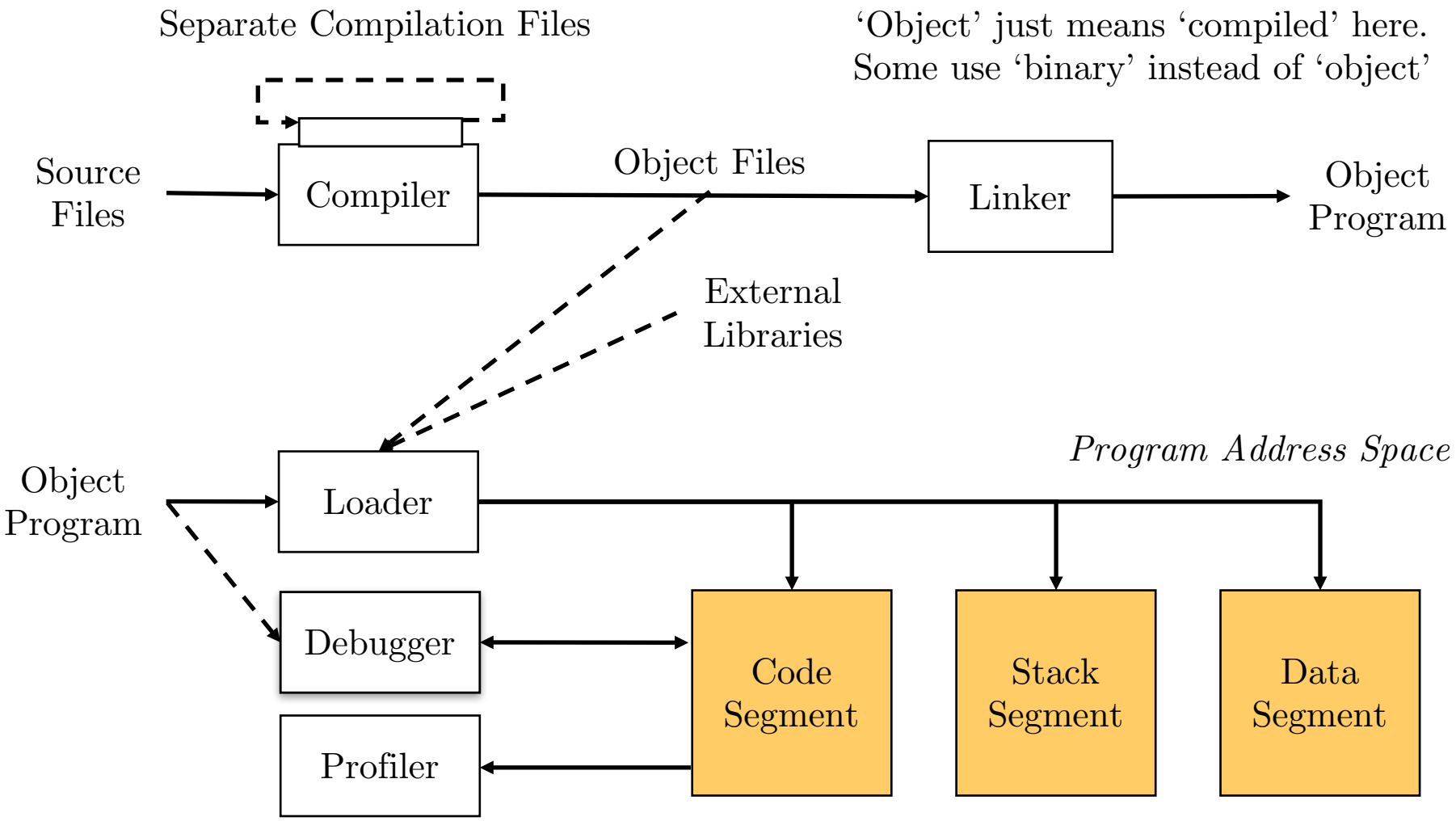
# Runtime (Memory) Organisation

Naranker Dulay
n.dulay@imperial.ac.uk

https://www.doc.ic.ac.uk/~nd/compilers

# Approach


Separate Compilation Files
'Object' just means 'compiled' here.
Some use 'binary' instead of 'object'
Source Files
Compiler
Object Files
Linker
Object Program
External Libraries
Program Address Space
Object Program
Loader
Debugger
Profiler
Code Segment
Stack Segment
Data Segment

# Primitive types

Compilers map variables of primitive types to memory locations and/or registers. They may also **align variables** to memory boundaries in order to keep memory accesses optimal. We'll use memory-based layout to access variables and objects.

| **Primitive Type** | **Typical Sizes(Bytes)** | **Typical Representation** |
|---|---|---|
| **Boolean** | 1 | 0 for false, 1 or non-zero for True |
| **Integer** | 1,2,4,8 | 2's-Complement. |
| **Real** | 4,8,16 | IEEE Floating Point |
| **Char** | 1,2 | 8-bit ASCII or 16-bit UNICODE |

Some languages support **pointer variables** (variables that hold addresses). Ensuring that pointer variables do not hold a 'bad' value is difficult if the language (e.g. C) doesn't support the notion of pointer type safety. Since pointers have largely been superseded by **type-safe object references**, we won't consider the handling of unsafe pointers any further.

# Records/Structures & Arrays

**Records** (structures in C/C++) are data types where a number of variables called fields are grouped together and typically allocated consecutively in memory. Record fields can be of different types and sizes and are accessed by a constant byte offset from the start of the record.

```
Record           = (Field1, ..., FieldK, ..., FieldN)
Size(Record)     = Size(Field1) + ... + Size(FieldN)     # Sizes are in bytes
Address(FieldK) = StartAddress(Record) + Size(Field1) ... + Size (FieldK-1)
```

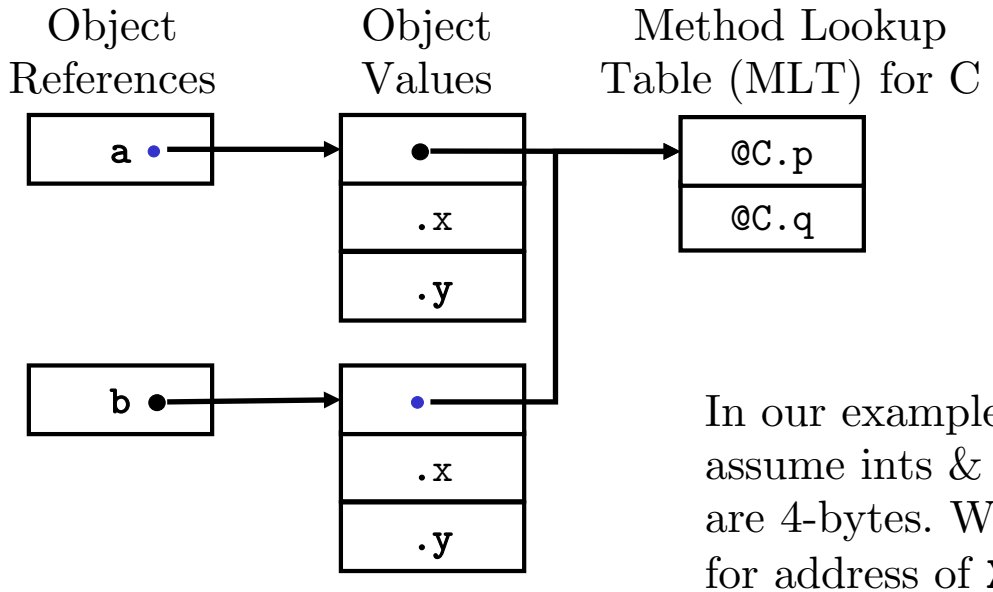
**Arrays** are data types where a number of variables called elements are grouped together and allocated consecutively in memory. The elements of an array are of the same type and size and are accessed by a run-time integer expression that gives the byte offset of the element from the start address of the array. If the size of an array can change then its bounds need to be saved in memory for bound-checking.

```
Array               = ElementType[TotalElements]       #  0 to TotalElements-1
Size(Array)         = Size(ElementType) * TotalElements
Address(Element K) = StartAddress(Array) + K * Size(ElementType)
```

# Objects

The representation of **Objects** is similar to records/structures except that we also need a scheme to handle the additional features of objects, such as methods, inheritance, method overriding and dynamic binding. Many schemes are possible. In our scheme, each object will be represented by a reference (a pointer) to a record on the heap consisting of a reference to the object's class **method lookup table (MLT)** followed by the data fields (instance variables) of the object. For **interface values** see exercise.

