14 Run-time organization

- Data representation
- Storage organization:
  - stack
  - heap
  - garbage collection

Data representation

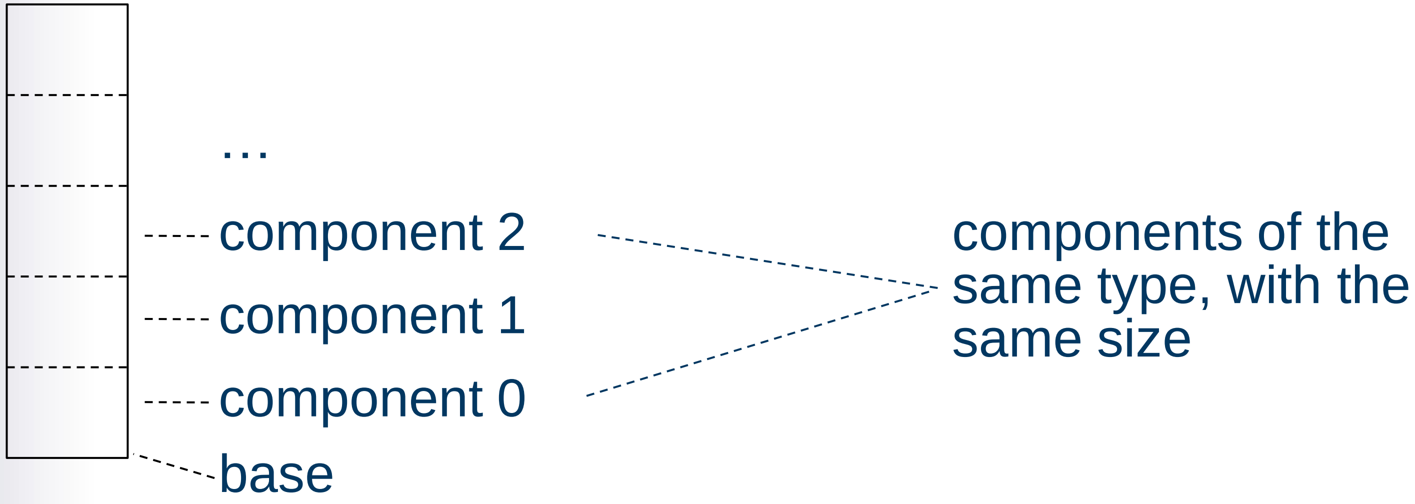
- Assumptions:
  - The PL is statically-typed.
  - The compiler decides the size and layout of each type.
  - All variables of the same type have the same size.
- Here consider representation of:
  - primitive types
  - cartesian products
  - arrays
  - objects.

Representation of primitive types

- Representation of each primitive type may be language-defined or implementation-defined.
- Typically 8, 16, 32, or 64 bits.
- BOOL: 00000000 or 00000001.
- CHAR: 00000000, …, 11111111 (if 8-bit)
- INT: 16-bit or 32-bit or 64-bit twos-complement.
- FLOAT: 32-bit or 64-bit IEEE floating-point.

Representation of arrays *(1)*

- Represent an array by juxtaposing its components.
- Represention of arrays of type {0, 1, 2, …} → *T*:



Representation of arrays *(2)*

- The offset of array component *i* (relative to the array’s base address) is linearly related to *i*:

offset of component *i* = (size of type *T*) × *i*

known to the compiler

unknown

- Since *i* is unknown, the offset of component *i* must be calculated at run-time.

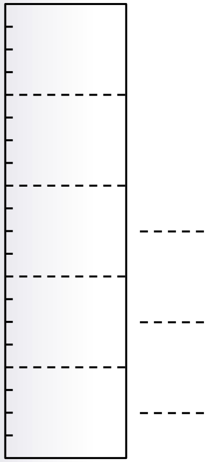
Example: representation of C arrays

- C type definition:

  `typedef int[] Arr;`

  Assume size of INT is 4 bytes

- Possible representation of values of type `Arr`:



