Compiler Construction
Lent Term 2013
Lectures 7 & 8 (of 16)

- The heap and garbage collection
  - Reference counting
  - Mark and sweep
  - Copy collection
  - Generational collection
- Implementation techniques for Object-oriented constructs

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

In-Flight Course Correction

illustrated with	Lecture	Concepts
Slang.1 VRM.0 and VSM.0	1.	Overview
	2.	Simple lexical analysis, recursive descent parsing (thus "bad" syntax), and simple type checking
	3.	Targeting a Virtual Register Machine (VRM)
	4.	Targeting a Virtual Stack Machine (VSM) . Simple "peep hole" optimization
	5.	Block structure, simple functions, stack frames
	6.	Tuples, records, first-class functions. Heap allocation, closures
	7.	Memory Management ("garbage collection")
	8.	Object-oriented constructs.
	9.	Tuples, records, and other complex data types
	10.	Assorted topics : bootstrapping, exceptions, linking and loading
Slang.2 VRM.1 and VSM.1 (call stack and heap extensions)	11.	Targeting a VRM, targeting a VSM
	12.	Improving the generated code. Enhanced VM instruction sets, improved instruction selection, more "peep hole" optimization, simple register allocation for VRM
mosmllex and mosmlyacc	13.	Return to lexical analysis : application of Theory of Regular Languages and Finite Automata
	14.	Generating Recursive descent parsers
	15.	Beyond Recursive Descent Parsing I
	16.	Beyond Recursive Descent Parsing II

What is Garbage?

An object in the heap is “garbage” if it will not be used in any subsequent computation by the program.

In general, determining what is and is not garbage is **not decidable.**

Read Chapter 13 of Appel

Memory Management

- Every modern programming language allows programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Every modern language needs facilities for reclaiming and recycling the storage used by programs
- COST: It’s usually the most complex aspect of the run-time system for any modern language (Java, ML, Lisp, Scheme, Scala, …)

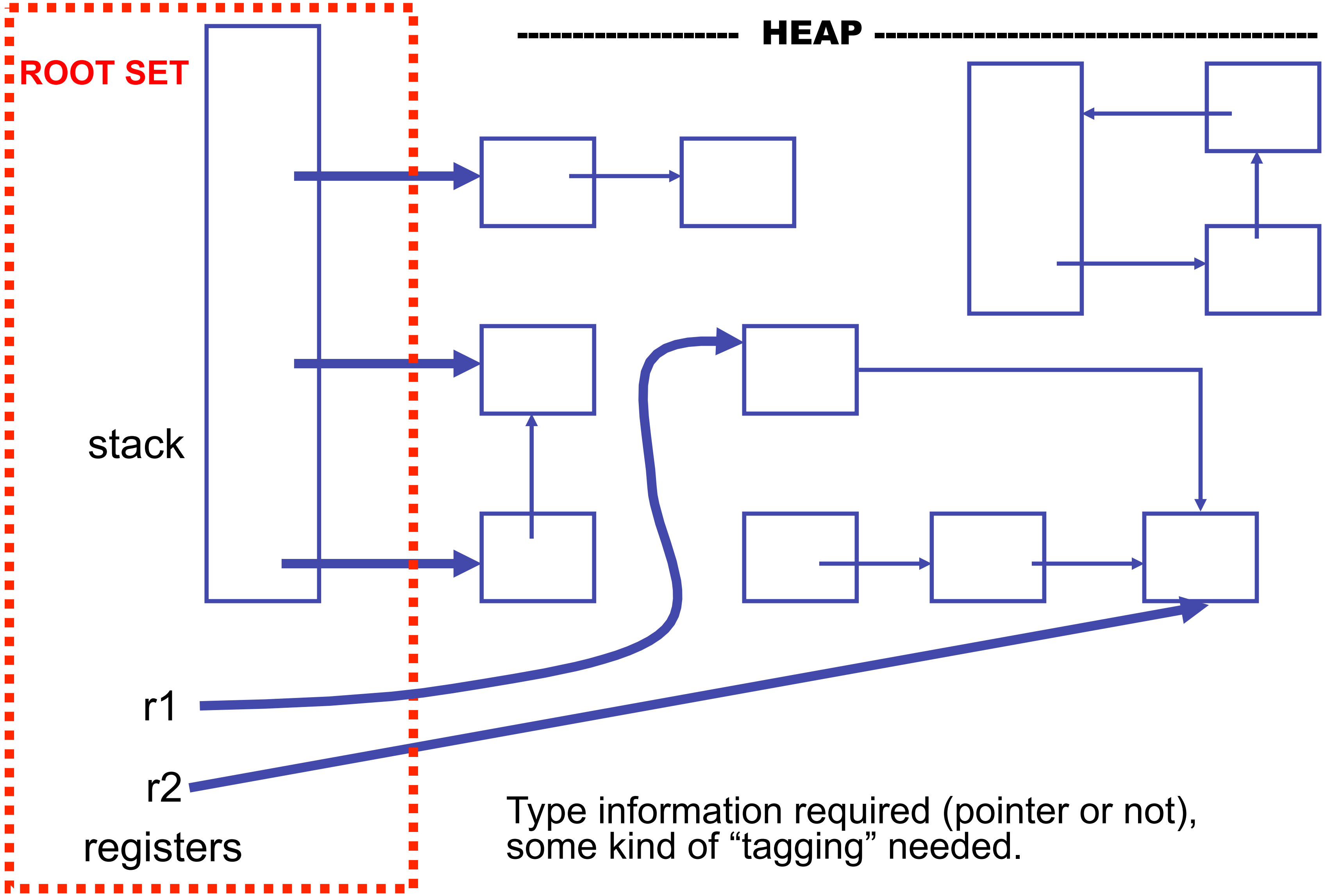
Solutions

- Ignore the problem
- Restrict programming language so the problem goes away! That is, use FORTRAN.
- Force programmer to worry about it (use **malloc** and **free** in C…)
- Automatic “garbage collection”
  - Reference Counting
  - Mark and Sweep
  - Copy Collection
  - Generational Collection
  - … there are other GC techniques…

