Languages and Compilers (SProg og Oversættere)

Lecture 17
Storage Allocations and Run Time Management

Bent Thomsen
Department of Computer Science
Aalborg University

Learning goals

- Understand
  - Data representation (direct vs. indirect)
  - Storage allocation strategies:
    - static vs. dynamic (stack and heap)
  - Activation records (sometimes called frames)
  - Why may we need heap allocation
- Gain an overview of
  - Garbage collection strategies (Types of GCs)

Issues in Code Generation

- ~~Code Selection:~~

  ~~Deciding which sequence of target machine instructions will be used to implement each phrase in the source language.~~

- Storage Allocation

  Deciding the storage address for each variable in the source program. (static allocation, stack allocation etc.)

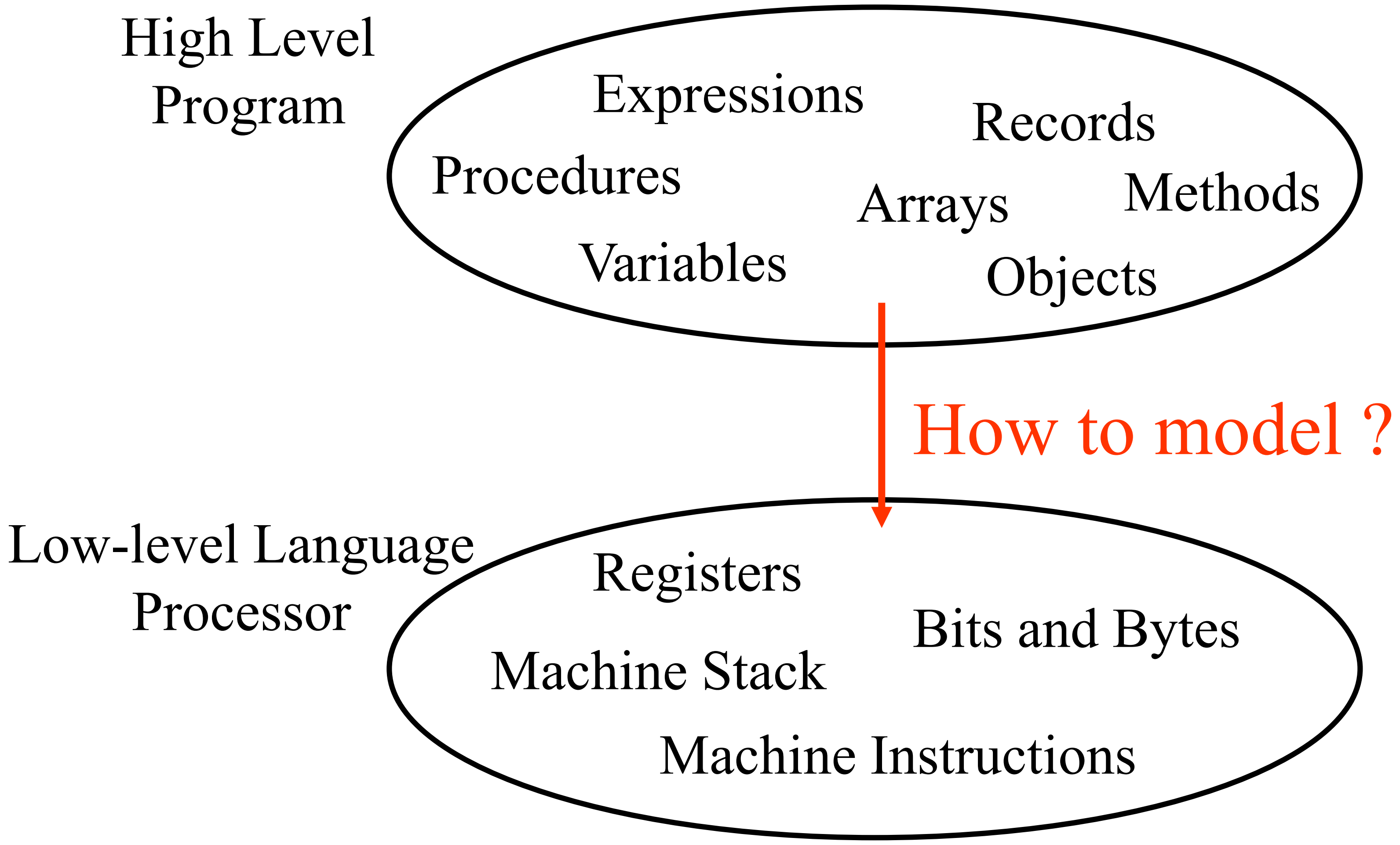
- ~~Register Allocation (for register-based machines)~~

  ~~How to use registers efficiently to store intermediate results.~~

We will look at register allocation in later lectures

What are (some of) the issues

How to model high-level computational structures and data structures in terms of low-level memory and machine instructions.



Easy for Java (or Java like) on the JVM

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	`L`*t*`;`
Array of type *a*	`[`*a*

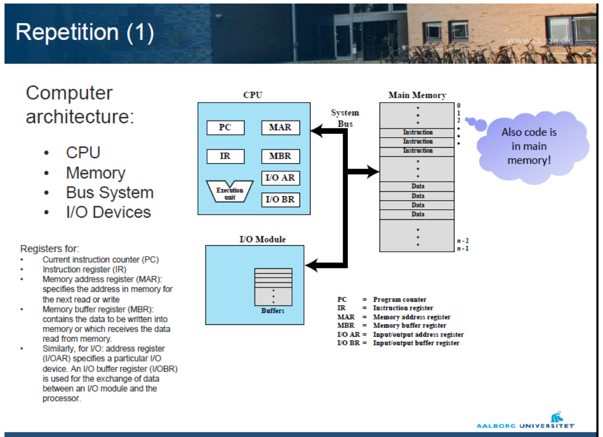
Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

For other Languages on the JVM some thoughts
Are needed on a suitable mapping

Back in the olden days....

- No memory organization
- Programs had access to all of memory
- Memory was one big array of bytes
- No distinction between code and data

- Not just so in the old days also so for:
  - Low level VMs
  - Assember/Machine code
  - Connection with the CART and PSS courses:



Repetition (1)

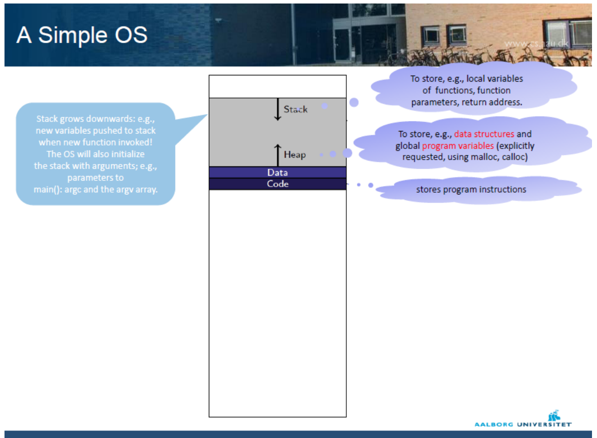
Computer architecture:

- CPU
- Memory
- Bus System
- I/O Devices

Registers for:
- Current instruction counter (PC)
- Instruction register (IR)
- Memory address register (MAR): specifies the address in memory for the next read or write
- Memory buffer register (MBR): contains the data to be written into memory or which receives the data read from memory.
- Similarly, for I/O: address register (I/OAR) specifies a particular I/O device. An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the processor.

PC = Program counter
IR = Instruction register
MAR = Memory address register
MBR = Memory buffer register
I/O AR = Input/output address register
I/O BR = Input/output buffer register

www.cs.aau.dk
CPU
PC
MAR
IR
MBR
I/O AR
I/O BR
Execution unit
System Bus
Main Memory
0
1
2
Instruction
Instruction
Instruction
Data
Data
Data
Data
n - 2
n - 1
Also code is in main memory!
I/O Module
Buffers
AALBORG UNIVERSITET



A Simple OS

Stack grows downwards: e.g., new variables pushed to stack when new function invoked! The OS will also initialize the stack with arguments; e.g., parameters to main(): argc and the argv array.

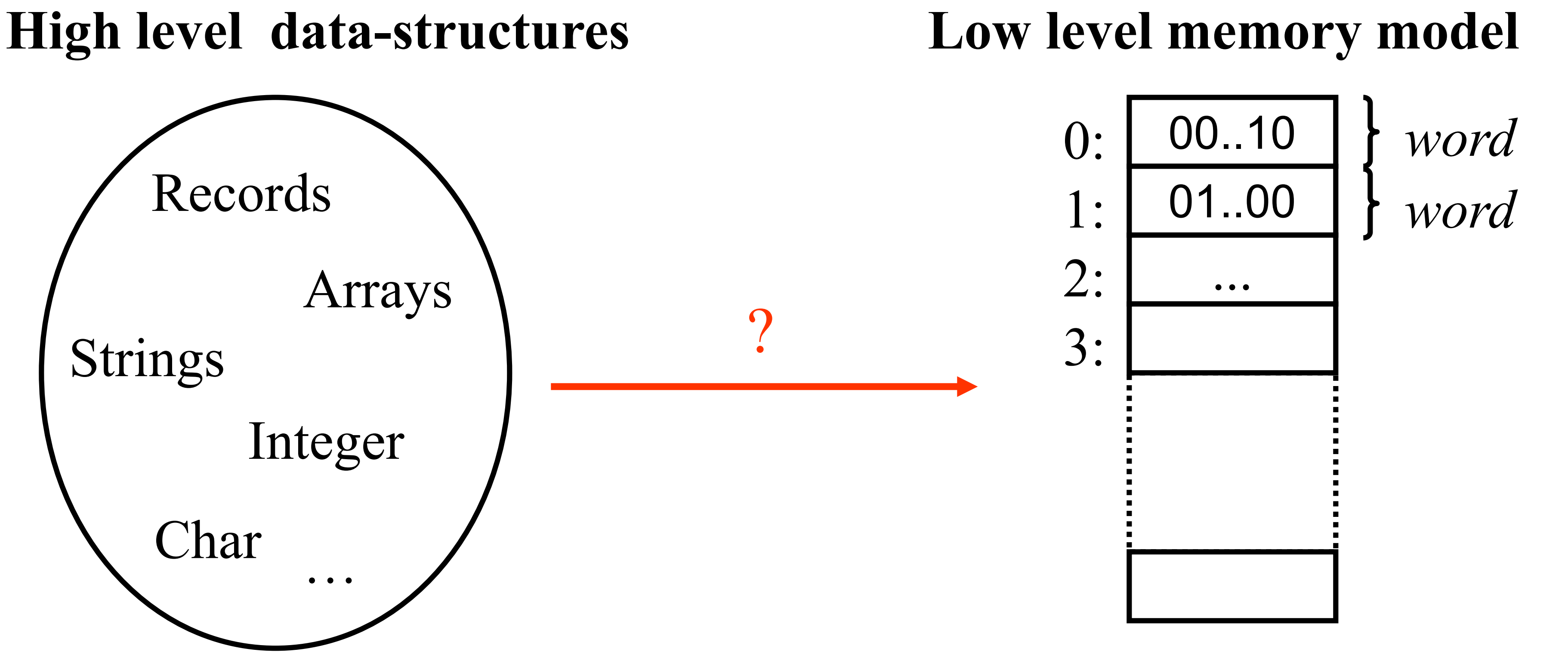
To store, e.g., local variables of functions, function parameters, return address.

To store, e.g., data structures and global program variables (explicitly requested, using malloc, calloc)

stores program instructions

Data Representation

- **Data Representation:** how to represent values of the source language on the target machine.



Note: addressing schema and size of "memory units" may vary

Data Representation

Important properties of a representation schema:

- **non-confusion:** different values of a given type should have different representations
- **uniqueness:** Each value should always have the same representation.

These properties are very desirable**,** but in practice they are not always satisfied:

Example:

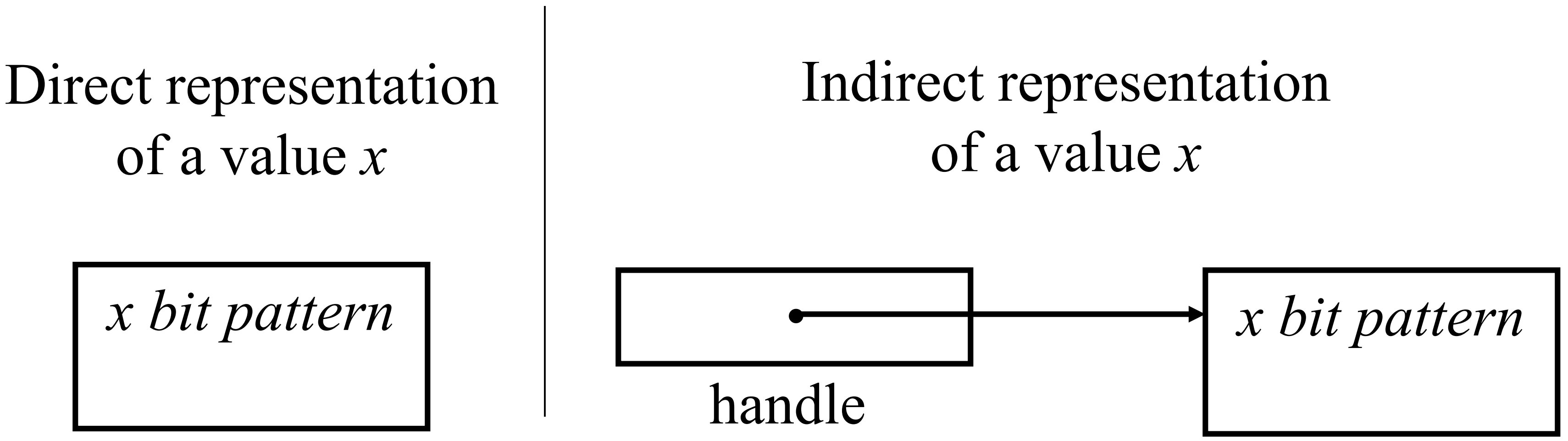
• confusion: approximated floating point numbers.

• non-uniqueness: one's complement representation of integers
+0 and -0

Data Representation

Important issues in data representation:

- **constant-size representation:** The representation of all values of a given type should occupy the same amount of space.
- **direct** versus **indirect** representation

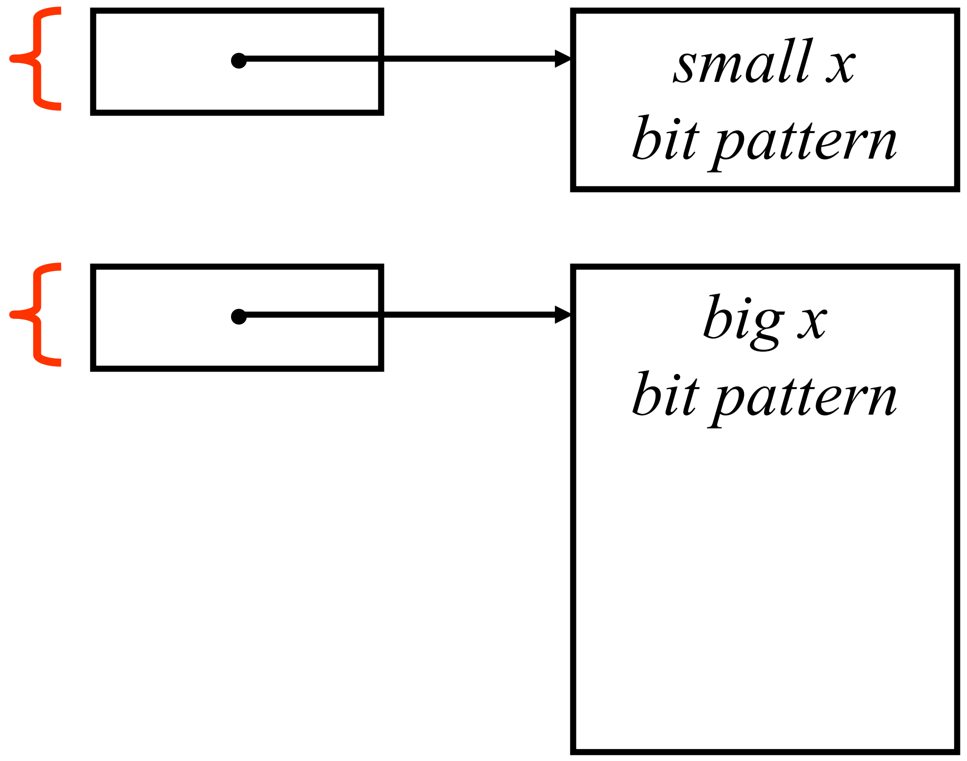

Direct representation of a value *x*
x bit pattern
Indirect representation of a value *x*
handle
x bit pattern

Indirect Representation

Q: What reasons could there be for choosing indirect representations?