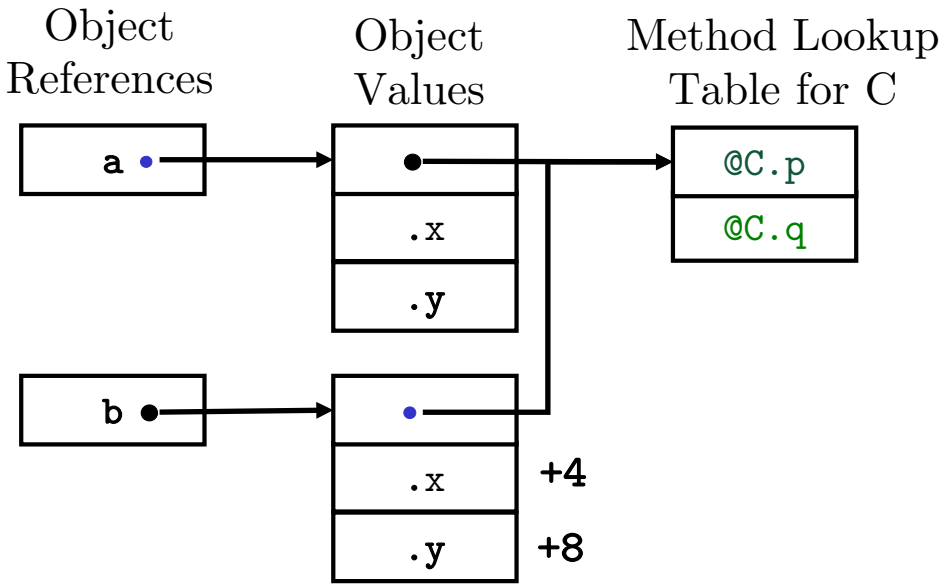
```
class C {
   int x
   int y
   method p ...
   method q ...
}
C a = new C()
C b = new C()
```




In our examples we'll assume ints & addresses are 4-bytes. We'll use `@X` for address of `X`

# Accessing Object Fields

```
class C {
   int x
   int y
   method p ...
   method q ...
}
C a = new C()
C b = new C()
```




We can perform accesses to the fields of an object as follows:

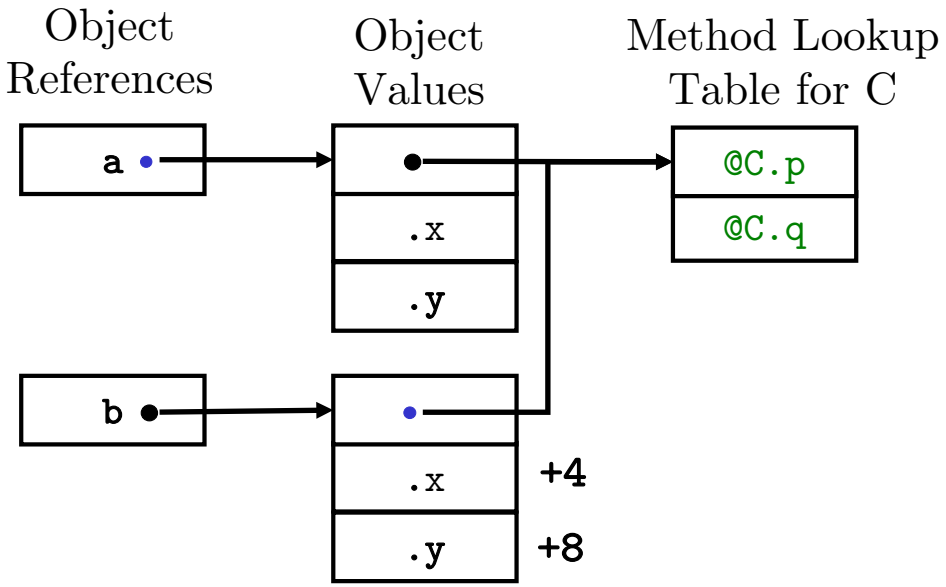
```
Object.Field => Memory[Memory[@ObjectRef] + ByteOffsetToField]
```

*Example:* If addresses & integers are 4 bytes then:

```
b.y => Memory[Memory[@b] + 8]
```

# Calling Methods

```
class C {
   int x
   int y
   method p ...
   method q ...
}
C a = new C()
C b = new C()
```




To call `Object.method(arg1, ..., argN)` we perform the following call instead

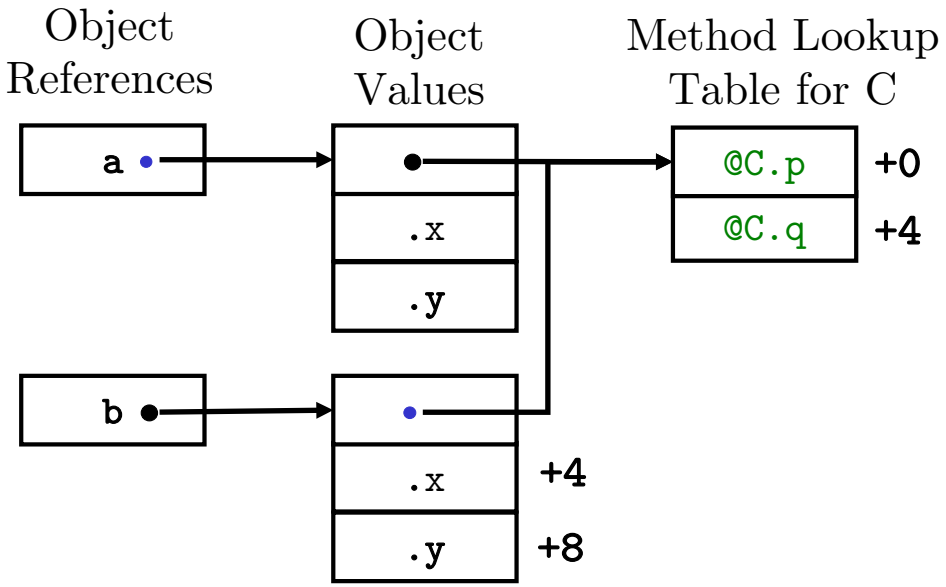
`Class.method(ObjectRef, arg1, ..., argN)` - here `ObjectRef` is passed as a hidden parameter (i.e. *this*, *self*)

Which translates to:

```
CALL Mem[Mem[Mem[@ObjectRef]] + ByteOffsetToEntryInMLT]
```

# Calling Methods II

```
class C {
   int x
   int y
   method p ...
   method q ...
}
C a = new C()
C b = new C()
```



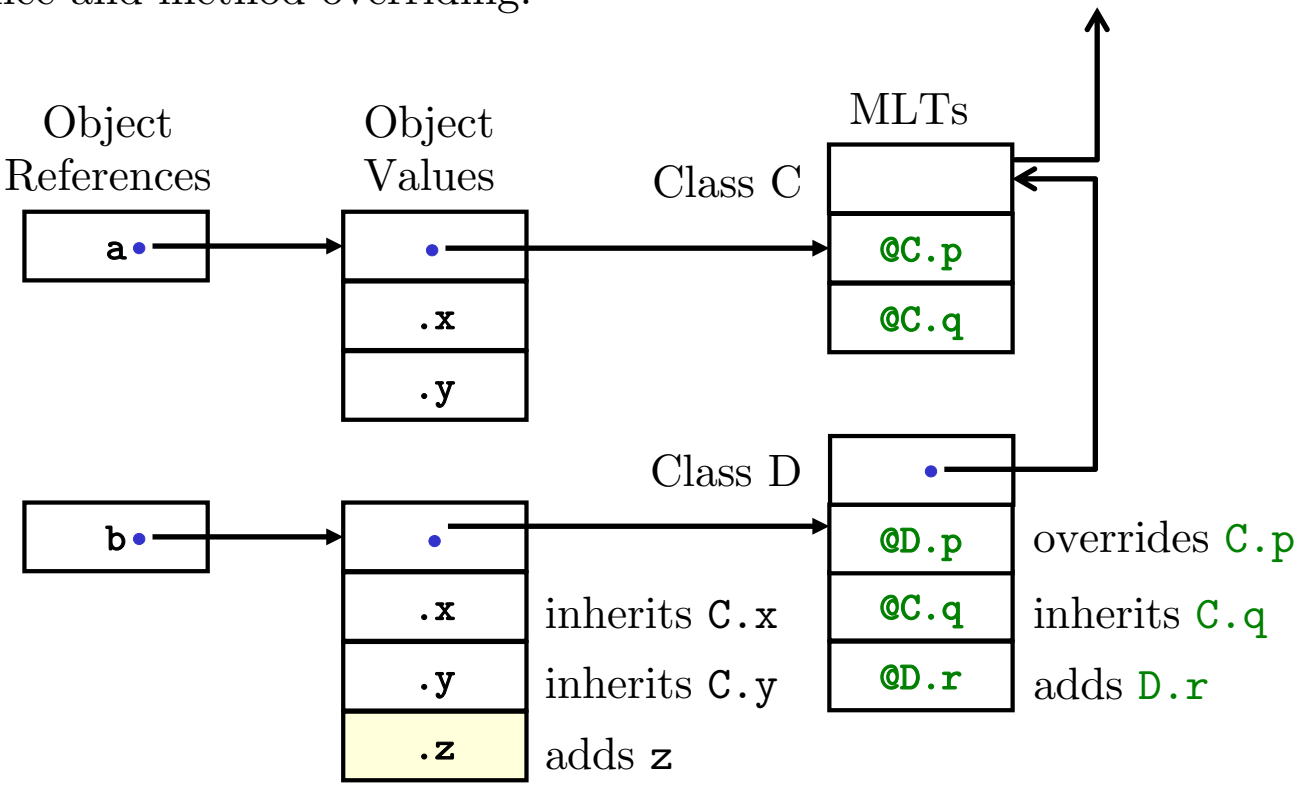

**Example**: On an Intel IA-32 the method call `b.q()` would translate to:

```
mov     eax,   <b>      # Load contents of b into eax
push    eax             # Push ObjectRef as a hidden inner parameter
mov     eax, [eax]      # Load Address of Method Lookup Table into eax
call    [eax + 4]       # Call method q at byte offset 4 in the MLT for Class C
```