Explicit MM

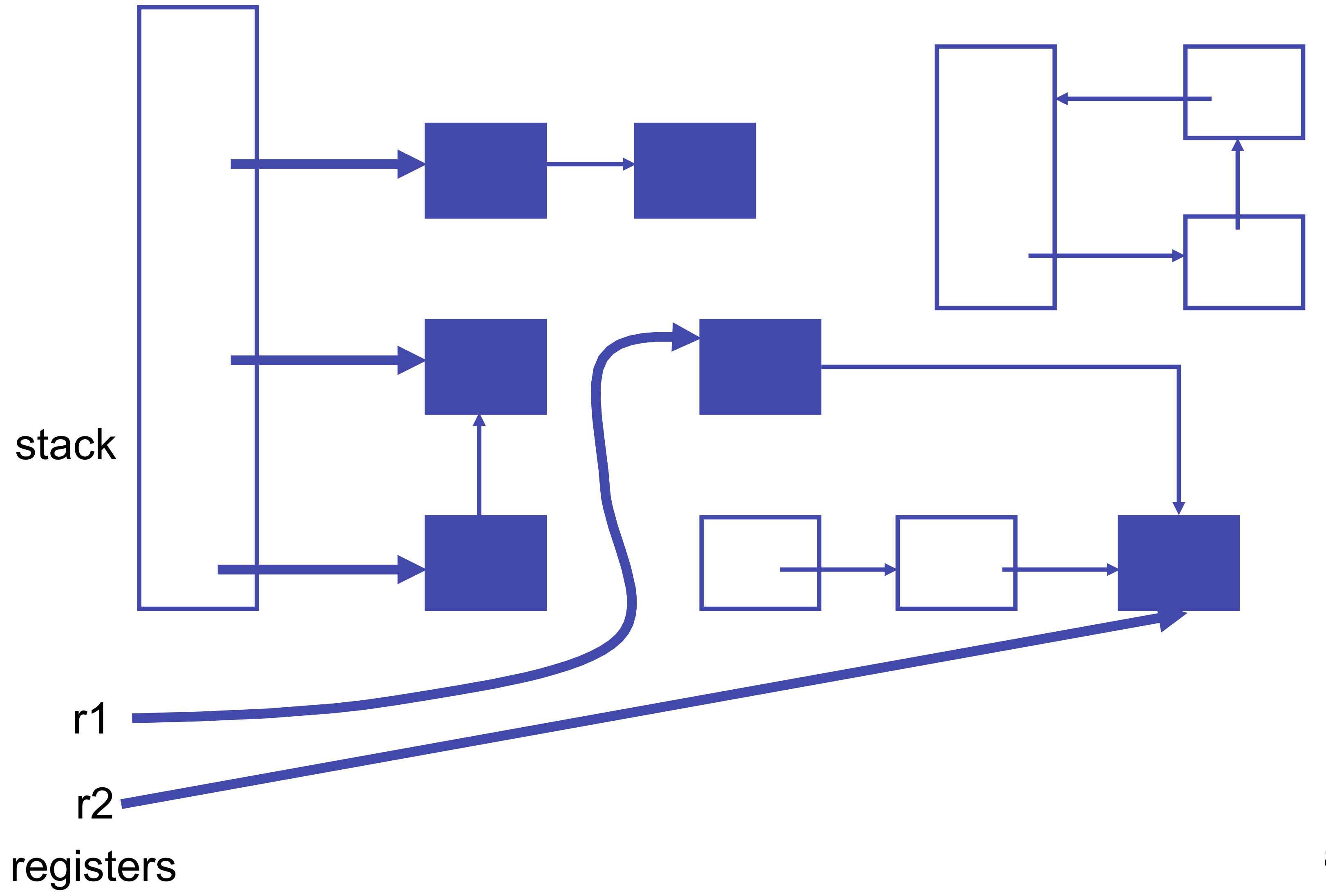
- User library manages memory; programmer decides when and where to allocate and de-allocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - Advantage: Gives programmer a lot of control.
  - Disadvantage: people too clever and make mistakes. Getting it right can be costly. And don't we want to automate-away tedium?
  - Advantage: With these procedures we can implement garbage collection for other languages ;-)

Automation is based on an approximation : if data cannot be reached from a root set then it is garbage