…

component 2

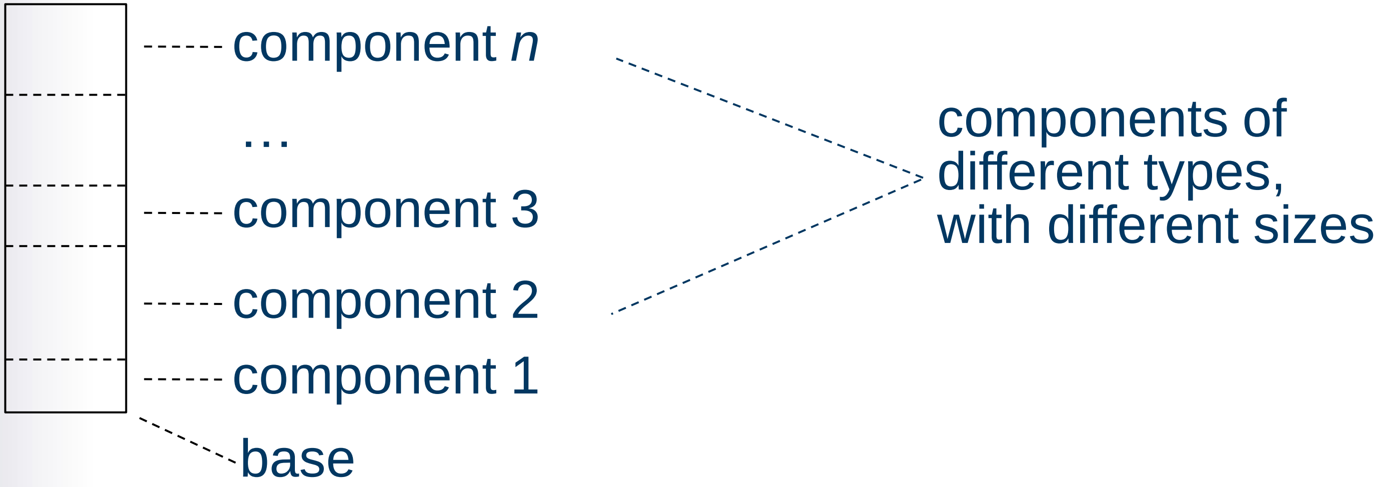
component 1

component 0

(offset of component *i* is 4*i* bytes)

Representation of cartesian products *(1)*

- Represent a tuple (or record or struct) by juxtaposing its components.
- Representation of tuples of type $T_1 \times T_2 \times \ldots \times T_n$:



component *n*

…

component 3

component 2

component 1

base

components of different types, with different sizes

Representation of cartesian products *(2)*

- The compiler knows the offset of each tuple component (relative to the tuple's base address):

offset of component 1 = 0

offset of component 2 = size of type T_1

offset of component 3 = size of type T_1 + size of type T_2

…

offset of component n = size of type T_1 + size of type T_2 + ... + size of type T_{n-1}

all sizes known to the compiler

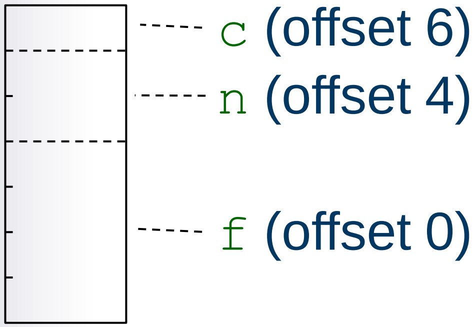
Example: representation of C structs

- C struct type definition:

```
struct Str {
   float f;
   int n;
   char c;
};
```

Assume sizes:
FLOAT 4 bytes
INT 2 bytes
CHAR 1 byte

- Possible representation of structs of type `Str`:



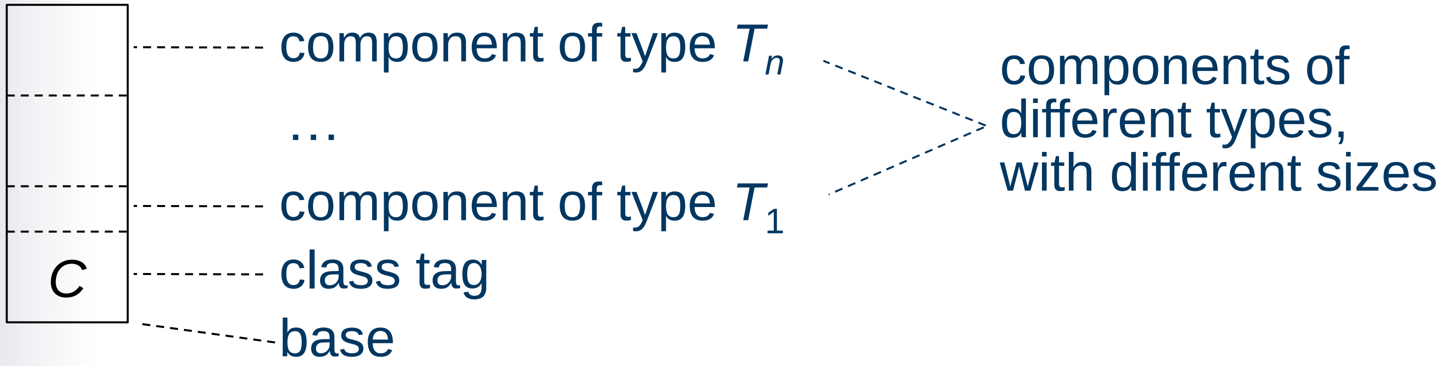
`c` (offset 6)
`n` (offset 4)
`f` (offset 0)

Representation of objects *(1)*

- *Recall:* Objects are tagged tuples.
- Represent an object by juxtaposing its components (instance variables) with a **class tag**.

Representation of objects *(2)*

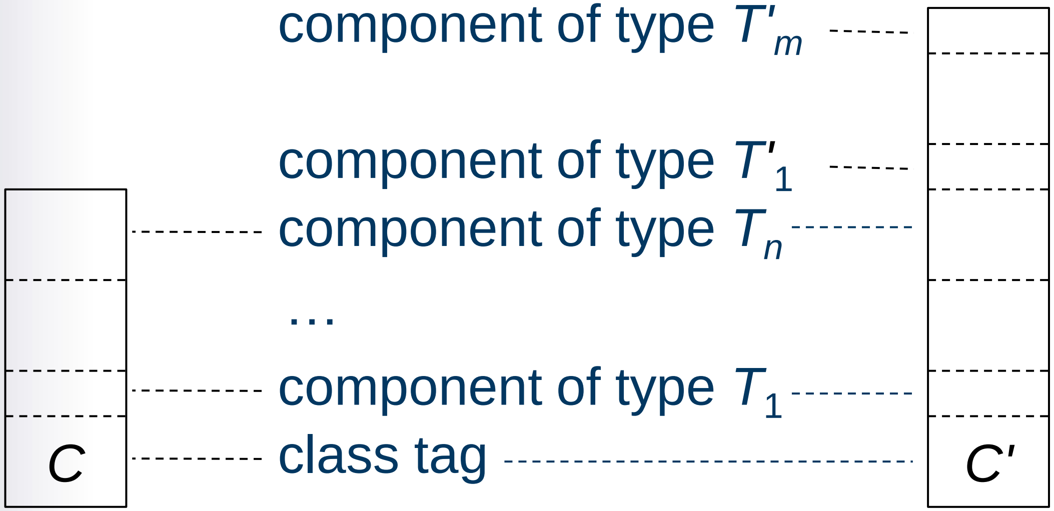
- Consider class *C* with components (instance variables) of types $T_1, \ldots, T_n$.
- Representing objects of class *C*:



- The compiler knows the offset of each component (relative to the object's base address).

Representation of objects *(3)*

- Now consider class *C*′ (a subclass of *C*) with *additional* instance variables of types T'_1, …, T'_m.
- Representation of objects of classes *C* and *C*′:



component of type T'_m

component of type T'_1

component of type T_n

…

component of type T_1

class tag

- Each component has a known offset in objects of a given class *C and all subclasses of C*.