# Inheritance & Overriding

The scheme we've used appears over-elaborate, but there is a reason for the design. Let's consider inheritance and method overriding.

```
class C {
   int x
   int y
   method p ...
   method q ...
}

class D extends C
{
   int z
   method p ...
   method r ...
}

C a = new C()
D b = new D()
```

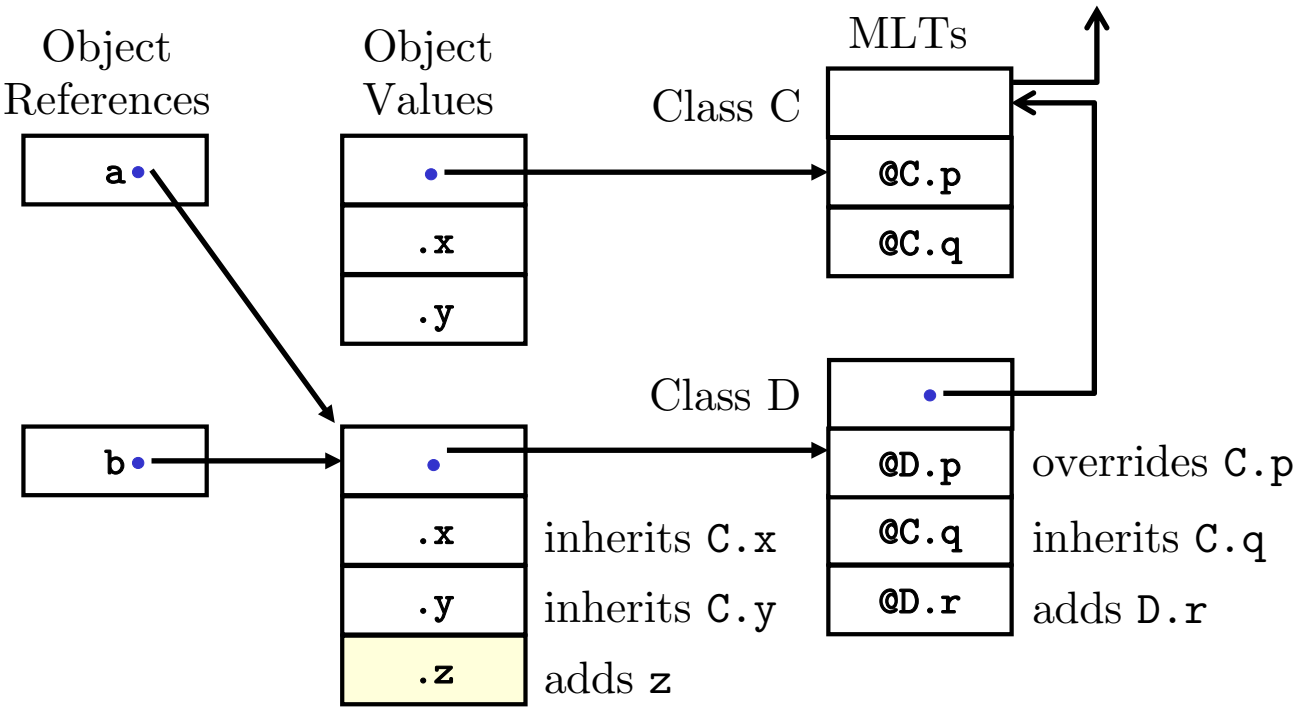



Observe how D's object layout and D's method lookup table layout are extensions of C's. We've also chained the MLTs to allow type-tests for inherited classes. The new pointer could point at a runtime type descriptor for the class instead.

# Dynamic Binding

Provided the object layout for a subclass is an extension of the object layout for its superclass, then we can implement the assignment **a** = **b** by copying **b**'s object reference to **a**. Further, the fields **a.x**, **a.y** are accessed as before with the same byte offset and a call to **a.p** will call the overridden method!

```
class C {
   int x
   int y
   method p ...
   method q ...
}
class D extends C
{
   int z
   method p ...
   method r ...
}
C a = new C()
D b = new D()
a = b
```




Observe that the original 'a' object is now inaccessible.

# Java

What's the output of the following program:

```
class A {
    public int n = 1;
    public int get() { return n; }
}

class B extends A {
    public int n = 2;
    public int get() { return n; }
}
```

```
class javaprog {
  public static void main(String args[]) {
    A a = new A();
    System.out.println("A a = new A()");
    System.out.println(a.get());
    System.out.println(a.n);

    B b = new B();
    System.out.println("B b = new B()");
    System.out.println(b.get());
    System.out.println(b.n);
  }
}
```

# Java Ha Ha!!

What about the following program?

```
class A {
    public int n = 1;
    public int get() { return n; }
}

class B extends A {
    public int n = 2;
    public int get() { return n; }
}
```

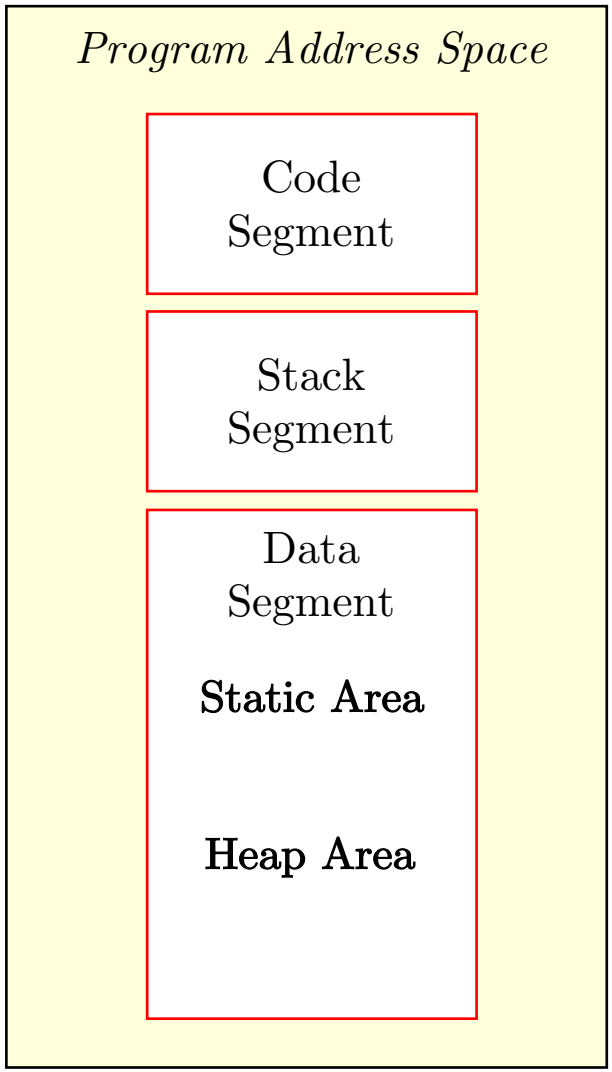
```
class javaprog {
  public static void main(String args[]) {
    B b = new B();
    A a = b;

    System.out.println("a = b");
    System.out.println(a.get());
    System.out.println(a.n);

  }
}
```

# Program Address Space

On modern architectures and operating systems, a compiled program is normally given a large (virtual) **address space**.

This address space is often divided into **segments** based on usage. We'll consider 3 segments only. The **code segment** to hold the compiled program code, the **stack segment** for method calls and to hold local variables, and the **data segment** to hold both global variables (**static area**) and dynamic variables (**heap area**).

Segments can of fixed sizes or allowed to expand at runtime. An OS can often also protect segments, for example, to prevent writes to the code segment, instruction execution from the stack segment. Segments have also been used for mixed language runtime-systems.