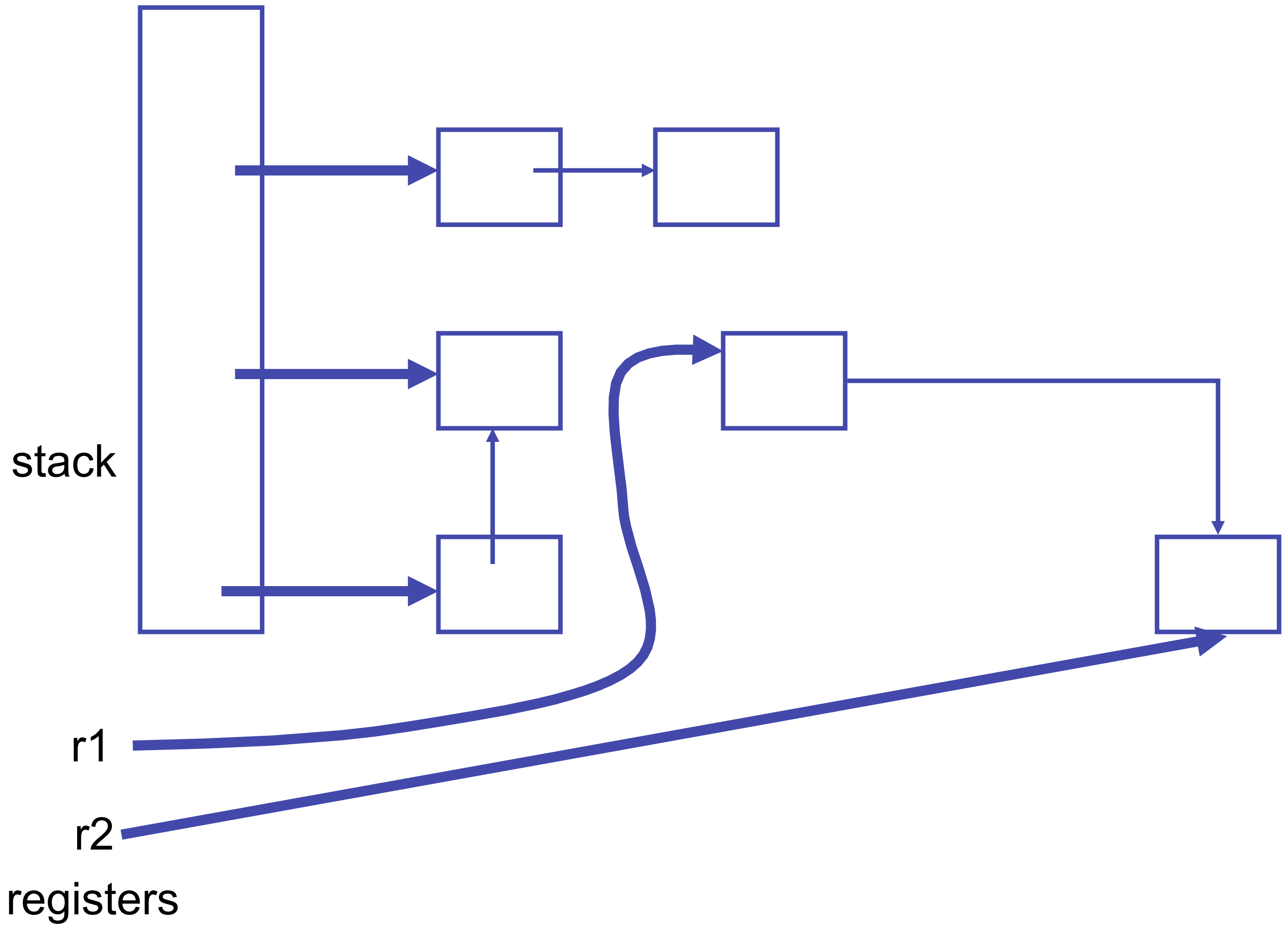


Type information required (pointer or not), some kind of “tagging” needed.