Example: representation of Java objects (1)

- Java class declarations:

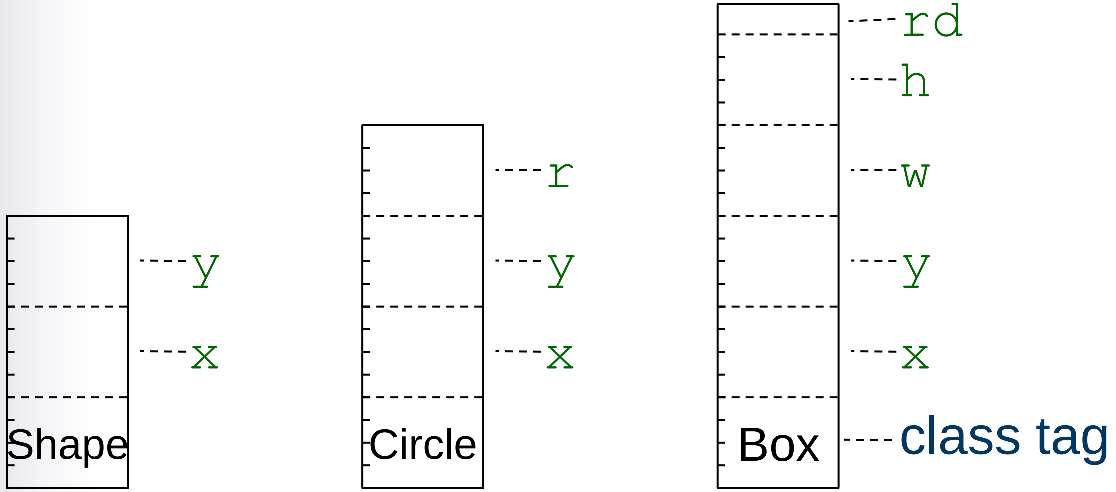
```
class Shape {
    int x, y;
    …
}

class Circle extends Shape {
    int r;
    …
}

class Box extends Shape {
    int w, h;
    boolean rd; // true if corners are rounded
    …
}
```

Example: representation of Java objects (2)

- Representing objects of above classes (simplified):



Storage organization

- Each variable occupies storage space throughout its lifetime. That storage space must be:
  - allocated at the start of the variable’s lifetime
  - deallocated at the end of the variable’s lifetime.
- Assumptions:
  - The PL is statically typed, so every variable’s type is known to the compiler.
  - All variables of the same type occupy the same amount of storage space.

Storage for global and local variables *(1)*

- Recall: A *global variable*’s lifetime is the program’s entire run-time.
- For global variables, the compiler allocates **fixed** storage space.
- Recall: A *local variable*’s lifetime is an activation of the block in which the variable is declared. The lifetimes of local variables are nested.
- For local variables, the compiler allocates storage space on a **stack**.

Storage for global and local variables *(2)*

- At any given time, the stack contains one or more **activation frames**:
  - The frame at the base of the stack contains the global variables.
  - For each *active* procedure *P*, there is a frame containing *P*'s local variables.
- A frame for procedure *P* is:
  - pushed on to the stack when *P* is called
  - popped off the stack when *P* returns.