To make the representation "constant size" even if representation requires different amounts of memory for different values.

Both are represented by pointers =>Same size



Indirect versus Direct

The choice between indirect and direct representation is a key decision for a language designer/implementer.

- Direct representations are often preferable for efficiency:
  - More efficient access (no need to follow pointers)
  - More efficient "storage class" (e.g stack rather than heap allocation)
- For types with widely varying size of representation it is almost a must to use indirect representation (see previous slide)

Languages like Pascal, C, C++ try to use direct representation wherever possible.

Languages like Scheme, ML, Python use mostly indirect representation everywhere (because of polymorphic higher order functions)

Java: primitive types direct, "reference types" indirect, e.g. objects and arrays.

Data Representation

We now survey representation of the data types found in C-like languages (Triangle), assuming direct representations wherever possible.

We will discuss representation of values of:

- Primitive Types
- Record Types
- Static Array Types
- Dynamic Array Types

We will use the following notations (if T is a type):

#[T] The cardinality of the type (i.e. the number of possible values)

size[T] The size of the representation (in number of bits/bytes)

Data Representation: Primitive Types

What is a primitive type?

The primitive types of a programming language are those types that cannot be decomposed into simpler types. For example `integer, boolean, char`, etc.

Type: `boolean`

Has two values *true* and *false*

=> #[`boolean`] = 2

=> *size*[`boolean`] ≥ 1 bit

	Possible Representation		
Value	1bit	byte(option 1)	byte(option2)
false	0	00000000	00000000
true	1	00000001	11111111

Note: In general if #[T] = n then $size[T] \geq \log_2 n$ bits

Data Representation: Primitive Types

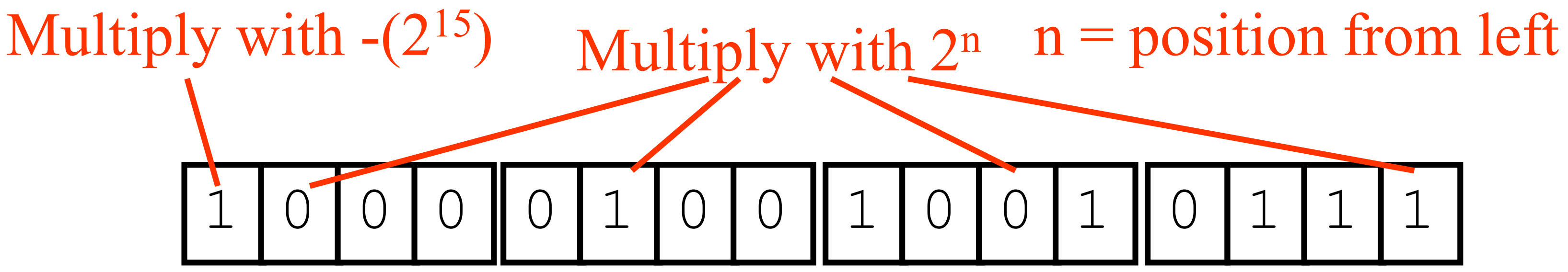
Type: `integer`

Fixed size representation, usually dependent (i.e. chosen based on) what is efficiently supported by target machine. Typically uses one word (16 bits, 32 bits, or 64 bits) of storage.

size[`integer`] = *word (= 16 bits)*

=> #[`integer`] $\leq 2^{16} = 65536$

Modern processors use two's complement representation of integers



Value = $-1.2^{15} + 0.2^{14} + \ldots + 0.2^{3} + 1.2^{2} + 1.2^{1} + 1.2^{0}$

Data Representation: Composite Types

Composite types are types which are not “atomic”, but which are constructed from more primitive types.

- Records (called structs in C)

  Aggregates of several values of several different types
- Arrays

  Aggregates of several values of the same type
- Variant Records or Disjoint Unions
- Pointers or References
- (Objects)
- Functions

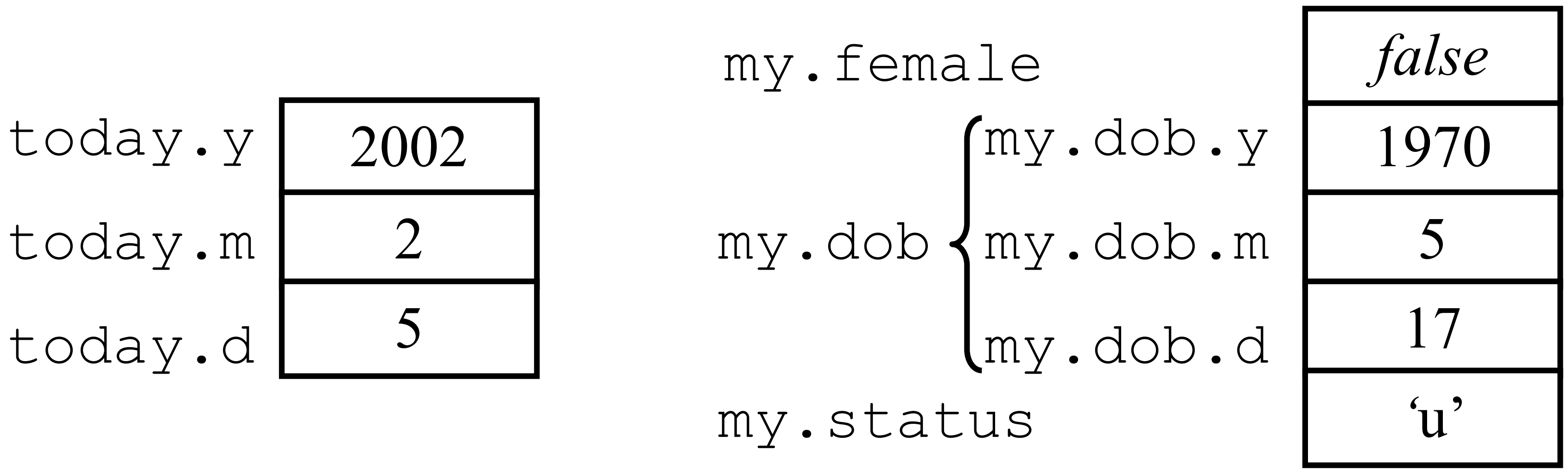
Data Representation: Records

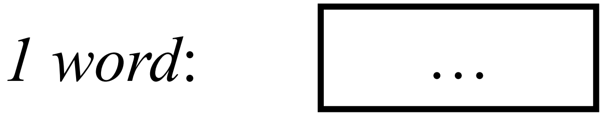
Example: Triangle Records

```
type Date = record
               y : Integer,
               m : Integer,
               d : Integer
            end;
type Details = record
                    female : Boolean,
                    dob :    Date,
                    status : Char
               end;
var today: Date;
var my:   Details
```

Data Representation: Records

Example: Triangle Record Representation

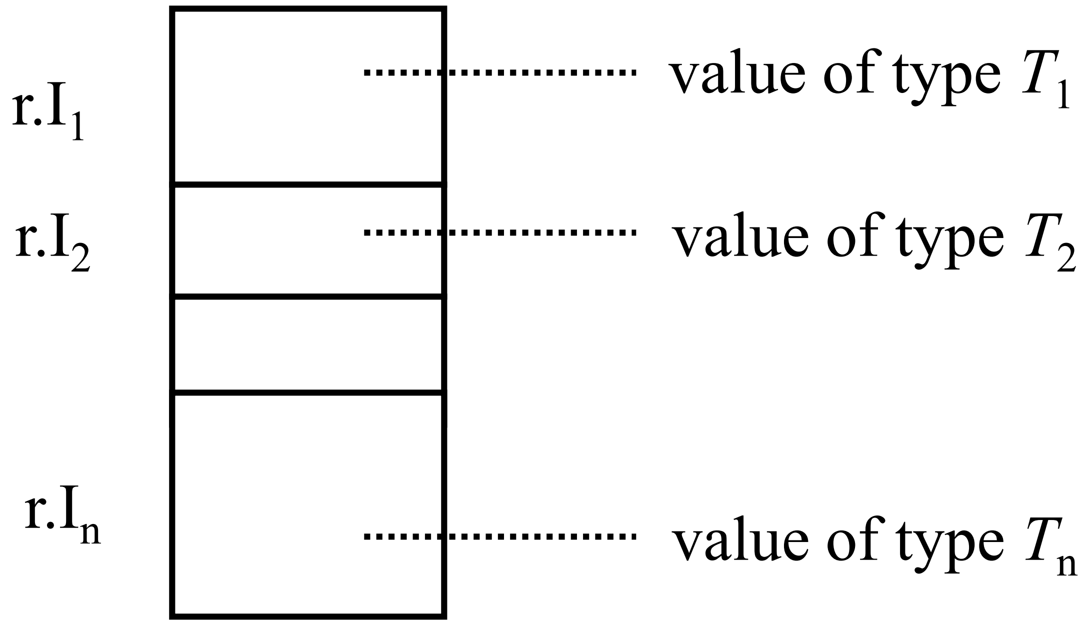




Data Representation: Records

Records occur in some form or other in most programming languages:
Ada, Pascal, Triangle (here they are actually called records)
C, C++, C# (here they are called structs).

The usual representation of a record type is just the concatenation of individual representations of each of its component types.



Data Representation: Records

Q: How much space does a record take up?
And how to access record elements?

Example:
size[`Date`] = 3*size[`integer`] = 3 words
address[today.y] = address[today]+0
address[today.m] = address[today]+1
address[today.d] = address[today]+2

address[my.dob.m] = address[my.dob]+1 = address[my]+2

Note: these formulas assume that addresses are indexes of words (not bytes) in memory (otherwise multiply offsets by 2)

Data Representation: Disjoint Unions

What are disjoint unions?
Like a record, has elements which are of different types. But the elements never exist at the same time. A "type tag" determines which of the elements is currently valid.

Example: Pascal variant records

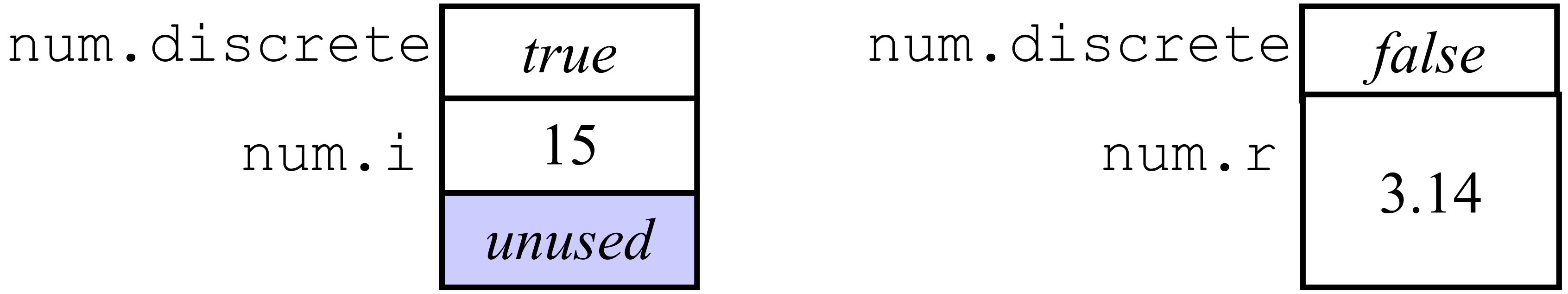
```
type Number = record
                 case discrete: Boolean of
                    true: (i: Integer);
                    false: (r: Real)
              end;
var num: Number
```

Mathematically we write disjoint union types as: $T = T_1 \mid \ldots \mid T_n$

Data Representation: Disjoint Unions

Example: Pascal variant records representation

```
type Number = record
                 case discrete: Boolean of
                    true: (i: Integer);
                    false: (r: Real)
              end;
var num: Number
```



Assuming *size*[Integer]=size[Boolean]=1 and *size*[Real]=2, then
size[Number] = *size*[Boolean] + MAX(size[Integer], size[Real])
= 1 + MAX(1, 2) = 3

Data Representation: Disjoint Unions

```
type T = record
  case I_tag: T_tag of
    v_1: (I_1: T_1);
    v_2: (I_2: T_2);
   ...
    v_n: (I_n: T_n);
end;
var u: T
```

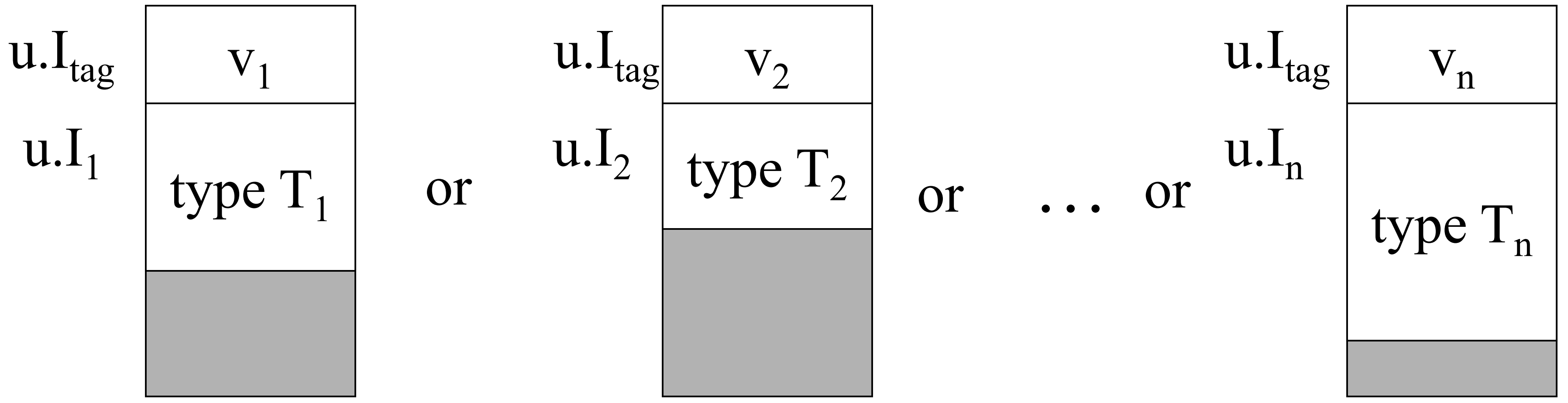
$$size[T] = size[T_{tag}] + \mathrm{MAX}(size[T_1], ..., size[T_n])$$

$$address[u.I_{tag}] = address[u]$$

$$address[u.I_1] = address[u]+size[T_{tag}]$$

...

$$address[u.I_n] = address[u]+size[T_{tag}]$$



Arrays

An array is a composite data type, an array value consists of multiple values of the same type. Arrays are in some sense like records, except that their elements all have the same type.

The elements of arrays are typically indexed using an integer value (In some languages such as for example Pascal, also other "ordinal" types can be used for indexing arrays).