… Identify Cells Reachable From Root Set…



… reclaim unreachable cells, and repeat …

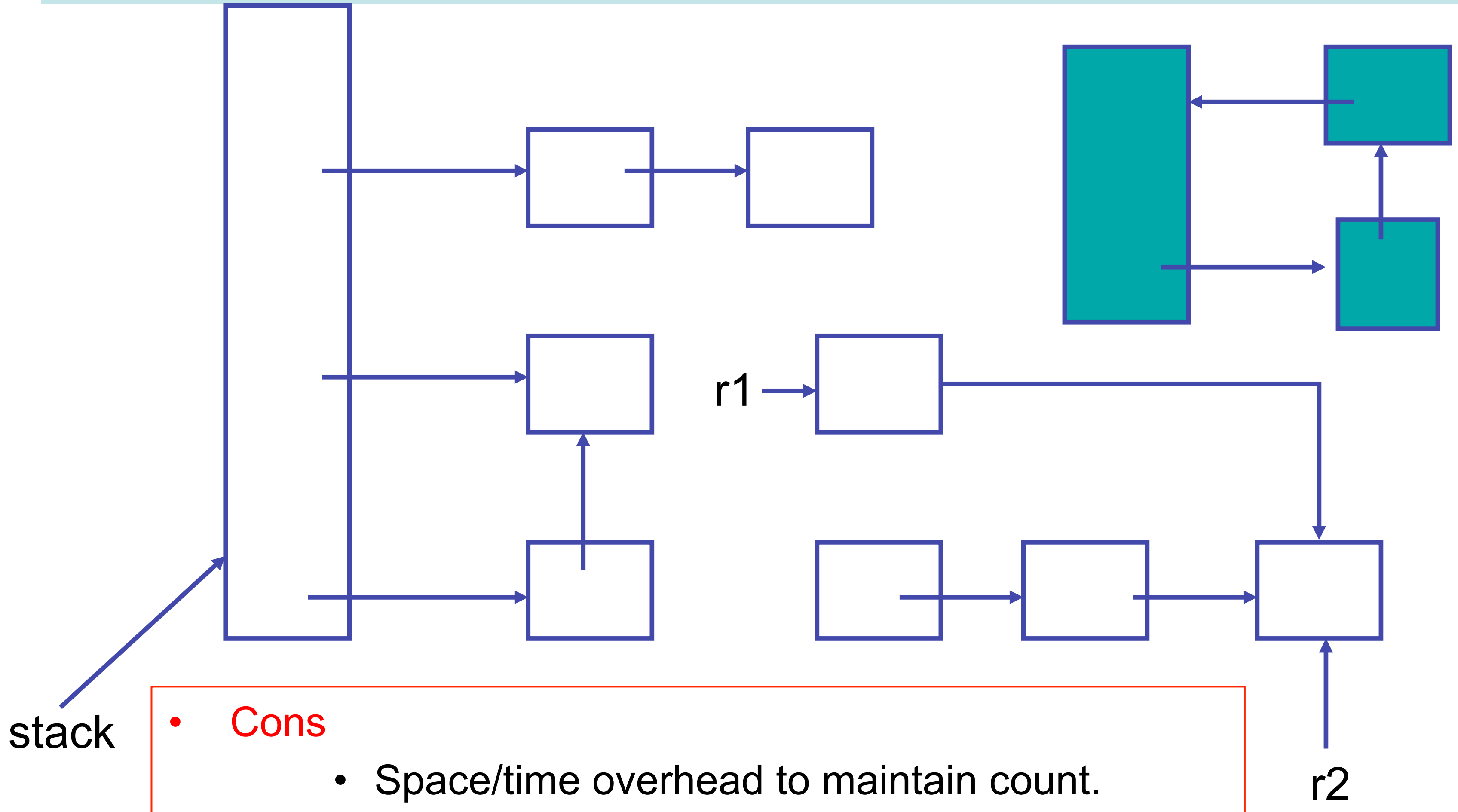


Reference Counting, basic idea:

- Keep track of the number of pointers to each object (the reference count).
- When Object is created, set count to 1.
- Every time a new pointer to the object is created, increment the count.
- Every time an existing pointer to an object is destroyed, decrement the count
- When the reference count goes to 0, the object is unreachable garbage

Clearly --- this can be VERY costly….

Reference counting can't detect cycles!



- Cons
  - Space/time overhead to maintain count.
  - Memory leakage when cycles in data.
- Pros
  - <u>Incremental</u> (no long pauses to collect…)

Mark and Sweep

- A two-phase algorithm
  - Mark phase: Depth first traversal of object graph from the roots to mark live data
  - Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list

Cost of Mark Sweep

- Cost of mark phase:
  - O(R) where R is the # of reachable words
  - Assume cost is c1 * R (c1 may be 10 instr’s)
- Cost of sweep phase:
  - O(H) where H is the # of words in entire heap
  - Assume cost is c2 * H (c2 may be 3 instr’s)
- Analysis
  - The “good” = each collection returns H - R words reclaimed
  - Amortized cost = time-collecting/amount-reclaimed
    - ((c1 * R) + (c2 * H)) / (H - R)
    - If R is close to H, then each collection reclaims little space..
  - R / H must be sufficiently small or GC cost is high.

