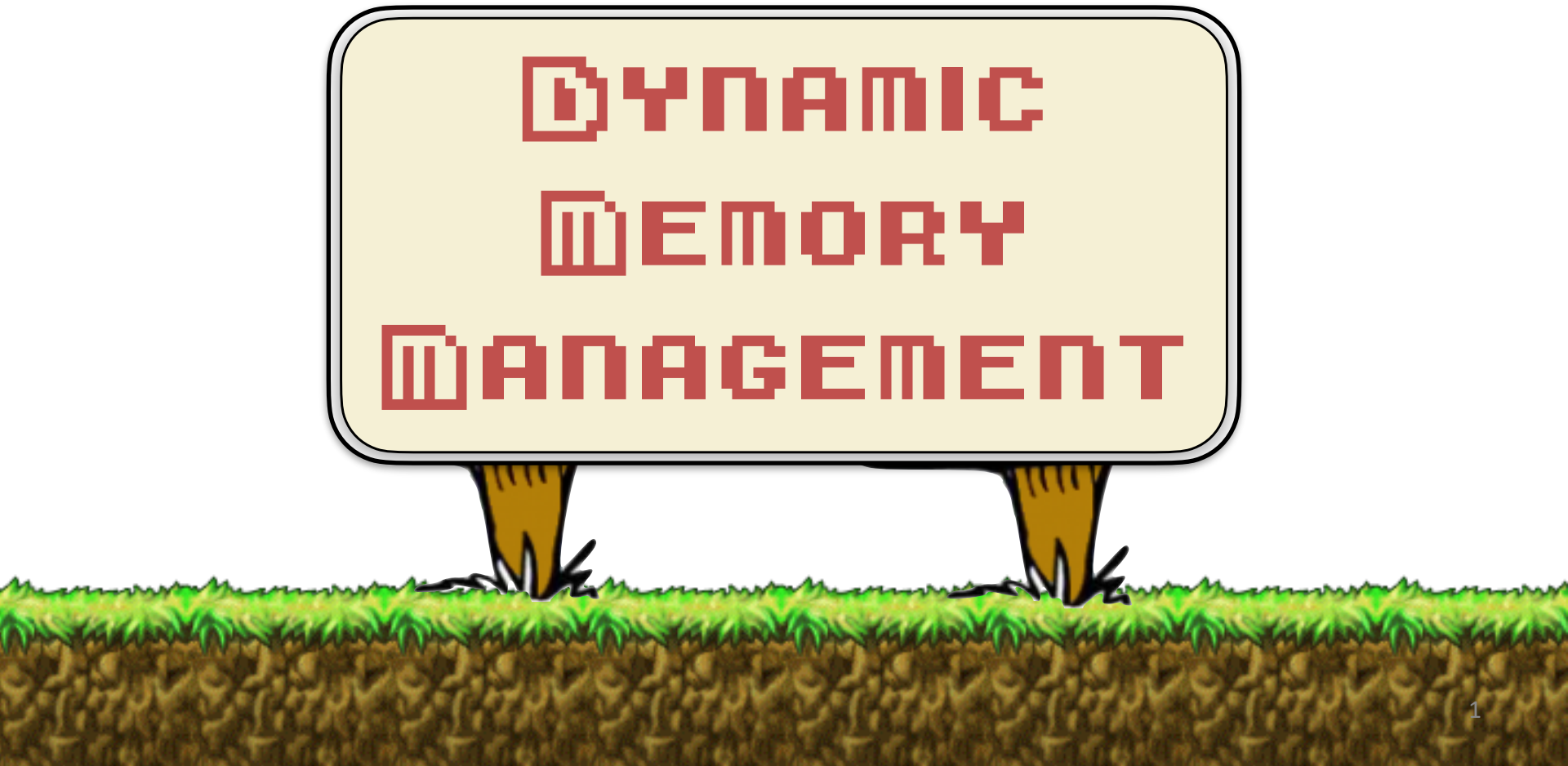
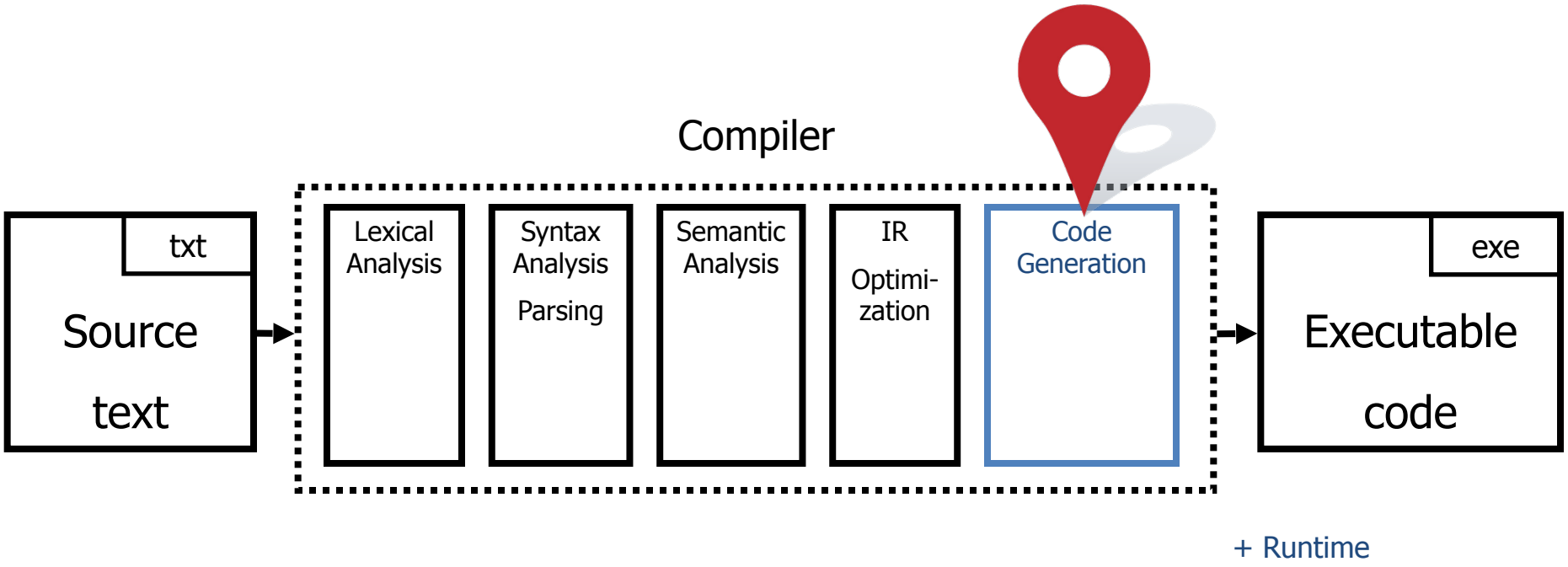
Theory of Compilation

Lecture 10



You are here


Compiler
txt
Source text
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
IR
Optimi-zation
Code Generation
exe
Executable code
+ Runtime

Representing Data at Runtime

- Source language types
  - int, boolean, string, **object types (incl. structs & arrays)**
- Target language types
  - Single bytes, integers, address representation
- Compiler should map source types to some combination of target types
  - Implement source types using target types

Pointer Types

- Represent addresses of source language data structures
- Usually implemented as an unsigned integer
- Pointer dereferencing — retrieves value at address
- Pointer arithmetic — access elements of array/buffer

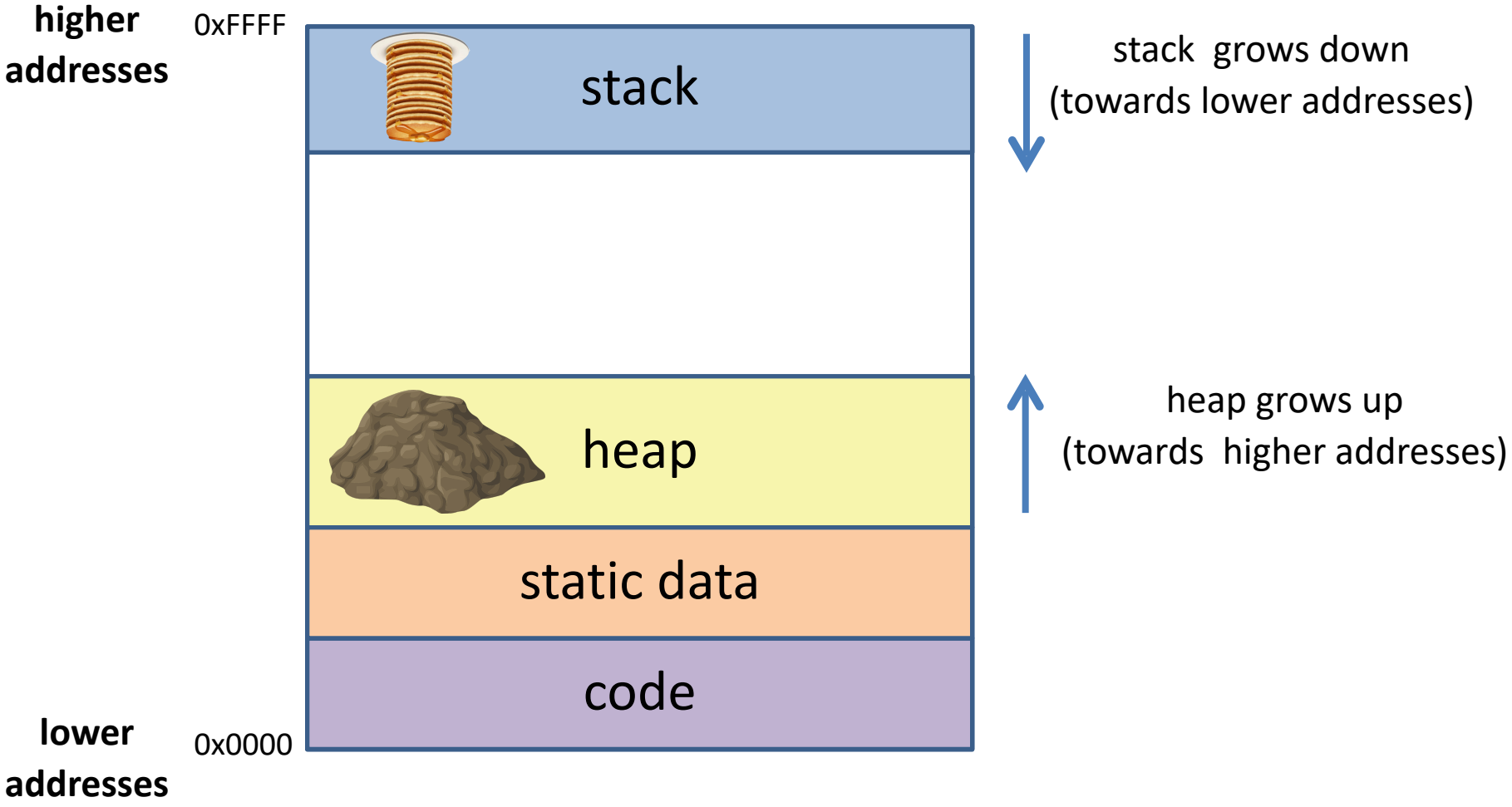
- May produce an error
  - Null pointer dereference (p = 0)
  - Segmentation violation (p outside heap area)

Where do we allocate data?

- Activation records — on the stack
  - Lifetime of allocated data limited by procedure lifetime
  - Stack frame deallocated (popped) when procedure return
  - (that was the whole idea)

- **Dynamic memory allocation — on the heap**

Memory Layout



Allocating memory

- In C — malloc

```
void *malloc(size_t size)
```

- Why does malloc return void* ?
  - It just allocates a chunk of memory, without regard to its contents
- How does malloc guarantee alignment?
  - After all, you don't know what type it is allocating for
  - It has to align for the largest primitive type
  - In practice optimized for 8 byte alignment (glibc-2.17)

Memory Management

Manual Deallocation

Automatic Deallocation

Manual Memory Management

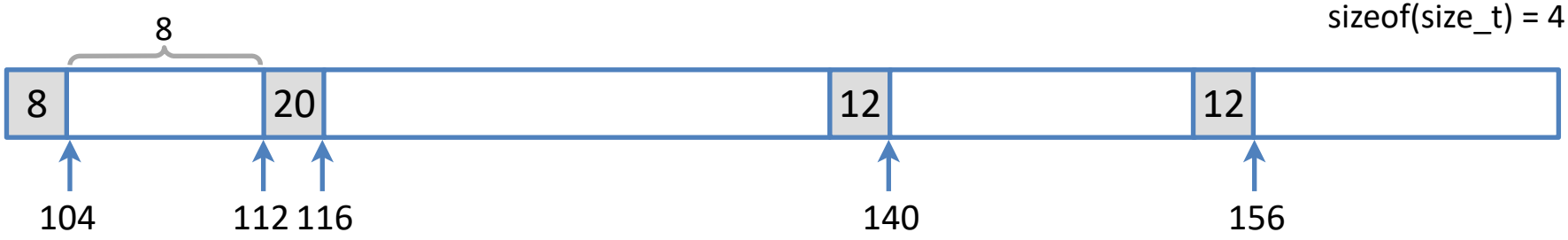
- malloc (new)
- free (delete)

```
a = malloc(size);
   ⋮
// do something with 'a'
   ⋮
free(a);
```

size = *(size_t*)(a – sizeof(size_t))

Implicit Block List

- Idea: the heap is a contiguous area in memory. Split it into a sequence of logical blocks
  - We can always find the "next block" using the current block's size



- Problem: how will we know which blocks are allocated and which are free?

Implicit Block List

- Idea: the heap is a contiguous area in memory. Split it into a sequence of logical blocks
  - We can always find the "next block" using the current block's size

sz = sz

sz = sz | 0x1

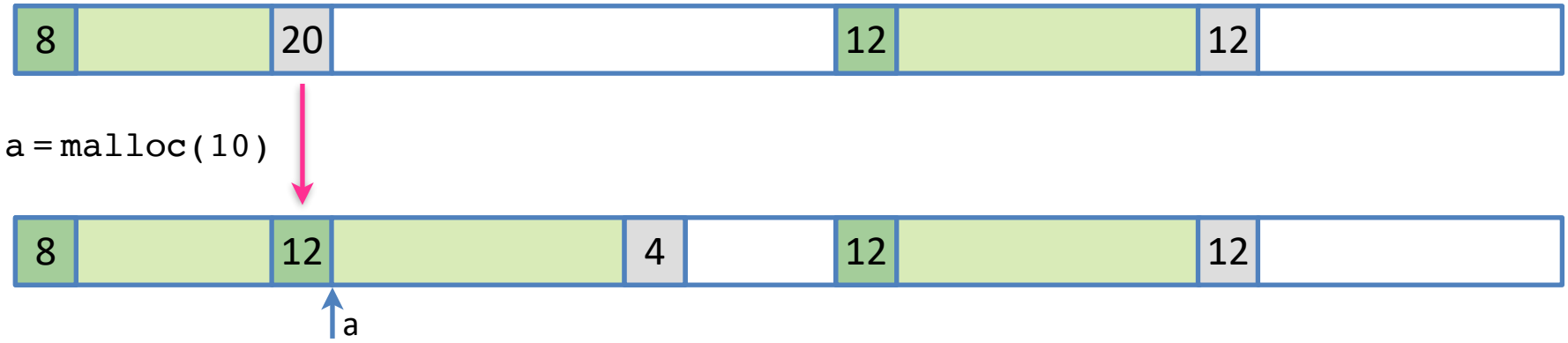
Trick: use the LSB of the size field to indicate whether the block is allocated.

(don't worry; sizes must be multiples of word size due to alignment)

size = *(size_t*)(a – sizeof(size_t)) & ~0x1

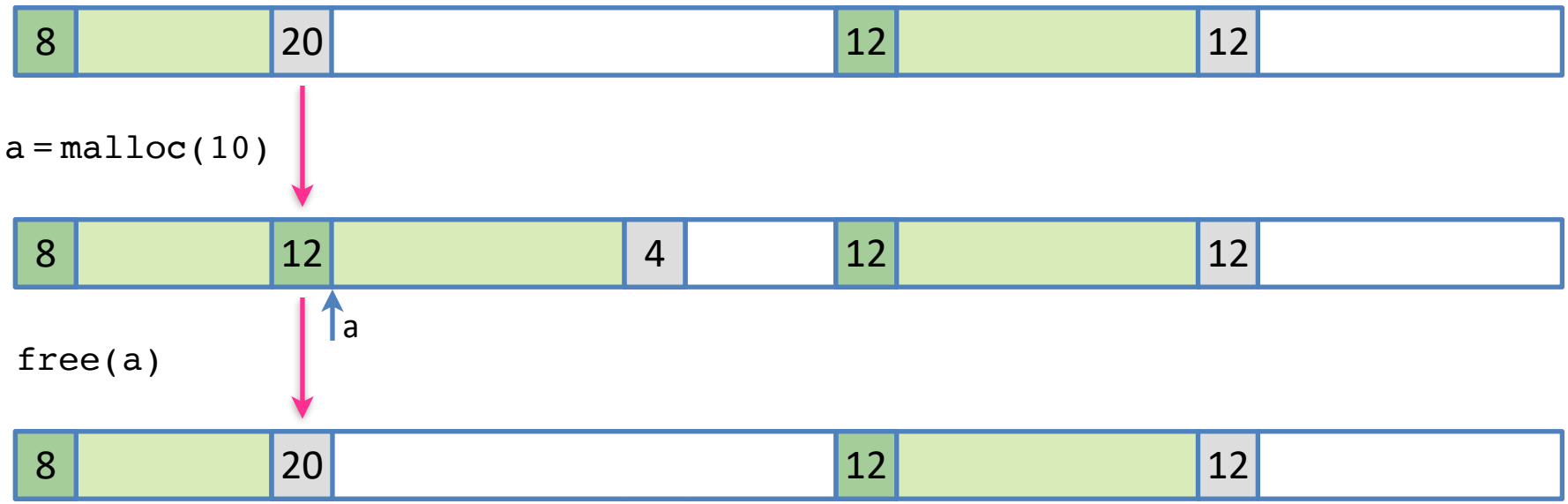
Implicit Block List

- malloc(*n*): find a block that is large enough for the requested size *n*
  - May have to split the block to be able to use the unallocated portion of it