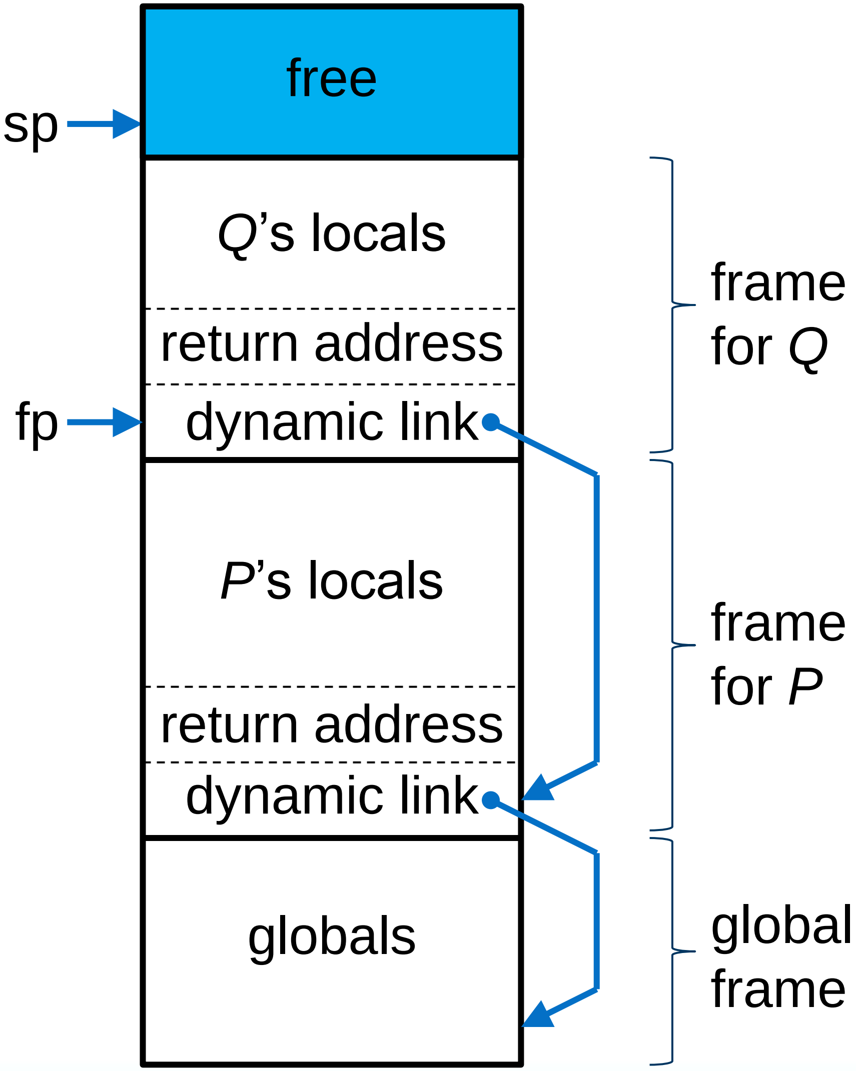
An active procedure is one that has been called but not yet returned.

Storage for global and local variables *(3)*

- The compiler fixes the size and layout of each frame.