    Could dynamically adjust. Say, if R / H is larger than .5, increase heap size

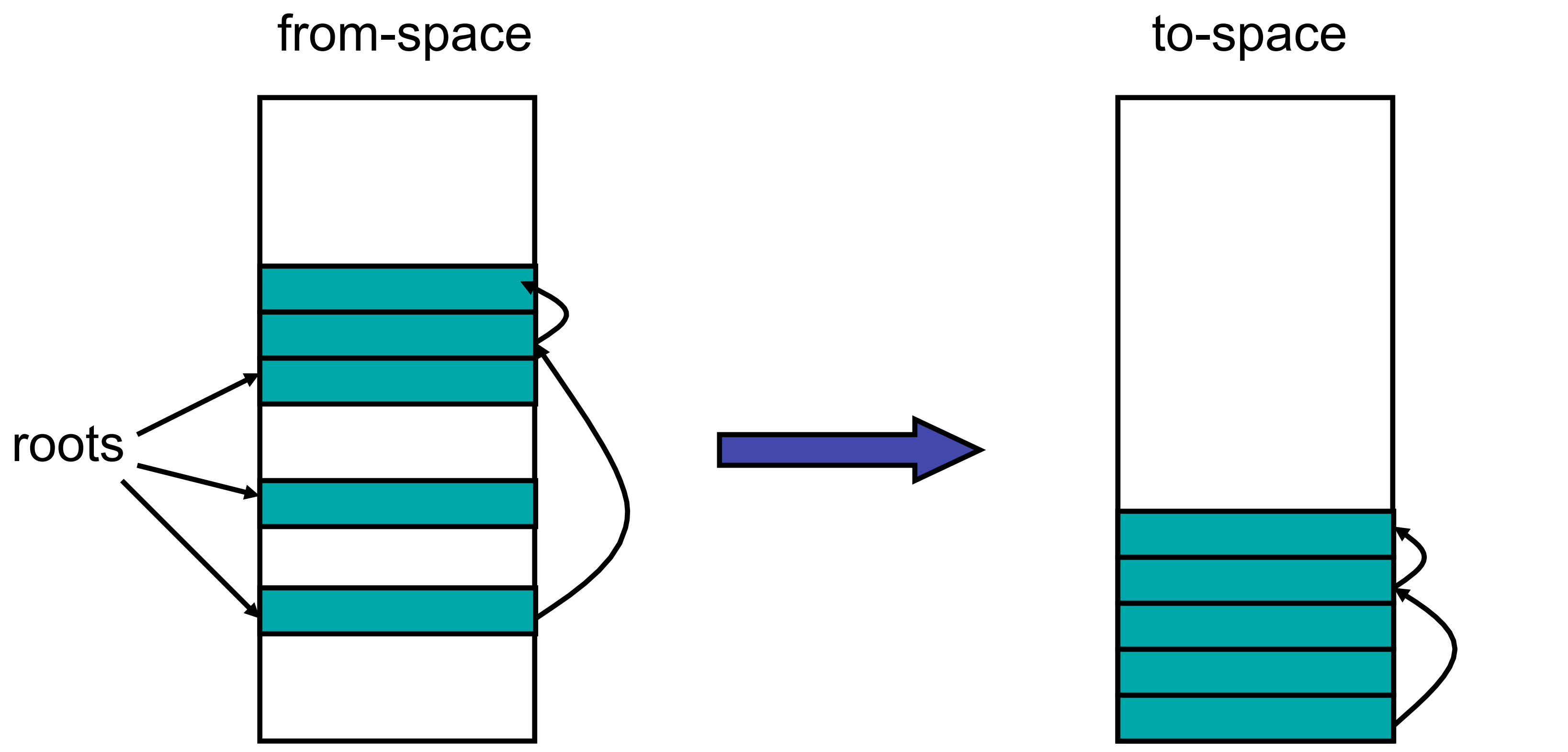
Other Problems

- Depth-first search is usually implemented as a recursive algorithm
  - Uses stack space proportional to the longest path in the graph of reachable objects
    - one activation record/node in the path
    - activation records are big
  - If the heap is one long linked list, the stack space used in the algorithm will be greater than the heap size!!
  - What do we do? Pointer reversal [See Appel]
- Fragmentation

Copying Collection

- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

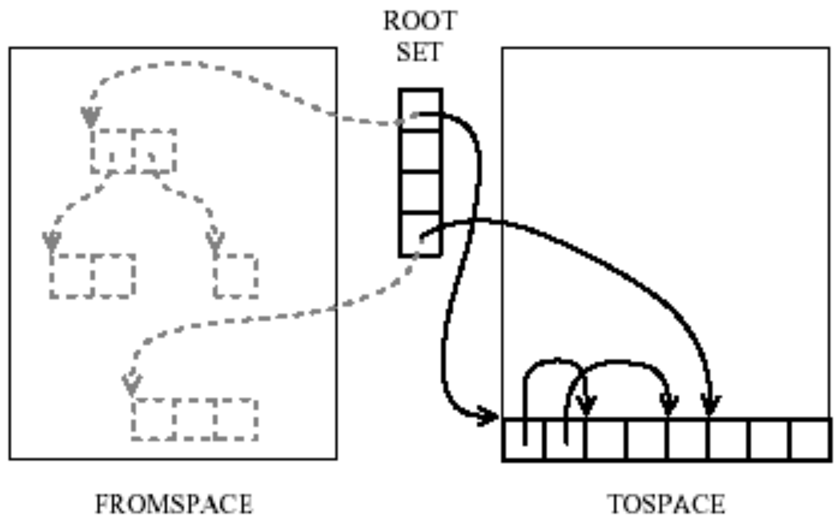
Copying Collection



Copying GC

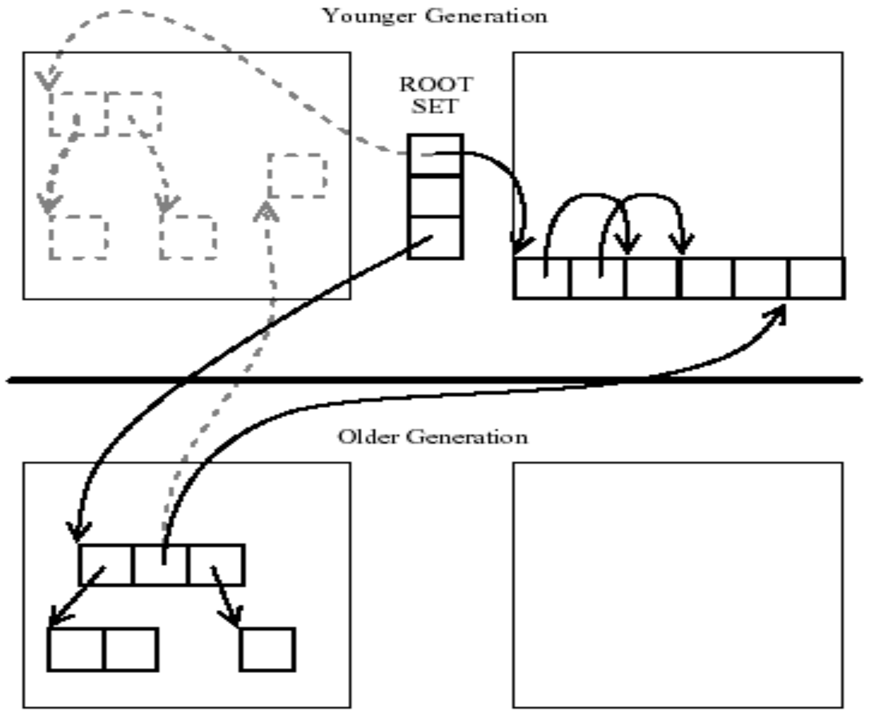
- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
- Cons
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps

OBSERVATION: for a copying garbage collector



- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It’s a inefficient that long-lived objects be copied over and over.

IDEA: Generational garbage collection



Segregate objects into multiple areas by age, and collect areas containing older objects less often than the younger ones.

Other issues…

- When do we promote objects from young generation to old generation
  - Usually after an object survives a collection, it will be promoted
- Need to keep track of older objects pointing to newer ones!
- How big should the generations be?
  - Appel says each should be exponentially larger than the last
- When do we collect the old generation?
  - After several minor collections, we do a major collection
- Sometimes different GC algorithms are used for the new and older generations.
  - Why? Because the have different characteristics
  - Copying collection for the new
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - Mark-sweep for the old

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.