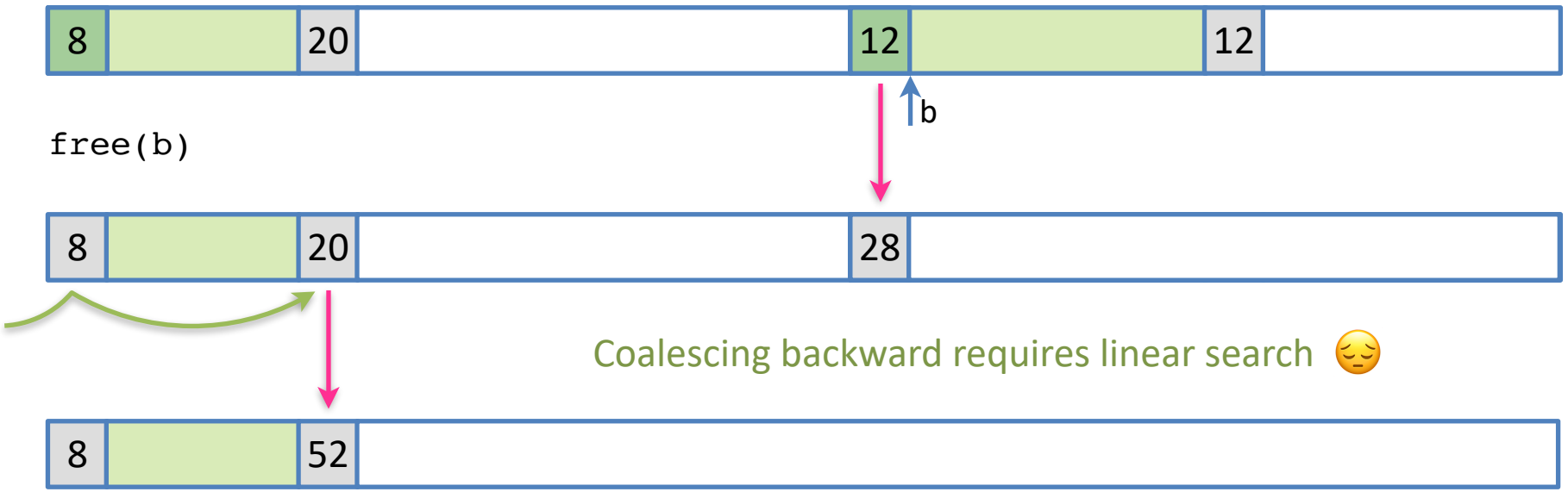
Implicit Block List

- free(*ptr*): straightforward — mark block as free (*i.e.* clear LSB of size field)
  - Would be beneficial to *coalesce* adjacent free blocks though



Implicit Block List

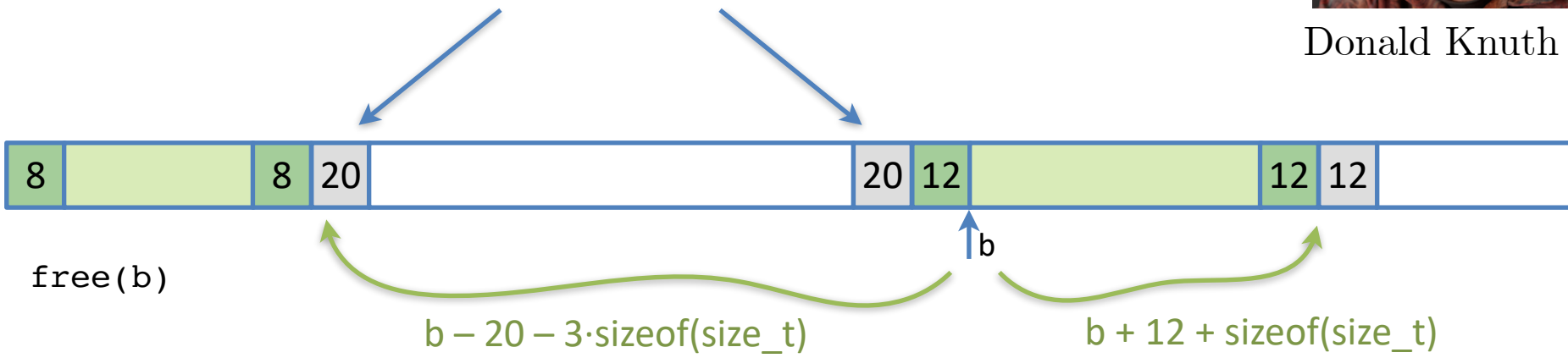
- free(*ptr*): straightforward — mark block as free (*i.e.* clear LSB of size field)
  - Would be beneficial to *coalesce* adjacent free blocks though



Implicit Block List: Bidirectional

- *Boundary tags*
  - Store size both before **and** after allocated area

Donald Knuth



⊕ Coalescing (both directions) in constant time

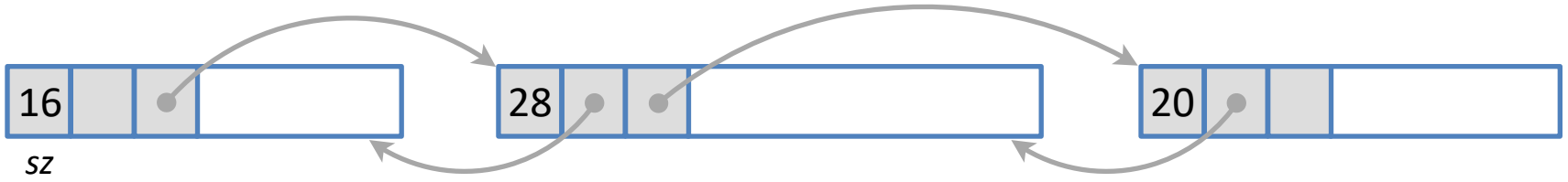
⊖ 2× bookkeeping overhead

this is true both with and without boundary tags

⊖ Allocation = linear time in **total number of blocks**

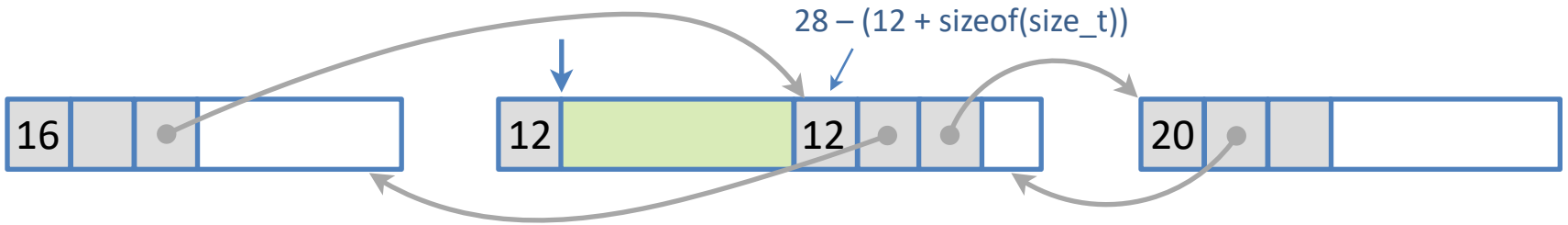
Free-list

- Idea: maintain a list of only the **free** blocks in the heap using a doubly-linked list.



(+) When allocating, only need to traverse the free blocks
(good when most of the heap is allocated)

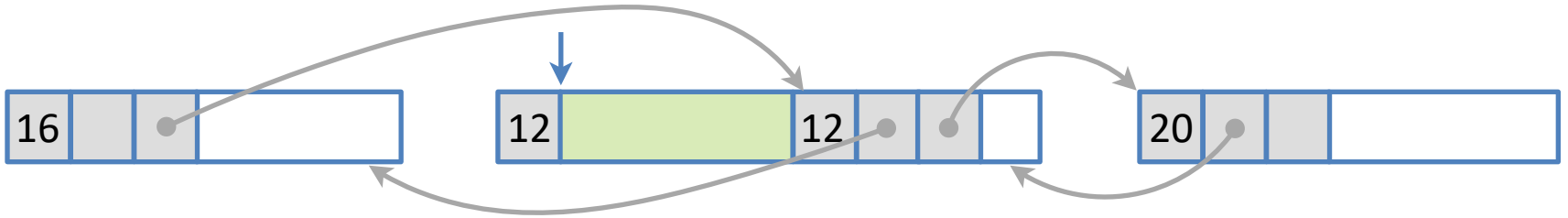
```
a = malloc(10);
```



(+) Pointers only needed in free blocks — **no extra overhead**

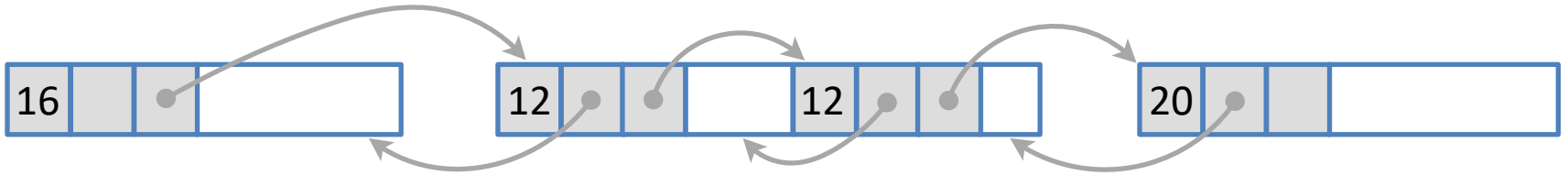
Free-list

- free(*ptr*): put block back in free-list and merge with adjacent block(s) if possible



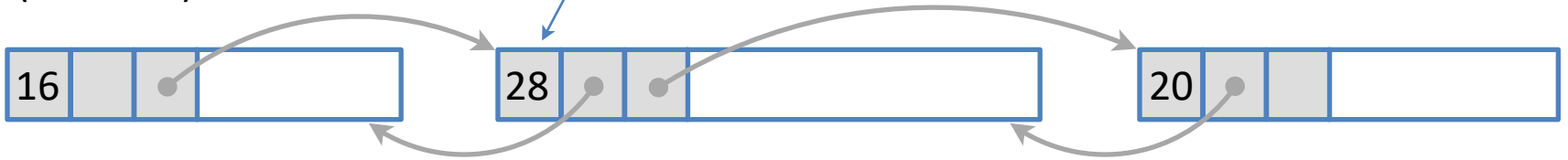
⊖ Coalesce = linear time (in **# of free blocks**)

```
free(a);
```



(coalesce)

12 + 12 + sizeof(size_t)



When to free() ?

- Free too late — waste memory (memory leak)
- Free too early — dangling pointers / crashes
- Free twice — error
- Try to free stack-allocated objects — also error

The burden is on the **programmer**.

When to free() ?

```
a = malloc(…) ;
b = a;
free (a); 🤔
c = malloc (…);
if (b == c)
    printf(“unexpected equality!”);
```

Cannot free an **object** if it has a reference with a **future use** (through *some* variable)

Automatic Memory Management

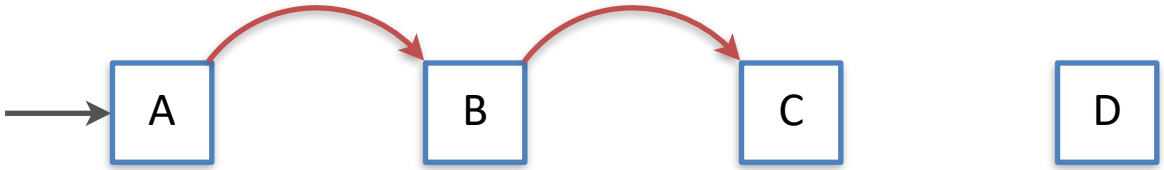
- Idea: automatically free memory when it is no longer needed

- Pervasive in OO languages such as Java, C#
  - but also functional languages (Haskell, OCaml)

Challenge: how can we know an object will **not be** used, since we cannot tell the future??

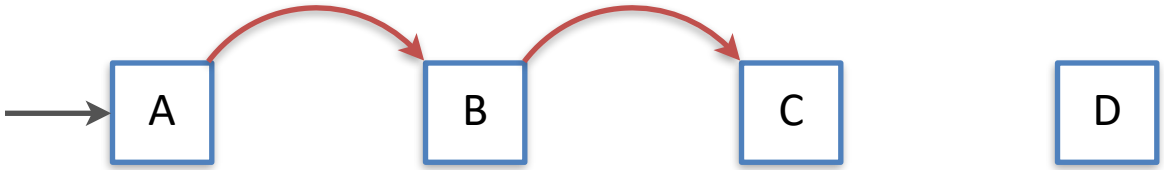
Garbage Collection

- **Approximate** reasoning about object liveness
- Use *reachability* to approximate liveness
  - an object is *reachable* if its address can be obtained starting from pointers in variables (local and global)
- **Assume reachable objects are live**
  - non-reachable objects are (definitely) dead



Garbage Collection – Classical Techniques

- Reference counting
- Mark and sweep
- Copying

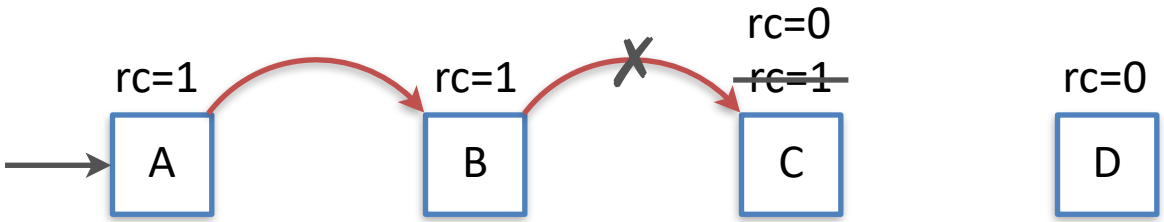


GC using Reference Counting

- Add a reference-count field to every object
  - tracks how many pointers refer to it

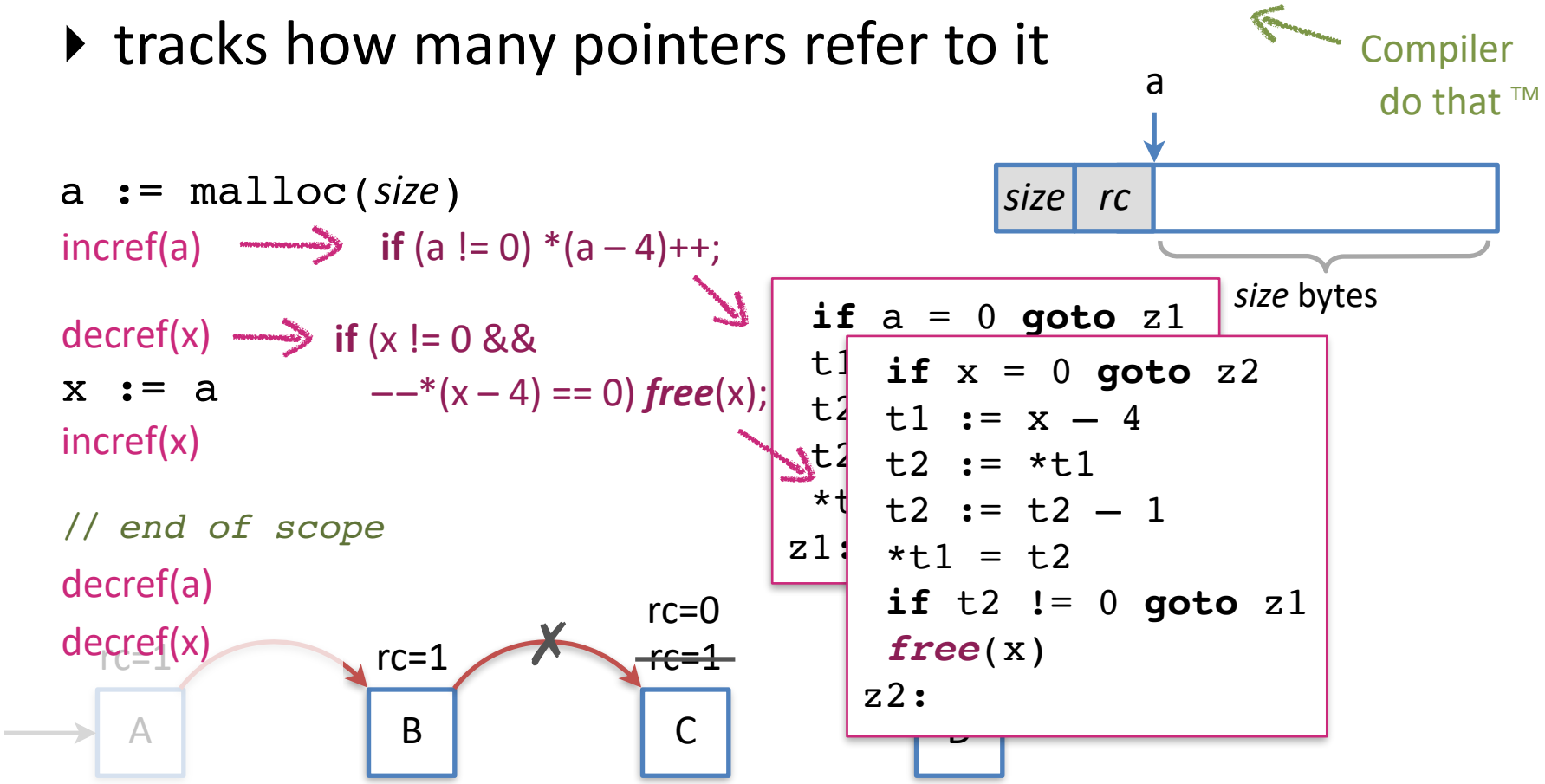
Compiler do that ™

- when (refcnt==0), the object is unreachable
  - not reachable ⇒ dead
  - can be collected (deallocated)



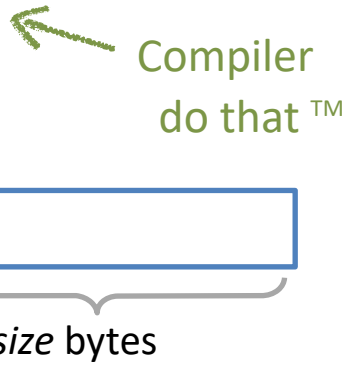
GC using Reference Counting

- Add a reference-count field to every object
  - tracks how many pointers refer to it



GC using Reference Counting

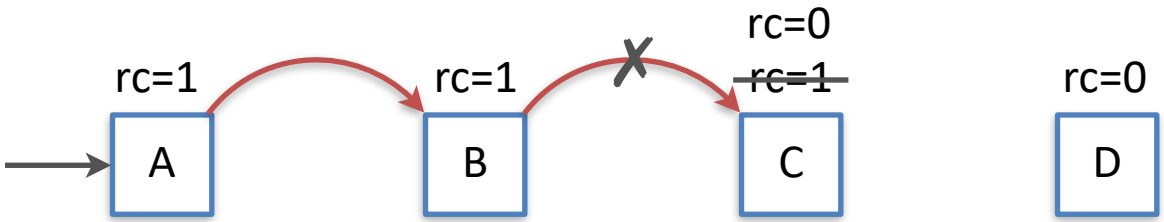
- Add a reference-count field to every object
  - tracks how many pointers refer to it




```
decref(y.ptr)
y.ptr := x
incref(y.ptr)
```

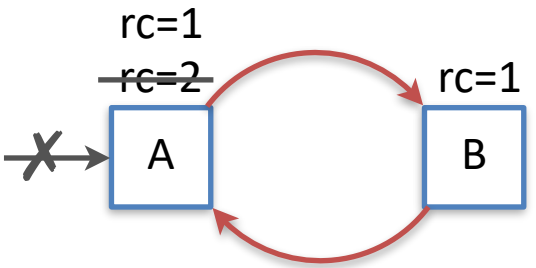
```
struct Y {
  int val;
  A *ptr;
}
```

```
// when freeing instance of type Y
decref(y.ptr)
```



The Problem — Cycles!