- The offset of each global/local variable (relative to the base of the frame) is known to the compiler.

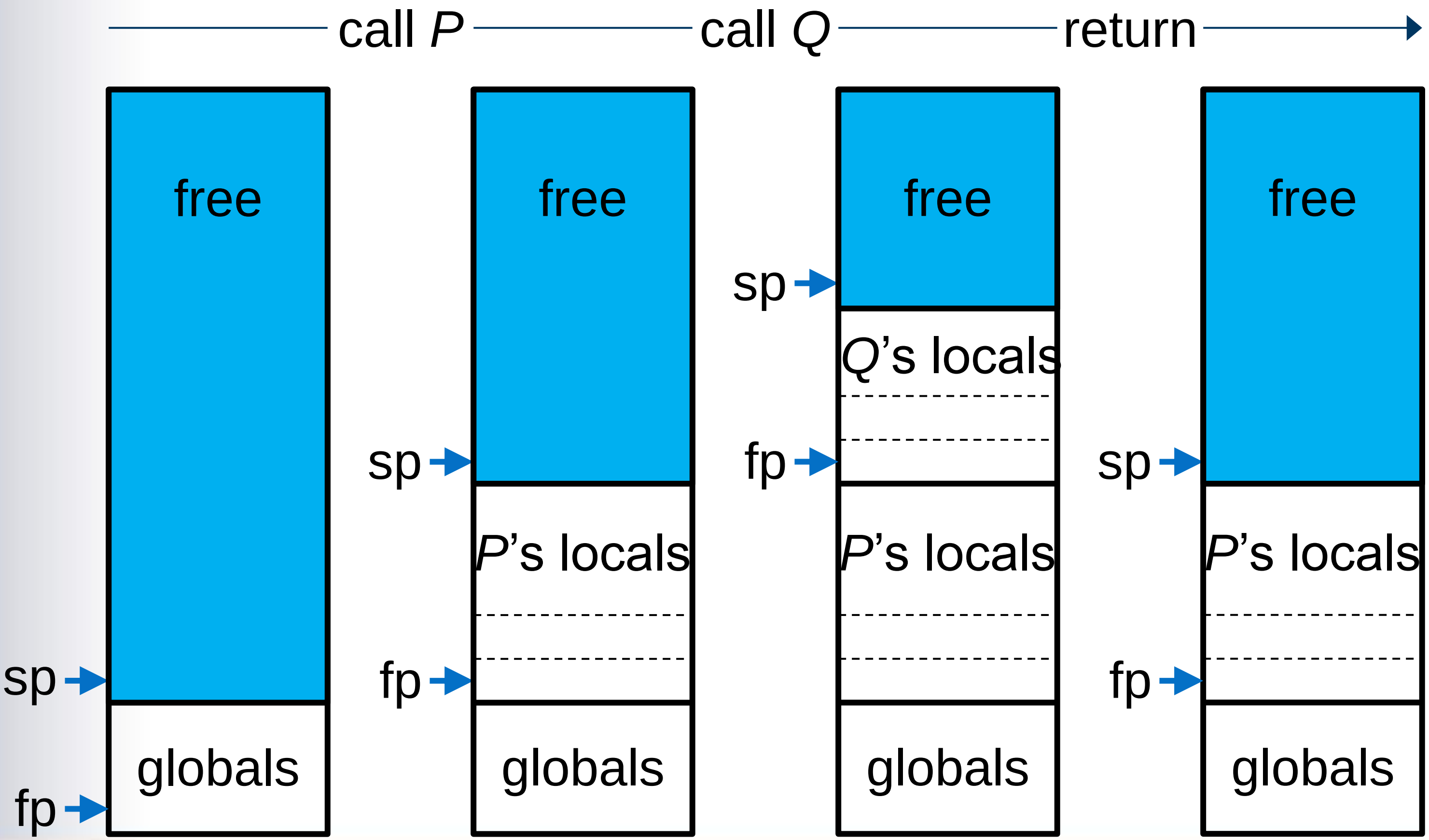
Example: storage for global and local variables in SVM *(1)*

