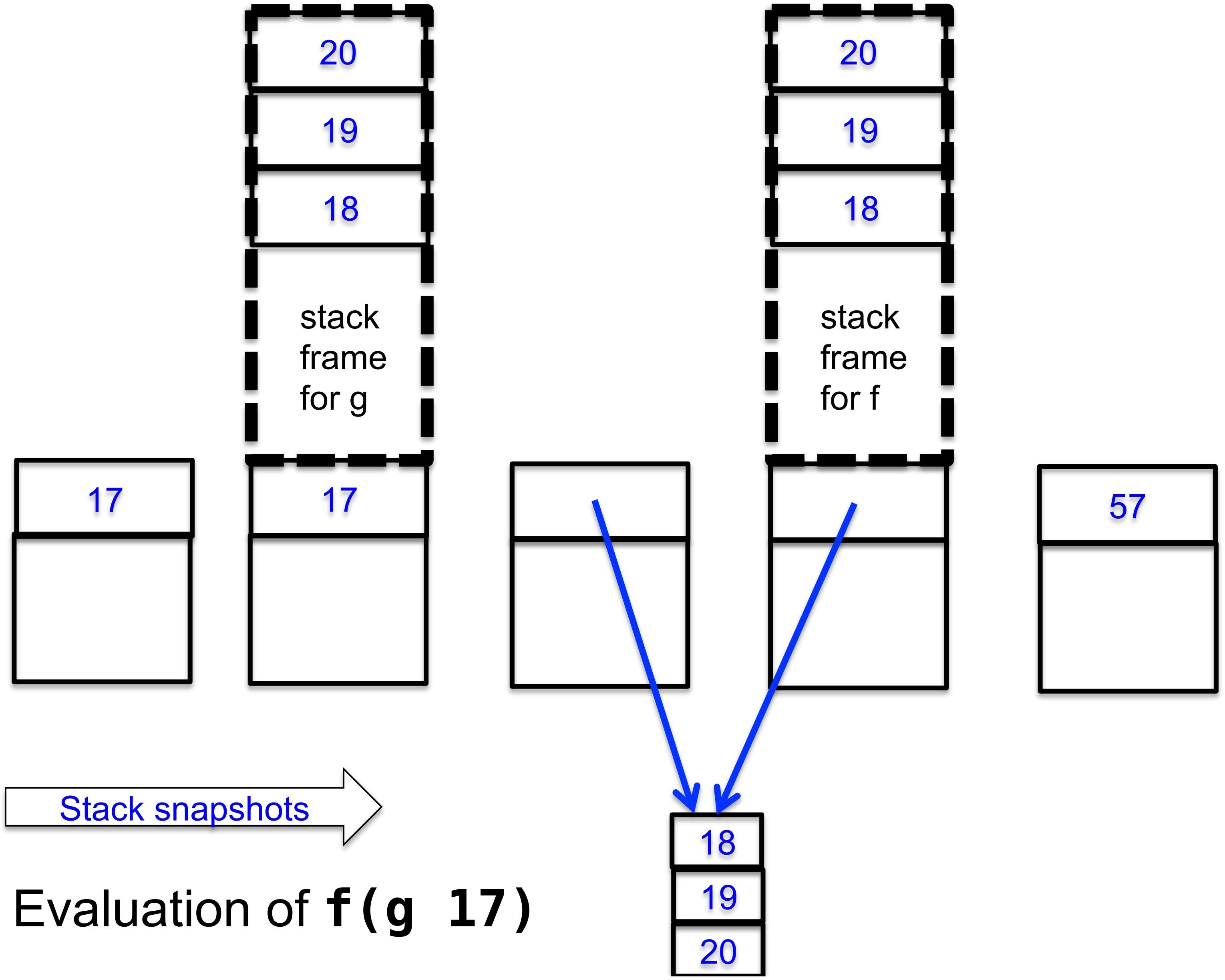


Tuples (in ML-like, L3-like languages)

```
fun g x = (x+1, x+1, x+3)

fun f (u, v, w) = u + v + w

   . . . f (g 17) . . .
```

- Does function f take 3 arguments or 1?
- How would you inline f?

How might we avoid this?



Evaluation of **`f(g 17)`**

New Topic : Objects (with single inheritance)

```
let start := 10

   class Vehicle extends Object {
      var position := start
      method move(int x) = {position := position + x}
   }
   class Car extends Vehicle {
      var passengers := 0
      method await(v : Vehicle) =
         if (v.position < position)
         then v.move(position - v.position)
         else self.move(10)
   }
   class Truck extends Vehicle {
      method move(int x) =
         if x <= 55 then position := position +x
   }
   var t := new Truck
   var c := new Car
   var v : Vehicle := c
in
   c.passengers := 2;
   c.move(60);
   v.move(70);
   c.await(t)
end
```

method override

subtyping allows a Truck or Car to be viewed and used as a Vehicle

Object Implementation?