- Cannot identify non-reachable cycles
  - reference counts for nodes on the cycle will never decrement to 0
- several approaches for dealing with cycles
  - ignore
  - weak references
  - periodically invoke a tracing algorithm to collect cycles

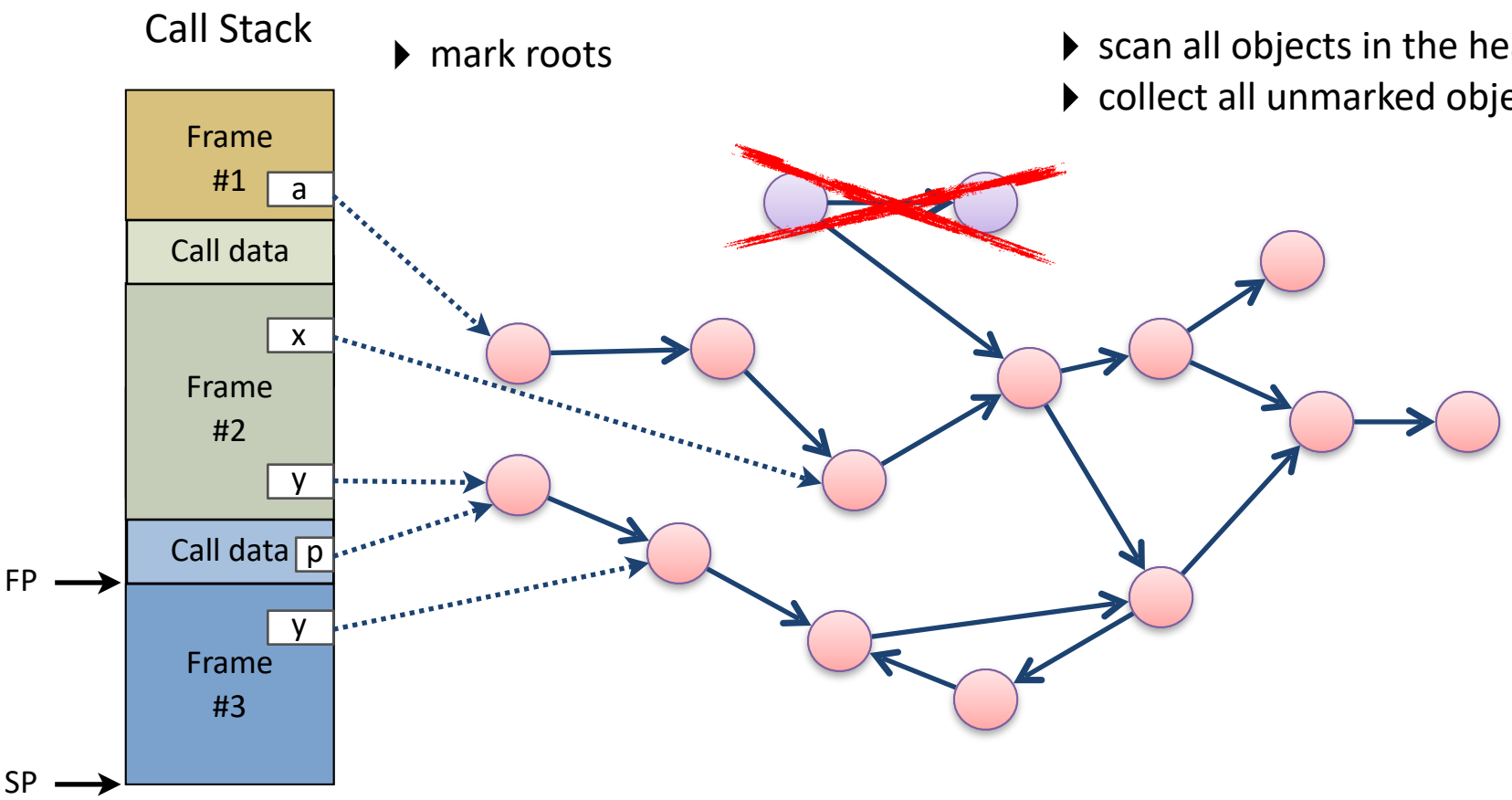


The Mark-and-Sweep Algorithm
[McCarthy 1960]

- Marking phase
  - mark roots
  - trace all objects transitively reachable from roots
  - mark every traversed object

- Sweep phase
  - scan all objects in the heap
  - collect all unmarked objects

The Mark-and-Sweep Algorithm



Triggering

- Garbage collection is triggered by allocation

```
new(A):
    if free_list is empty
            mark_sweep()
            if free_list is empty
                    return ("out-of-memory")
    pointer = malloc(sizeof(A))
    return pointer
```

- (Remember: free_list is a collection of free memory blocks)

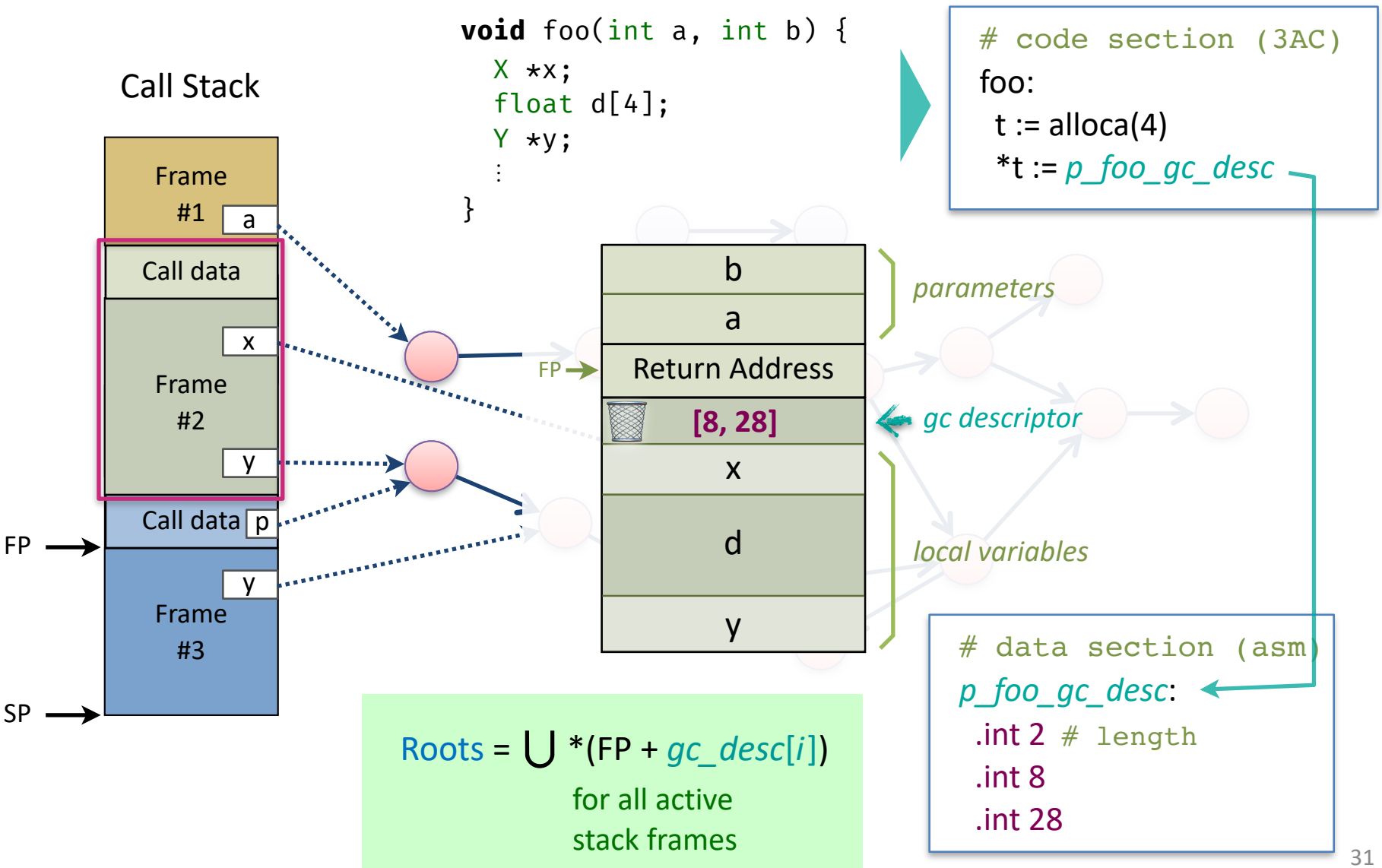
Basic Algorithm

```
mark_sweep():
    for Ptr in Roots
        mark(Ptr)
    sweep()
```

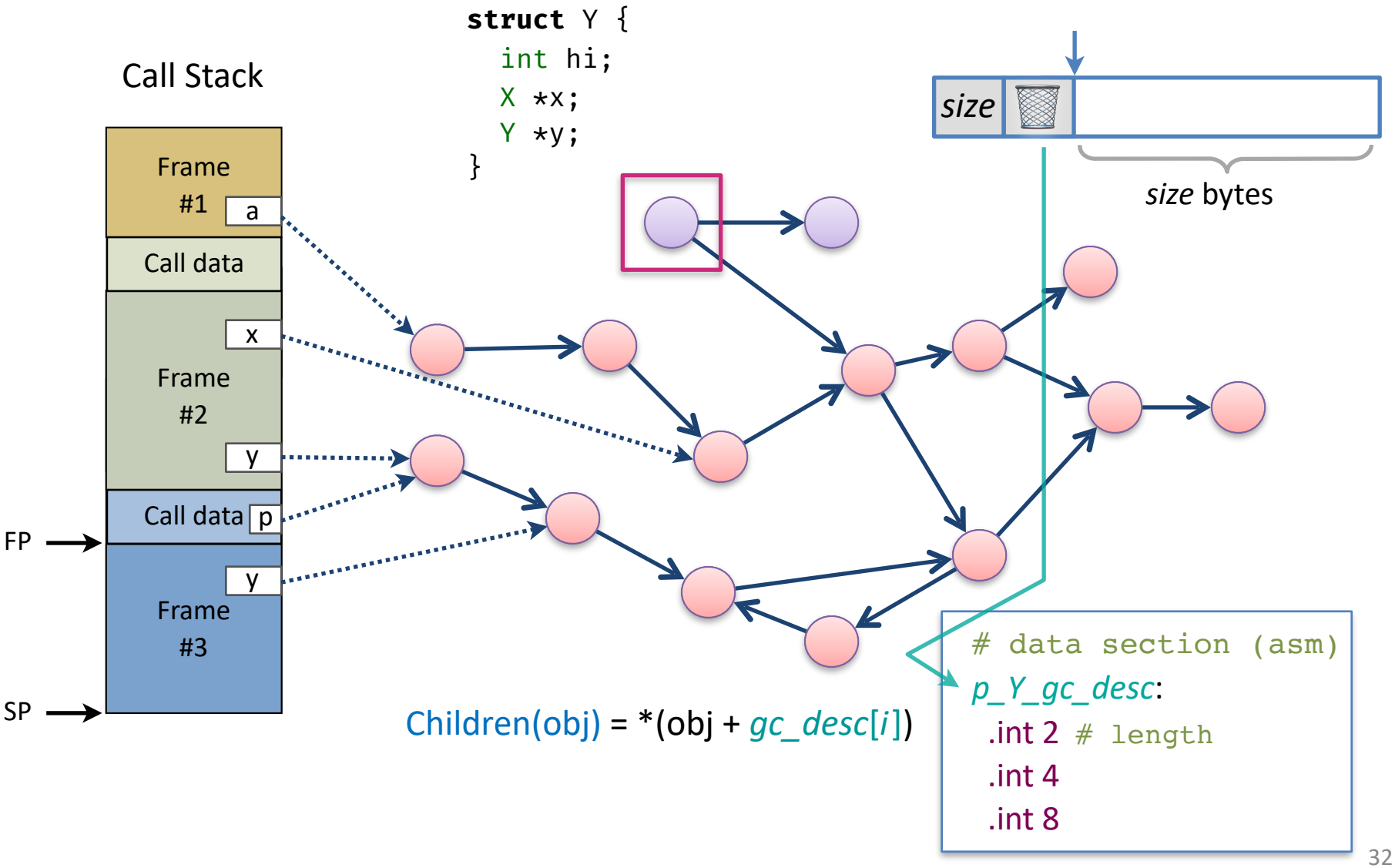
```
mark(Obj):
    if mark_bit(Obj) == unmarked
        mark_bit(Obj) = marked
        for C in Children(Obj)
            mark(C)
```

```
sweep():
    p = Heap_bottom
    while (p < Heap_top)
        if (mark_bit(p) == unmarked) then free(p)
        else mark_bit(p) = unmarked;
        p = p + size(p)
```

The Mark-and-Sweep Algorithm



The Mark-and-Sweep Algorithm



Properties of Mark & Sweep

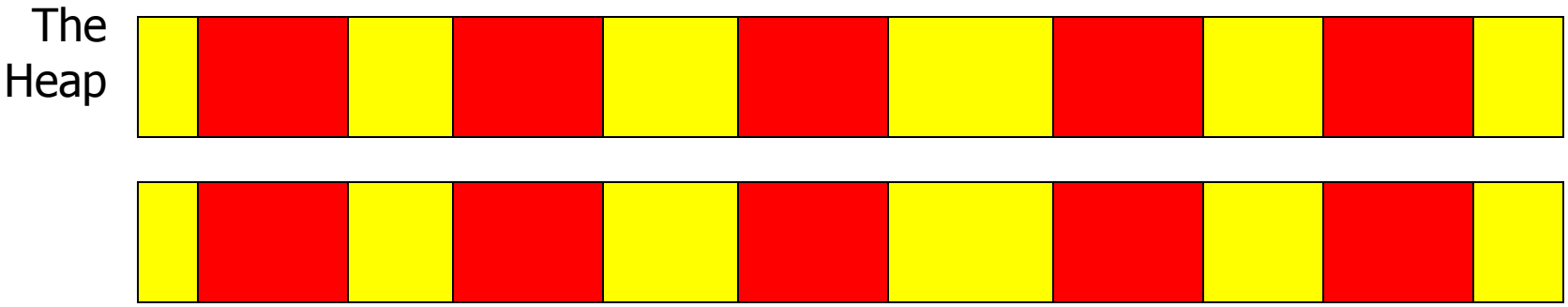
- Most popular method today
- Simple
- Does not move objects, and so **heap may fragment**

- Termination: each pointer traversed once
- Complexity
  - ☺ Mark phase: live objects (dominant phase)
  - ☹ Sweep phase: heap size
- Engineering tricks used to improve performance

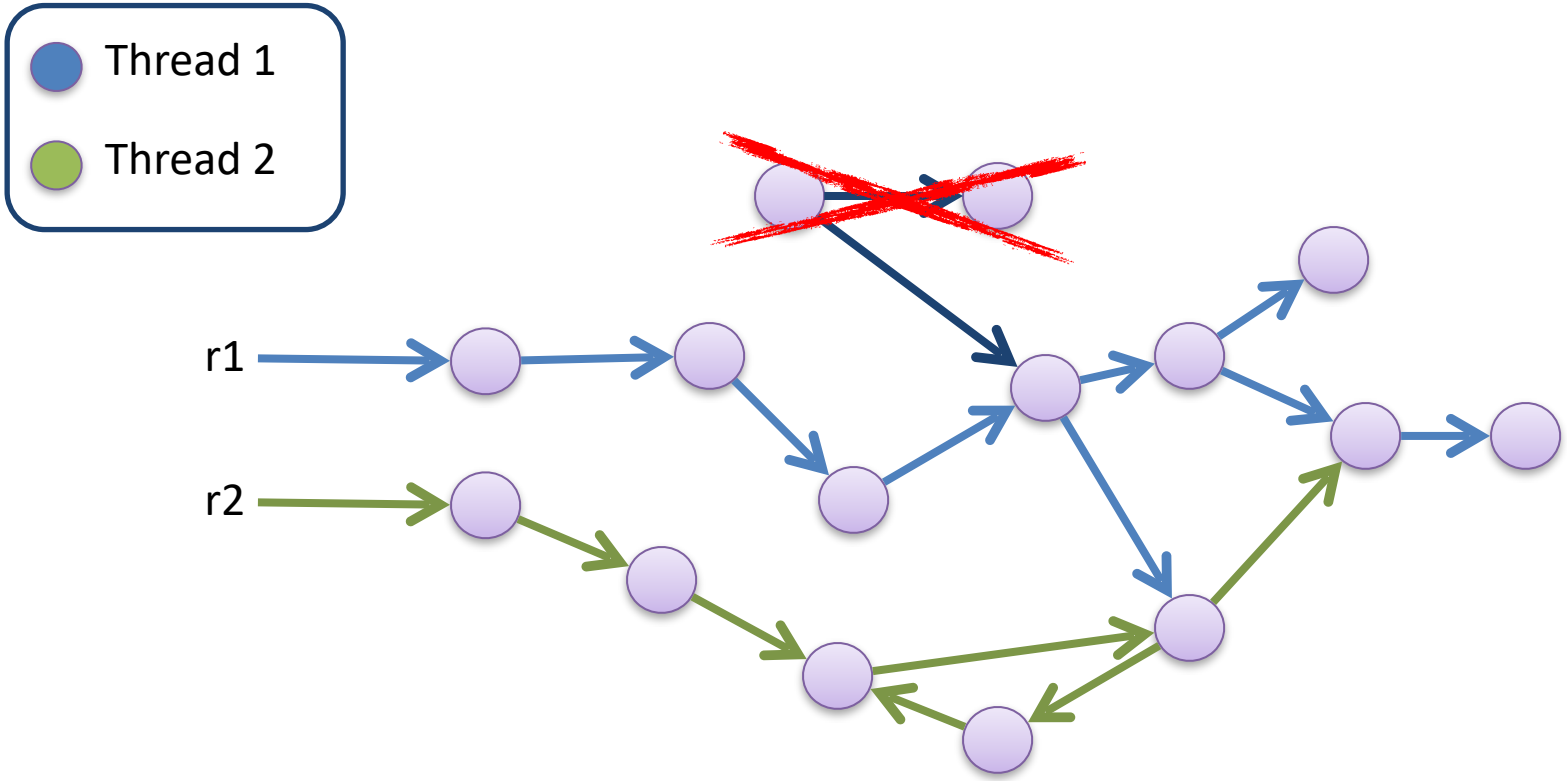
Mark-Compact

- During execution, objects are allocated and reclaimed
- Gradually, the **heap** gets **fragmented**
- When space is too fragmented to allocate, a compaction algorithm is used
- **Move all live objects to the beginning of the heap and update all pointers to reference the new locations**
- Compaction is very costly and we attempt to run it infrequently, or only partially