Two kinds of arrays (with different runtime representation schemas):

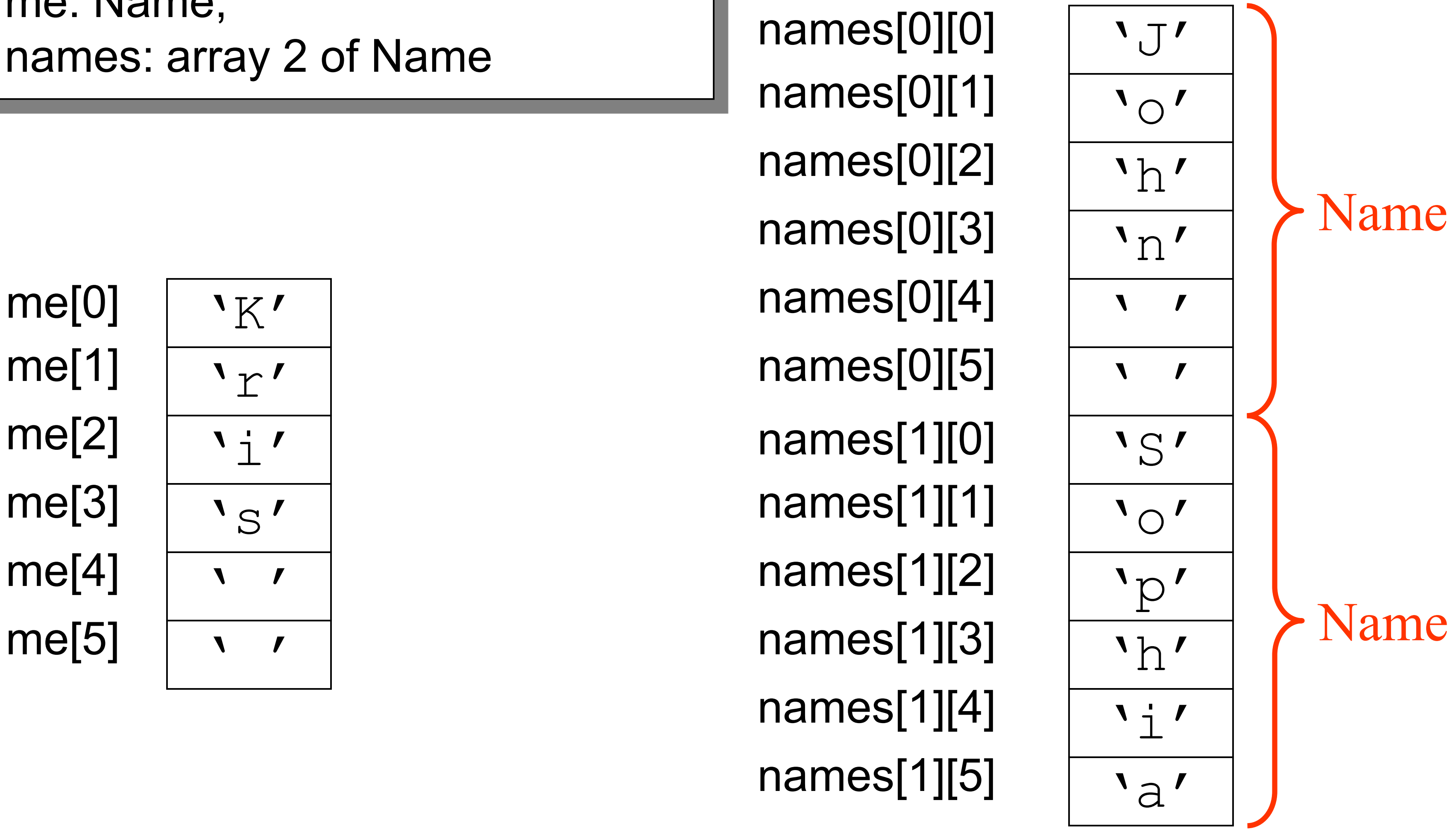
- **static** arrays: their **size** (number of elements) is **known** at **compile time**.
- **dynamic** arrays: their size can not be known at compile time because the number of elements is computed at run-time and sometimes may vary at run-time (Flex-arrays).

Q: Which are the "cheapest" arrays? Why?

Static Arrays

Example:

```
type Name = array 6 of Char;
var me: Name;
var names: array 2 of Name
```

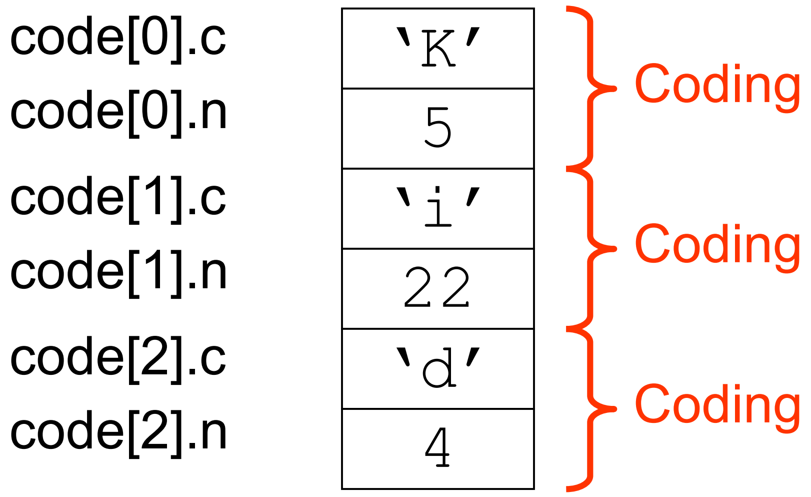


Static Arrays

Example:

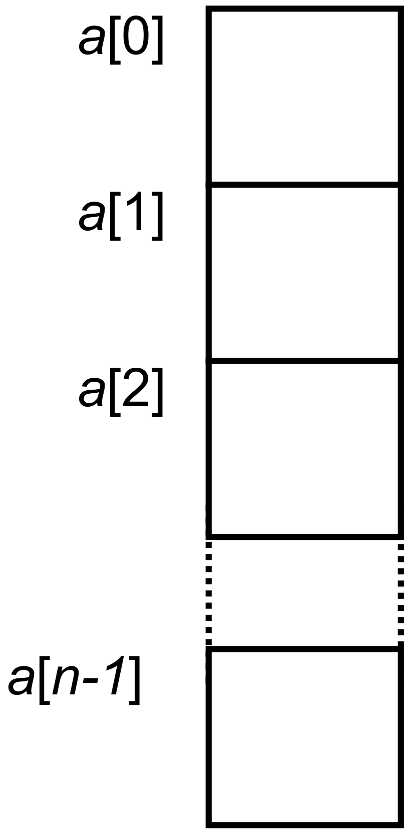
```
type Coding = record
   Char c, Integer n
end

var code: array 3 of Coding
```



Static Arrays

type *T* = **array** *n* **of** *TE*;
var *a* : *T*;



size[T] = n * *size*[TE]

address[a`[0]`] = *address*[a]
address[a`[1]`] = *address*[a]+*size*[TE]
address[a`[2]`] = *address*[a]+2**size*[TE]
…
address[a[*i*]] = *address*[a]+*i***size*[TE]
…

Dynamic Arrays

Dynamic arrays are arrays whose size is not known until run time.

Example: Java Arrays (<u>all</u> arrays in Java are dynamic)

```
char[ ] buffer;

buffer = new char[buffersize];

...
for (int i=0; i<buffer.length; i++)
   buffer[i] = ‘ ’;
```

Dynamic array: no size given in declaration

Array creation at runtime determines size

Can ask for size of an array at run time

Q: How could we represent Java arrays?

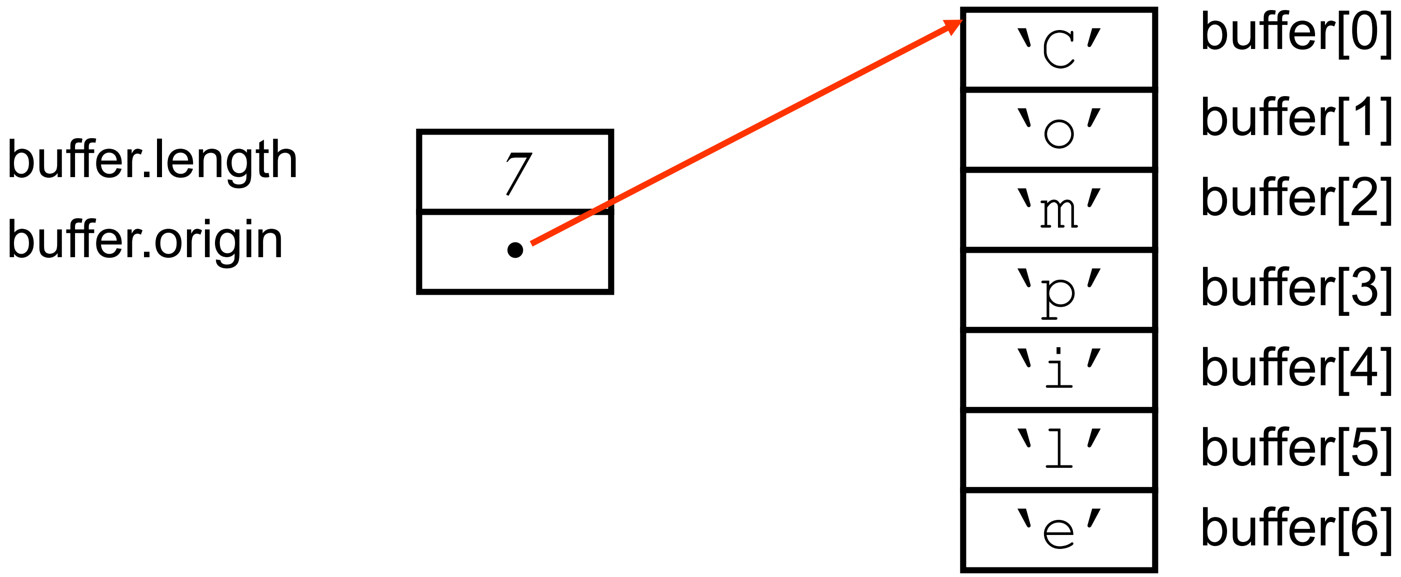
Dynamic Arrays

Java Arrays

```
char[ ] buffer;

buffer = new char[7];
```

A possible representation for Java arrays



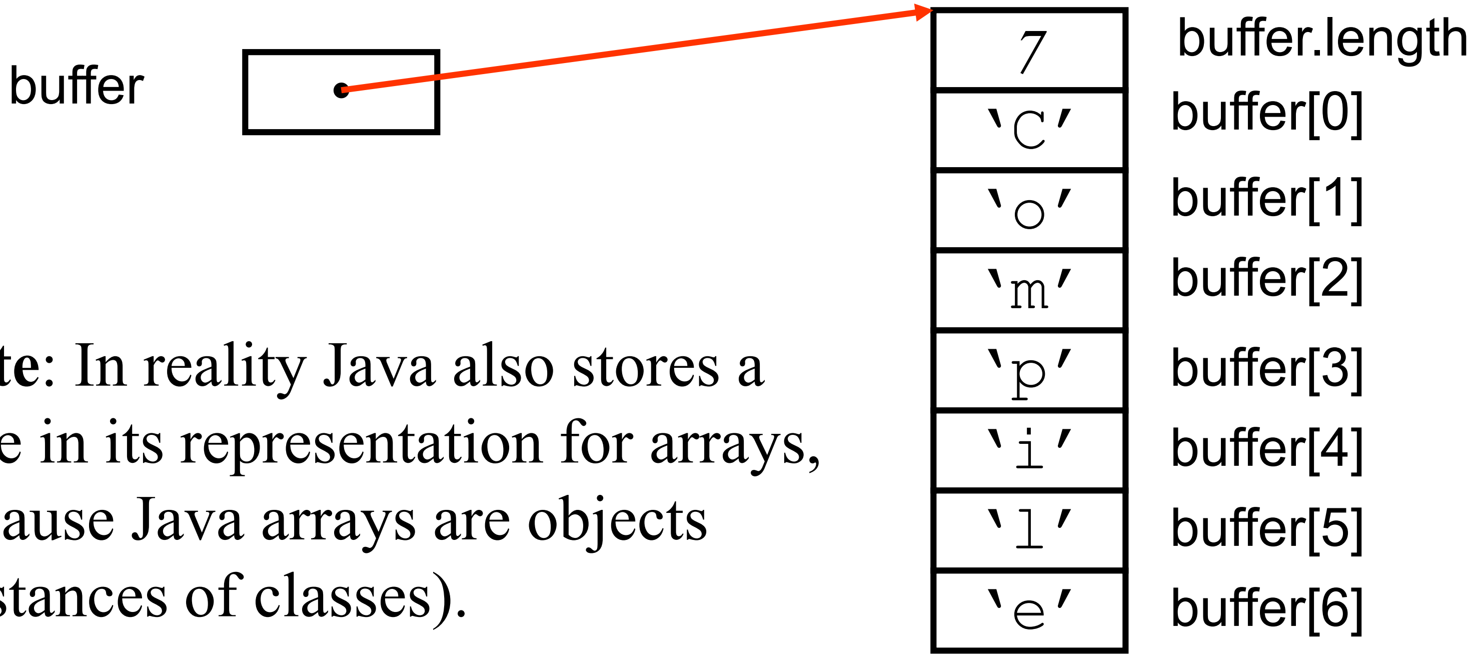
Dynamic Arrays

Java Arrays

```
char[ ] buffer;

buffer = new char[7];
```

Another possible representation for Java arrays



Note: In reality Java also stores a type in its representation for arrays, because Java arrays are objects (instances of classes).

Where to put data?

Now we have looked at how program structures are implemented in a computer memory

Next we look at where to put them

We will cover 3 methods:

1) static allocation,
2) stack allocation, and
3) heap allocation.

Static Allocation

Originally, all data were global.

Correspondingly, all memory allocation was static.

During compilation, data was simply placed at a fixed memory address for the entire execution of a program. This is called static allocation.

Examples are all assembly languages, Cobol, and Fortran.

Note: code is (still) usually allocated statically

Static Allocation (Cont.)

Static allocation can be quite wasteful of memory space. To reduce storage needs, in Fortran, the *equivalent* statement overlays variables by forcing two variables to share the same memory locations. In C,C++, *union* does this too.

Overlaying hurts program readability, as assignment to one variable changes the value of another.

In more modern languages, static allocation is used for global variables and literals (constant) that are fixed in size and accessible throughout program execution.

It is also used for static and extern variables in C/C++ and for static fields in C# and Java classes.

Stack Allocation

Recursive languages require dynamic memory allocation. Each time a recursive method is called, a new copy of local variables (frame) is pushed on a runtime stack. The number of allocations is unknown at compile-time.

A frame (or activation record) contains space for all of the local variables in the method. When the method returns, its frame is popped and the space reclaimed.