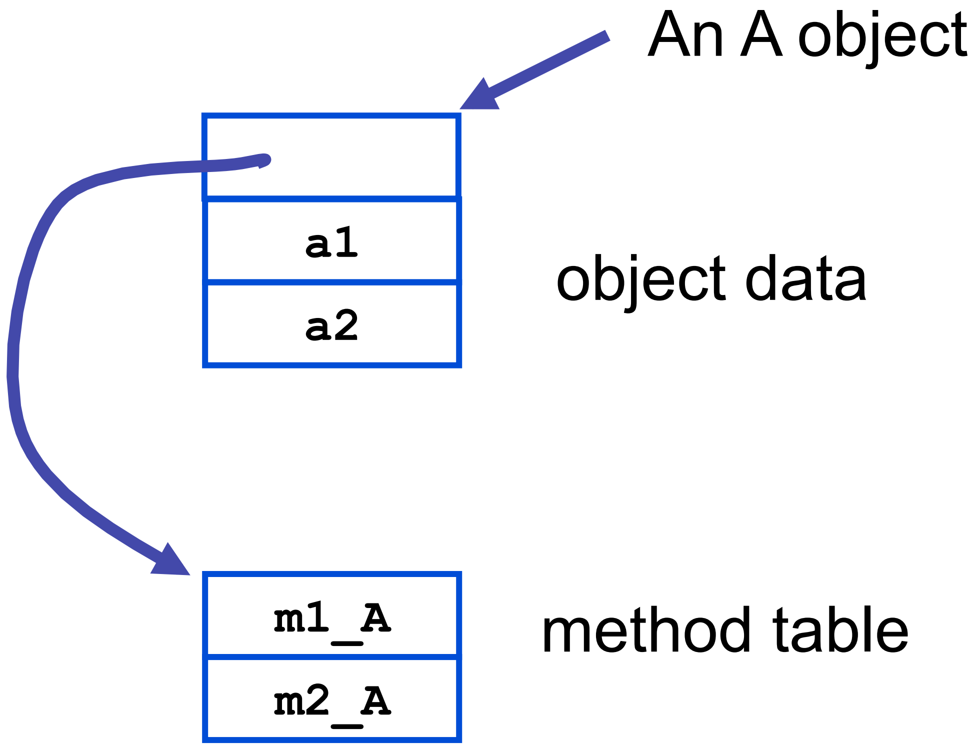
Static & Dynamic Methods

- The result of compiling a method is some machine code located at a particular address
  - at a method invocation point, we need to figure out what code location to jump to
- Java has static & dynamic methods
  - to resolve static method calls, we look at the static type of the calling object
  - to resolve dynamic method calls, we need the dynamic type of the calling object

Object representation

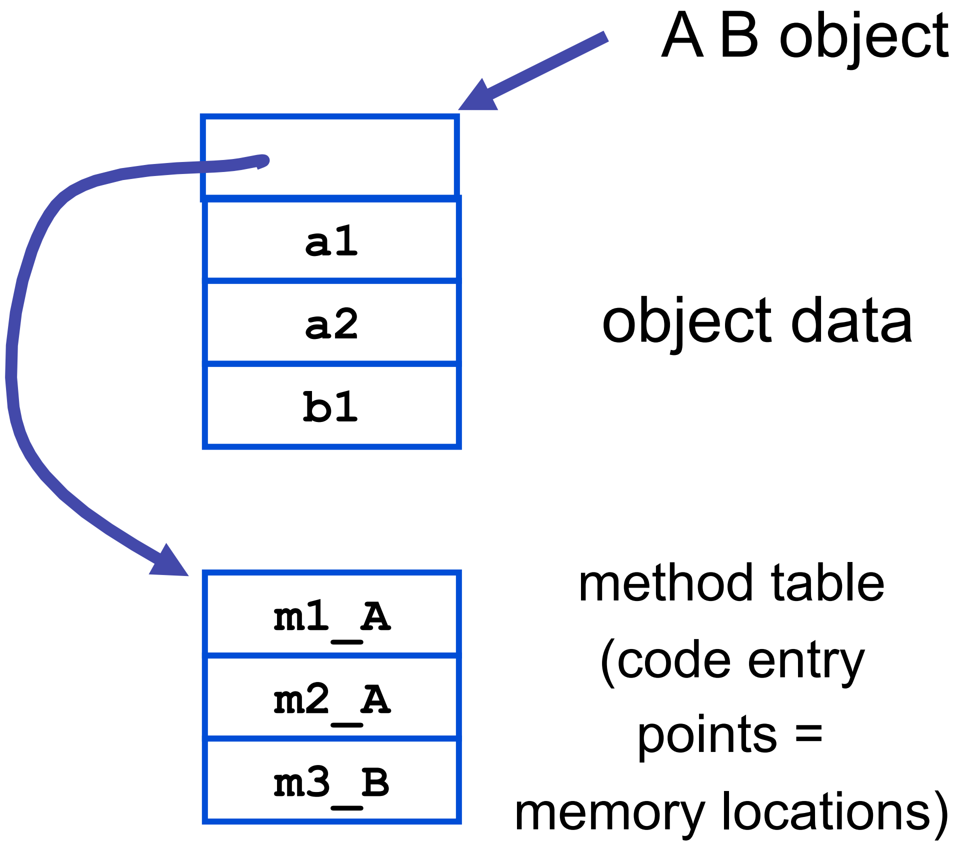
C++

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

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



Inheritance (“pointer polymorphism”)