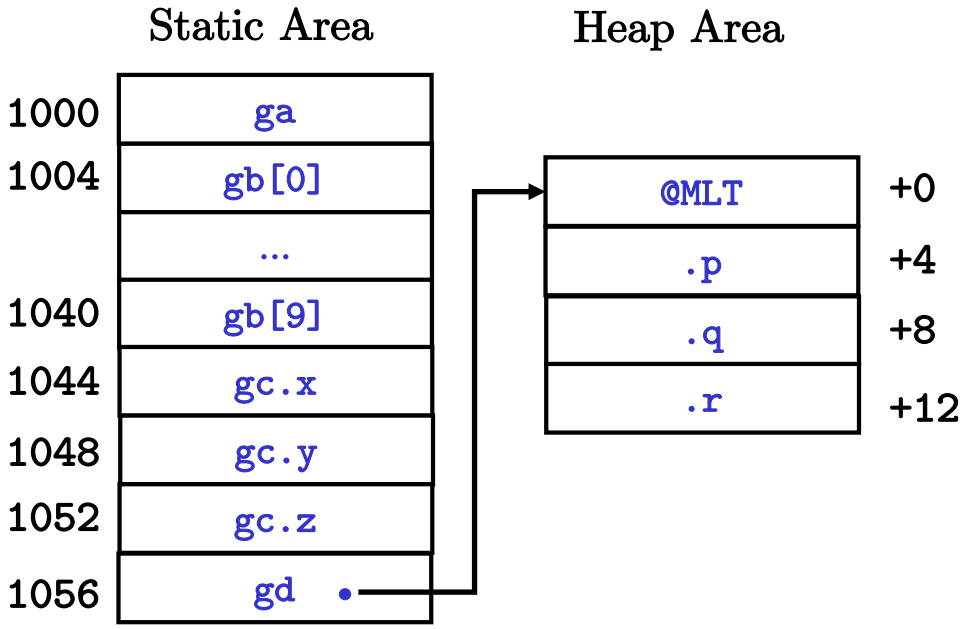
# Global/Static Variables

**Global/Static variables** exist for the lifetime of a program's execution and are normally mapped to 'absolute' locations in the **static area** of the **data segment** and accessed directly via the start address of the allocated location.

**Note:** Constants and Method Lookup Tables can also be held in the static area.

Recall in our examples we're assuming int's & addresses are 4-byte.

```
class  Triple {int p, q, r}
static int ga
static int gb[10]
static struct {int x, y, z} gc
static Triple gd = new Triple()
```

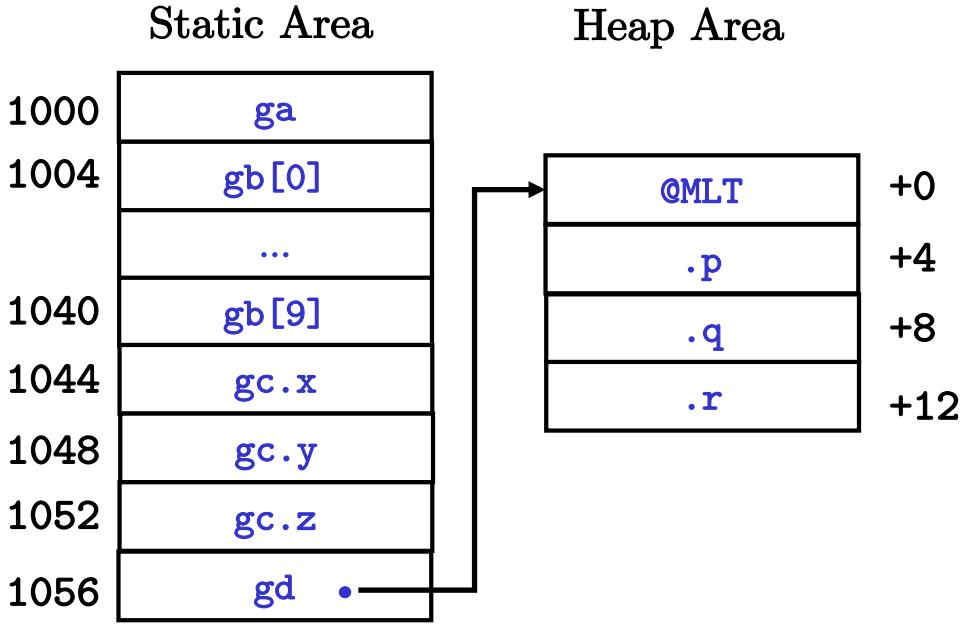


# Global/Static Variables 2

| **Variable** | **Access With** |
|---|---|
| ga | Mem [1000] |
| gb[$k$] | Mem [1004 + 4*$k$] |
| gc.z | Mem [1052] |
| gd.q | Mem [Mem[1056] + 8] |

| **Instr.** | **IA-32 Assembly Code** |
|---|---|
| ga++ | inc [1000] |
| gb[$k$]++ | mov eax, $k$ |
| | inc [1004+4*eax] |
| gc.z++ | inc [1052] |
| gd.q++ | mov eax, [1056] |
| | inc [eax + 8] |

```
class  Triple {int p, q, r}
static int ga
static int gb[10]
static struct {int x, y, z} gc
static Triple gd = new Triple()
```



# Local Variables

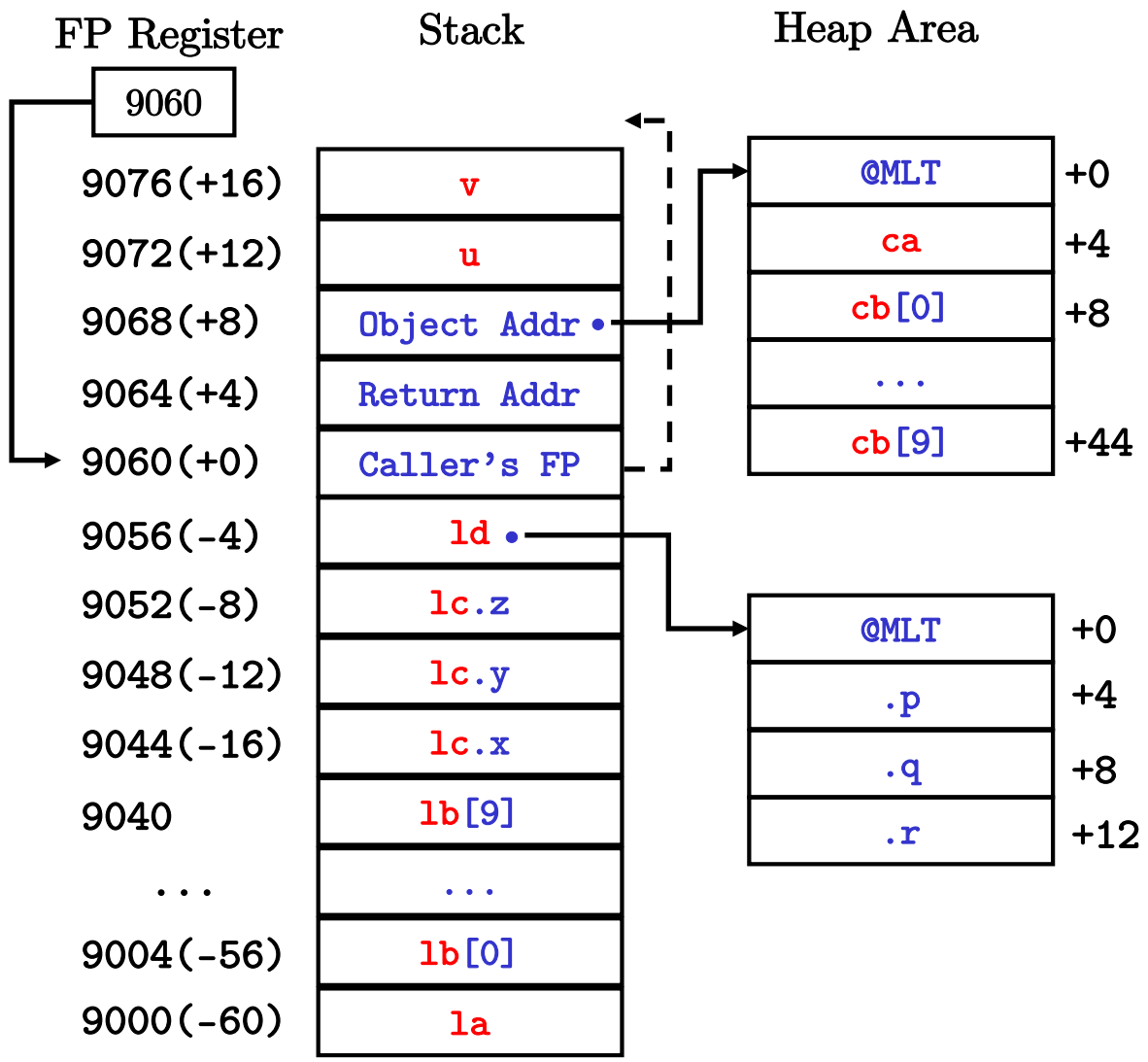
- **Local variables** (those declared within a method) map to locations in the **stack segment** and exist only for the duration of the method call in the stack frame of the method.
- If a method is called recursively, each activation of the method gets its own stack frame with its own set of the local variables.
- Access to local variables is via a **Frame Pointer Register** FP (EBP register on the Intel IA-32) and a byte offset. The FP points to a location in the stack frame and the offset gives the relative position of the local variable from the FP location.
- The FP is set on entry and restored on exit from the method.
- **Parameters** are often also found in the stack frame and can be accessed in the same way as local variables.
- In addition to the local variables and parameters, the stack frame for a method activation holds:

  - the method's **return address**
  - the *address of the method's object* (e.g. **this** in Java) - this address is not needed for functions (static methods in Java)
  - the *saved values of any registers* which are used in the method (incl. the Caller's FP) Saved registers are restored on method exit.
  - space for function return value if not returned in a register

# Example

```
class Triple {int p, q, r}
class MyClass {
   int ca
   int cb [10]
   method M(int u,v) {
     int la
     int lb[10]
     struct {int x, y, z} lc
     Triple ld = new Triple()
   }
}
```

| Var | Access With |
|---|---|
| v | Mem [FP + 16] |
| lb[*k*] | Mem [FP - 56 + 4**k*] |
| ld.q | Mem [Mem[FP-4] + 8] |
| ca | Mem [Mem[FP+8] + 4] |
| cb[*k*] | Mem [Mem[FP+8]+8+4**k*] |



# Dynamic/Heap Variables

**Dynamic variables** (object instances and arrays in Java) are normally mapped to locations in the **heap area** of the data segment and sometimes referred to as **heap variables**.

Dynamic variables are allocated when the program (explicitly or implicitly) calls an allocator function (**`new`** in Java, **`malloc`** in C) and de-allocated either when the program calls an explicit de-allocator function (**`free`** in C) or automatically when the runtime system detects that the dynamic variable is no longer accessible from any place in the program.

This latter approach requires a so-called **garbage collector** to be active at run-time that can reclaim such 'garbage' ("dead" objects)

# Explicit Heap Allocation

**Allocate(int N) → address**

Search a List of "Free Memory Blocks" for a block of size N or a bigger block
If a free block of exactly size N is found:
*Remove block from FreeList*
*Return start address of block*
Otherwise if a free block bigger than N is found :
*Split block into N-sized block and residual block*
*Leave residual block in Free List*
*Return start address of N-sized block*
If the Search did not return a suitable free block :
*Request more heap memory from the Operating System & allocate*
If the OS does not comply:
*Report failure e.g. return None or raise an exception*

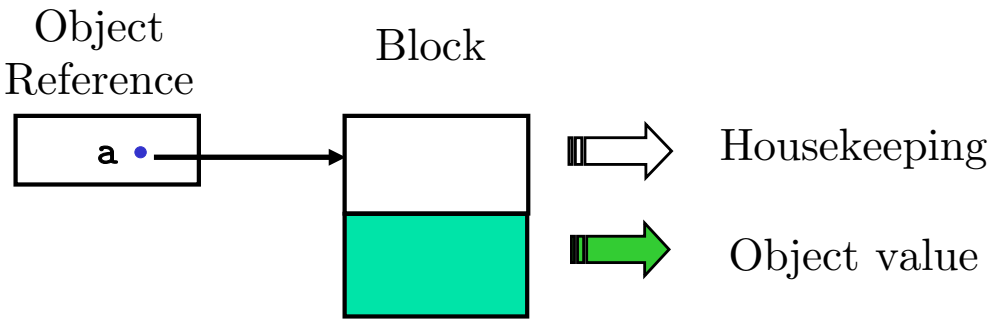
---

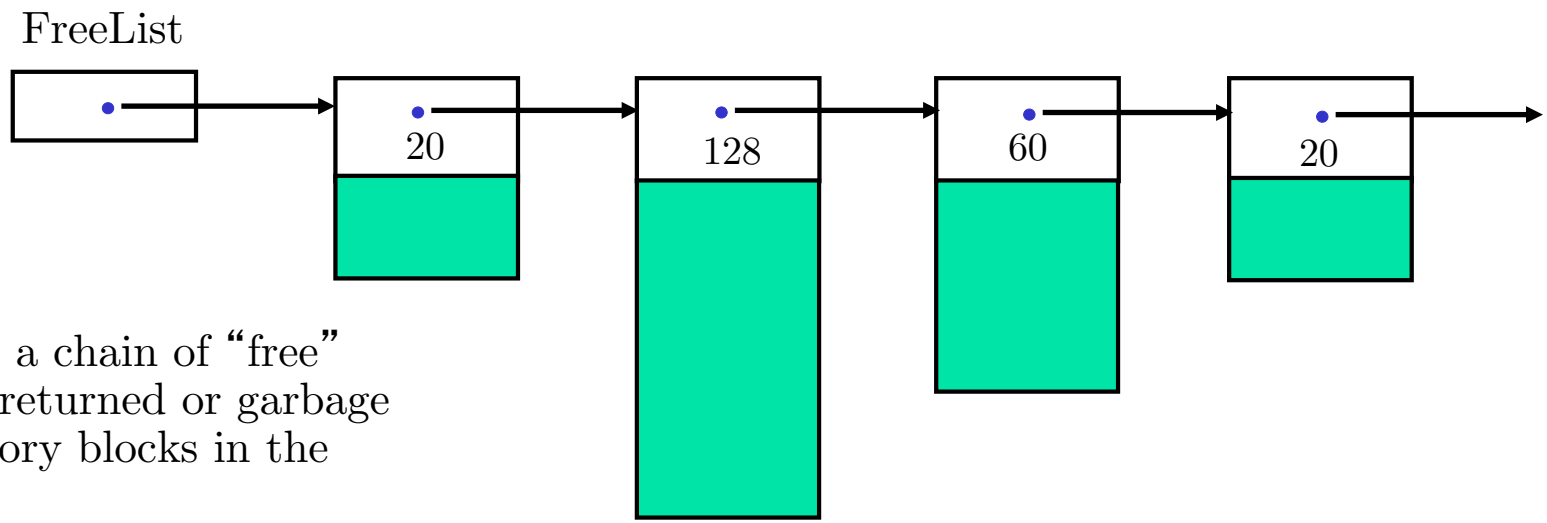
**DeAllocate(address)** or sometimes **Deallocate(address, blocksize)**

Return block at given address to list of Free Memory Blocks
***Coalesce*** *returned block with adjacent block(s) if possible => reduces* ***fragmentation***

# Heap Blocks and the Freelist

Each heap block is typically prefixed with a small amount of **housekeeping** data that is used for heap management, e.g. the size of the block, and a pointer to the next free block. Note: object references point to the start of the object not the start of the block.




The **FreeList** is a chain of "free" (i.e. unused or returned or garbage collected) memory blocks in the heap area.

# Explicit Heap Allocation II

**Even with this simple scheme, many improvements are possible:**

- Maintain different free lists for different block-size intervals -> Faster allocation.

- Allocate blocks that are just big enough.

- For extensible arrays, increase the array size by a constant size.

**Other issues to consider are:**

- Memory alignment. Some architectures require certain data types to be aligned, for example on the SPARC, 64-bit IEEE values must be aligned on 64-bit boundaries.

  Even when alignment is not required, not-aligning can lead to a noticeable performance loss.

- Locality of reference

# Garbage Collection

Modern programming languages (rightly) prevent programmers from manipulating pointers/addresses directly. They free programmers from the burden of ensuring that dynamic variables are properly de-allocated.

This liberation has led to the adoption of garbage collection (GC) as a core component of the runtime system of a programming language.

Alas like sorting algorithms there are many, many GC algorithms, each with its own pluses and minuses.

# GC Requirements

**Correctness**

Bugs in GC can be extremely hard to identify, its critical that the GC works correctly, e.g. that the GC never collects live data, that the GC will always make progress even in low memory situations.

**Speed**

Programmers demand that the GC is fast. Pauses due to the GC are not acceptable in some applications.

**Low Memory Overhead**

The additional memory costs of the GC should be low, particularly for languages that create/destroy many small objects.

**Types**

GC algorithms can be 1-shot (stop-the-world), on-the-fly, concurrent

**Compiler Support**

In order to perform GC, the compiler needs to provide the Garbage Collector with details of:

- Which variables in the program point into the heap (and their types).
- For each block (e.g. object) pointed to, what pointers does it contain, the position of those pointers in the block and the type of the blocks that they point to.
- These details can be provided either in 'type descriptor data' that the compiler generates and/or via special routines that the GC can call and the compiler generates for the program.

# Heap Compaction

After GC, the heap may contain many small blocks which may mean that its impossible to allocate larger blocks (Fragmentation problem again).

It's useful to move all 'live' blocks together, which effectively allows the free-blocks to be replaced by one large free-block.

Heap compaction is integrated in some fashion in most garbage collectors.