Compiler do that ™

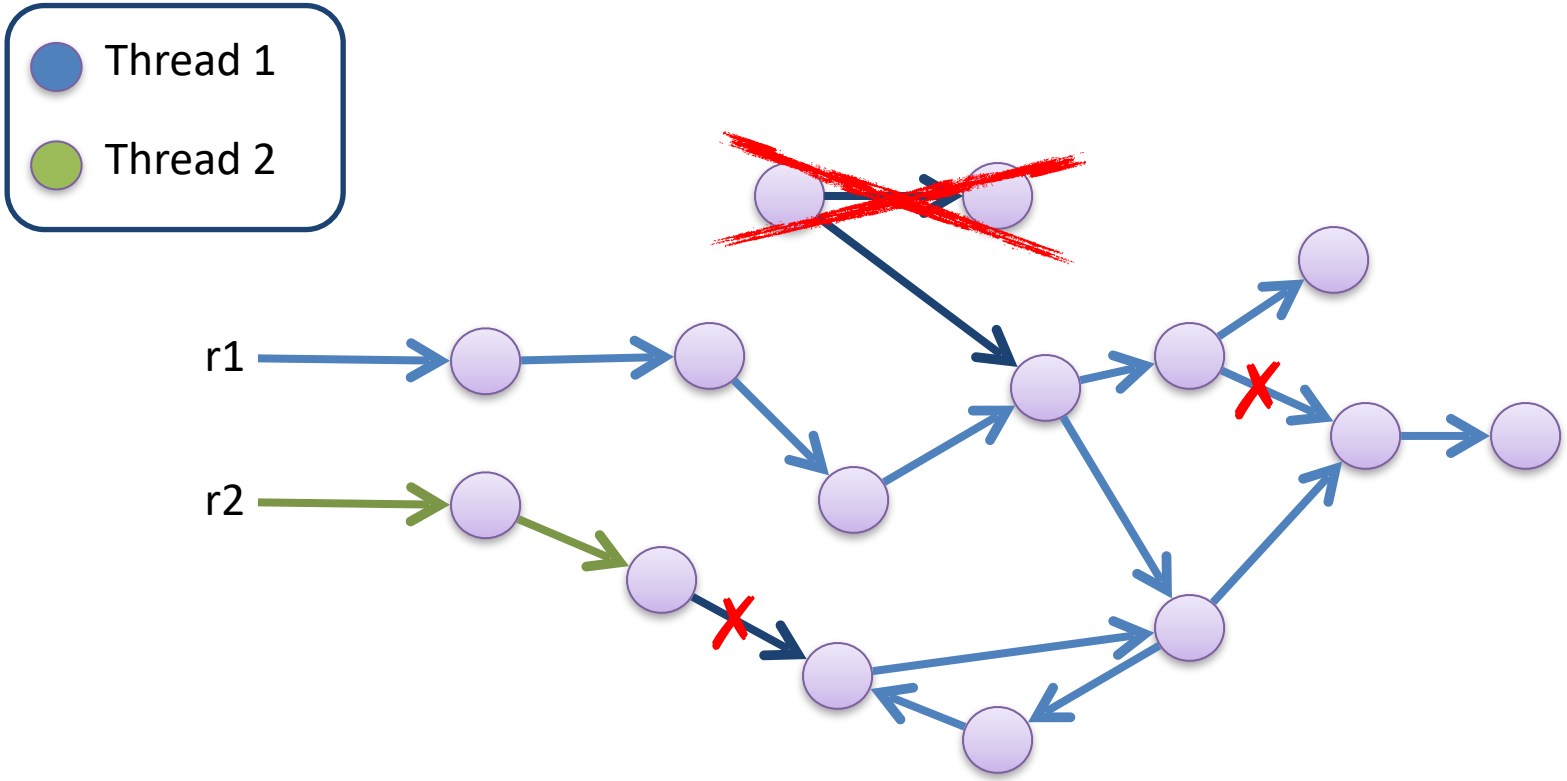


Parallel Mark & Sweep GC



Parallel = mutator (main program) is stopped,
GC threads run in parallel

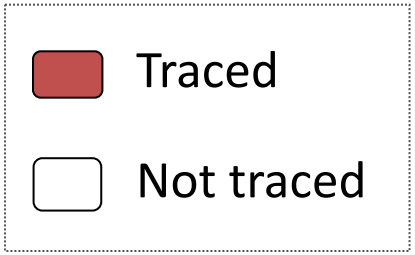
Concurrent Mark & Sweep GC

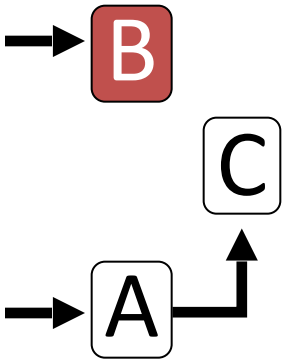


Concurrent = mutator and GC threads run in parallel,
no need to stop mutator

Problem: Interference

SYSTEM = MUTATOR || GC

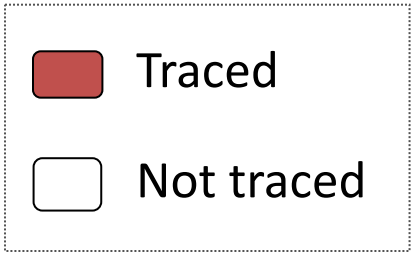


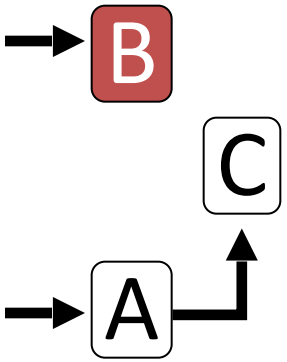


1. GC traced B

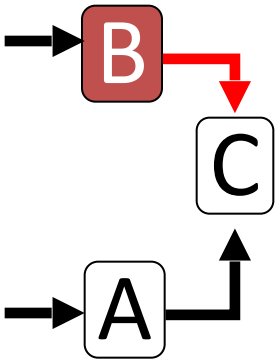
Problem: Interference

SYSTEM = MUTATOR || GC





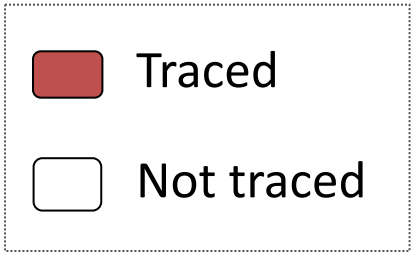
1. GC traced B

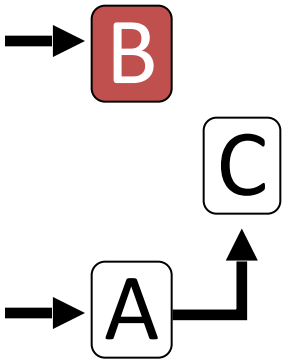


2. Mutator links C to B

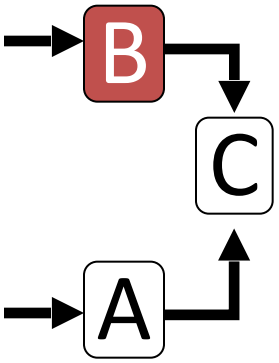
Problem: Interference

SYSTEM = MUTATOR || GC

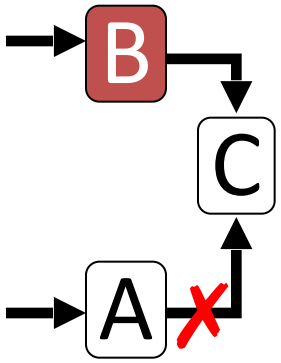




1. GC traced B



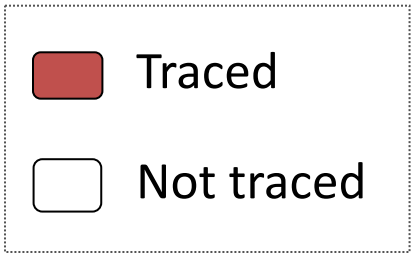
2. Mutator links C to B

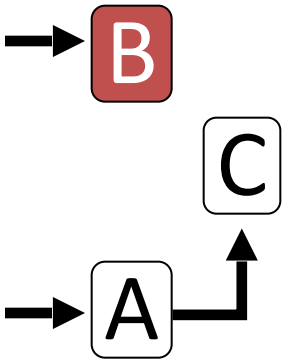


3. Mutator unlinks C from A

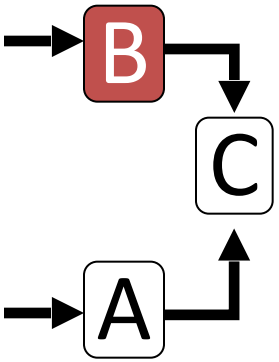
Problem: Interference

SYSTEM = MUTATOR || GC

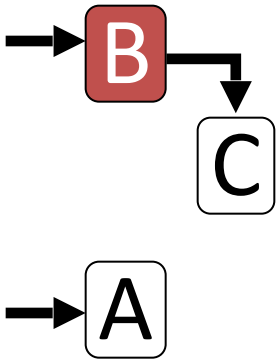




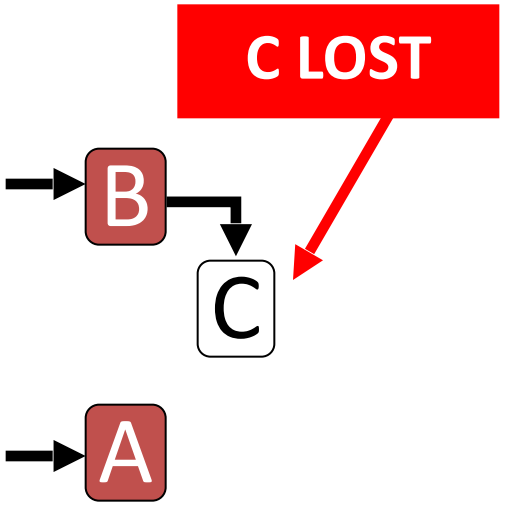
1. GC traced B



2. Mutator links C to B



3. Mutator unlinks C from A



4. GC traced A

Approach: Tricolor Abstraction

SYSTEM = MUTATOR || GC

Insight: object can **only** be lost when

Mutator creates a reference from a black object to a white object

and

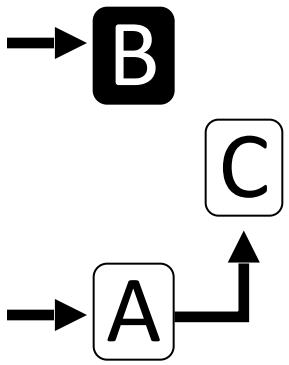
No paths from gray objects to that white object remain

All successors traced; may not have pointers to white objects.