- SVM data store when the main program has called *P*, and *P* has called *Q*:
- **sp** (stack pointer) points to the first free cell above the top of the stack.
- **fp** (frame pointer) points to the first cell of the topmost frame.



Example: storage for global and local variables in SVM *(2)*

- Effect of calls and returns:



Storage for heap variables *(1)*

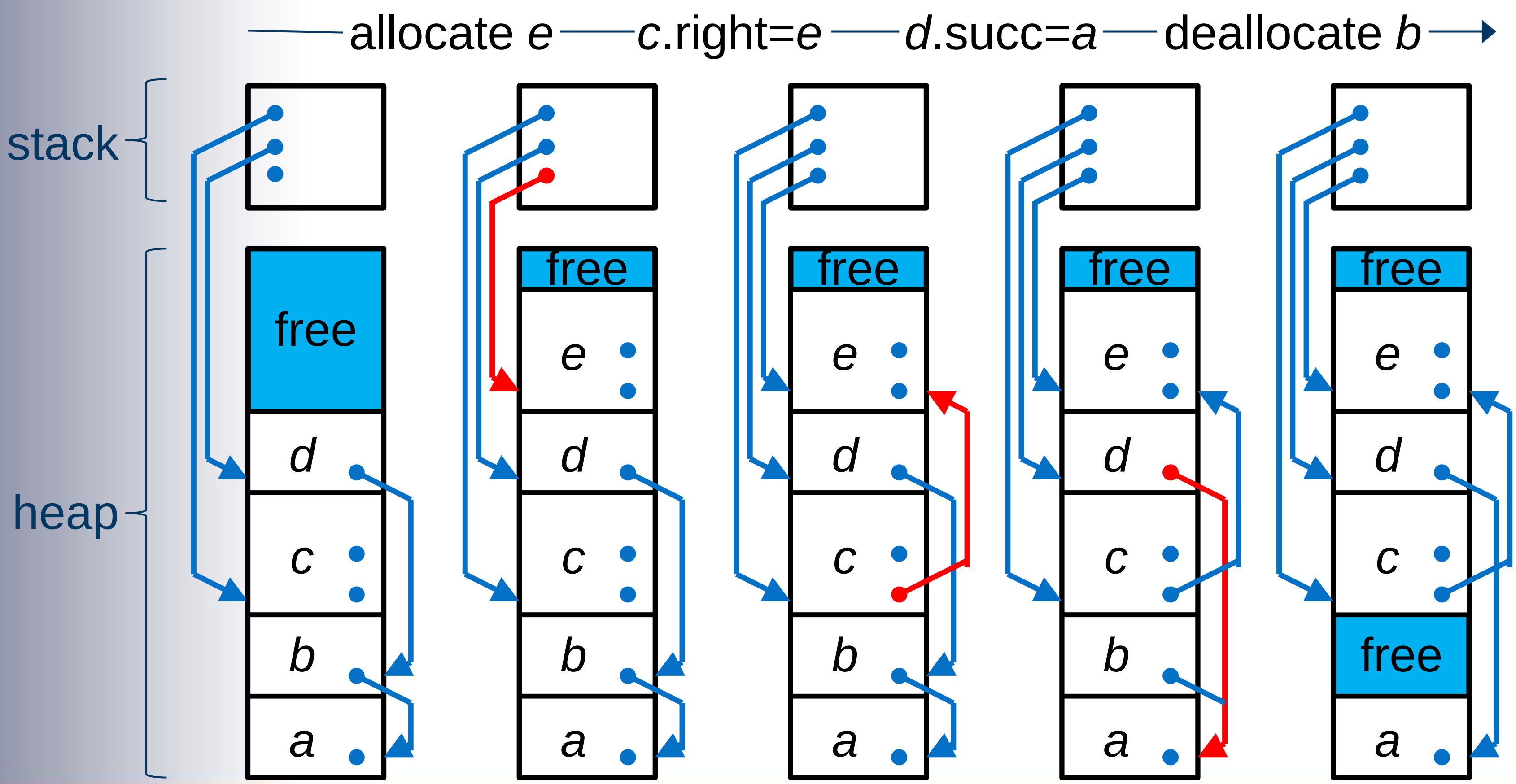
- Recall: A *heap variable*’s lifetime starts when the heap variable is created and ends when it is destroyed or becomes unreachable. The lifetimes of heap variables follow no pattern.
- Heap variables occupy a storage region called the **heap**. At any given time, the heap contains all currently-live heap variables, interspersed with free space.
  - When a new heap variable is to be created, some free space is allocated to it.
  - When a heap variable is to be destroyed, its allocated space reverts to being free.