Compaction can be done in 3 phases:

- the 1st phase marks the live blocks
- the 2nd phase compacts the heap by co-locating 'live' blocks.
- the 3rd phase updates the pointers that referred to co-located blocks

# Reference-Counting GC

In Reference-Counting GC we record in the housekeeping part of each block the number of pointers (i.e. references) that point to the block.

Whenever a reference to a block is copied, e.g. in an assignment or passed to a parameter we increment the count.

Whenever a reference to the block is removed, e.g. a variable is re-assigned or when a method returns and its local variables holding references are removed we decrement the count.

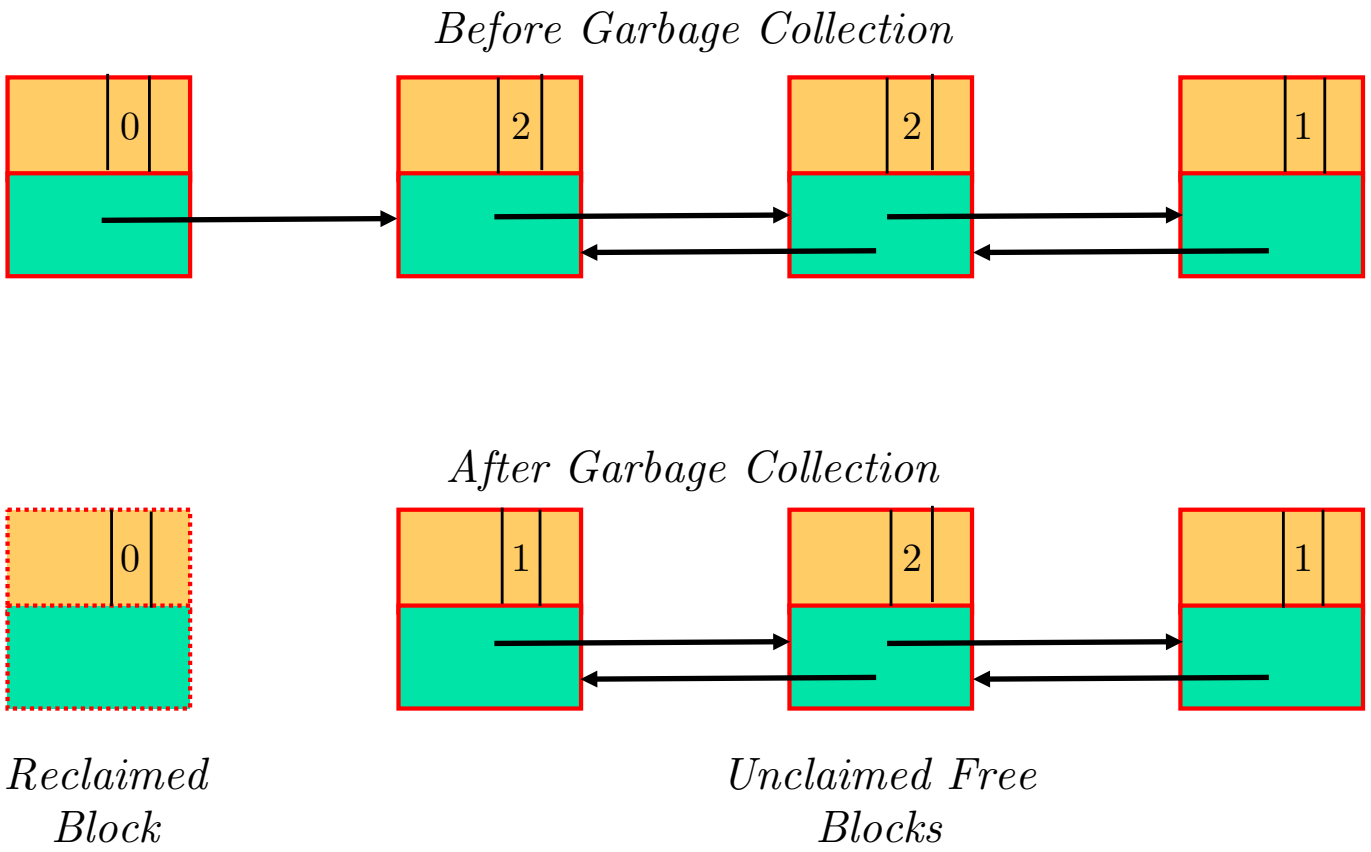
Reference-counting GC is simple and fairly efficient, but requires the compiler to generate extra code for all pointer manipulations. More problematic is that reference-counting GCs need techniques to reclaim cyclic data structures.

Example: For the reference assignment $p = q$, the compiler generates:

```
if q points into the heap:
        q.count ++
if p points into the heap:
        p.count --
        if p.count == 0:
            # reclaim p's block & recursively reclaim any linked block with zero count
p = q   # finally copy reference
```

# Cyclic Data Structures

Although reference-counting GC is easy to implement it cannot reclaim cyclic data structures, without special techniques to detect cycles. Fpr example:



# Mark-Sweep GC

Unlike reference-counting GC, **mark-sweep GC** will reclaim all dead blocks (all garbage).

**Mark phase:** Mark as live, all blocks that are reachable from non-heap references (from top-level global/local heap variables).

**Sweep phase**: Scan all blocks, reclaiming dead blocks and unmarking live blocks (for next collection).

When combined with compaction, mark-sweep provides the largest possible block available at that time.

The housekeeping area of a block needs 1 bit to indicate 'marked'. Alternatively we can keep mark bits in a bitmap.

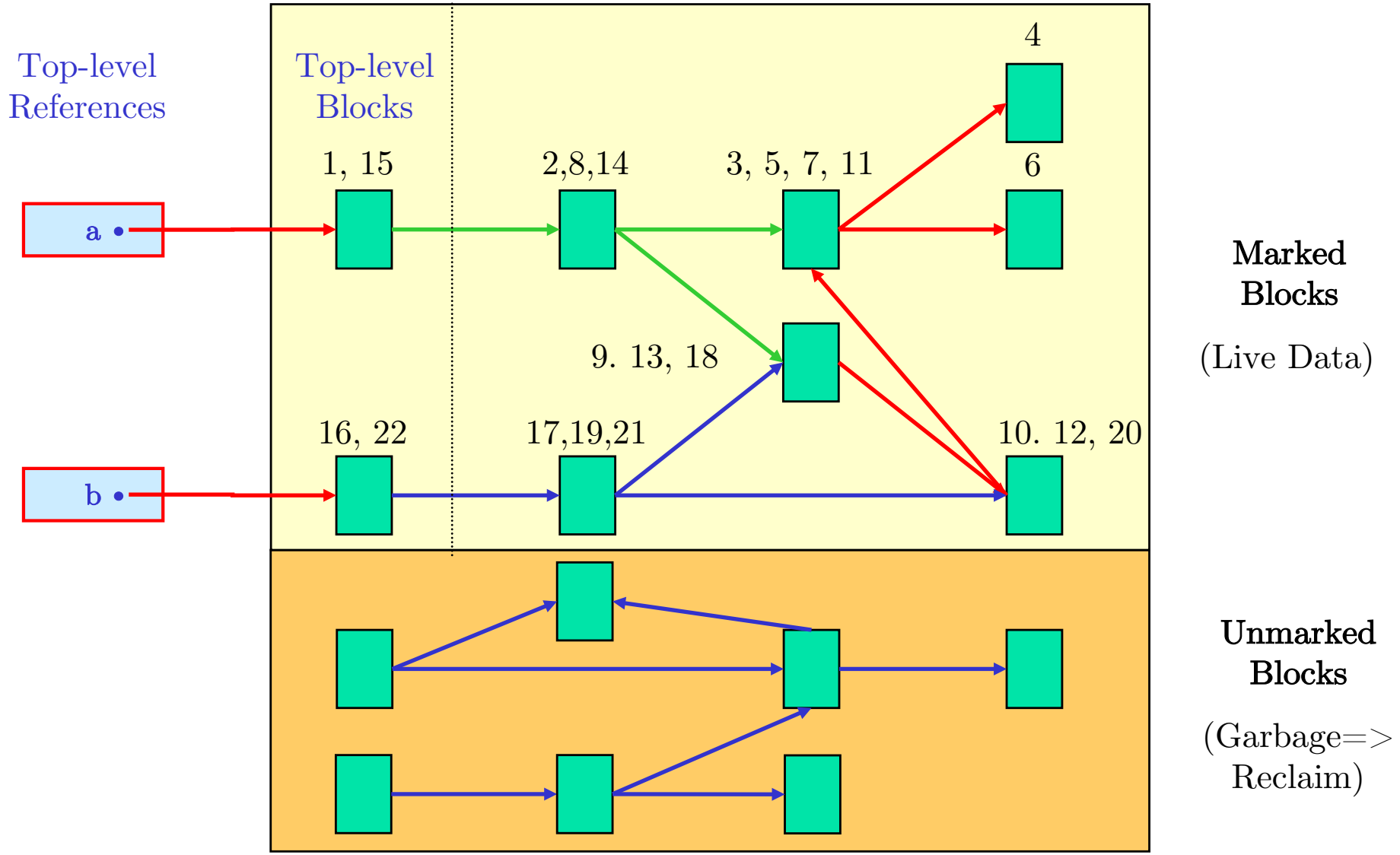
```
def New(size):
    if FreeList == None:
        Mark_Sweep()
    return Allocate(size)
```

```
def Mark_Sweep()
    foreach Global-Local-HeapVariable Var:
        Mark(Var.block) # as live
    Sweep()
    if FreeList == None:
        Error("Out of memory")
```

```
def Mark(Block):
    if Block.MarkBit == 0: # 0=not marked
        Block.MarkBit = 1:  # 1=marked(live)
    foreach ChildRef in Block.Children:
        Mark(ChildRef)
```

```
def Sweep():
    Block = HeapStart
    while Block < HeapEnd:
        if Block.MarkBit == 0:    # dead block
            Deallocate(Block)     # reclaim
        else Block.MarkBit = 0    # reset
        Block = Block + Block.size
```