- how do we access object fields?
  - both inherited fields and fields for the current object?
- how do we access method code?
  - if the current class does not define a particular method, where do we go to get the inherited method code?
  - how do we handle method override?
- How do we implement subtyping ("object polymorphism")?
  - If B is derived from A, then need to be able to treat a pointer to a B-object as if it were an A-object.

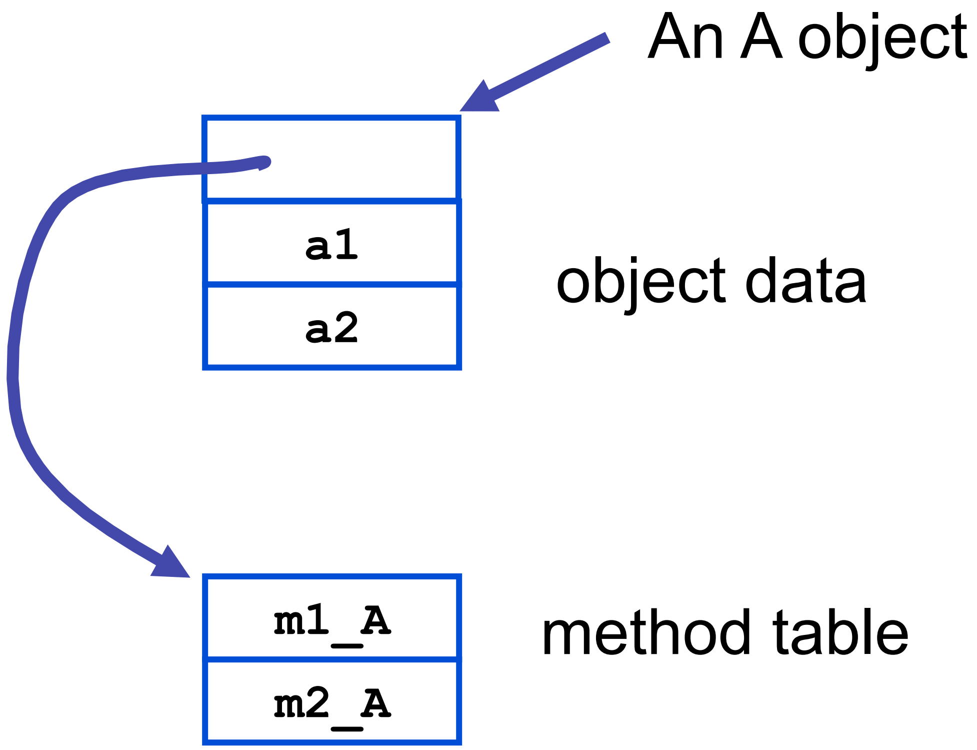
Another OO Feature

- Protection mechanisms
  - to encapsulate local state within an object, Java has “private” “protected” and “public” qualifiers
    - private methods/fields can’t be called/used outside of the class in which they are defined
  - This is really a scope/visibility issue! Front-end during semantic analysis (type checking and so on), the compiler maintains this information in the symbol table for each class and enforces visibility rules.

Object representation

C++

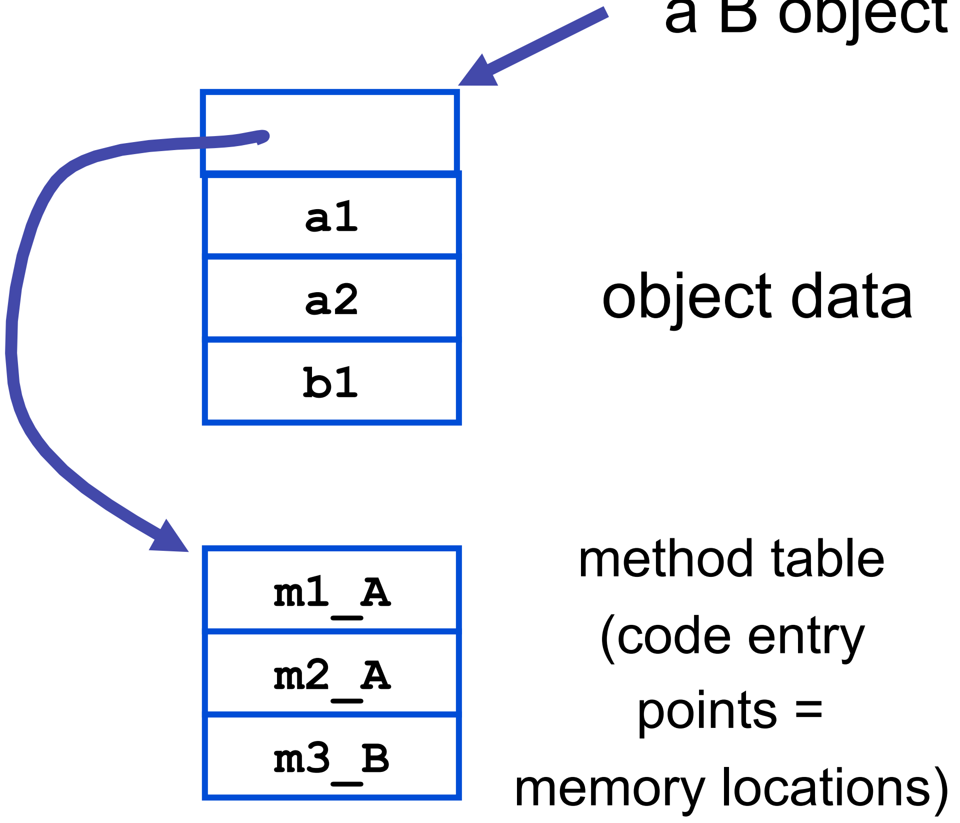
```
class A {
public:
    int a1, a2;

    void m1(int i) {
        a1 = i;
    }
    void m2(int i) {
        a2 = a1 + i;
    }
}
```