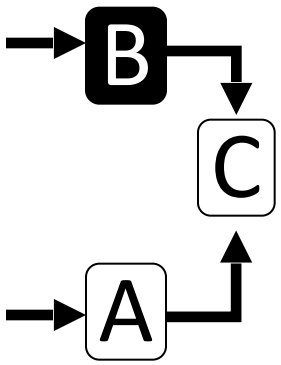
Processed

Reached

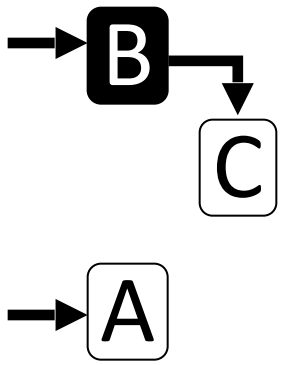
Not traced



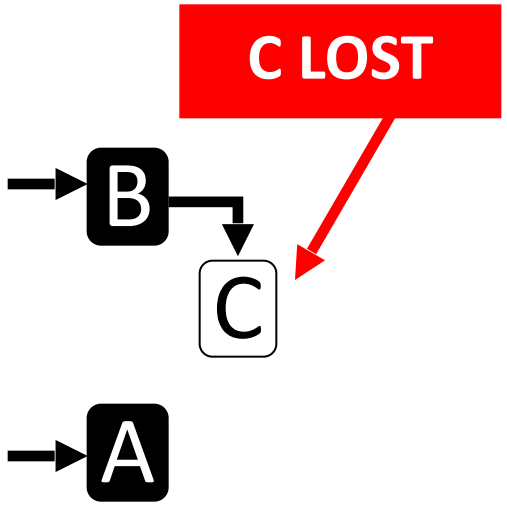
1. GC traced B



2. Mutator links C to B



3. Mutator unlinks C from A

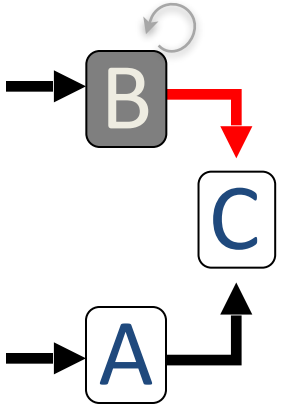


4. GC traced A

The 3 Families of Concurrent GC Algorithms

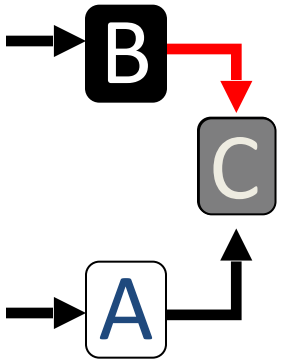
STEELE 1976

Revert B when
C is linked to B



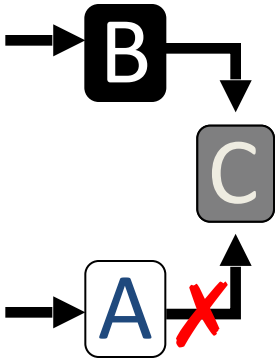
DIJKSTRA 1978

Shade C when
C is linked to B



YUASA 1990

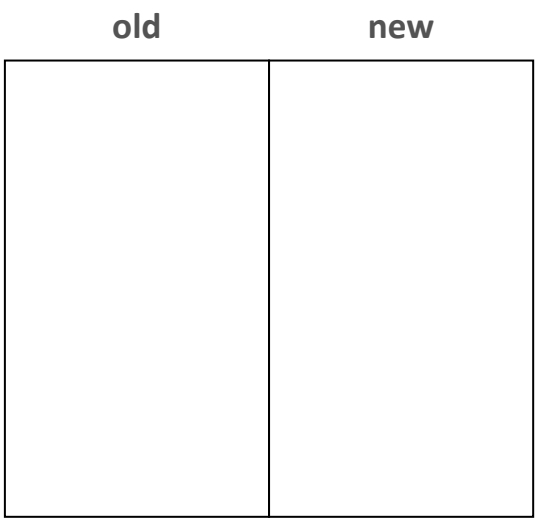
Shade C when
link to C is removed



Compiler
do that ™

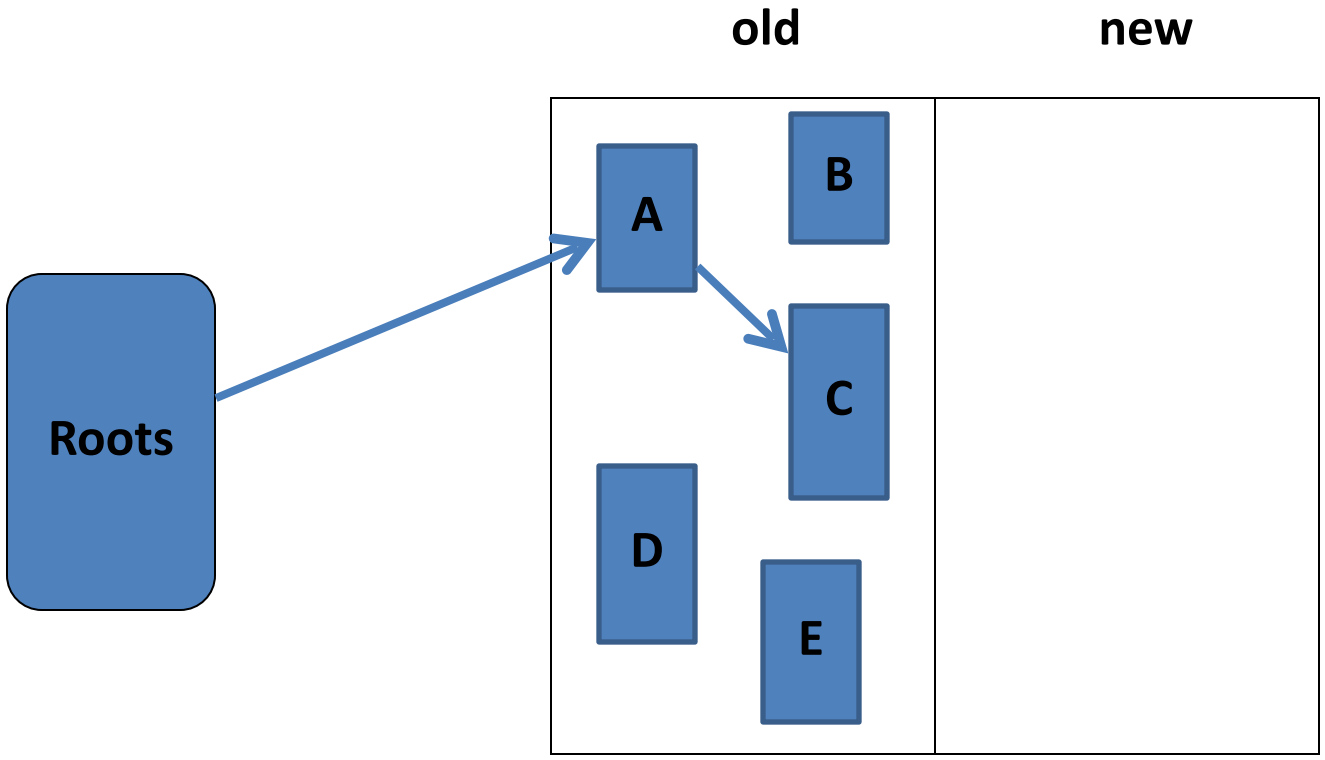
Copying Garbage Collection

- Partition the heap into two parts
  - old space
  - new space

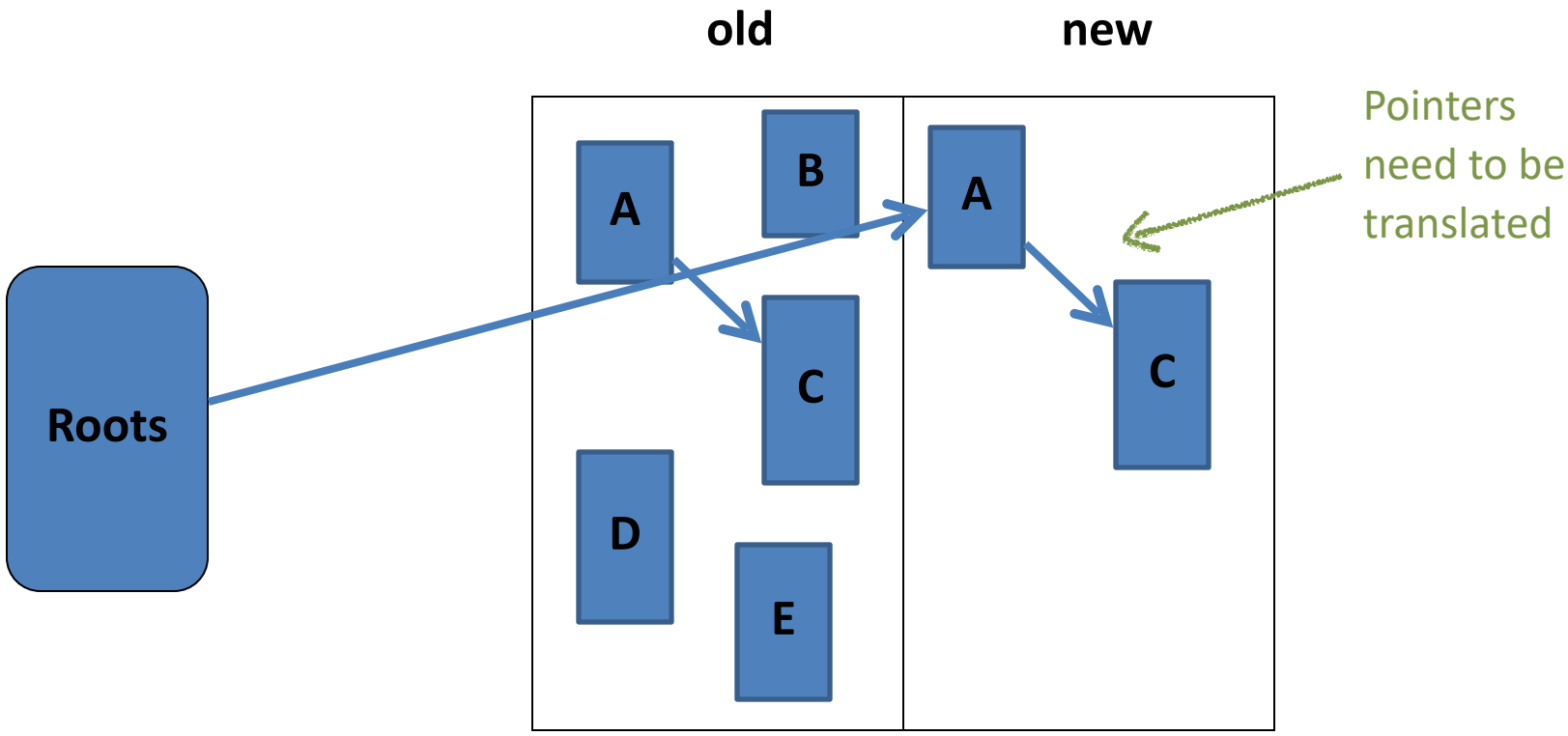


- Copying GC algorithm
  - copy all **reachable** objects from old space to new space
  - then, swap roles of old/new space

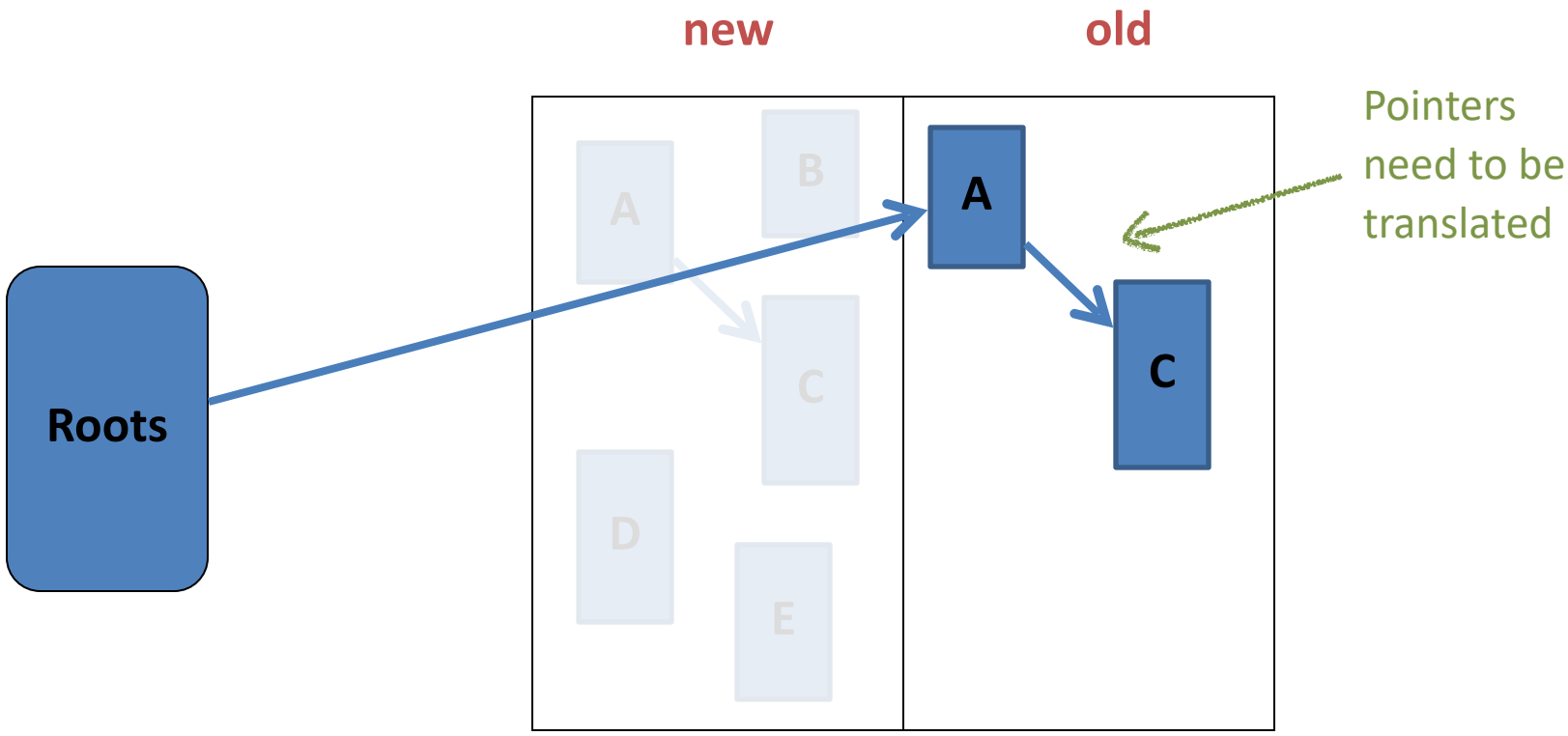
Example



Example



Example



Properties of Copying GC

- Compaction is obtained for free
- Disadvantage: **half of the heap is not used**
- Advantage: “touches” only live objects
  - Good when most objects are dead

Generational Garbage Collection

- Observation: usually, most new objects are dead
- Idea: manage "young" objects separately from "old" objects.
  - The heap is split into two "generations".
  - The "young" objects are managed using a copying GC (copying cost is low, since most of them are dead)
  - The "old" objects are managed using *e.g.* Mark & Sweep
  - Transition from "young" to "old" is done based on some heuristic;
    e.g., an object is moved to old after having been copied *k* times.

Conservative GC

- How do you track pointers in languages such as C?
  - Any value can be cast down to a pointer
- Easy – be conservative, consider anything that can be a pointer to be a pointer
- Practical! (*e.g.*, Boehm collector)

Conservative GC

- Can you implement a conservative **copying GC**?

- What is the problem?

- **Cannot update pointers to the new address… not sure whether the value is a pointer — it is unsafe update it**

A note about Rust

- Ownership types
  - Embed information about which function owns which memory object in the *types* of variables and parameters.

`T`	`&T`	`&mut T`
owned	borrowed (immutable)	borrowed (mutable)

```
let mut v : Vec<i32> = vec![1, 2, 3];
v.push(4);
v.push(5);

send(v);
```

Allocate a vector on the heap

Method `push` *borrows* a reference

```
fn push(self : &mut Vec<i32>, value : i32)
```

Transfer ownership to a function `send`

```
fn send(u : Vec<i32>)
```

A note about Rust

- Embed information about which function owns which memory object in the *types* of variables and parameters.

`T`	`&T`	`&mut T`
owned	borrowed (immutable)	borrowed (mutable)

- The compiler ensures that
  - Only *one context* may own an object at any given time
  - The scope of a borrow (&) is *contained* in the scope of the object's owner
  - Mutable borrows are *exclusive* (preclude all other borrows)
- As a consequence
  - Only the owner ever has to free an object.
  - And several other benefits (*esp.*, no data races).

No manual `free()`!
No memory leaks!
No runtime overhead!

A note about Rust

- Embed information about which function owns which memory object in the *types* of variables and parameters.
  - Only *one context* may own an object at any given time
  - The scope of a borrow (&) is *contained* in the scope of the object's owner
  - Mutable borrows are *exclusive* (preclude all other borrows)

`T`	`&T`	`&mut T`
owned	borrowed (immutable)	borrowed (mutable)

Vector is owned by the outer scope

```
let mut v : Vec<i32> = vec![1, 2, 3];

join(|| println!("thread #1: {}", (&v)[1]),
     || println!("thread #2: {}", (&v)[2]));

v.push(4);
```

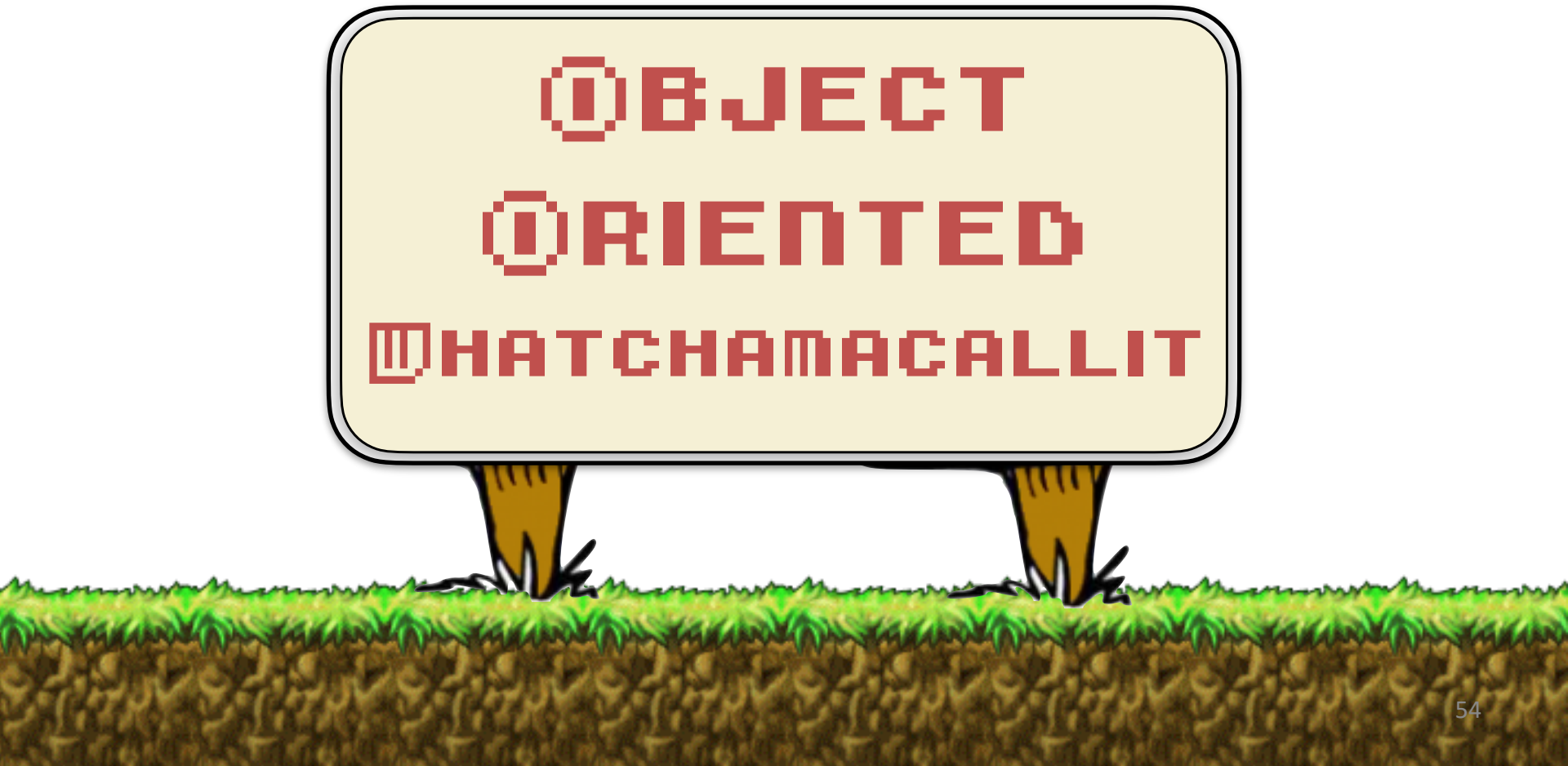
Creates a shared reference of type `&Vec<i32>`

- Borrowed until `join` exits
- Can be aliased
- Object cannot be changed

Here we can mutate the object again

THEORY OF COMPILATION

LECTURE 11



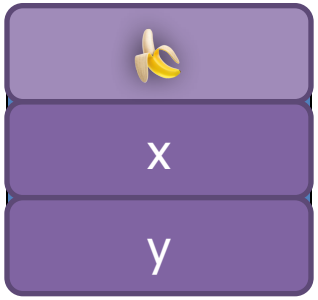
Object Types

- Basic operations
  - Field selection
    - computing address of field, dereferencing address
  - Copying
    - copy block (shallow) or field-by-field copy (deep)
  - Method invocation
    - Identifying method to be called, calling it

- How does it look at runtime?

Object Types

```
class Foo {
   int x;
   int y;

   void rise() {…}
   void shine() {…}
}
```

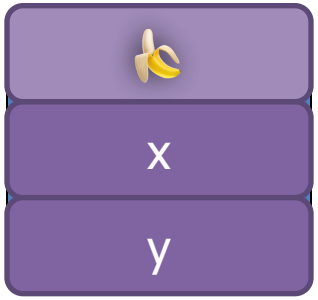


Runtime memory layout for object of class Foo

Compile time information

Field Selection

```
Foo f;
int q;

q = f.x;
```



Runtime memory layout for object of class Foo

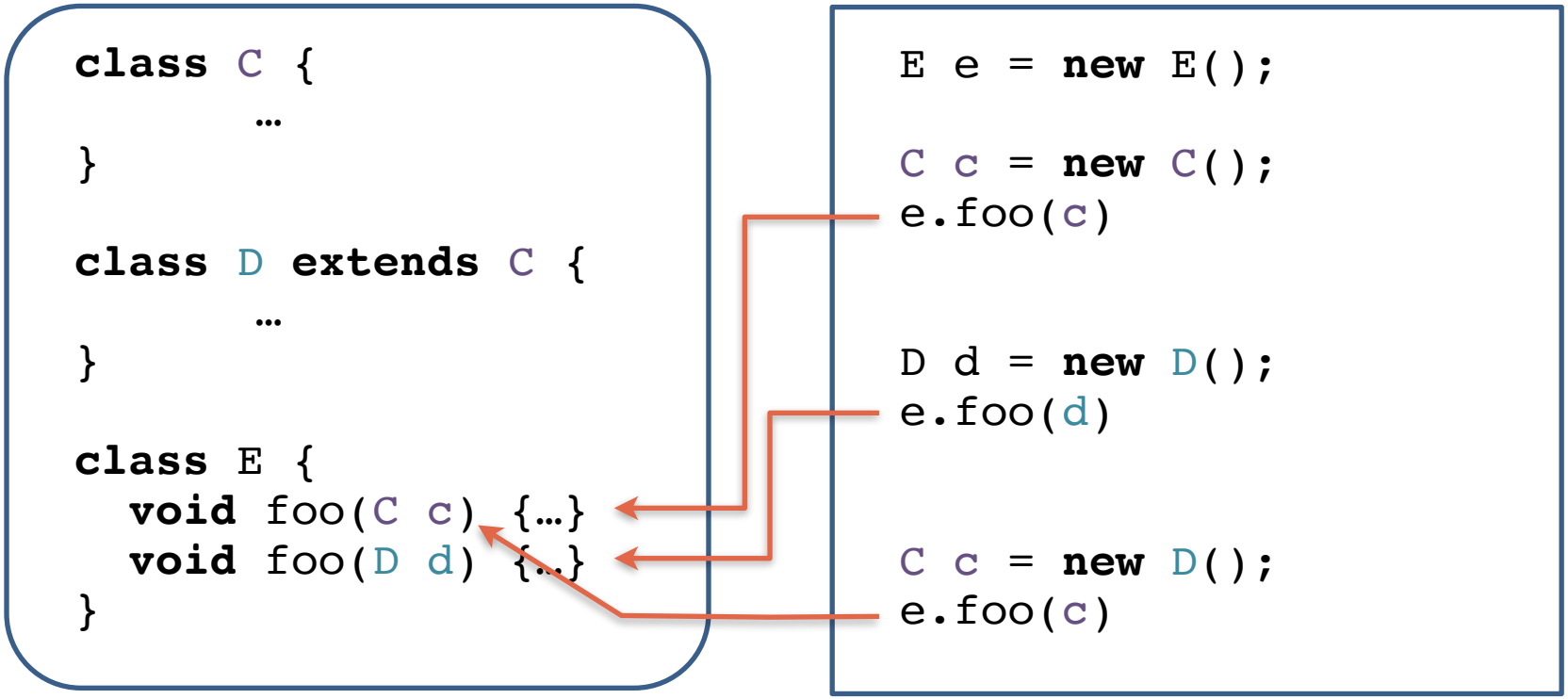
base pointer

field offset relative to base pointer

```
t1 := f
t2 := t1 + 4
q := *t2
```

Compile time information

Function Overloading



- Methods with same name but different parameter types are compiled as separate subroutines.
- During typechecking, the appropriate alternative is chosen for each call site.