Thus, only the methods that are actually executing are allocated memory space in the runtime stack. This is called stack allocation.

```
p(int a) {
   double b;
   double c[10];
   b = c[a] * 2.51;
}
```

Figure 12.1: A Simple Subprogram

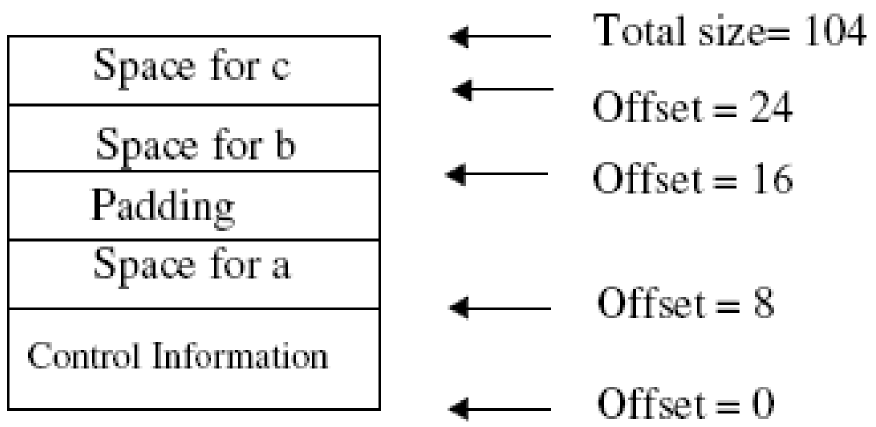
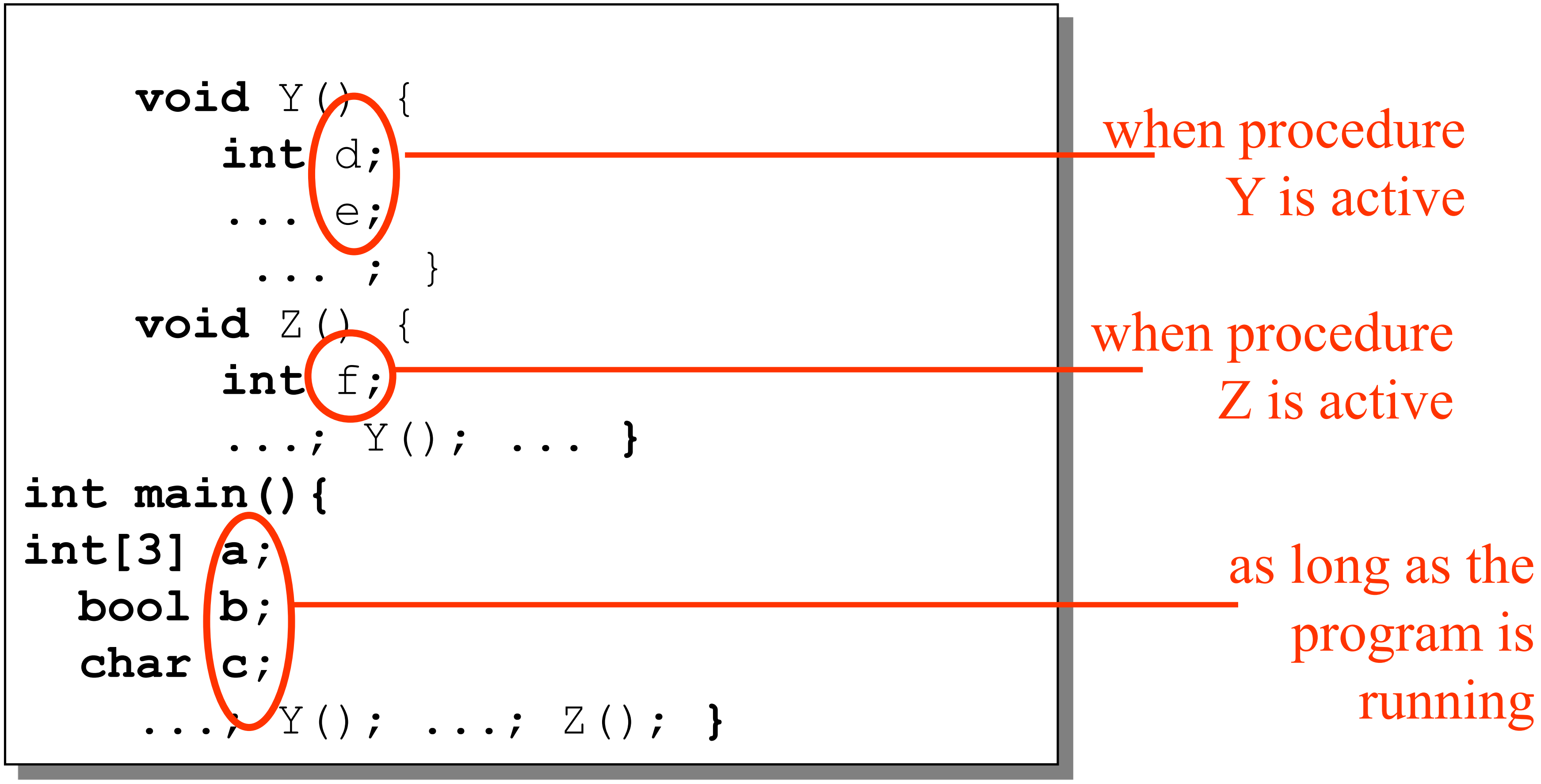


Figure 12.2: Frame for Procedure p

Stack Storage Allocation

Now we will look at allocation of local variables

Example: When do the variables in this program "exist"

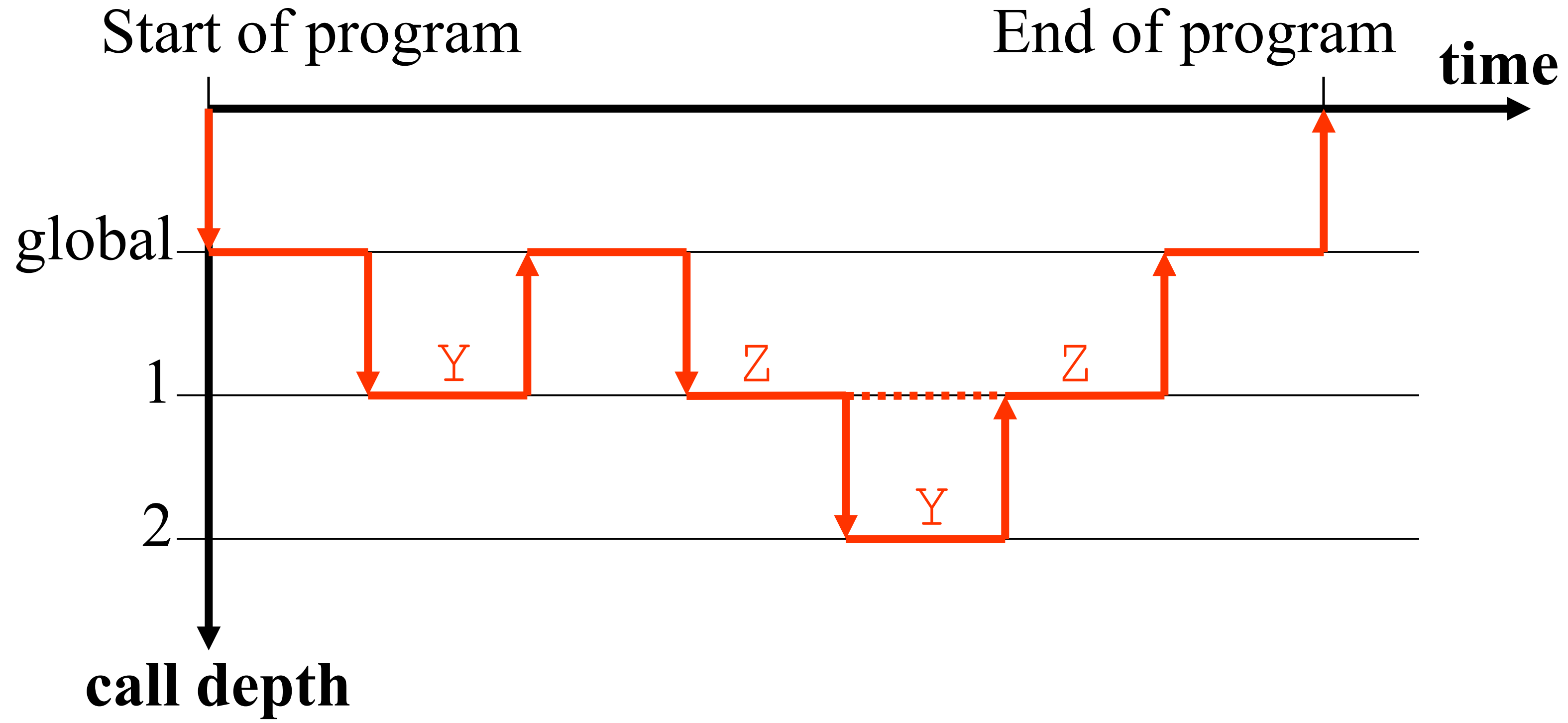


```
     void Y() {
         int d;
         ... e;
          ... ; }
     void Z() {
         int f;
         ...; Y(); ... }
int main(){
int[3] a;
  bool b;
  char c;
     ...; Y(); ...; Z(); }
```

when procedure Y is active
when procedure Z is active
as long as the program is running

Stack Storage Allocation

A "picture" of our program running:



1) Procedure activation behaves like a stack (LIFO).

2) The local variables "live" as long as the procedure they are declared in.

1+2 => Allocation of locals on the "call stack" is a good model.

Recursion

```
int fact (int n) {
    if (n>1) return n* fact (n-1);
    else return 1;
}
```

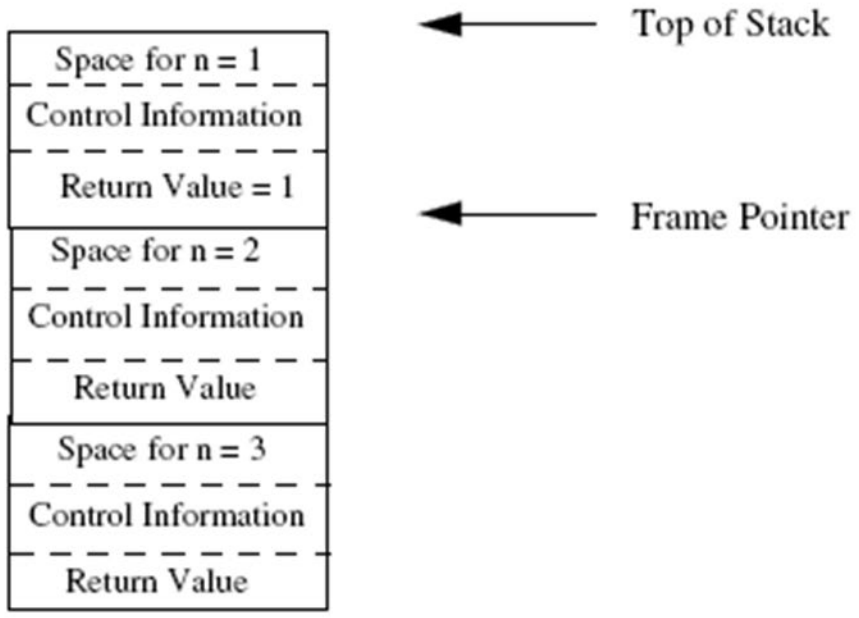
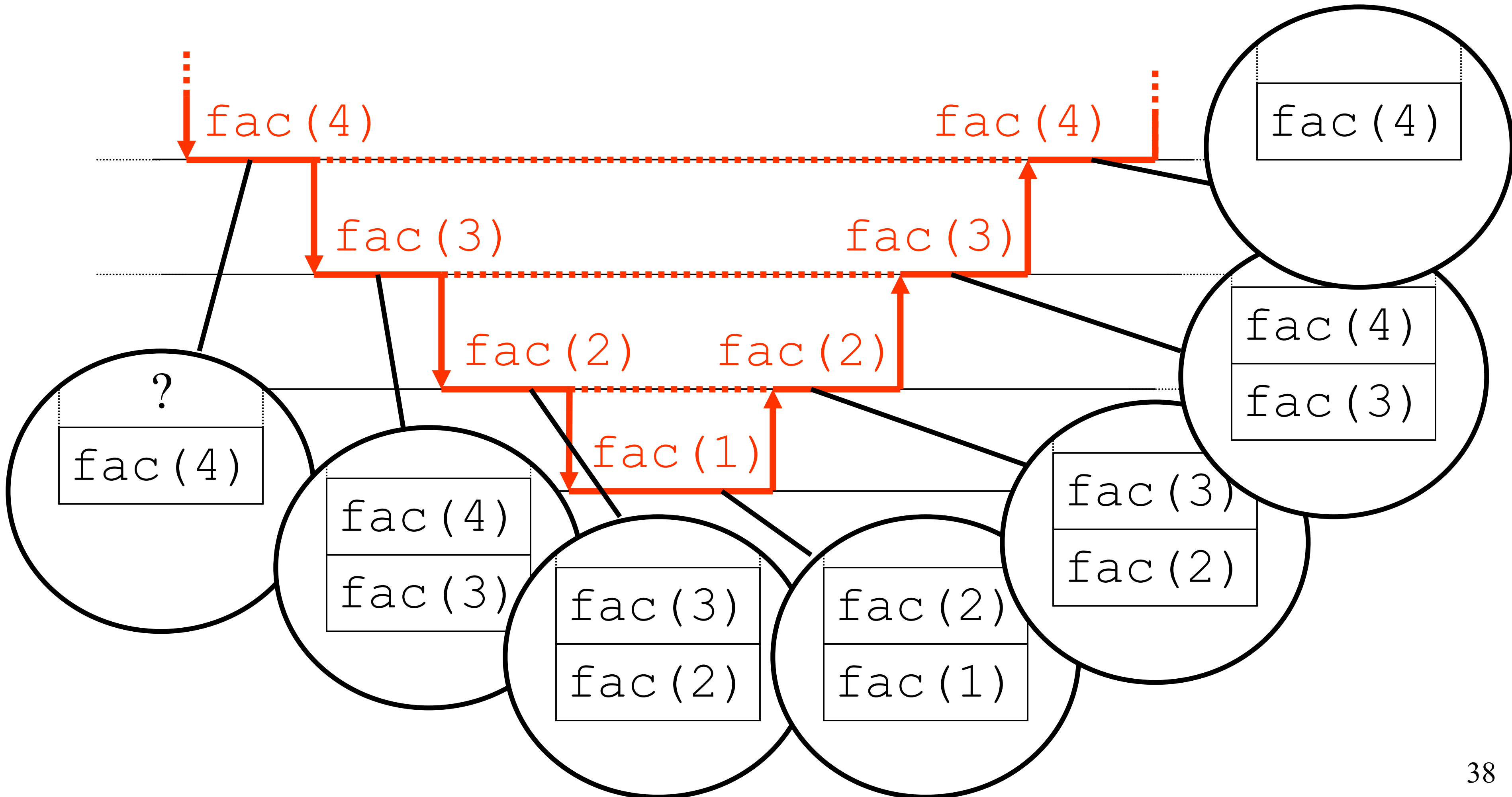


Figure 12.3: Runtime Stack for a Call of `fact(3)`

Recursion: General Idea

Why the stack allocation model works for recursion:
Like other function/procedure calls, lifetimes of local variables and parameters for recursive calls behave like a stack.



Dynamic link

Because stackframes may vary in size and because the stack may contain more than just frames (e.g., registers saved across calls), dynamic link is used to point to the preceding frame (Fig. 12.4).