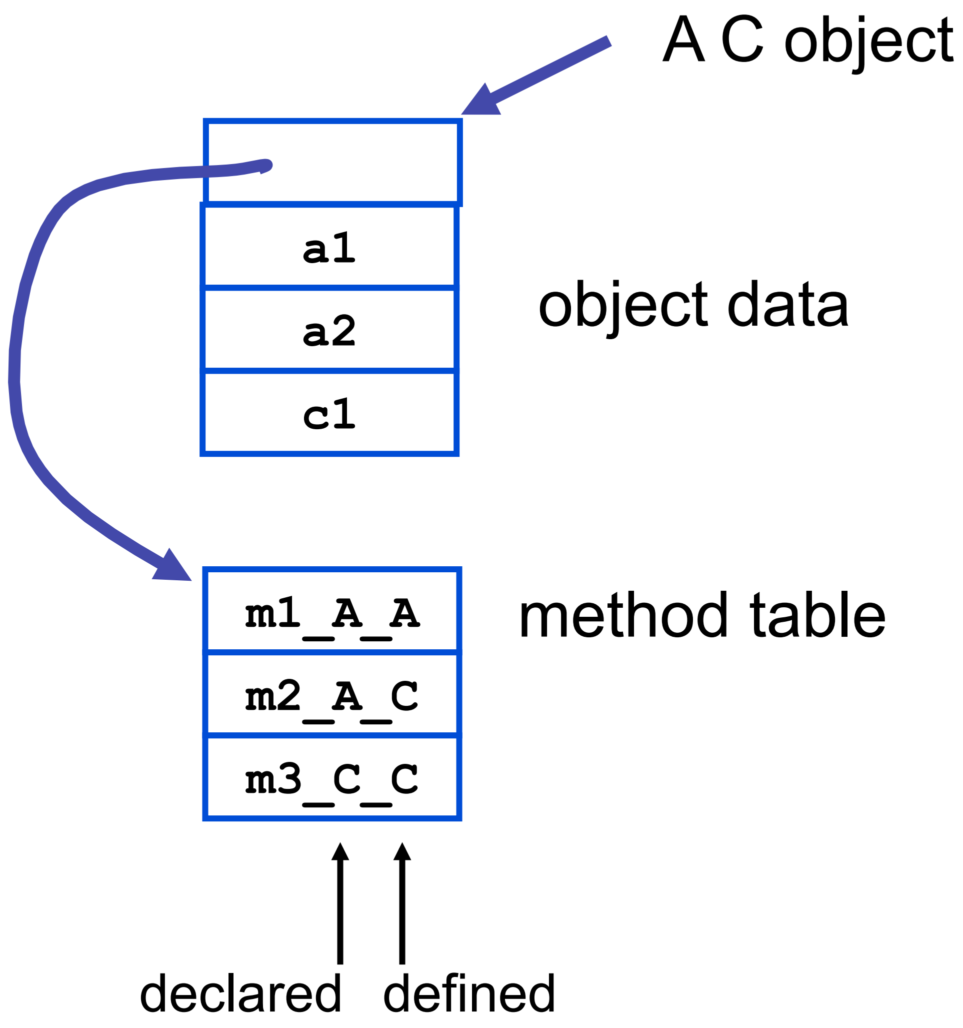
```
class B : public A {
public:
   int b1;

   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;
   }
}
```



Example

```
abstract class Shape {
   boolean IsShape() {return true;}
   boolean IsRectangle() {return false;}
   boolean IsSquare() {return false;}
   abstract double SurfaceArea();
}
class Rectangle extends Shape {
   double SurfaceArea { ... }
   boolean IsRectangle() {return true;}
}
class Squrae extends Rectangle {
   boolean IsSquare() {return true;}
}
```

Method table for **`Rectangle`**

`IsShape_Shape_Shape`
`IsRectangle_Shape_Rectangle`
`IsSquare_Shape_Shape`
`SurfaceArea_Shape_Rectangle`

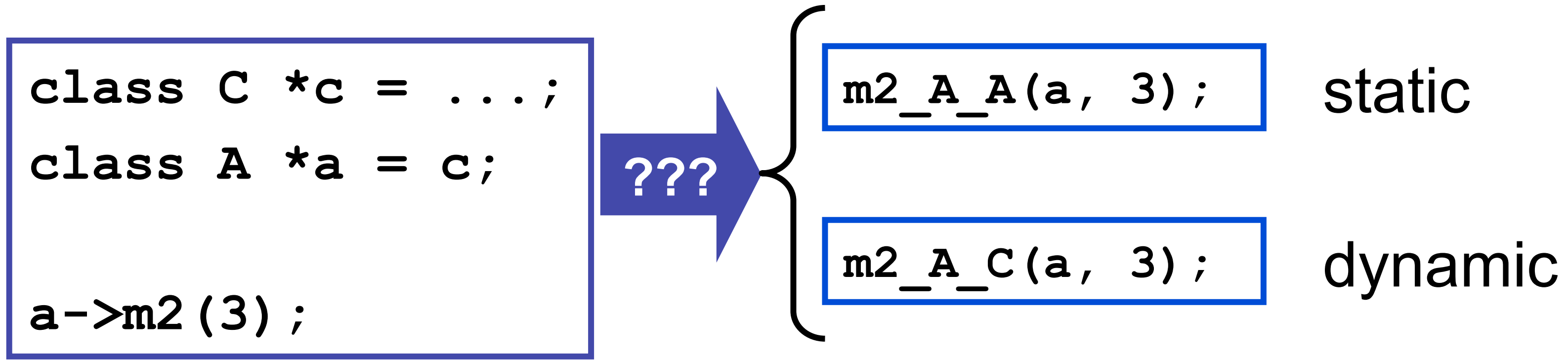
Method table for **`Square`**

`IsShape_Shape_Shape`
`IsRectangle_Shape_Rectangle`
`IsSquare_Shape_Square`
`SurfaceArea_Shape_Rectangle`

The compiler needs to us a SYMBOL TABLE to keep track of object types and relationships

Static vs. Dynamic

- which method to invoke on overloaded polymorphic types?