Function Overloading

```
class C {
        …
}
class D extends C {
        …
}
class E {
  void foo(C c) {…}
  void foo(D d) {…}
  void foo(C c, D d) {…}
  void foo(D d, C c) {…}
}
```

```
E e = new E();

C c = new C();
e.foo(c)
e.foo(c, c)

D d = new D();
e.foo(d)
e.foo(c, d)
e.foo(d, d)
```

No suitable match

Ambiguous matches

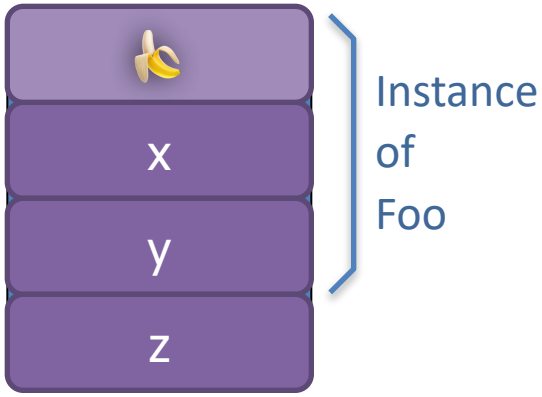
- The compiler always looks for the most suitable match.
- Two kinds of errors may arise:

No suitable match

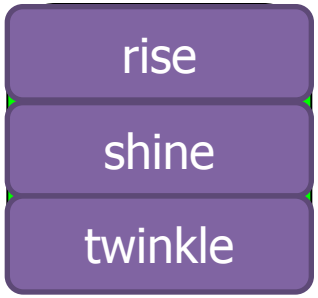
Ambiguous matches

Object Types - Inheritance

```
class Foo {
  int x;
  int y;

  void rise() {…}
  void shine() {…}
}

class Bar extends Foo {
  int z;
  void twinkle() {…}
}
```



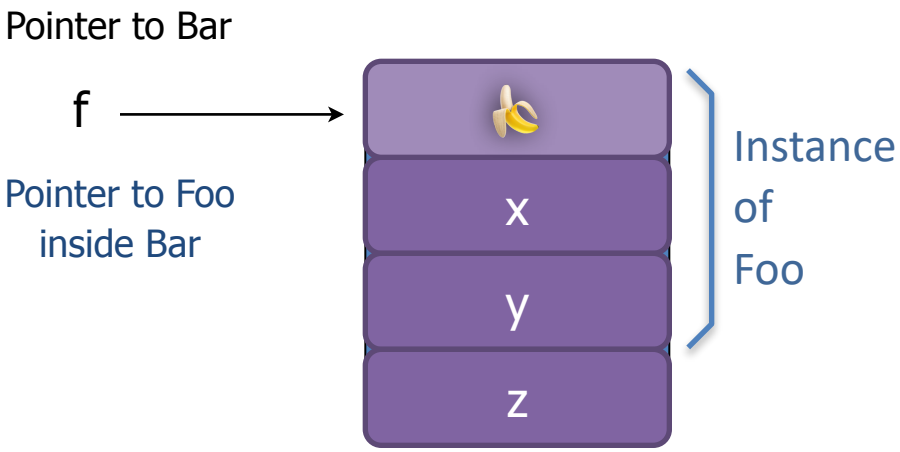
Runtime memory layout
for object of class Bar



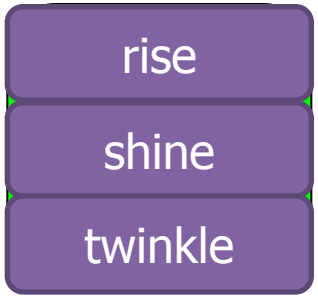
Compile time information

Object Types - Polymorphism

```
class Foo {
  …
  void rise() {…}
  void shine() {…}
}

class Bar extends Foo {
  …
}

class Main {
  void main() {
    Foo f = new Bar();
    f.rise();
}
```



Runtime memory layout for object of class Bar



Compile time information

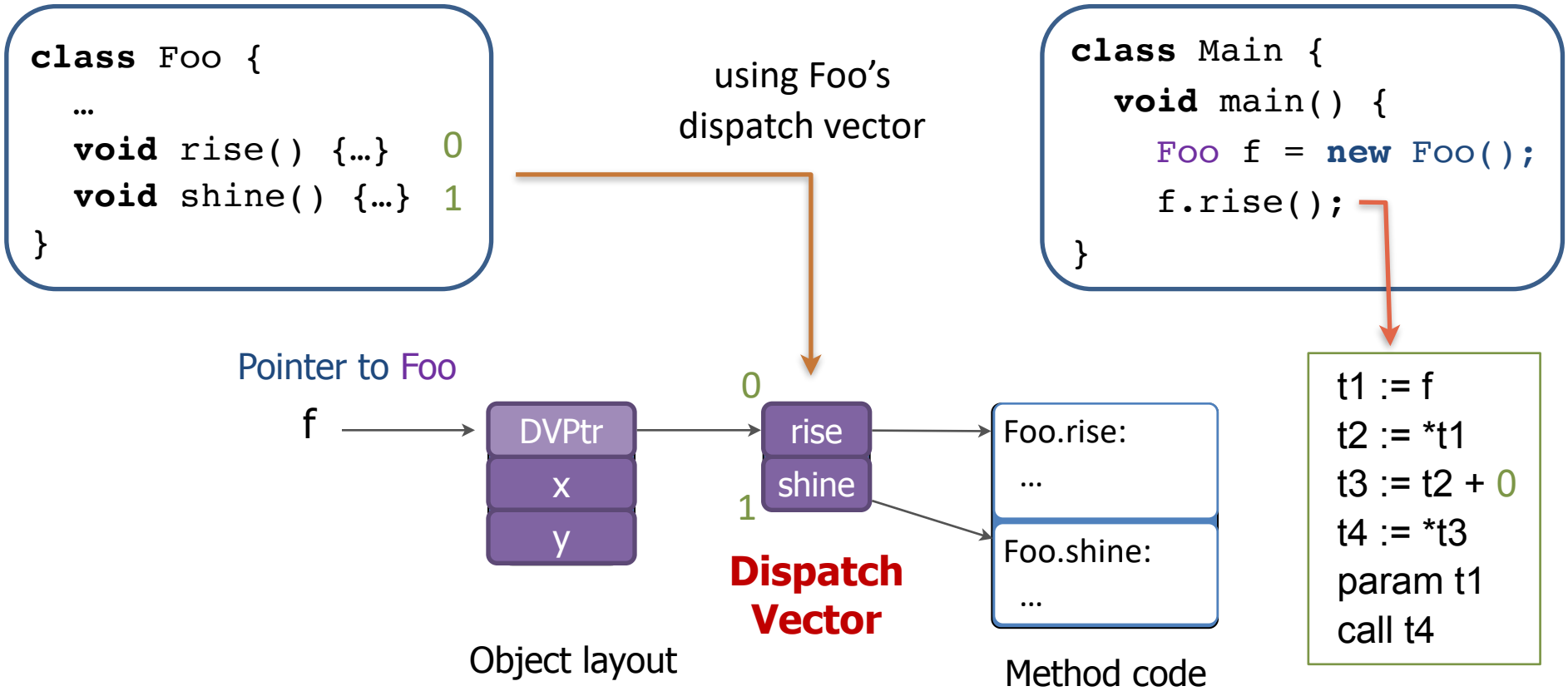
Dynamic Dispatch

```
class Foo {
  …
  void rise() {…}
  void shine() {…}
}

class Bar extends Foo {
  void rise() {…}
}

class Main {
  void main() {
    Foo f = new Bar();
    f.rise();
}
```

- Method defined in superclass, may be overridden in subclass.
  - Finding the right method implementation
  - Done at runtime according to concrete object type
  - Using a Dispatch Vector (sometimes called Dispatch Table, or Virtual Table, or vtable)

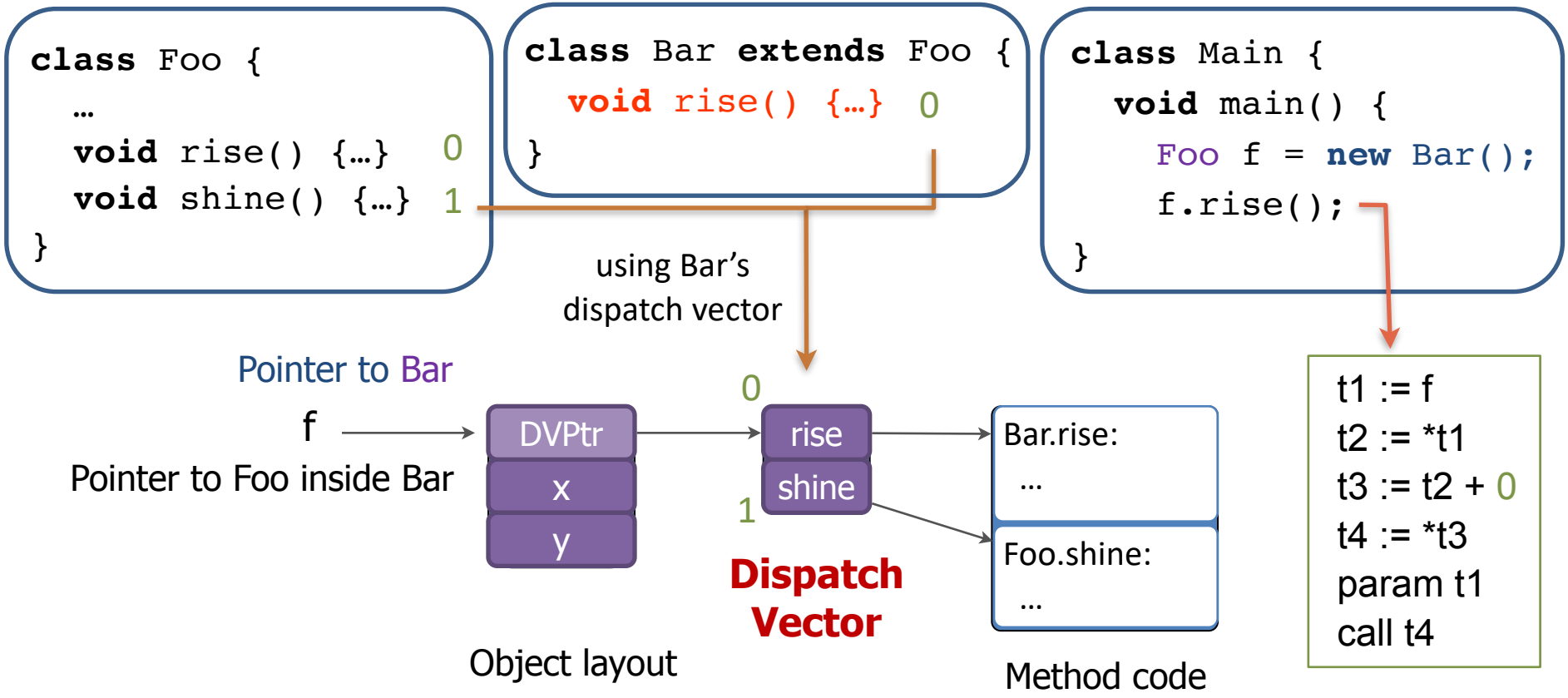
(*assume all the methods are virtual, Java-like*)

Dispatch Vectors



- Dispatch Vector contains addresses of methods
- Indexed by method-id number
- A method signature has the same id for all subclasses

Dispatch Vectors



- Every class has its own dispatch vector, built at compile time
- A pointer to the dispatch vector is set on every object in the constructor
- DVPtr can also serve as a runtime class identifier

Representing Dispatch Tables

```
class Foo {
  void rise() {…}     0
  void shine() {…}    1
   …
}
```

0	Foo::rise
1	Foo::shine

```
class Bar extends Foo {
  void rise() {…}       0
  void twinkle() {…}    2
}
```

0	Bar::rise
1	Foo::shine
2	Bar::twinkle

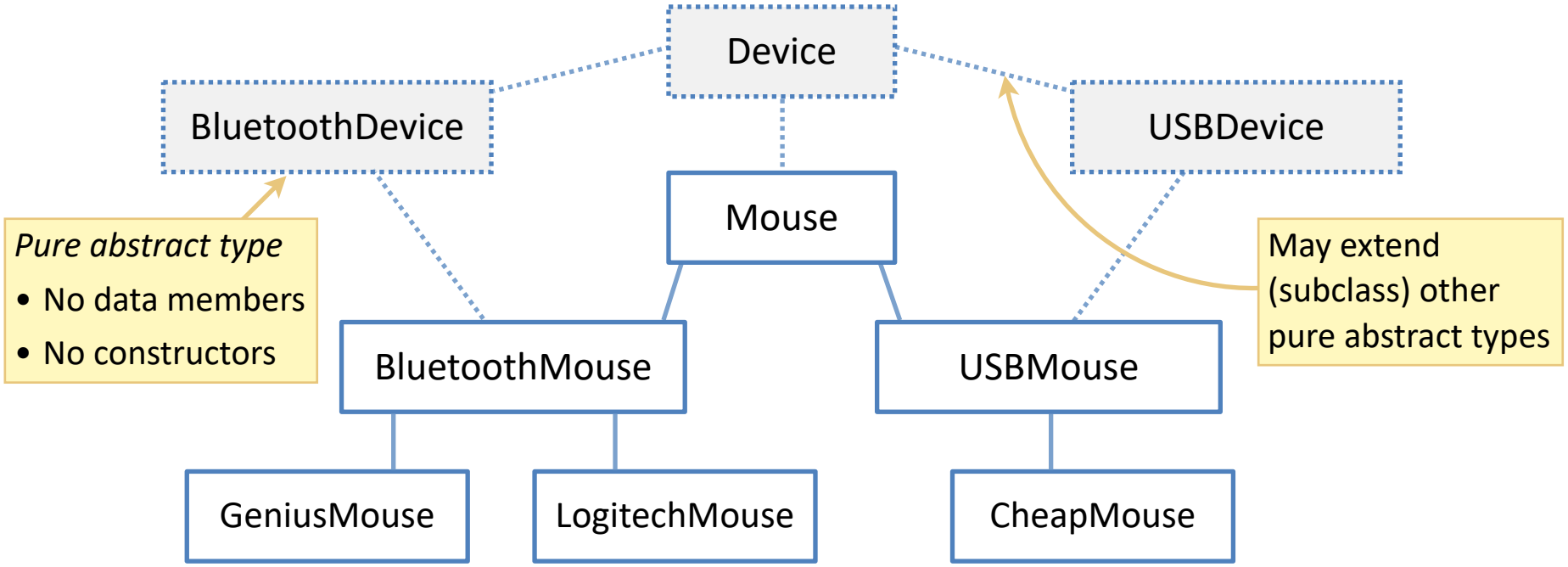
```
# data section (asm)
.data
  .align 4
_DV_Foo:
  .long _Foo_rise
  .long _Foo_shine
_DV_Bar:
  .long _Bar_rise
  .long _Foo_shine
  .long _Bar_twinkle
```

```
# code section (3AC)
_Foo_ctor:
   t1 := arg0          # (this)
   *t1 := _DV_Foo
   return

_Bar_ctor:
   t1 := arg0          # (this)
   call _Foo_ctor(t1)
   *t1 := _DV_Bar
   return
```

Multiple Inheritance

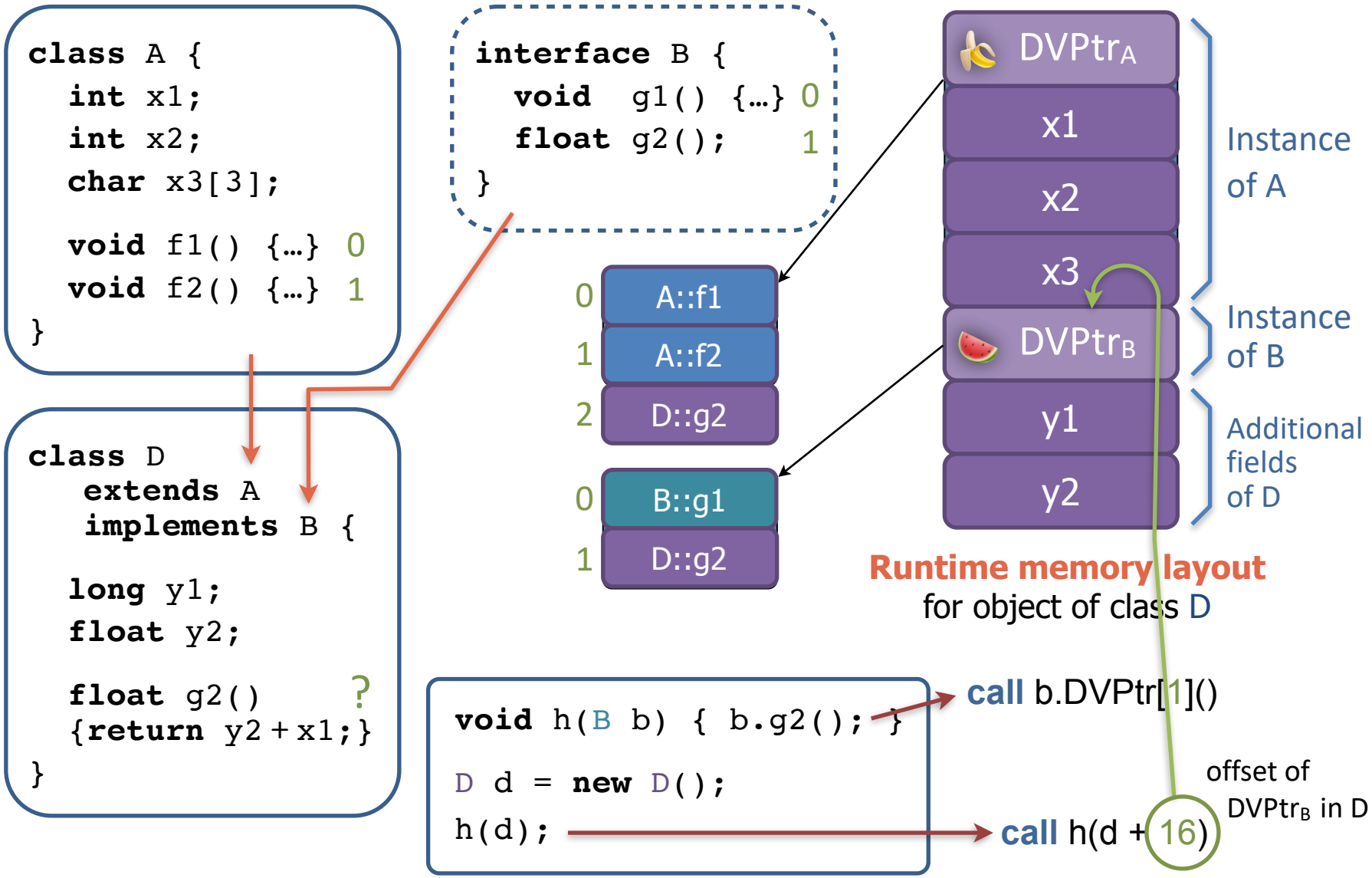
- Remember?