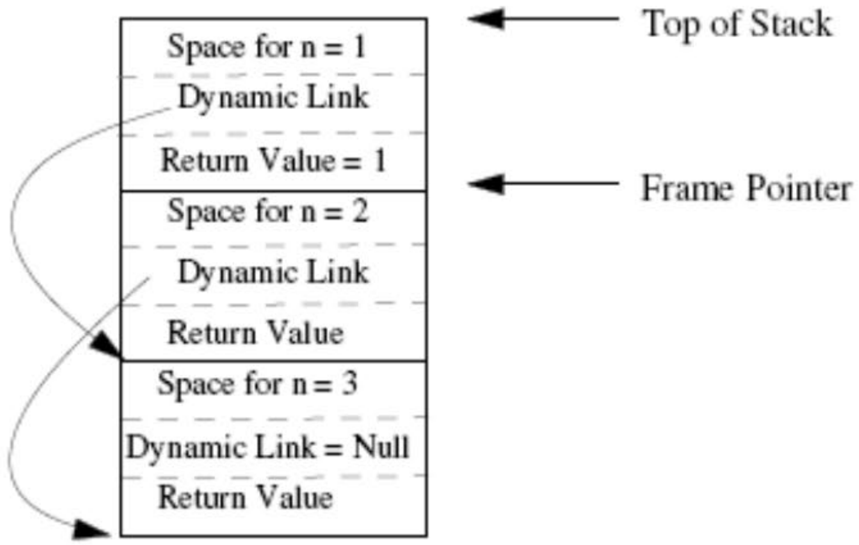


Figure 12.4: Runtime Stack for a Call of `fact(3)`with Dynamic Links

Nested functions/procedures

```
int p (int a) {
    int q (int b) { if (b <0) q (-b)  else return a+b; }
    return q (-10);
}
```

Methods cannot nest in C, Java, but in languages like Pascal, ML and Python they can. How to keep track of static block structure as above?

A static link points to the frame of the method that statically encloses the current method. (Fig. 12.6)

An alternative to using static links to access frames of enclosing methods is the use of a display. Here, we maintain a set of registers which comprise the display. (see Fig. 12,7)

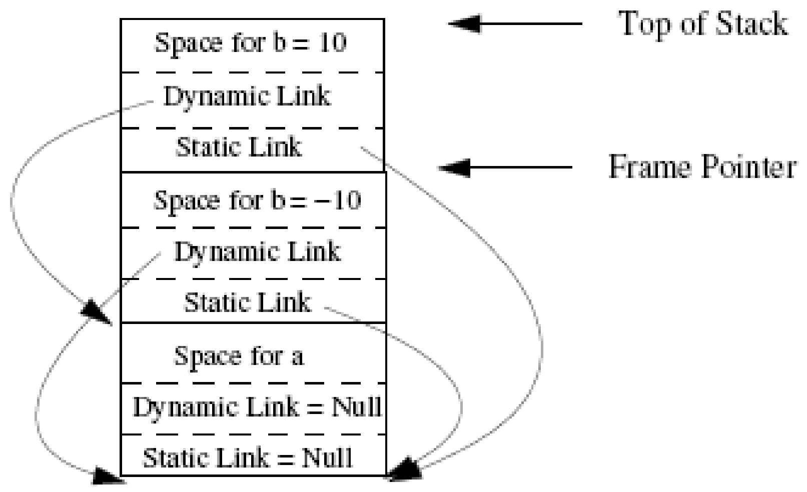


Figure 12.6: An Example of Static Links

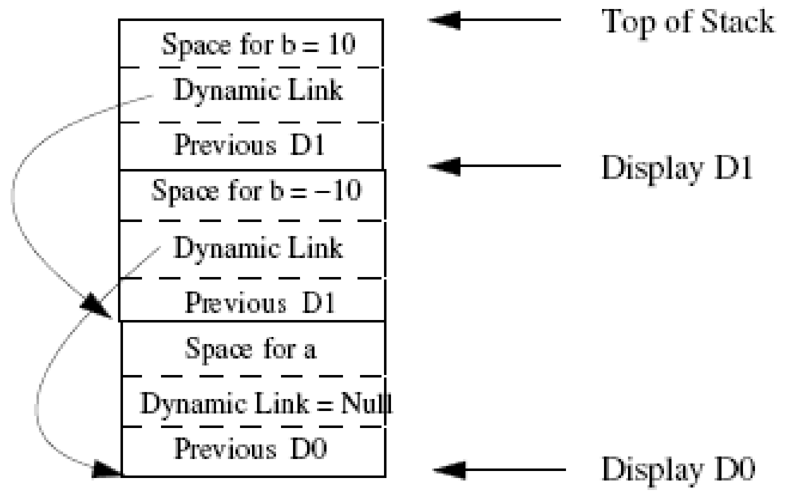


Figure 12.7: An Example of Display Registers

Blocks

```
void p (int a) {
   int b;
   if (a>0) {float c,d;  //body of block 1//}
   else      {int e[10]; //body of block 2//}
 }
```

We could view such blocks as parameter-less procedures and thus use procedure-level-frames to implement blocks, but because the then and else parts of the if statement above are mutually exclusive, variables in block 1 and block 2 can overlay each other. This is called **block-level frame**, as contrasted with **procedure-level frame** allocation. (Fig. 12.8)

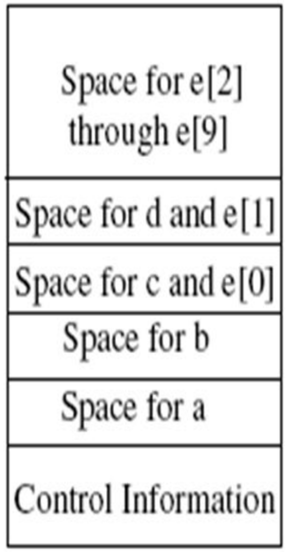


Figure 12.8: An Example of a Procedure-Level Frame

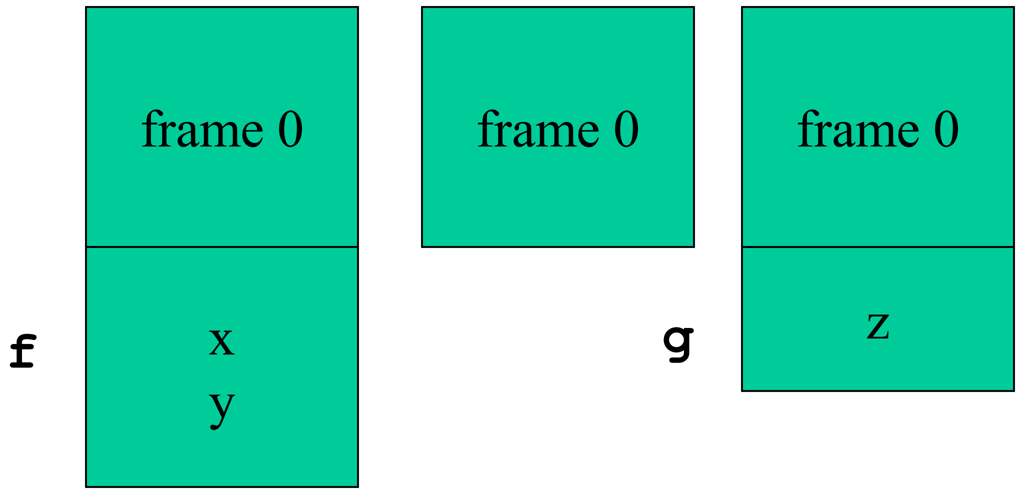
Higher-order functions

- Functions as values (first-class)
  - Pass as arguments
  - Return as values
  - Stored into data structures
- Implementation:
  - A code pointer, (i.e., a code address + an environment pointer)
  - Such a data structure is called a closure

Higher-order Nested Functions

```
void->int f(){
   int x;
   int y;
   int g (){
      int z;
      return z+x;
   }
   return g;
}

h = f();  // h==g
h();      // g()
```

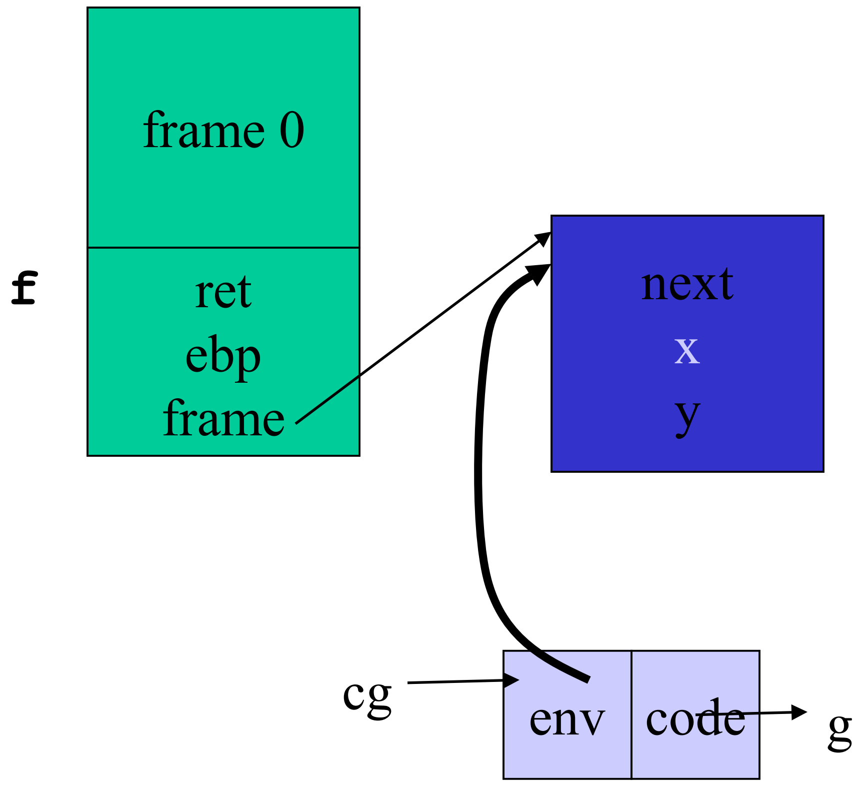


Function frames don't obey LIFO discipline any more. What one need to do is to keep frames live long enough! Heap-allocation!

Heap-allocated Frames

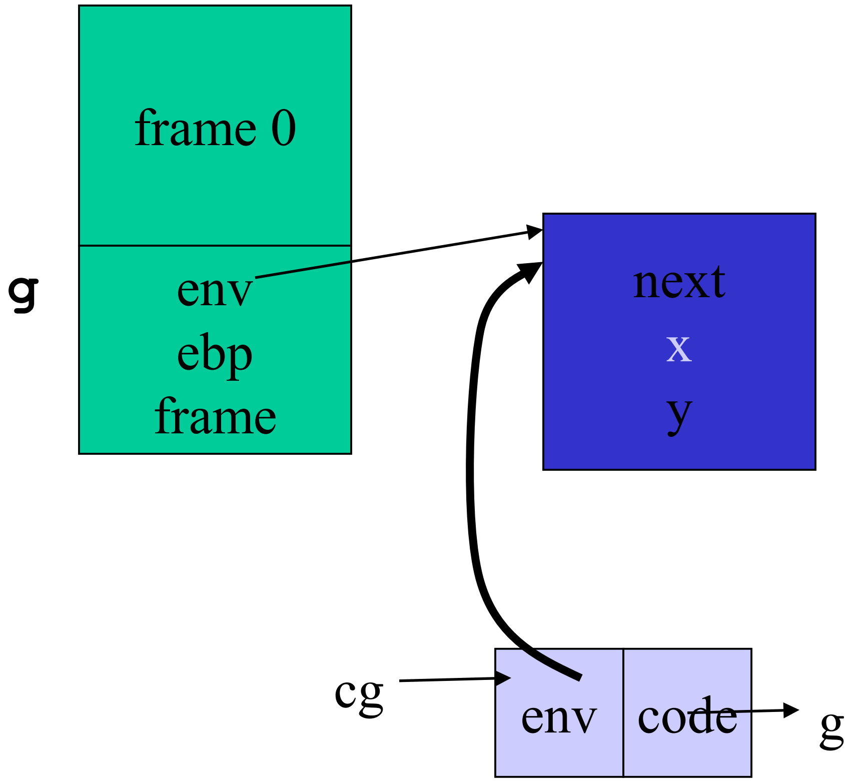
```
void->int f(){
  int x;
  int y;
  int g (){
    int z;
    return z+x;
  }
  return g;
}

h = f();
// h == cg
h();
```



Heap-allocated Frames

```
void->int f(){
   int x;
   int y;
  int g (){
     int z;
     return z+x;
  }
   return g;
}
```



```
h = f();
// h == cg
h();
```

h->code(h->env);

Pause

Memory Management

When a program is started, most operating systems allocate 3 memory segments for it:

1) **code segment: read-only**

   Code (normally doesn't change during execution)

   Global variables (sometimes stored at the bottom of the stack)

2) **stack segment (data)**:

   manipulated by machine instructions.

   local variables and arguments for procedures and functions

   lifetime follows procedure activation

3) **heap segment (data)**:

   manipulated by the programmer.

   some programs may ask for and get memory allocated on arbitrary points during execution

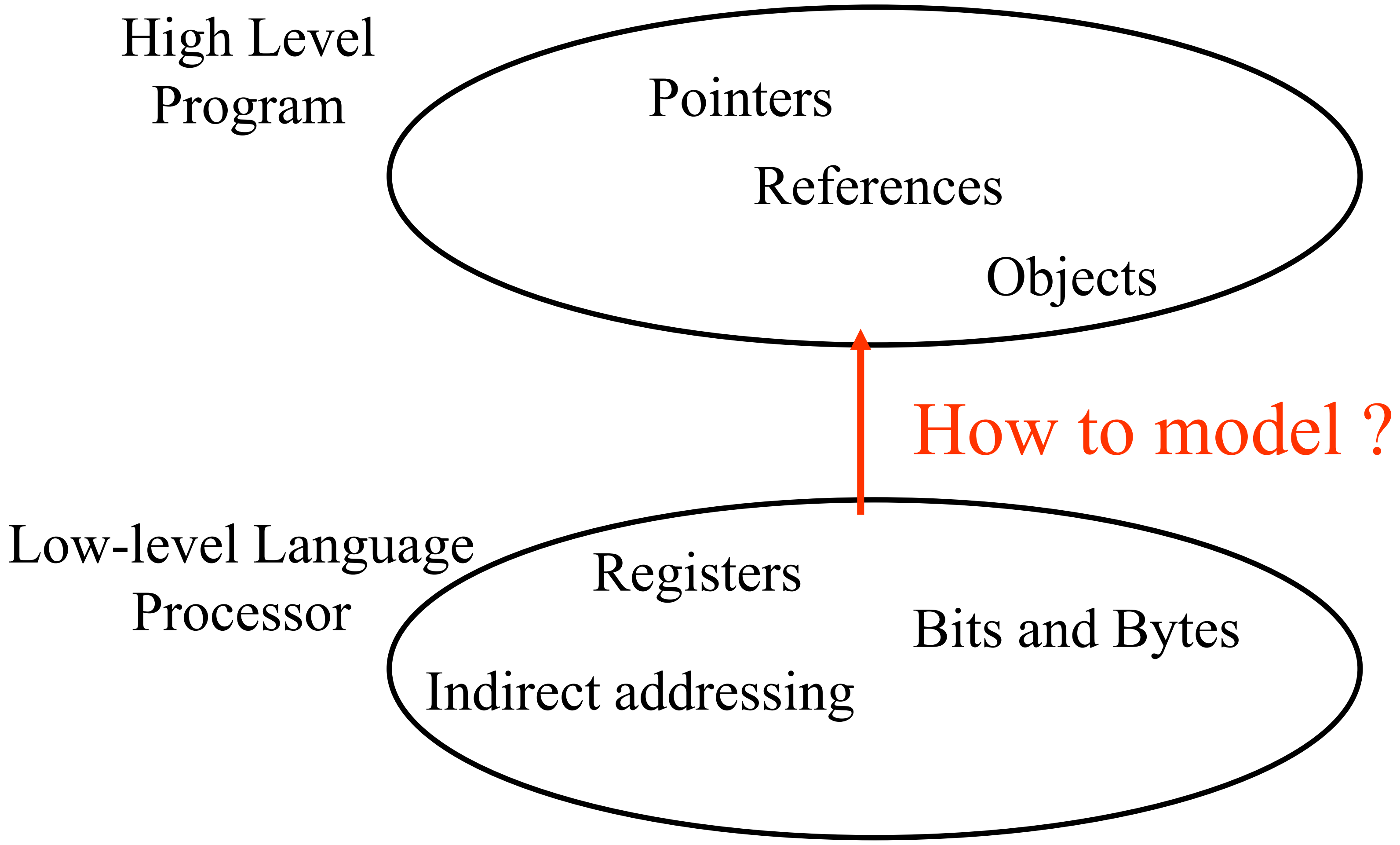
   When this memory is no longer used it should be freed

Heap Storage

- Memory allocation under explicit programmatic control
  - C malloc, C++ / Pascal / Java / C# new operation.
- Memory allocation implicit in language constructs
  - Lisp, Scheme, Haskell, SML, … most functional languages
  - Autoboxing/unboxing in Java 1.5 and C#
- Deallocation under explicit programmatic control
  - C, C++, Pascal (free, delete, dispose operations)
- Deallocation implicit
  - Java, C#, Lisp, Scheme, Haskell, SML, …

Data representation
Sometimes it goes the other way round

How to reflect low-level memory and machine data structures in terms of high-level computational structures.