Storage for heap variables *(2)*

- The **heap manager** (part of the PL's run-time system) keeps track of free space within the heap
  - usually by means of a **free-list**: a linked list of free areas of various sizes.
- The heap manager provides:
  - a routine to **create** a heap variable (called by the PL's allocator)
  - a routine to **destroy** a heap variable (called by the PL's deallocator, if any).

Example: storage for heap variables

- Effect of allocations and deallocations:



Garbage collection

- If the PL has no deallocator, the heap manager must be able to find and destroy any unreachable heap variables *automatically*. This is done by a **garbage collector**.
- A garbage collector must visit *all* reachable heap variables. This is inevitably time-consuming.
- A **mark-sweep** garbage collector is the simplest. It first marks all reachable heap variables, then deallocates all unmarked heap variables.

Mark-sweep garbage collection algorithm

- Mark-sweep algorithm:

To mark all heap variables reachable from pointer *p*:

```
1  Let heap variable v be the referent of p.
2  If v is unmarked:
   2.1   Mark v.
   2.2   For each pointer q in v:
         2.2.1  Mark all heap variables reachable from q.
```

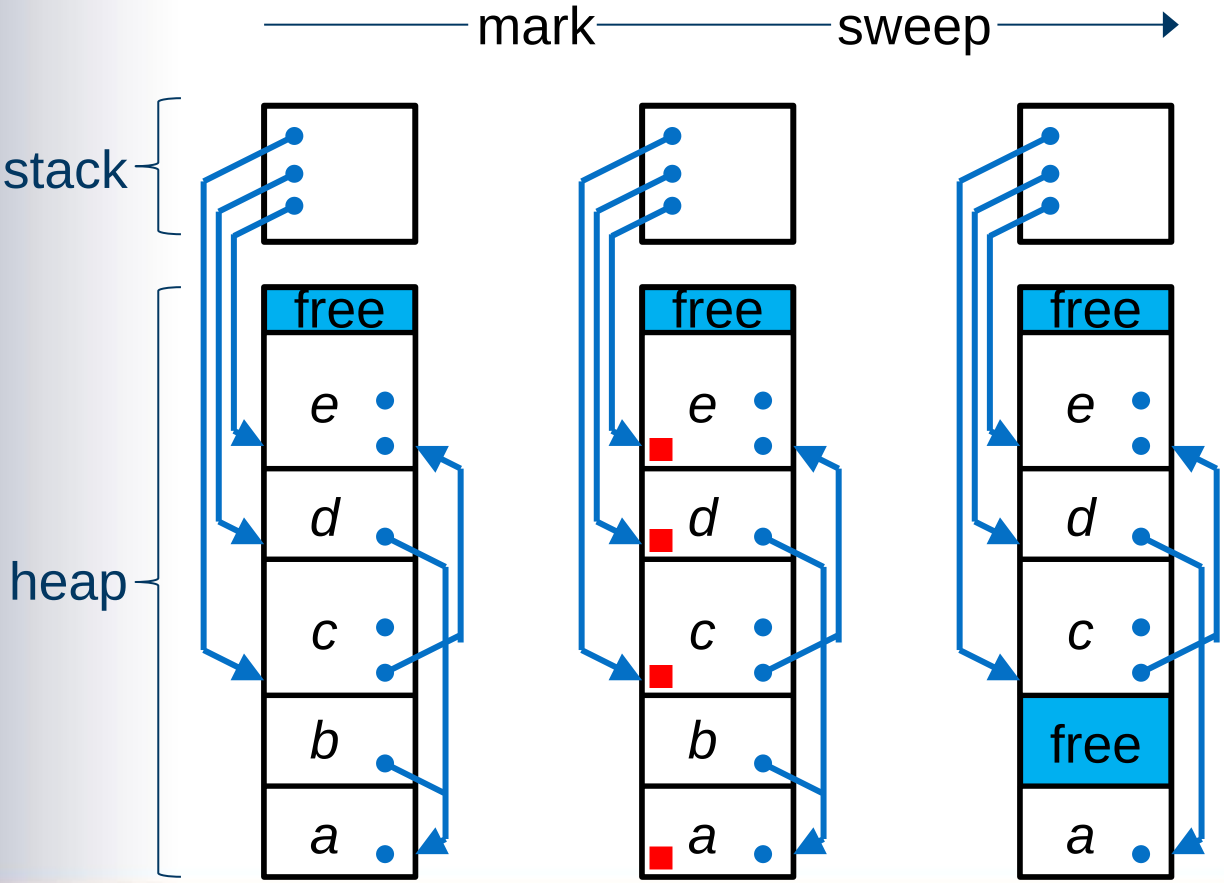
depth-first traversal

To collect garbage:

```
1  For each pointer p in a global/local variable:
   1.1   Mark all heap variables reachable from p.
2  For each heap variable v:
   2.1   If v is unmarked, destroy v.
   2.2   Else, if v is marked, unmark v.
```

Example: mark-sweep garbage collection

- Effect of mark and sweep:



Issues

- Time complexity of mark-sweep garbage collection is $O(n_r + n_h)$
  - n_r = number of reachable heap variables
  - n_h = total number of heap variables.
- The heap tends to become fragmented:
  - There might be many small free areas, but none big enough to allocate a new large heap variable.
  - Partial solution: coalesce adjacent free areas in the heap.
  - Better solution: use a copying or generational garbage collector.

Copying garbage collector

- A **copying** garbage collector maintains two separate heap spaces:
  - – Initially, space 1 contains all heap variables; space 2 is spare.
  - – Whenever the garbage collector reaches an unmarked heap variable *v*, it copies *v* from space 1 to space 2.
  - – At the end of garbage collection, spaces 1 and 2 are swapped.
- Pros and cons:
  - \+ Heap variables can be consolidated when copied into space 2.
  - – All pointers to a copied heap variable must be found and redirected from space 1 to space 2.

Generational garbage collector

- A **generational** garbage collector maintains two (or more) separate heap spaces:
  - One space (the *old generation*) contains only long-lived heap variables; the other space (the *young generation*) contains shorter-lived heap variables.
  - The old generation is garbage-collected *infrequently* (since long-lived heap variables are rarely deallocated).
  - The young generation is garbage-collected *frequently* (since short-lived heap variables are often deallocated).
  - Heap variables that live long enough may be promoted from the young generation to the old generation.
- Pro:
  - \+ Garbage collection is more focussed.