  We assume that any class may extend at most one non-pure-abstract class.

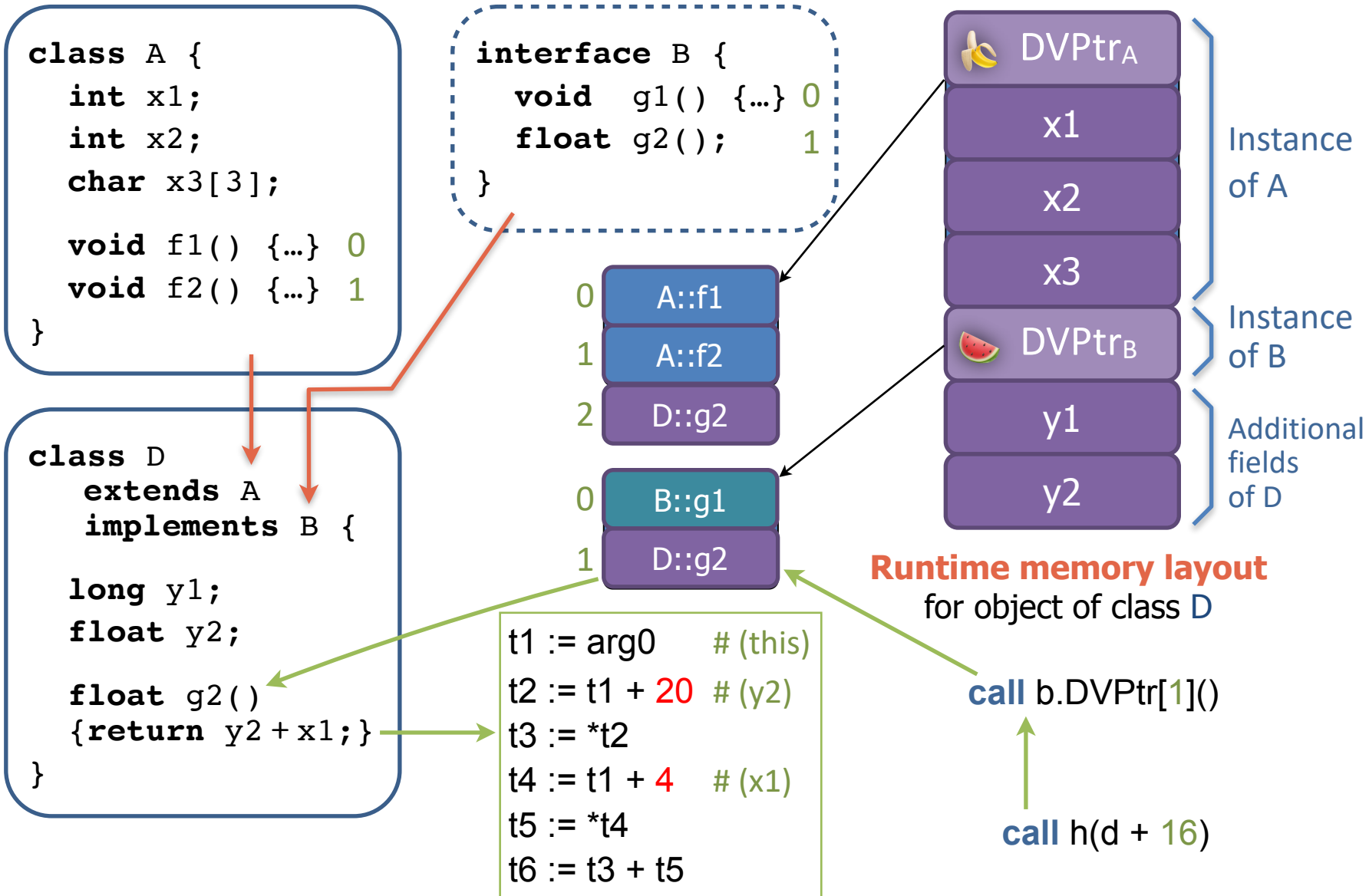


Inheritance ~~Tree~~ DAG

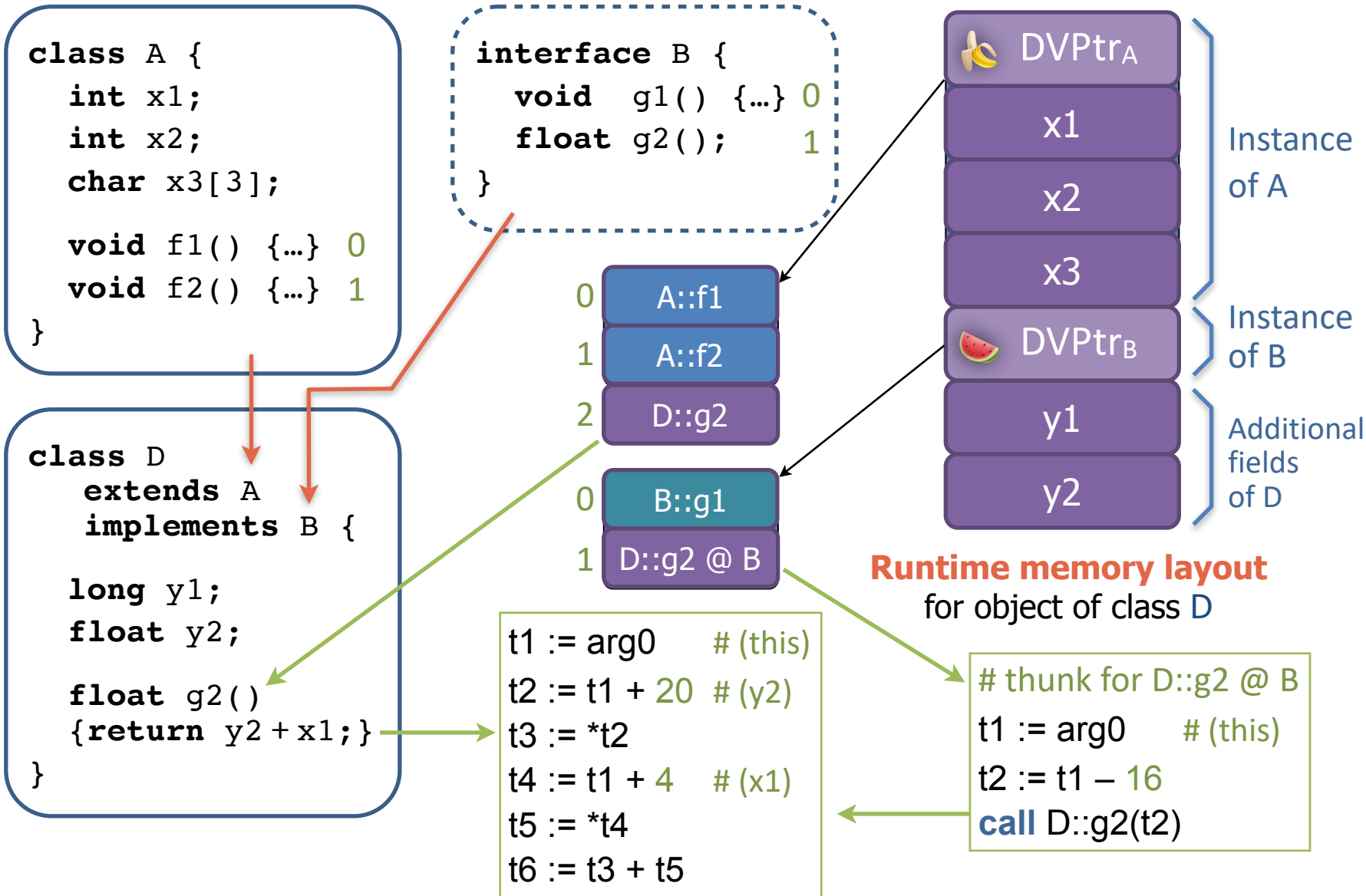
Multiple Inheritance



Multiple Inheritance

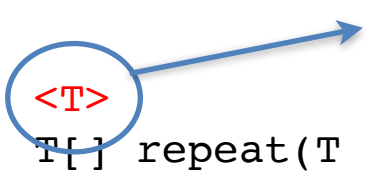


Multiple Inheritance



Generics

- Concept: reuse the same code for different data types.




```
T[] repeat(T value, int times) {
    T[] arr = new T[times];
    for (int i = 0; i < times; i++) arr[i] = value;
    return arr;
}

String[] s = repeat("theory of compilation", 236);
int[] k = repeat(42, 360);
A[] a = repeat(new A(), 101);
```

- Most common use case: collections library

Generics

- Implementation: two variants —
  - C++ templates: per-type specializations
  - Java generics: general type, insert coercions

```
T[] repeat(T value, int times) {
    T[] arr = new T[times];
    for (int i = 0; i < times; i++) arr[i] = value;
    return arr;
}

String[] s = repeat("theory of compilation", 236);
int[] k = repeat(42, 360);
A[] a = repeat(new A(), 101);
```

Templates: Specialization

```
<T>
T[] repeat(T value, int times) {
    T[] arr = new T[times];
    for (int i = 0; i < times; i++) arr[i] = value;
    return arr;
}

repeat("compi", 236);
repeat(42, 360);
repeat(new A(), 101);
```

```
String[] repeat@String(String value, int times) {
    String[] arr = new String[times];
    for (int i = 0; i < times; i++) arr[i] = value;
    return arr;
}

int[] repeat@int(int value, int times) {
    int[] arr = new int[times];
     …
}

A[] repeat@A(A value, int times) {
    A[] arr = new A[times]; …
```

- Code is replicated for every type assignment

Templates: Specialization

```
<T>
T repeat(T value, int times) {
    T t = value;
    for (int i = 0; i < times; i++) t = t + value;
    return t;
}
```

```
String repeat@string(String value, int times) {
    String t = value;
    for (int i = 0; i < times; i++) t = t + value;
    return arr;
}
```

✓

```
repeat("compi", 236);
```

```
repeat(42, 360);
```

```
int repeat@int(int value, int times) {
    int t = value;
    for (int i = 0; i < times; i++) t = t + value;
     …
}
```

✓

```
repeat(new A(), 101);
```

```
A repeat@A(A value, int times) { … t = t + value; …
```

✗

- Code is replicated for every type assignment
  - Duck-typing is used to check instances.

Java: Coercion Injection

```
<T>
T[] repeat(T value, int times) {
    T[] arr = new T[times];
    for (int i = 0; i < times; i++)
        arr[i] = value;
    return arr;
}
```

```
Object[] repeat@O(Object value, int times){
    Object[] arr = new T[times];
    for (int i = 0; i < times; i++)
        arr[i] = value;
    return arr;
}
```

```
repeat("compi", 236);   → (String[])repeat@O((Object)"compi", 236);
repeat(42, 360);        → (Integer[])repeat@O(
                                (Object)new Integer(42), 101);
repeat(new A(), 101);   → (A[])repeat@O((Object)new A(), 101);
```

- Single implementation — based on common interface (*e.g.* `Object`)
  - Primitive types require special treatment: autoboxing

Java: Coercion Injection

```
<T extends Arith>
T repeat(T value, int times) {
    T t = value;
    for (int i = 0; i < times; i++)
        t = t.plus(value);
    return t;
}
```

```
Arith[] repeat@G(Arith value, int times){
    Arith t = value;
    for (int i = 0; i < times; i++)
        t = t.plus(value);
    return t;
}
```

method of `Arith`

```
repeat("compi", 236);
```

→ `Error: 'String' is not a subtype of 'Arith'`

```
repeat(new A(), 101);
```

→

```
(A)repeat@G((Arith)new A(), 101);
```

(*assuming* `A extends Arith`)

- Subtyping constraints are a required feature
  - Introduce further assumptions about the types being used.

Generics: Pros and Cons

C++ Templates: Specialization

- ✓ Allows for partial specializations (with duck typing)
- ✓ Enables type-specific optimizations
- ✓ Agnostic to memory management issues
- ✗ Code bloat

Java Generics: Coercion Injection

- ✓ Early detection of type errors
- ✓ Small code footprint
- ✗ Depends on garbage collection (autoboxing)

Switching Topic Again

Error Handling

Runtime Checks

- Generate code for checking attempted illegal operations
  - Null pointer check
    - Field access, array access, get array length, call virtual method
    - Pointer-typed arguments to library functions that should not be null
  - Array bounds check
  - Array allocation size check
  - Division by zero
  - …
- If check fails — jump to **error handler** code that prints a message and gracefully exits program
- Alternatively, use an exception-handling mechanism

Null Pointer Check

Before `ptr->field, ptr[i], ptr->method(…)`

```
# null pointer check
t1 := ptr
if (t1==0) goto labelNPE
```

Single generated handler for entire program

```
labelNPE:
    push $strNPE    # error message
    call __println
    push $1         # error code
    call __exit
```

Array allocation size check

Before `new T[size]`

```
# array size check
t1 := size
if (t1<=0) goto labelASE
```

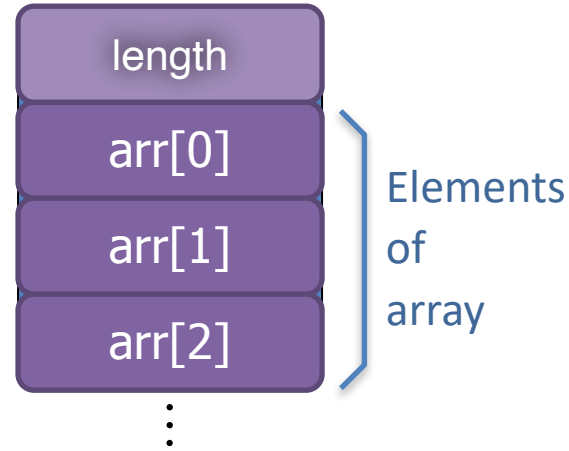
Single generated handler for entire program

```
labelASE:
    push $strASE   # error message
    call __println
    push $1        # error code
    call __exit
```

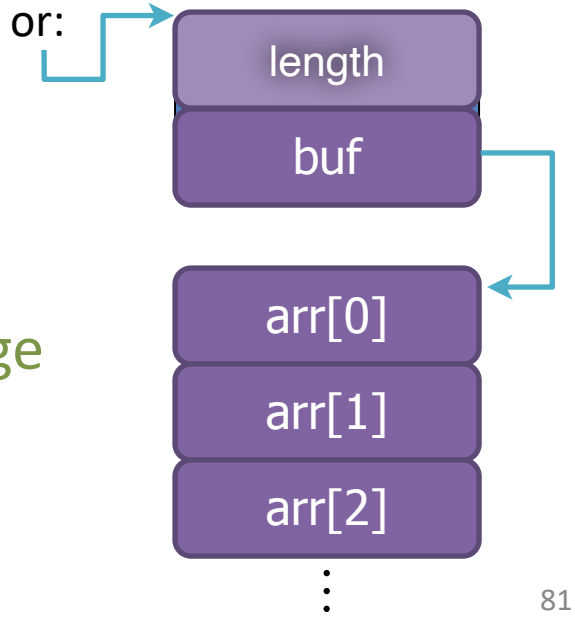
Array Bounds Check

Before `arr[index], arr[index] = expr`

```
# array bounds check
t1 := @arr
t2 := *t1
t3 := index
if (t3 < 0) goto labelABE
if (t3 >= t2) goto labelABE
```

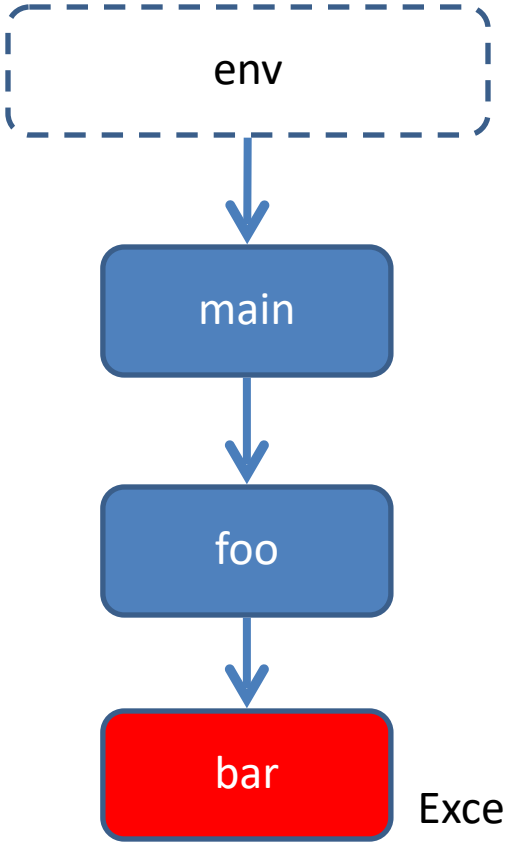


or:




```
labelABE:
    push $strABE      # error message
    call __println
    push $1           # error code
    call __exit
```

Exceptions



Exception thrown

```
main() {
  foo(1)
}

foo(int n) {
  bar(n – 1, 1)
}

bar(int x, int y) {
  if (x < y)
    throw new Exception()
}
```