# Example


Top-level References
Top-level Blocks
a
b
1, 15
2,8,14
3, 5, 7, 11
4
6
9. 13, 18
16, 22
17,19,21
10. 12, 20
Marked Blocks
(Live Data)
Unmarked Blocks
(Garbage=> Reclaim)

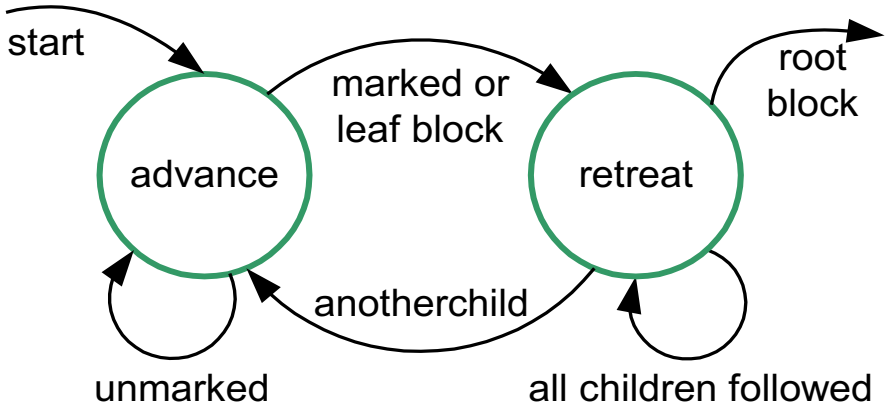
# Pointer-Reversal Marking

Marking is a recursive process and potentially costly in low-memory situations. Pointer reversal (Schorr and Waite) is a clever technique that allows us to visit all nodes of a Directed Graph without additional stack space:

While visiting a child block C of a block P, the parent pointer of P (grand-parent) is stored in the location in P which held the pointer to C.

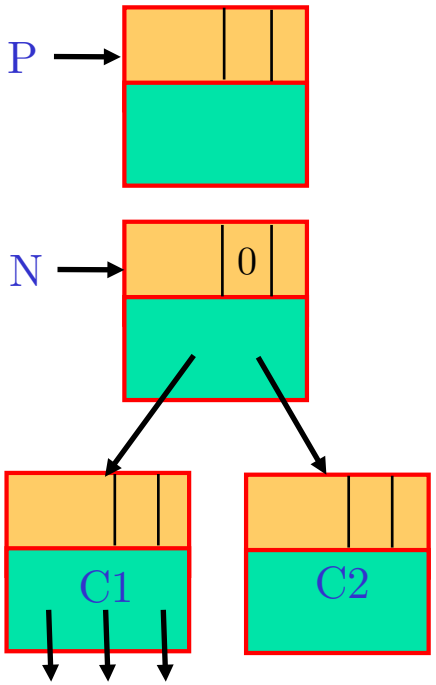
The visit to the child block C returns a pointer to C, which is used to restore the pointer in P by swapping it with the parent pointer.

This then restores the parent pointer which can then be swapped with the location of the next pointer in C.

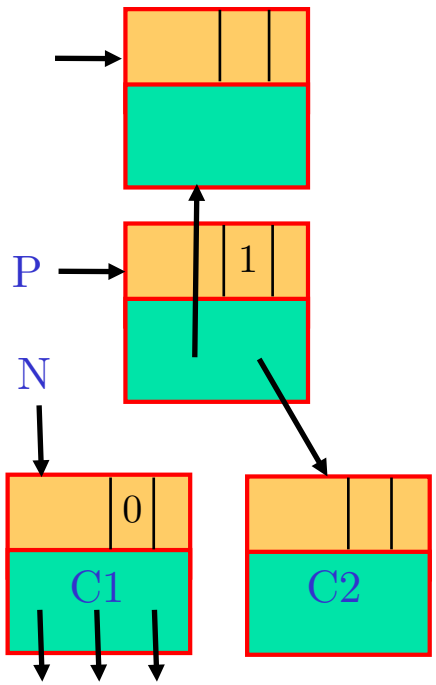



The housekeeping data in blocks needs to keep a count of the number of marked children.