NB: a compiler typically generates methods with an extra argument representing the object (self) and used to access object data.

Inheritance (“pointer polymorphism”)

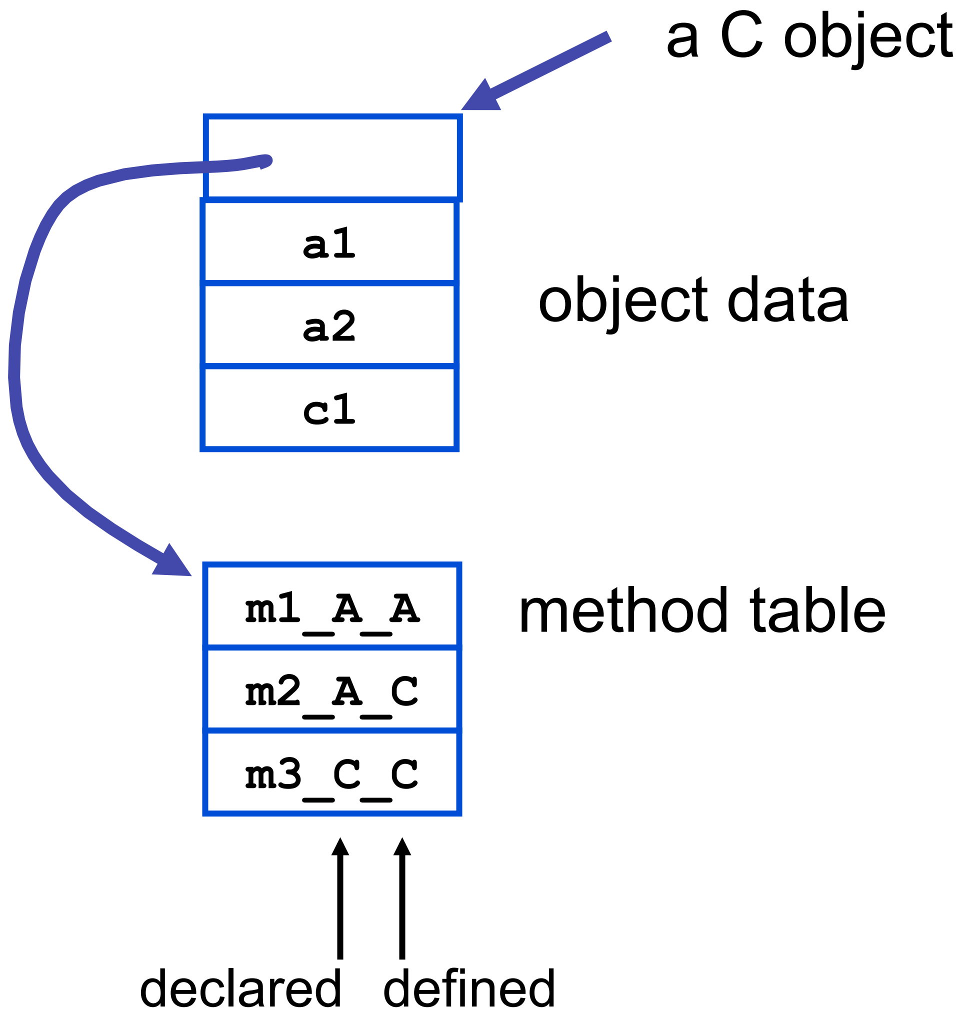
```
class B : public A {
public:
   int b1;

   void m3(void) {
      b1 = a1 + a2;
   }
}
```



Note that a pointer to a B object can be treated as if it were a pointer to an A object!