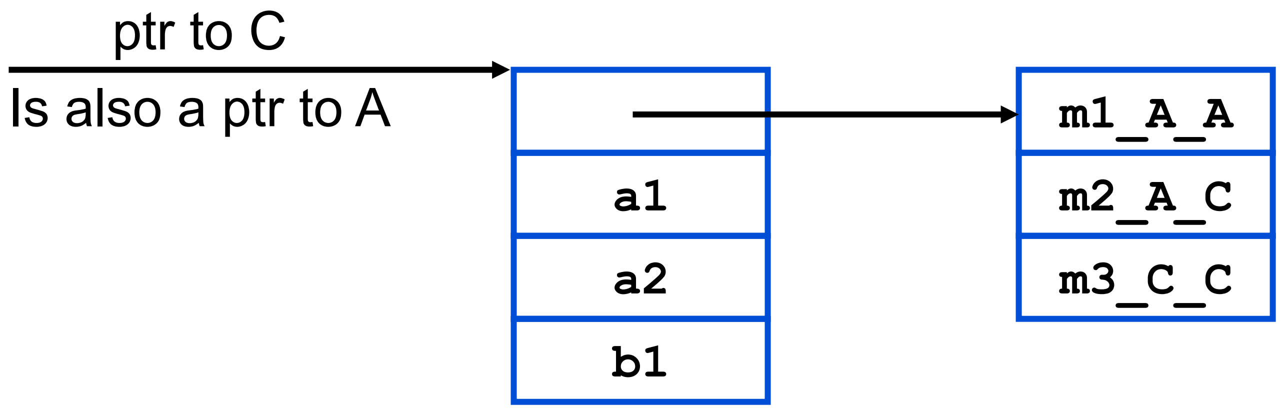
```
class C *c = ...;
class A *a = c;

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);
```

Dynamic typing:implementation requires 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;
}
```

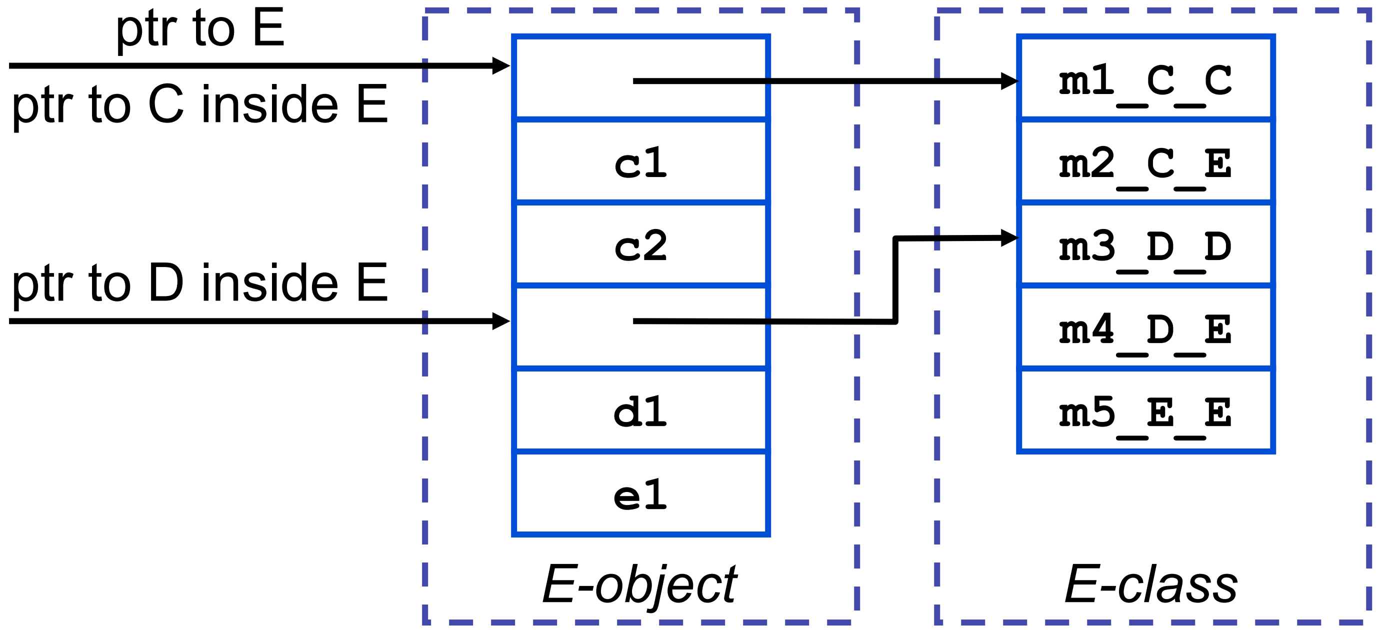
Multiple inheritance

```
class C {
public:
    int c1, c2;
    void m1() {...}
    void m2() {...}
}
```

```
class D {
public:
    int d1;
    void m3() {...}
    void m4() {...}
}
```

```
class E : public C, D {
public:
    int e1;
    void m2() {...}
    void m4() {...}
    void m5() {...}
}
```

Multiple inheritance



supertyping

```
convert_ptrE_to_ptrC(e)  ≈ e
convert_ptrE_to_ptrD(e)  ≈ e + sizeof(Class_C)
```

subtyping

```
convert_ptrC_to_ptrE(c)  ≈ c
convert_ptrD_to_ptrE(d)  ≈ d - sizeof(Class_C)
```

given an object `e` of class `E`

`e.m1()`	`(*(e->dispatch_table[0]))((Class_C *) e)`

`e.m3()`	`(*(e->dispatch_table[2]))(`
	`    (class_D *)((char *)e + sizeof(Class_C)))`

`e.m4()`	`(*(e->dispatch_table[3]))(`
	`    (class_D *)((char *)e + sizeof(Class_C)))`

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.