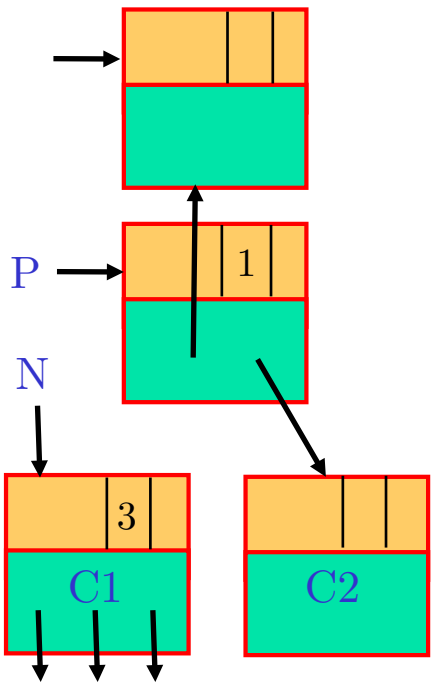
# Example



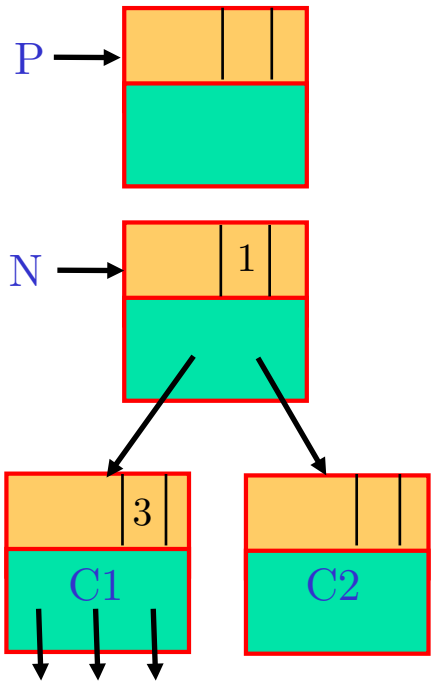

Before advance to C1




Advance to C1

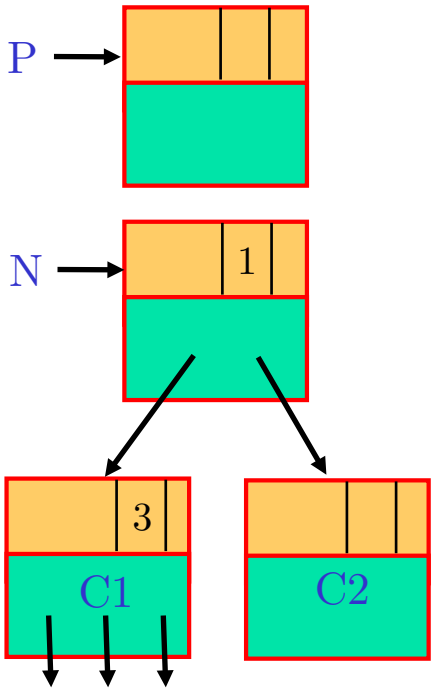



About to retreat from C1. *Note 3 children of C1 were followed*

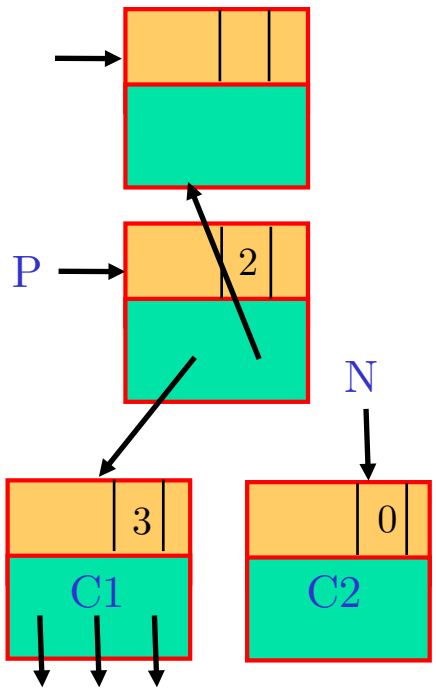



Back at N, about to advance to C2

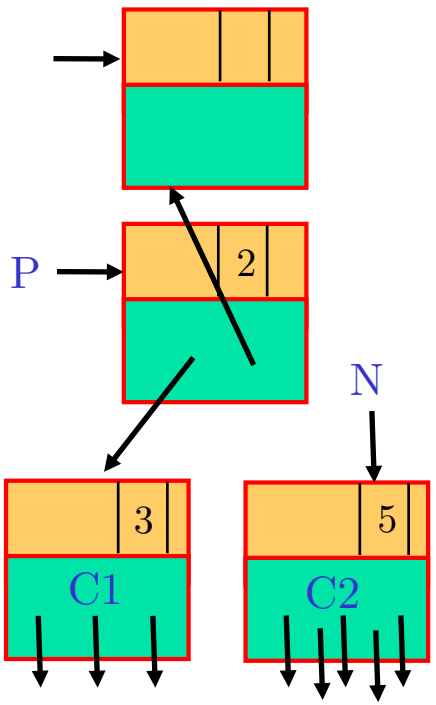
# Example Continued



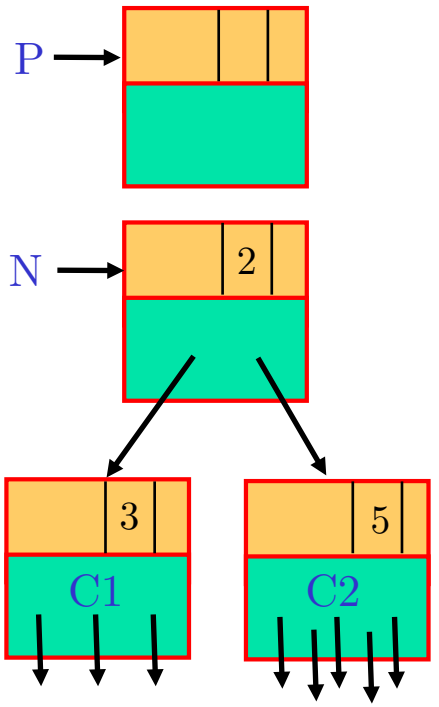

Before advance to C2




Advance to C2




About to retreat from C2. *Note 5 children of C2 were followed*



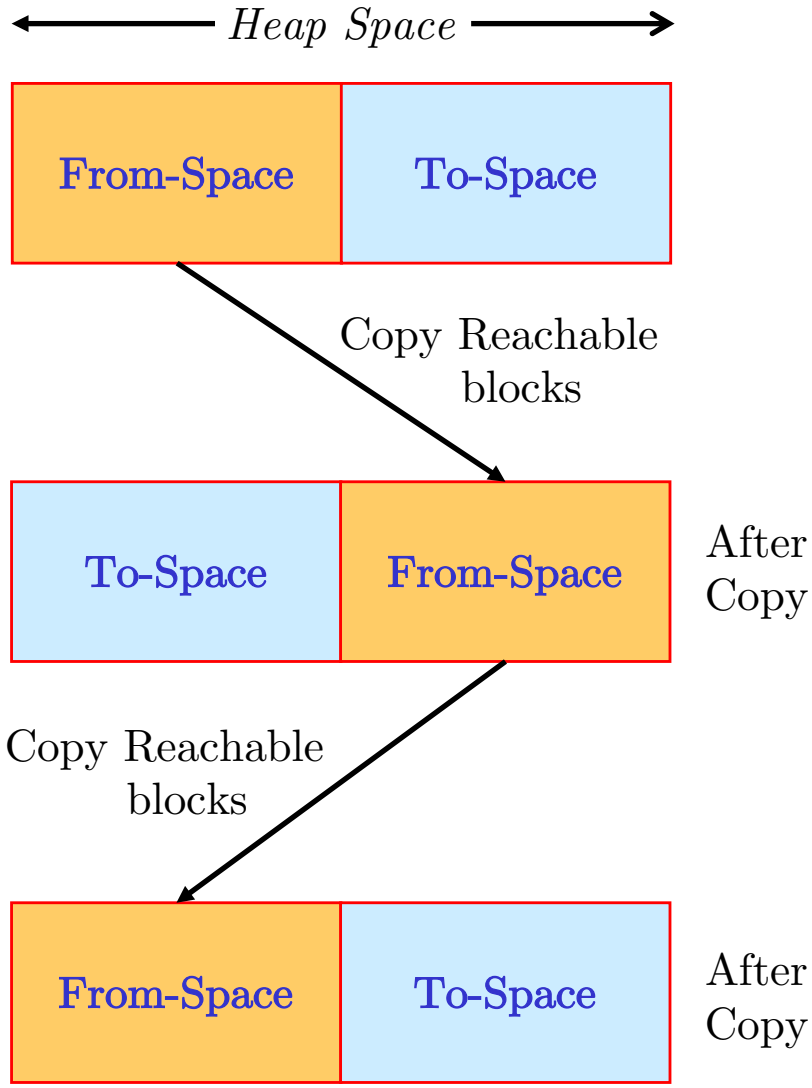

Back at N, about to retreat to P

# Two-Space GC

In two-space GC we split the heap into two spaces, the **From-Space** and the **To-Space**. We allocate blocks from the From-Space. When the From-Space is exhausted, we copy live (reachable) blocks from the From-Space to the To-Space, and then the swap roles of From-Space and the To-Space.

Fast. No pointer manipulations. Copying automatically compacts the heap. Wastes half of the memory and will copy blocks that are long-lived.

Co-locating linked blocks on copying improves locality of reference for memory caches and virtual memory.



# Generational GC

Newly created objects usually die young, while objects that survive many collections often survive even longer.

In generational GC we divide the heap into several areas (generations) based on age of block (object).

Allocate new blocks from the youngest generation and move objects to older generation based on age of block.

Perform GC on youngest generation more often then next generation, etc.

Can apply different GC techniques to different generations, e.g. can use Mark-and-sweep GC for youngest generation and 2-space GC for next generation.

Java 8 has 4 garbage collectors to choose from:

- Serial, Parallel, Concurrent Mark-Sweep, and Garbage First (G1)

One is of these is selected depending on the amount of memory and the number of cores available. However developers and users can select a different GC and configure it to better support application requirements. For futher details see:

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html

# Debuggers

- **Post-mortem debugging**
- **Interactive debuggers:**
  For natively compiled programs, we need support from the Architecture/OS in order to set *breakpoints* on machine instructions
- Debuggers should not affect the behaviour of a program except perhaps its execution time.
- Require information about identifiers and a mapping of the program counter to source lines from the compiler.
- Optimising compilers and debuggers do not always co-exist.
- How to support mixed language debugging, concurrent and distributed programs.

Reverse look-up Program Counter to Source Line

- **Stack Traceback** returns:
  – Name of method
  – Values of its local variables
  – Name of calling method, its locals
  etc.
- Contents of global variables
- Contents of dynamic variables
- Note: variables may be in registers at various times - the debugger needs to know when variables are in registers and when in memory.
- Etc.

# Profilers

**Profilers** allow us to identify which parts of a program are taking the most time.

Before optimising code however, remember that its:

- more important to get code right before trying to make it fast
- better to find a faster algorithm than tinker with a correct but slow algorithm
- worth leaving optimisations to an optimising compiler.
- worth considering faster hardware instead

Interrupt program every X milliseconds/microseconds.

Identify the method that was executing, e.g. Lookup value of Program Counter in a table sorted by start address of Method

| Method | StartAddr | Counter |
|---|---|---|
| D.A () | 1000 | 0 |
| D.B () | 1200 | 0 |
| D.C () | 4500 | 0 |

Increment a counter for the method.

On program completion, print out or graph counts.

# Summary

In addition to generating code for a program, compilers also need to systematically organise the various data objects that will exist when the program executes. At run-time these objects may need be managed by a garbage collector or inspected by a debugger, or a program that uses reflection. In order to support such components, compilers need to generate additional information such as a type-descriptions of data objects (for garbage collectors, debuggers, and reflection calls from the program), and mappings of program counter ranges to source line numbers for profilers and debuggers.

**The demands put on a compiler (writer) are endless !**

For more details on runtime organisation see [Cooper – Chapters 6&7 , Appel – Chapters 13&14, Aho Chapter 7]

For detailed information on garbage collection algorithms a good source is '**The Garbage Collection Handbook: The Art of Automatic Memory Management**' by Richard Jones, Antony Hosking, Eliot Moss, 2011. Chapman and Hall.