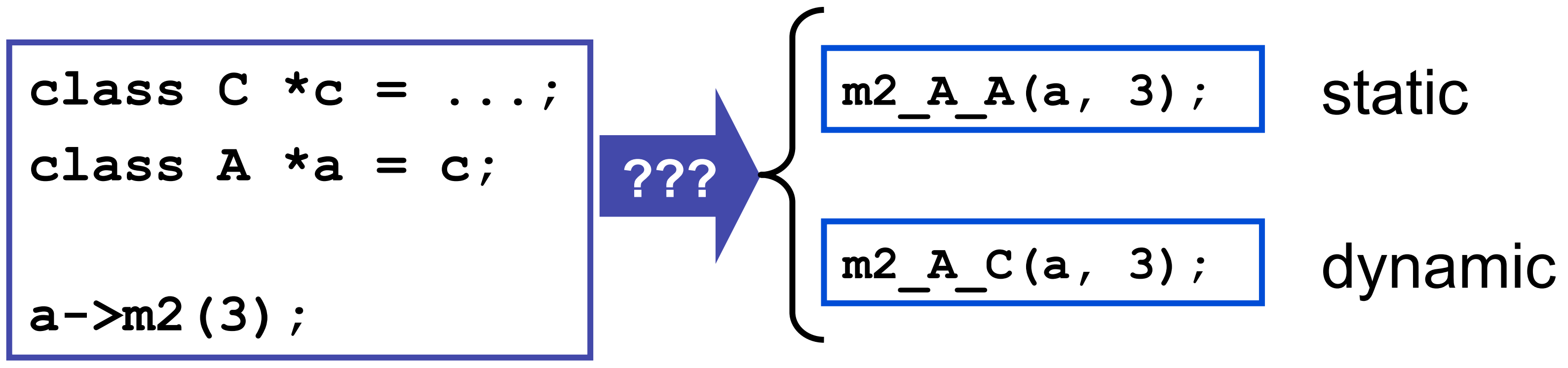
Method overriding

```
class C : public A {
public:
   int c1;

   void m3(void) {
      b1 = a1 + a2;
   }
   void m2(int i) {
      a2 = c1 + i;
   }
}
```



Static vs. Dynamic

- which method to invoke on overloaded polymorphic types?



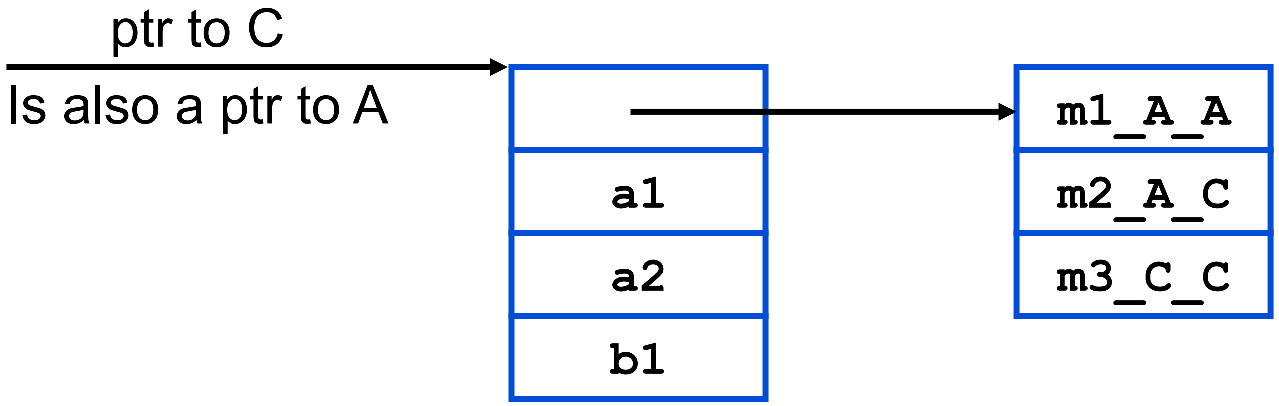
```
class C *c = ...;
class A *a = c;

a->m2(3);
```

```
m2_A_A(a, 3);
```

```
m2_A_C(a, 3);
```

Dynamic dispatch

- implementation: dispatch tables



```
class C *c = ...;
class A *a = c;

a->m2(3);
```

⇨

```
*(a->dispatch_table[1])(a, 3);
```

This implicitly uses some form of pointer subtyping

```
void m2(int i) {
      a2 = c1 + i;
}
```

```
void m2_A_C(class_A *this_A, int i) {
   class_C *this = convert_ptrA_to_ptrC(this_A);

   this->a2 = this->c1 + i;
}
```