How does things become garbage?

```
int *p, *q;
…
p = malloc(sizeof(int));
p = q;
```

Newly created space becomes garbage

```
for(int i=0;i<10000;i++){
    SomeClass obj= new SomeClass(i);
    System.out.println(obj);
}
```

Creates 10000 objects, which becomes garbage just after the print

Problem with explicit heap management

```
int *p, *q;
…
p = malloc(sizeof(int));
q = p;
free(p);
```

Dangling pointer in q now

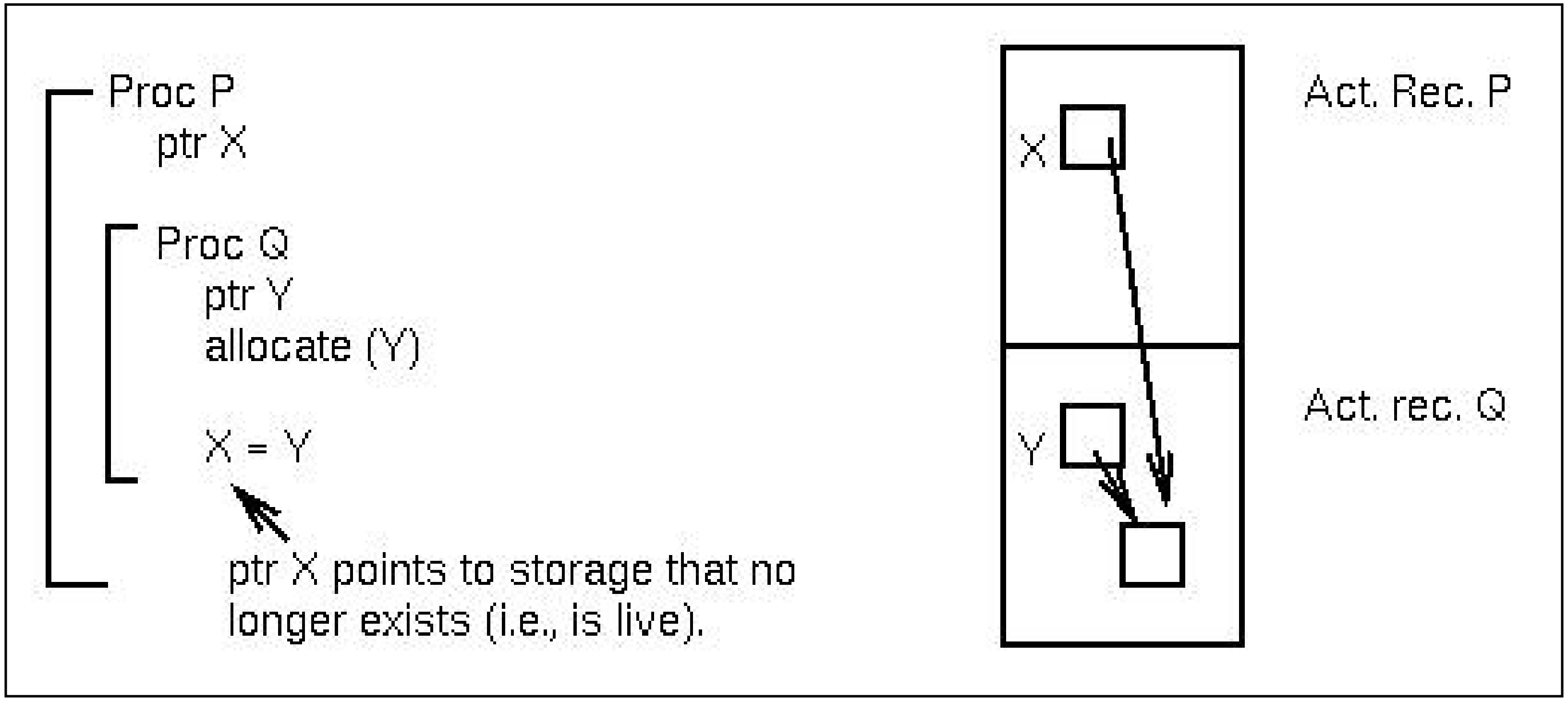
```
float myArray[100];

p = myArray;
*(p+i) = …   //equivalent to myArray[i]
```

They can be hard to recognize

Stacks and dynamic allocations are incompatible

Why can't we just do dynamic allocation within the stack?



Where to put the heap?

- The heap is an area of memory which is dynamically allocated.
- Like a stack, it may grow and shrink during runtime.
- Unlike a stack it is not a LIFO => more complicated to manage
- In a typical programming language implementation we will have both heap-allocated and stack allocated memory coexisting.

Q: How do we allocate memory for both

Where to put the heap?

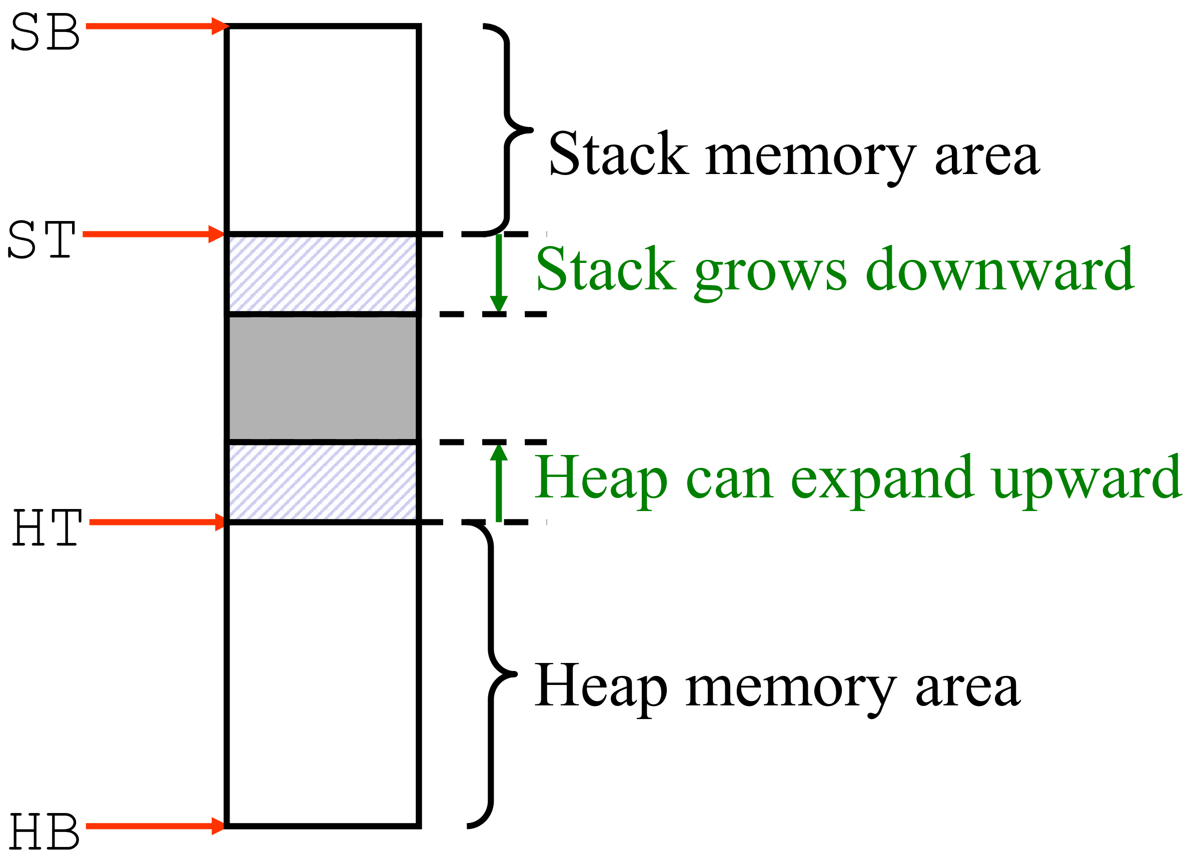
- A simple approach is to divide the available memory at the start of the program into two areas: stack and heap.
- Another question then arises
  - How do we decide what portion to allocate for stack vs. heap ?
  - Issue: if one of the areas is full, then even though we still have more memory (in the other area) we will get out-of-memory errors

Q: Isn’t there a better way?

Where to put the heap?

Q: Isn't there a better way?

A: Yes, there is an often used "trick": let both stack and heap share the same memory area, but grow towards each other from opposite ends!



Implicit memory management

- Current trend of modern programming language development: to give only implicit means of memory management to a programmer:
  - The constant increase of hardware memory justifies the policy of automatic memory management
  - The explicit memory management distracts programmer from his primary tasks: let everyone do what is required of them and nothing else!
  - The philosophy of high-level languages conforms to the implicit memory management
- Other arguments for implicit memory management:
  - Anyway, a programmer cannot control memory management for temporary variables!
  - The difficulties of combination of two memory management mechanisms: system and the programmer’s
- The history repeats: in 70’s people thought that the implicit memory management had finally replaced all other mechanisms

Automatic Storage Deallocation (Garbage Collection)

Everybody probably knows what a garbage collector is.

But here are two “one liners” to make you think again about what a garbage collector really is!

1) Garbage collection provides the “illusion of infinite memory”!

2) A garbage collector predicts the future!

It’s a kind of magic! :-)

Let us look at how this magic is done!

Types of garbage collectors

- The "Classic" algorithms
  - Reference counting
  - Mark and sweep
- Copying garbage collection
- Generational garbage collection
- Incremental Tracing garbage collection

- **Direct Garbage Collectors:** a record is associated with each node in the heap. The record for node *N* indicates how many other nodes or roots point to *N*.

- **Indirect/Tracing Garbage Collectors:** usually invoked when a user's request for memory fails. The garbage collector visits all live nodes, and returns all other memory to the free list. If sufficient memory has been recovered from this process, the user's request for memory is satisfied.

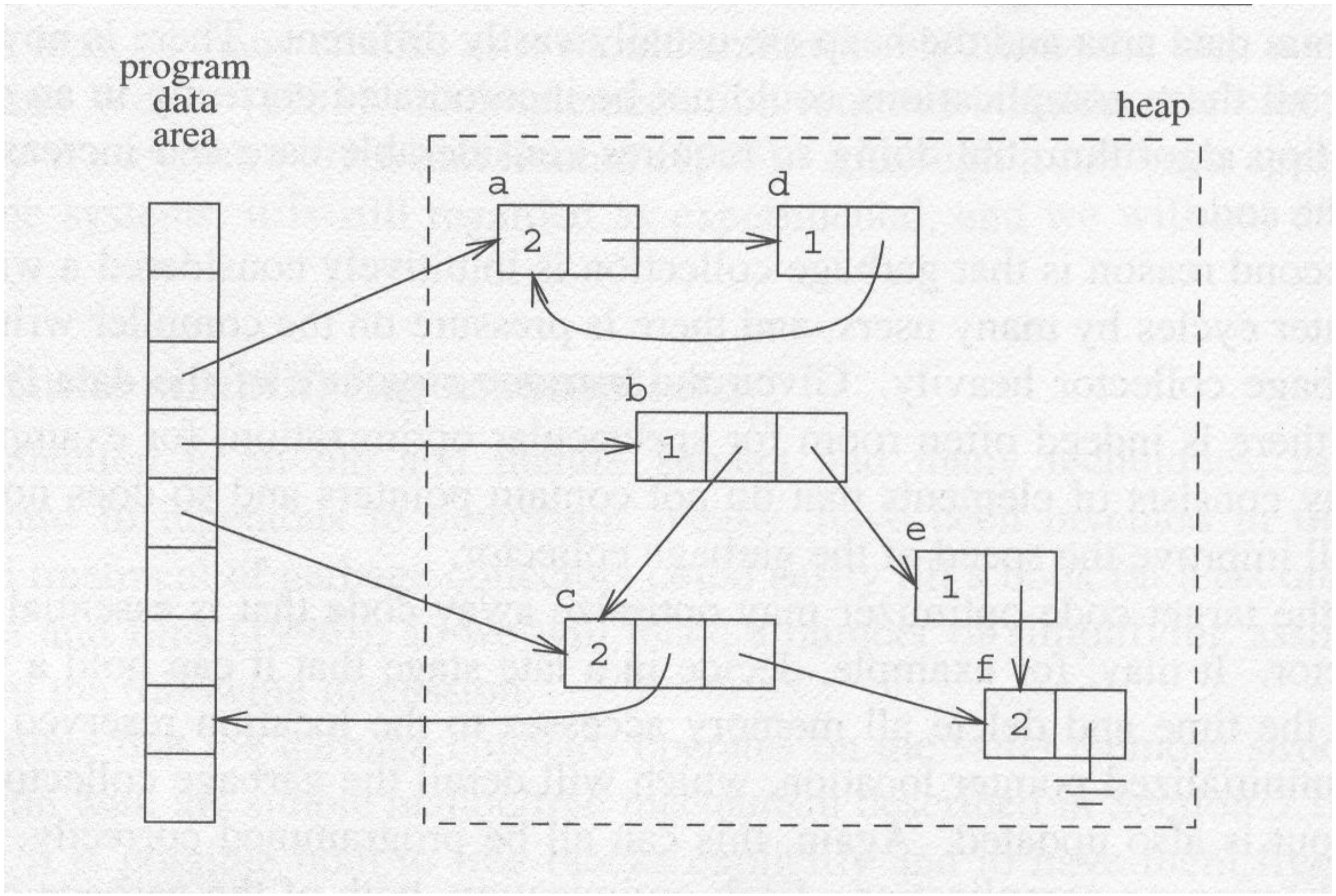
Terminology

- **Roots:** values that a program can manipulate directly (i.e. values held in registers, on the program stack, and global variables.)
- **Node/Cell/Object:** an individually allocated piece of data in the heap.
- **Children Nodes:** the list of pointers that a given node contains.
- **Live Node:** a node whose address is held in a root or is the child of a live node.
- **Garbage:** nodes that are not live, but are not free either.
- **Garbage collection:** the task of recovering (freeing) garbage nodes.
- **Mutator:** The program running alongside the garbage collection system.

Reference Counting

- Every cell has an additional field: the *reference count*. This field represents the number of pointers to that cell from roots or heap cells.
- Initially, all cells in the heap are placed in a pool of free cells, the *free list*.
- When a cell is allocated from the *free list*, its reference count is set to one.
- When a pointer is set to reference a cell, the cell's reference count is incremented by 1; if a pointer is to the cell is deleted, its reference count is decremented by 1.
- When a cell's reference count reaches 0, its pointers to its children are deleted and it is returned to the free list.