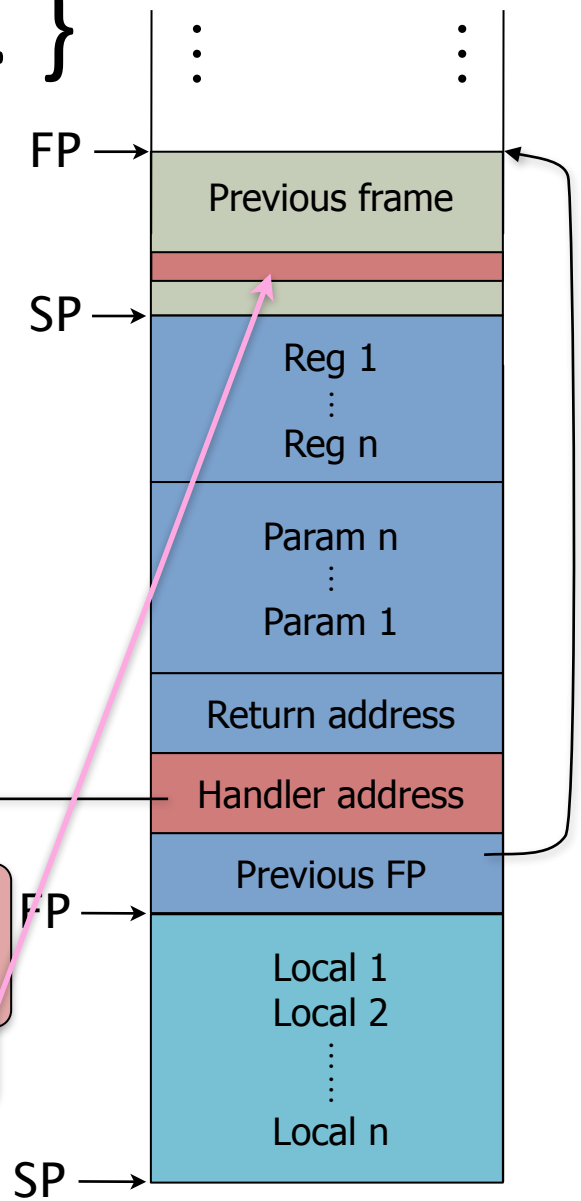
try { … } catch { … }

- Allows programmer to customize error handler

```
if (exc.type == IO_ERROR) {
        … something something …
        jump to <after catch block>
}
else if (exc.type == HW_ERROR) {
        … something something …
        jump to <after catch block>
}
else {
        unwind
        throw exc
}
```

Updates FP and SP, similar to epilogue
Return address is ignored

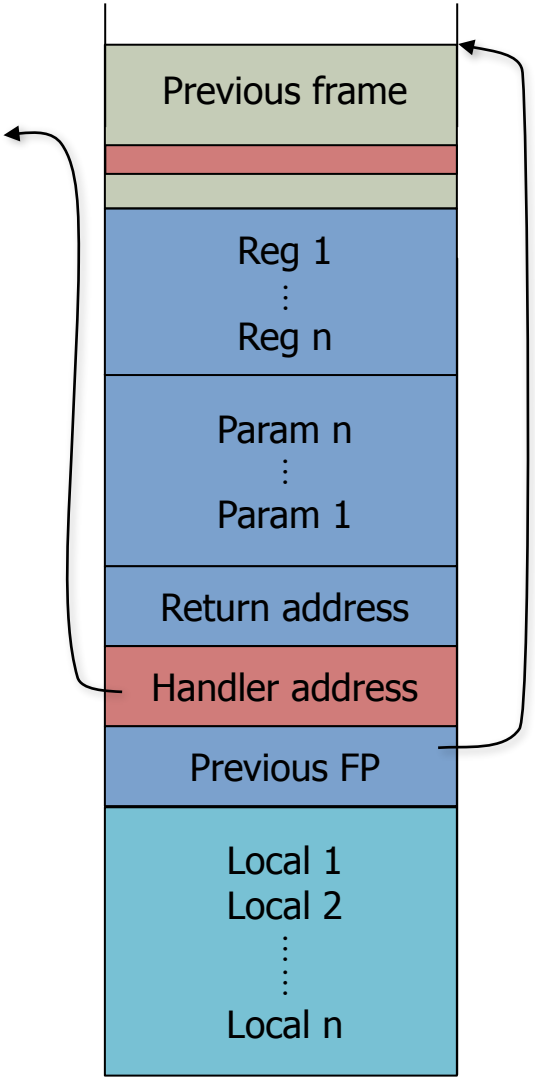
Invokes caller's exception handler



- (Default handler throws unconditionally)

Exception Handling Overhead

- Handler address has to be updated
  - whenever a `try { }` block is entered
  - and when it ends successfully
- Imposes constant cost even when no exception is thrown
  - C++ programmers cannot have that



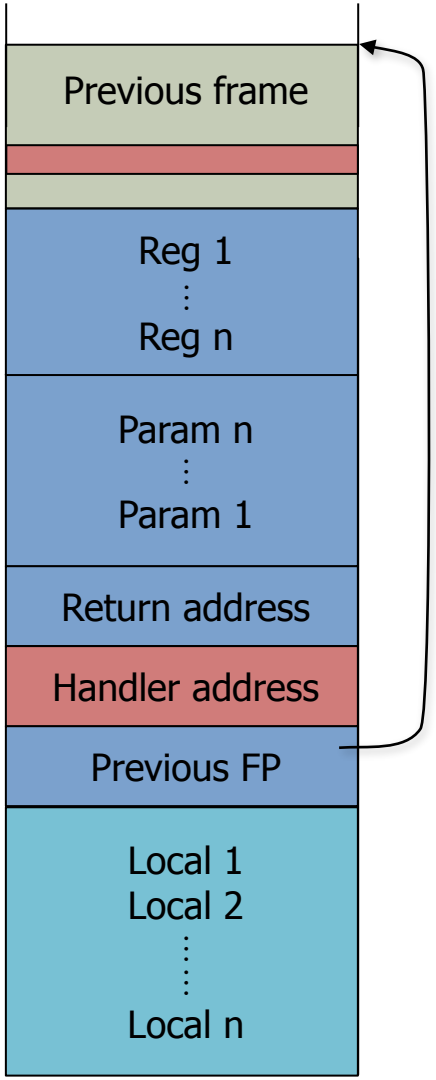
Exception Handling Overhead

- Idea: active handler is known, at compile time, per emitted location

3 ↦ 10
5 ↦ 7
8 ↦ 10
11 ↦ *default_eh*

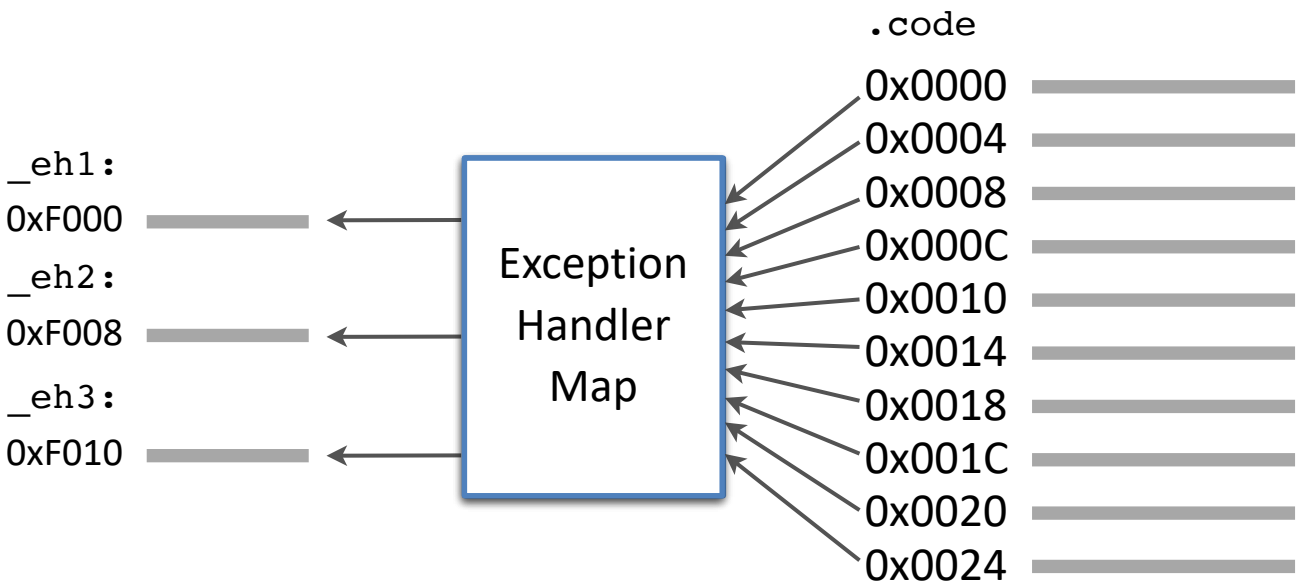
```
1  foo(int n) {
2    try {
3      throw
4      try {
5        throw
6      }
7      catch() { }
8      throw
9    }
10   catch()  { }
11   throw
12 }
```

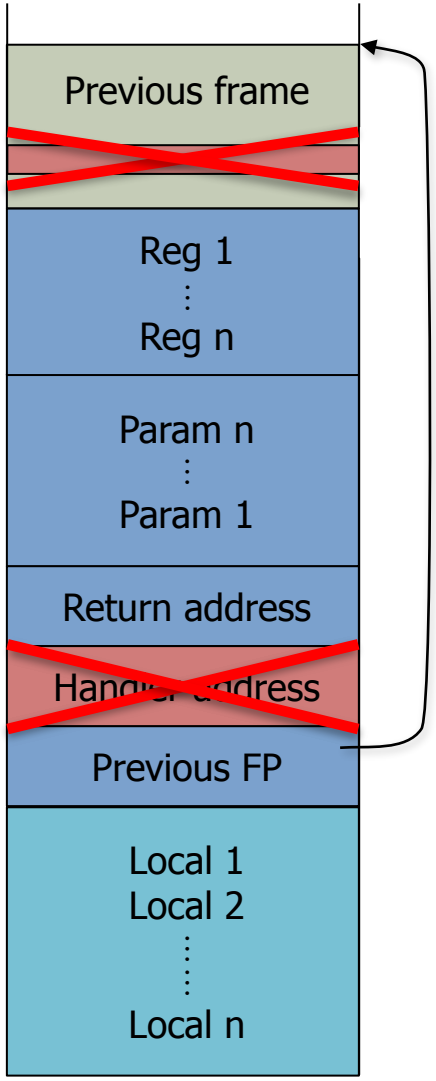
(default)



Exception Handling Overhead

- Solution: store a hash table that maps pc ⟶ handler address.
  - Default eh unwinds, then invokes next eh — looking up by return address.





Unhandled Exception?

```
org.eclipse.swt.SWTException: Graphic is disposed
    at org.eclipse.swt.SWT.error(SWT.java:3744)
    at org.eclipse.swt.SWT.error(SWT.java:3662)
    at org.eclipse.swt.SWT.error(SWT.java:3633)
    at org.eclipse.swt.graphics.GC.getClipping(GC.java:2266)
    at com.aelitis.azureus.ui.swt.views.list.ListRow.doPaint(ListRow.java:260)
    at com.aelitis.azureus.ui.swt.views.list.ListRow.doPaint(ListRow.java:237)
    at com.aelitis.azureus.ui.swt.views.list.ListView.handleResize(ListView.java:867)
    at com.aelitis.azureus.ui.swt.views.list.ListView$5$2.runSupport(ListView.java:406)
    at org.gudy.azureus2.core3.util.AERunnable.run(AERunnable.java:38)
    at org.eclipse.swt.widgets.RunnableLock.run(RunnableLock.java:35)
    at org.eclipse.swt.widgets.Synchronizer.runAsyncMessages(Synchronizer.java:130)
    at org.eclipse.swt.widgets.Display.runAsyncMessages(Display.java:3323)
    at org.eclipse.swt.widgets.Display.readAndDispatch(Display.java:2985)
    at org.gudy.azureus2.ui.swt.mainwindow.SWTThread.<init>(SWTThread.java:183)
    at org.gudy.azureus2.ui.swt.mainwindow.SWTThread.createInstance(SWTThread.java:67)
    ⋮
```

(Java)

Unhandled Exception?

```
Abort
```

(C++)

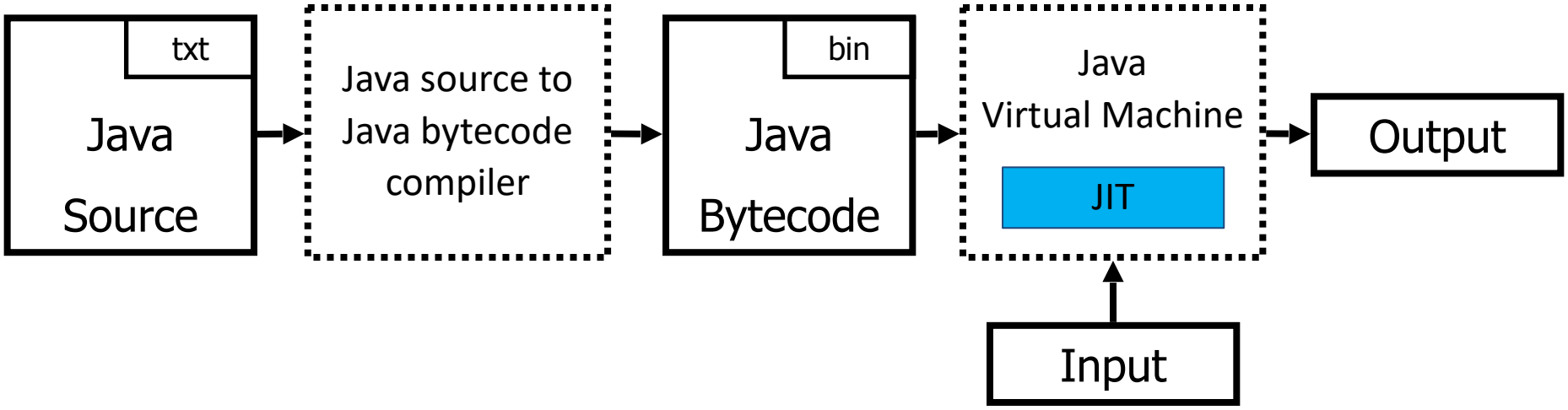
Summary

- Memory Management
  - ✓ Dynamic (= runtime) memory allocation
  - ✓ Manual deallocation
  - ✓ Garbage collection — refcount, mark & sweep, copying
- Object Oriented Features
  - ✓ Overloading — compile time
  - ✓ Inheritance and polymorphism — dispatch vector
- Error handling
  - ✓ Runtime checks
  - ✓ Exception handlers

Aside

Just-in-time Compiler

(Java example)

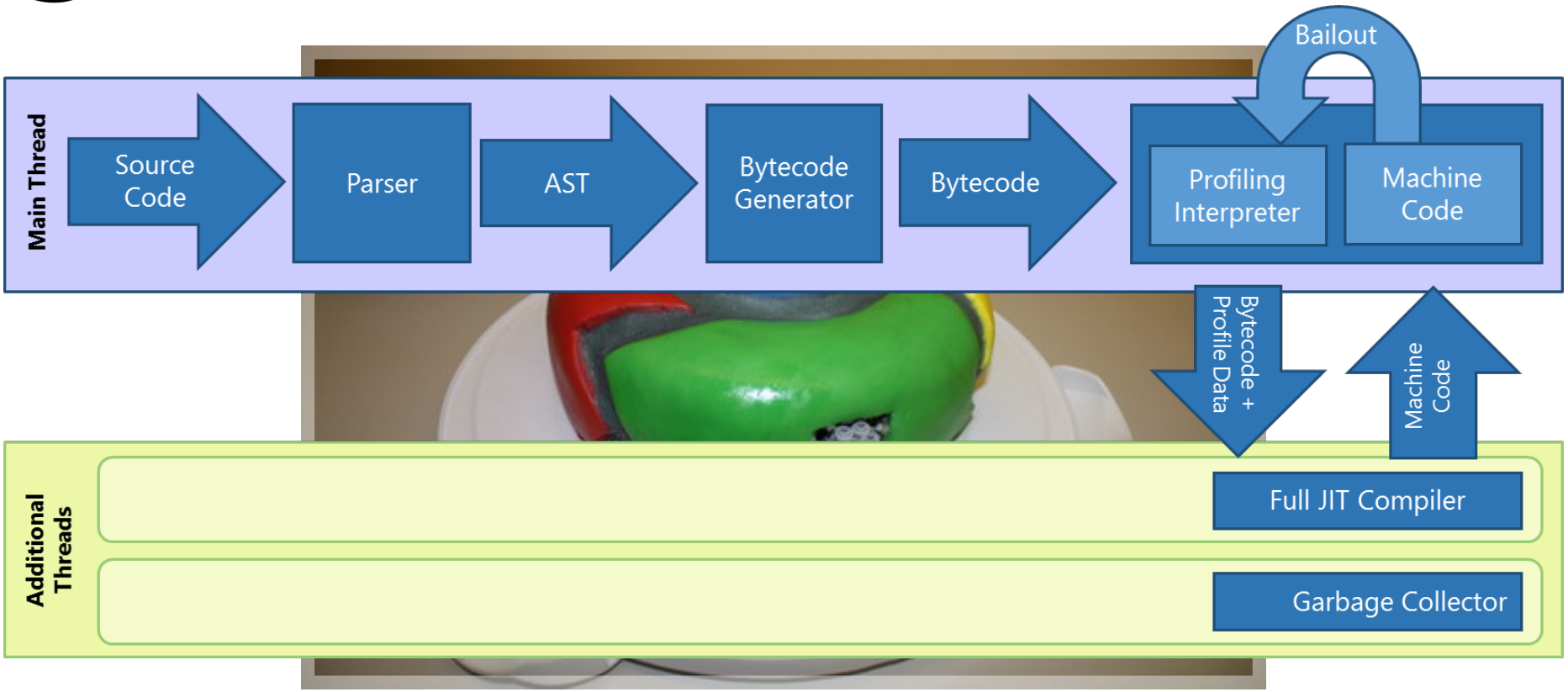


Just-in-time (JIT) compilation: bytecode interpreter (in the JVM) compiles program fragments during interpretation to avoid expensive re-interpretation.

Aside

Just-in-time Compiler

(Javascript example)



- The compiled code is optimized dynamically at runtime, based on ***runtime behavior***

Coming Up