Reference Counting



Reference Counting: Advantages and Disadvantages

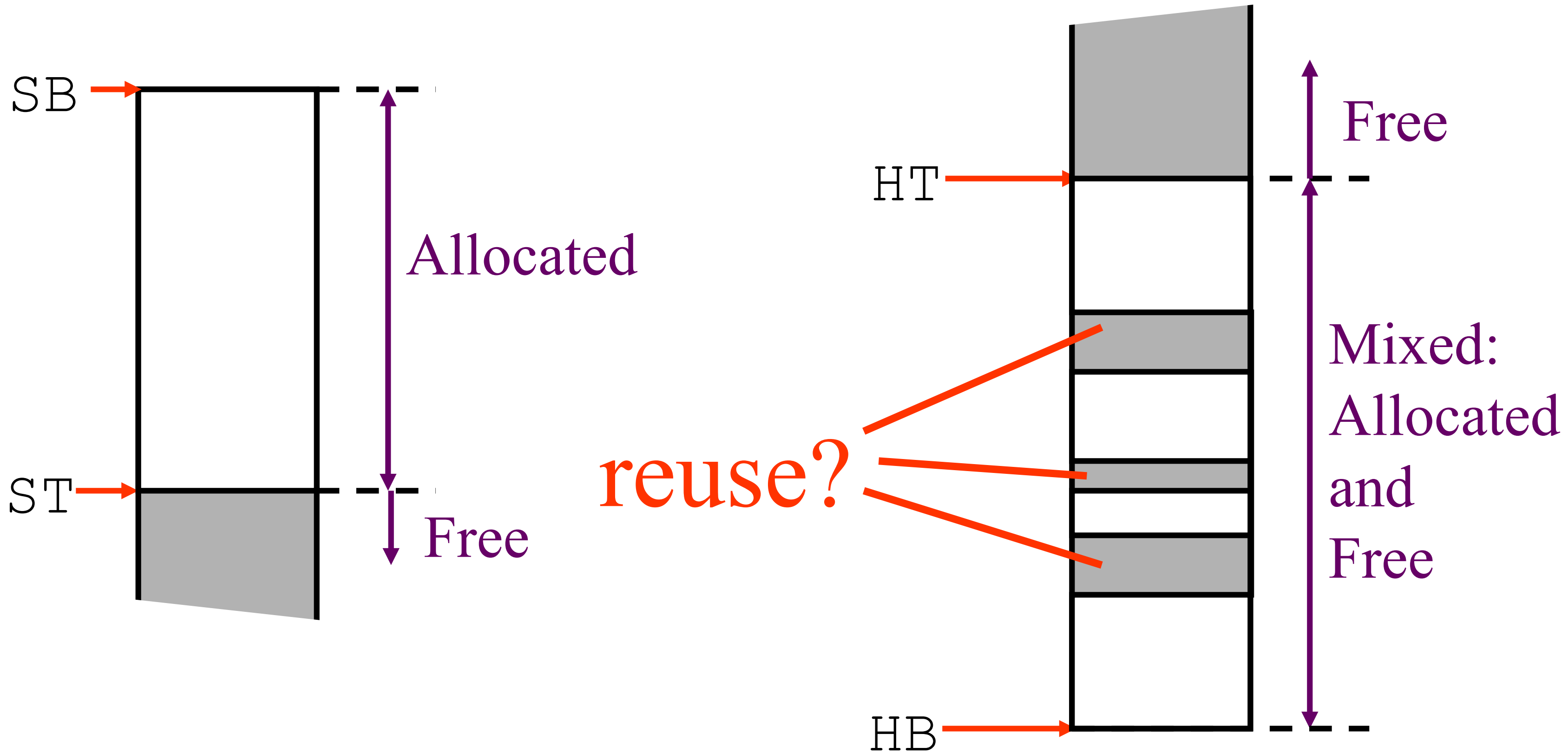
- Advantages:
  - Garbage collection overhead is distributed.
  - Locality of reference is no worse than mutator.
  - Free memory is returned to free list quickly.
- Disadvantages:
  - High time cost (every time a pointer is changed, reference counts must be updated).
    - In place of a single assignment x.f = p:

```
z = x.f
c = z.count
c = c – 1
z.count = c
If c = 0 call putOnFreeList(z)
x.f = p
c = p.count
c = c + 1
p.count = c
```

  - Storage overhead for reference counter can be high.
  - If the reference counter overflows, the object becomes permanent.
  - Unable to reclaim cyclic data structures.

How to keep track of free memory?

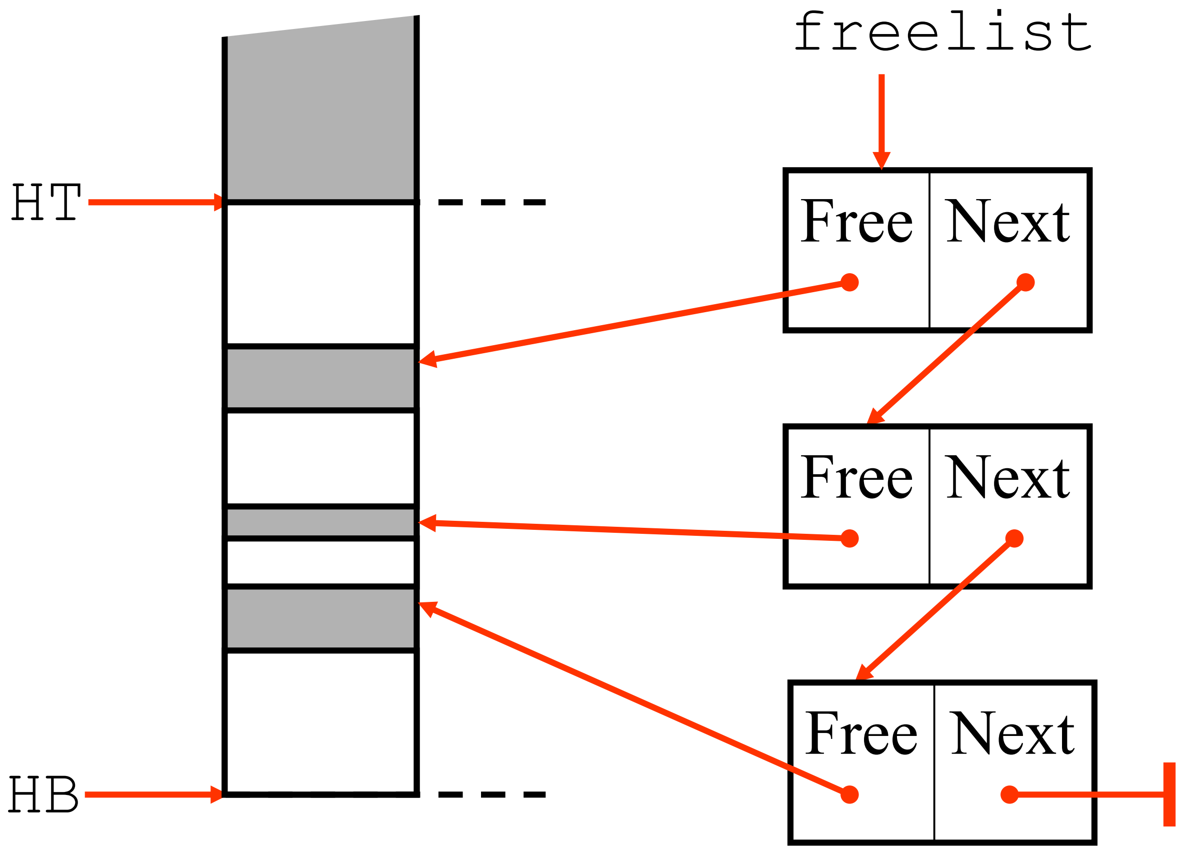
Stack is LIFO allocation => ST moves up/down everything above ST is in use/allocated. Below is free memory. This is easy! But …
Heap is not LIFO, how to manage free space in the “middle” of the heap?



How to keep track of free memory?

How to manage free space in the “middle” of the heap?

=> keep track of free blocks in a data structure: the “free list”. For example we could use a linked list pointing to free blocks.

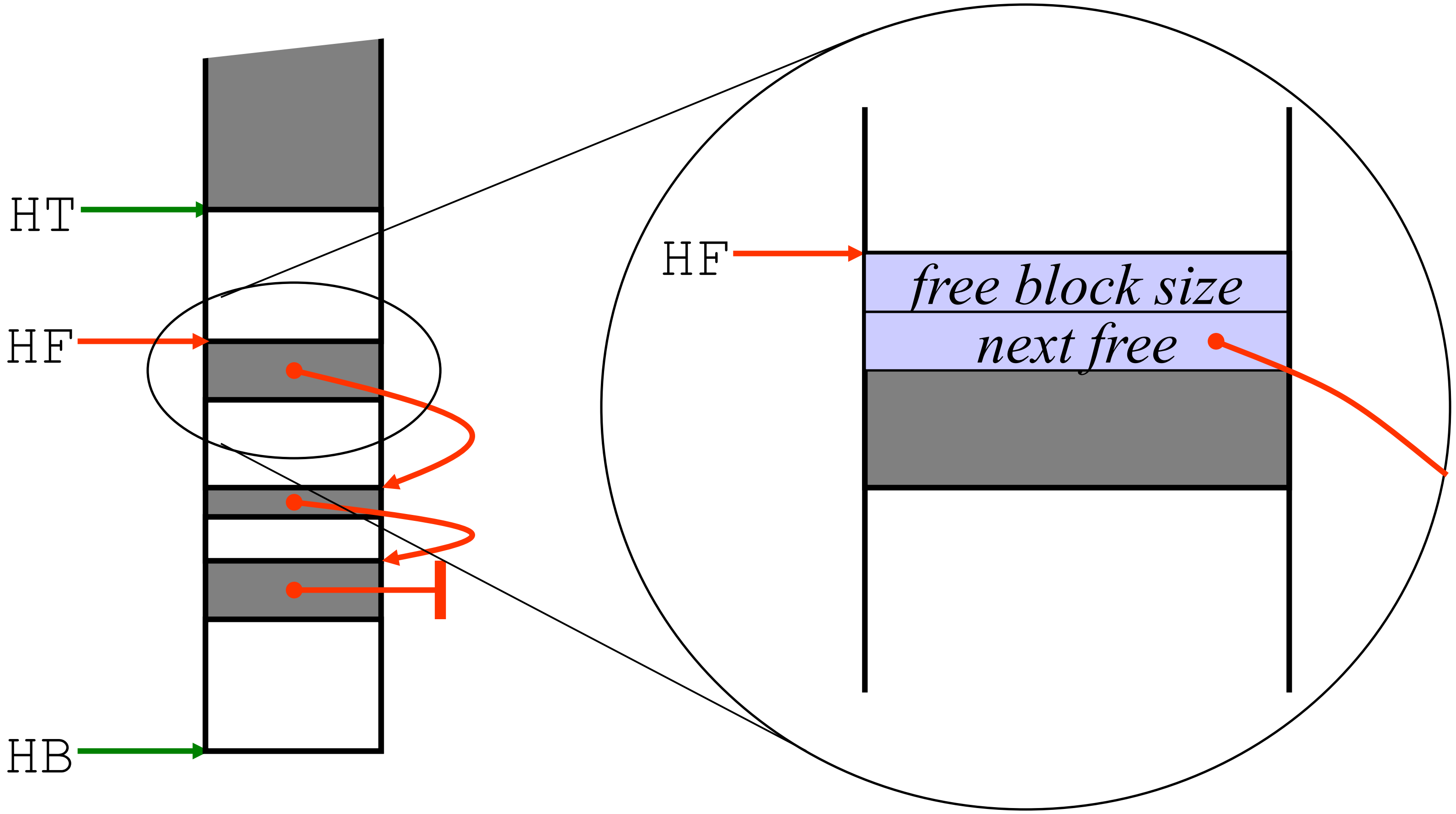


A freelist!
Good idea!

But where do we find the memory to store this data structure?

How to keep track of free memory?

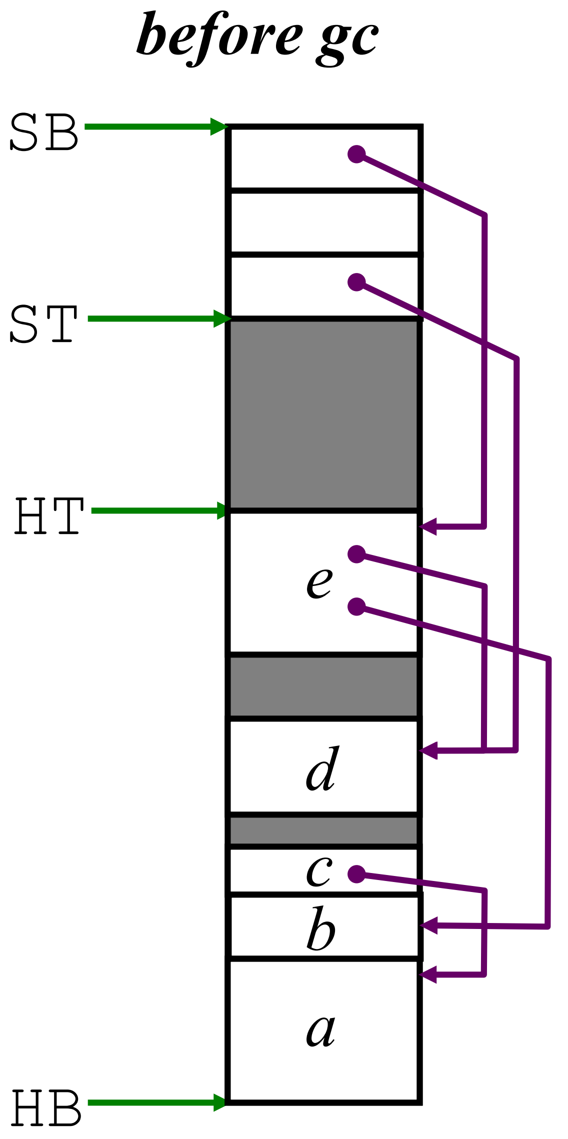
Q: Where do we find the memory to store a freelist data structure?
A: Since the free blocks are not used for anything by the program => memory manager can use them for storing the freelist itself.

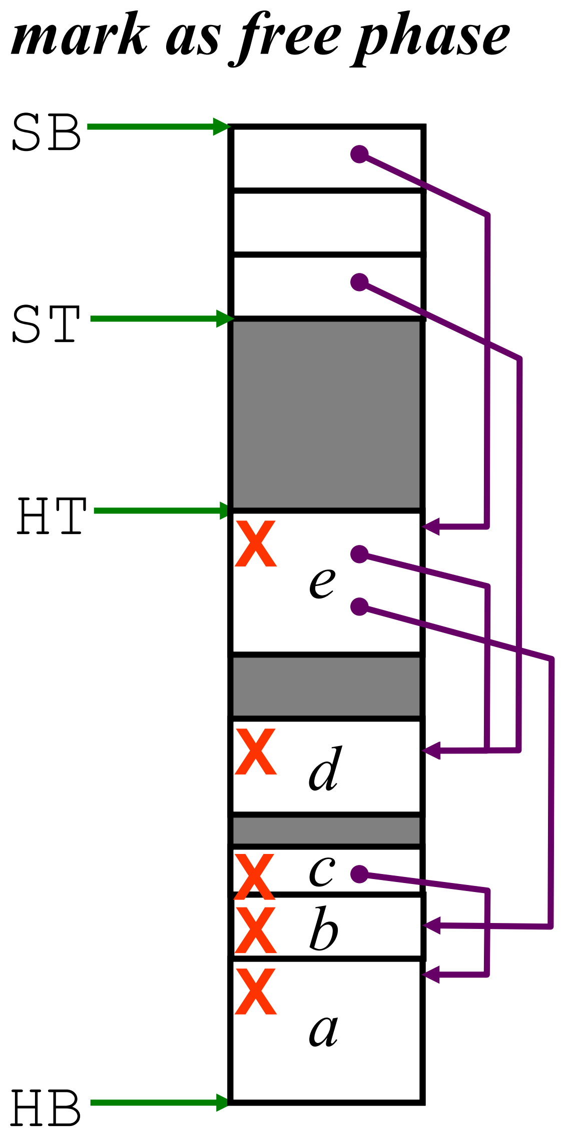


Mark-Sweep

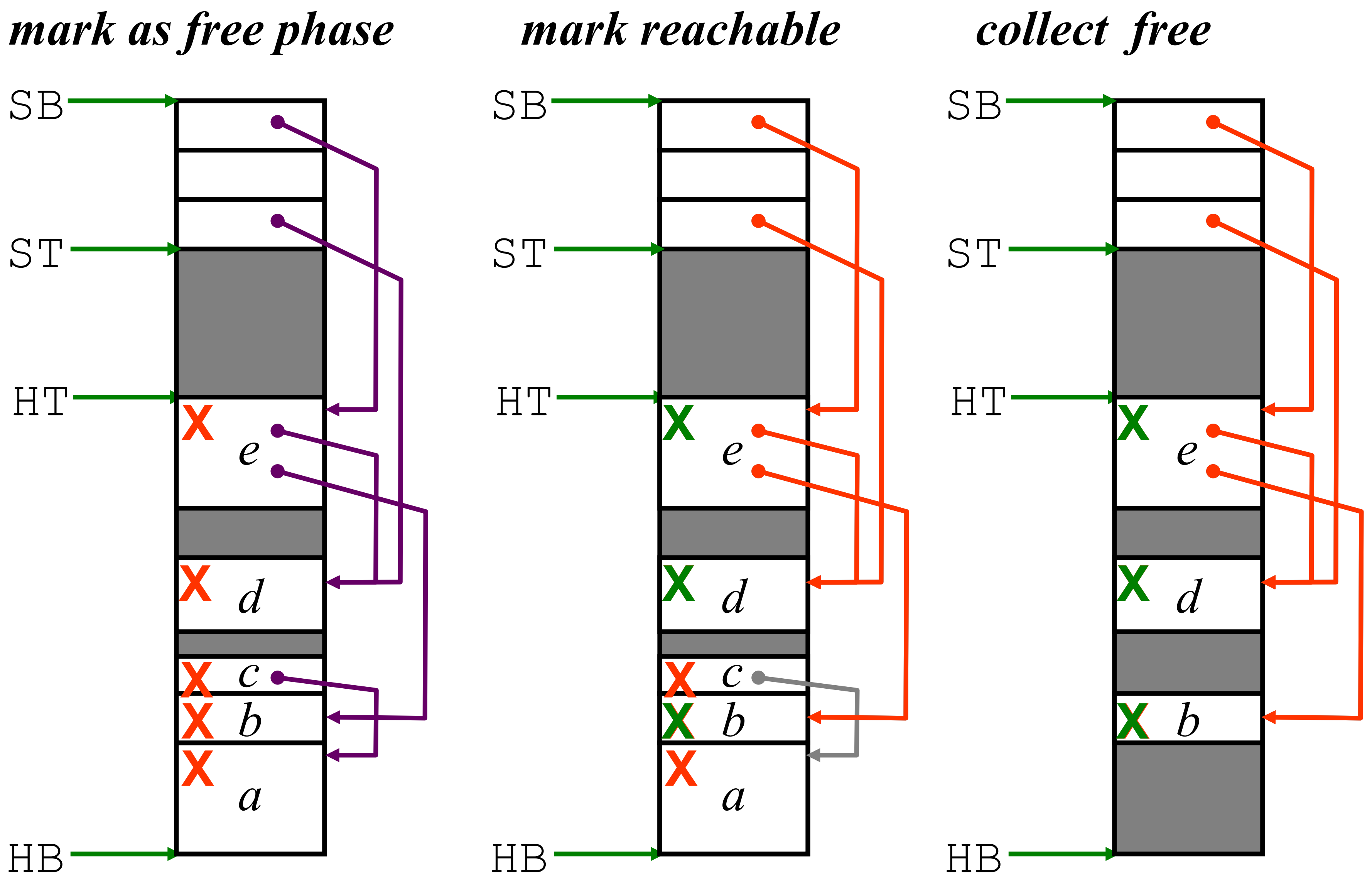
- The first tracing garbage collection algorithm
- Garbage cells are allowed to build up until heap space is exhausted (i.e. a user program requests a memory allocation, but there is insufficient free space on the heap to satisfy the request.)
- At this point, the mark-sweep algorithm is invoked, and garbage cells are returned to the free list.
- Performed in two phases:
  - **Mark:** identifies all live cells by setting a mark bit. Live cells are cells reachable from a root.
  - **Sweep:** returns garbage cells to the free list.

Mark and Sweep Garbage Collection





Mark and Sweep Garbage Collection



Mark and Sweep Garbage Collection

Algorithm pseudo code:

```
void garbageCollect() {
    mark all heap variables as free
    for each frame in the stack
        scan(frame)
    for each heapvar (still) marked as free
        add heapvar to freelist
}
void scan(region) {
    for each pointer p in region
        if p  points to region marked as free then
            mark region at p as reachable
            scan(region at p )
}
```

Q: This algorithm is recursive. What do you think about that?

Mark-Sweep:
Advantages and Disadvantages

- Advantages:
  - Cyclic data structures can be recovered.
  - Tends to be faster than reference counting.
- Disadvantages:
  - Computation must be halted while garbage collection is being performed
  - Every live cell must be visited in the mark phase, and every cell in the heap must be visited in the sweep phase.
  - Garbage collection becomes more frequent as residency of a program increases.
  - May fragment memory.

Mark-Sweep-Compact: Advantages and Disadvantages

- Advantages:
  - The contiguous free area eliminates fragmentation problem. Allocating objects of various sizes is simple.
  - The garbage space is "squeezed out", without disturbing the original ordering of objects. This improves locality.
- Disadvantages:
  - Requires several passes over the data are required. "Sliding compactors" takes two, three or more passes over the live objects.
    - One pass computes the new location
    - Subsequent passes update the pointers to refer to new locations, and actually move the objects

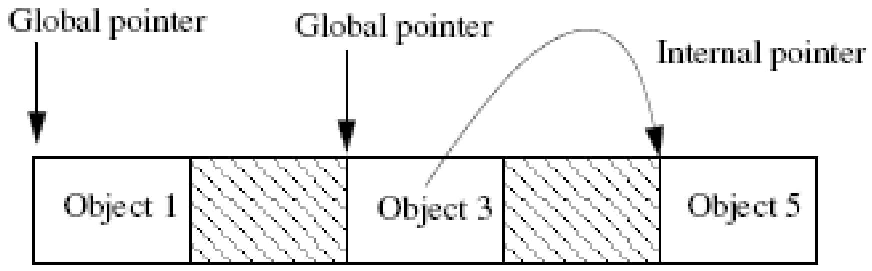


Figure 12.16: Mark-Sweep Garbage Collection

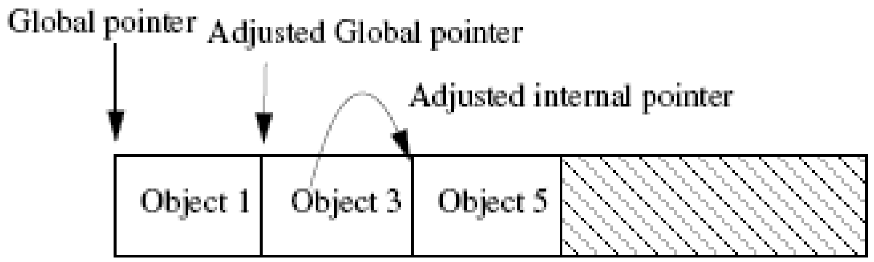


Figure 12.17: Mark-Sweep Garbage Collection with Compaction

Copying Garbage Collection (Cheney's algorithm)

- Like mark-compact, copying garbage collection, but does not really "collect" garbage.
- The heap is subdivided into two contiguous subspaces
  - (FromSpace and ToSpace).
- During normal program execution, only one of these semispaces is in use.
- When the garbage collector is called, all the live data are copied from the current semispace (FromSpace) to the other semispace (ToSpace), so that objects need only be traversed once.
- The work needed is proportional to the amount of live data (all of which must be copied).

Semispace Collector Using the Cheney Algorithm

- The heap is subdivided into two contiguous subspaces (*FromSpace* and *ToSpace*).
- During normal program execution, only one of these semispaces is in use.
- When the garbage collector is called, all the live data are copied from the current semispace (*FromSpace*) to the other semispace (*ToSpace*).

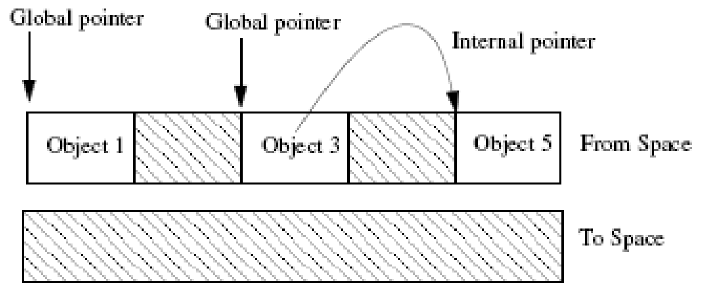


Figure 12.18: Copying Garbage Collection (a)

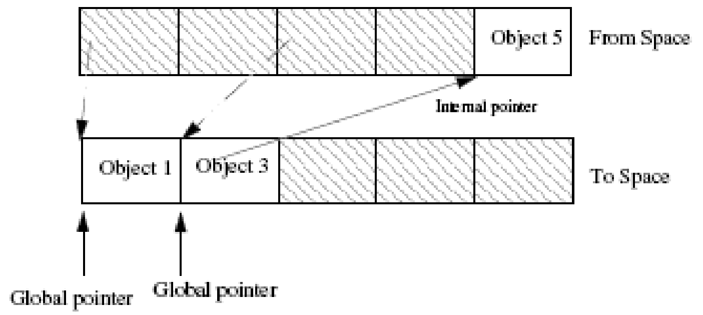


Figure 12.19: Copying Garbage Collection (b)

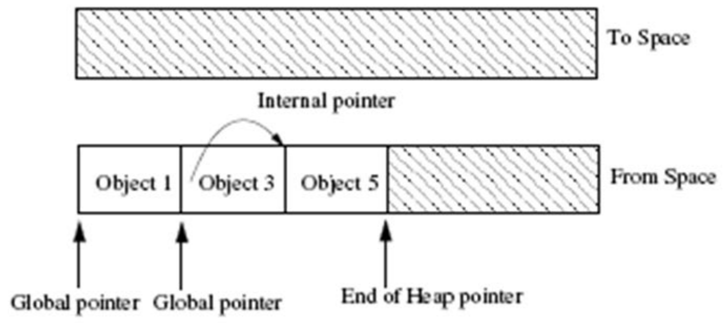


Figure 12.20: Copying Garbage Collection (c)

Copying Garbage Collection: Advantages and Disadvantages

- Advantages:
  - Allocation is extremely cheap.
  - Excellent asymptotic complexity.
  - Fragmentation is eliminated.
  - Only one pass through the data is required.
- Disadvantages:
  - The use of two semi-spaces doubles memory requirement
  - Poor locality. Using virtual memory will cause excessive paging.

Problems with Simple Tracing Collectors

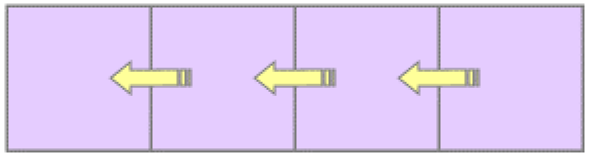
- Difficult to achieve high efficiency in a simple garbage collector, because large amounts of memory are expensive.
- If virtual memory is used, the poor locality of the allocation/reclamation cycle will cause excessive paging.
- Even as main memory becomes steadily cheaper, locality within cache memory becomes increasingly important.

Generational Garbage Collection

- Attempts to address weaknesses of simple tracing collectors such as mark-sweep and copying collectors:
  - All active data must be marked or copied.
  - For copying collectors, each page of the heap is touched every two collection cycles, even though the user program is only using half the heap, leading to poor cache behavior and page faults.
  - Long-lived objects are handled inefficiently.
- Generational garbage collection is based on the *generational hypothesis*:

Most objects die young.

- As such, concentrate garbage collection efforts on objects likely to be garbage: young objects.



Generational Garbage Collection: Multiple Generations

- Advantages:
  - Keeps youngest generation's size small.
  - Helps address mistakes made by the promotion policy by creating more intermediate generations that still get garbage collected fairly frequently.
- Disadvantages:
  - Collections for intermediate generations may be disruptive.
  - Tends to increase number of inter-generational pointers, increasing the size of the root set for younger generations.
- Performs poorly if any of the main assumptions are false:
  - That objects tend to die young.
  - That there are relatively few pointers from old objects to young ones.

Incremental Tracing Collectors

- Program (Mutator) and Garbage Collector run concurrently.
  - Can think of system as similar to two threads. One performs collection, and the other represents the regular program in execution.
- Can be used in systems with real-time requirements. For example, process control systems.
  - allow mutator to do its job without destroying collector's possibilities for keeping track of modifications of the object graph, and at the same time
  - allowing collector to do its job without interfering with mutator

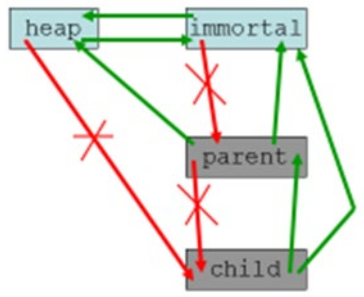
Garbage Collection: Summary

	Method	Conservatism	Space	Time	Fragmentation	Locality
Tracing	**Mark Sweep**	Major	Basic	1 traversal + heap scan	Yes	Fair
	Mark Compact	Major	Basic	Many passes of heap	No	Good
	Copying	Major	Two Semispaces	1 traversal	No	Poor
Incremental	**Reference Counting**	No	Reference count field	Constant per Assignment	Yes	Very Good
	Deferred Reference Counting	Only for stack variables	Reference Count Field	Constant per Assignment	Yes	Very Good
	Incremental	Varies depending on algorithm	Varies	Can be Guaranteed Real-Time	Varies	Varies
	Generational	Variable	Segregated Areas	Varies with number of live objects in new generation	Yes (Non-Copying) No (Copying)	Good

Different choices for different reasons

- JVM
  - Sun Classic: Mark, Sweep and Compact
  - SUN HotSpot: Generational (two generation + Eden)
    - -Xincgc an incremental collector that breaks that old-object region into smaller chunks and GCs them individually
    - -Xconcgc Concurrent GC allows other threads to keep running in parallel with the GC
  - BEA jRockit JVM: concurrent, even on another processor
  - IBM: Improved Concurrent Mark, Sweep and Compact with a notion of weak references
  - Real-Time Java
    - Scoped LTMemory, VTMemory, RawMemory
- .Net CLR
  - Managed and unmanaged memory (memory blob)
  - PC version: Self-tuning Generation Garbage Collector
  - .Net CF: Mark, Sweep and Compact

RTSJ Scoped Memory



- Scopes have fixed lifetimes
- Lifetime starts here:
  - `scopedMemArea.enter() { … }`
- Lifetime ends:
- All calls to new inside a scope, create an object inside of that scope
- When the scope’s lifetime ends, all objects within are destroyed
- Scopes may be nested

Region Based memory management

- Compiler (especially Type inference) automatically detects scopes or regions
- May require programmer to annotate types
- May sometimes have worse behaviour than GC and heap

Summary of Storage Allocation

- Data Representation
  - Non-confusion and uniqueness
  - Direct vs. indirect
- Data Allocation
  - Static
  - Stack
    - Frames, dynamic and static links/display regs, closures
  - Heap
    - Manual vs. automatic
    - Garbage Collection
      - Different algorithms have pros and cons