Data Types and Memory Management

CSCI 3136

Principles of Programming Languages

Faculty of Computer Science

Dalhousie University

Winter 2012

Reading: Chapter 7

What is a Type System?

A type system is a mechanism for defining types and associating them with operations that can be performed on objects of this type.

- Built-in types with built-in operations
- Custom operations for built-in and custom types

A type system includes rules that specify

- **Type equivalence:** Do two values have the same type? (Structural equivalence vs name equivalence)
- **Type compatibility:** Can a value of a certain type be used in a certain context?
- **Type inference:** How is the type of an expression computed from the types of its parts?

What is a Type System?

A type system is a mechanism for defining types and associating them with operations that can be performed on objects of this type.

- Built-in types with built-in operations
- Custom operations for built-in and custom types

A type system includes rules that specify

- **Type equivalence:** Do two values have the same type? (Structural equivalence vs name equivalence)
- **Type compatibility:** Can a value of a certain type be used in a certain context?
- **Type inference:** How is the type of an expression computed from the types of its parts?

Type errors are a nuisance. Why don't we just get rid of types completely?

Common Kinds of Type Systems

Strongly typed: Prohibits application of an operation to any object not supporting this operation.

Statically typed: Strongy typed and type checking is performed at compile time (Pascal, C, Haskell, . . .)

Dynamically typed: Types of operands of operations are checked at run time (LISP, Smalltalk, . . .)

In some languages (e.g., ML), the programmer does not specify types at all. They are inferred by the compiler.

Definition of Types

Similar to subroutines in many languages, defining a type has two parts:

- A type's ***declaration*** introduces its name into the current scope.
- A type's ***definition*** describes the type (the simpler types it is composed of).

Definition of Types

Similar to subroutines in many languages, defining a type has two parts:

- A type's ***declaration*** introduces its name into the current scope.
- A type's ***definition*** describes the type (the simpler types it is composed of).

Classification of types:

Denotational: A type is a set of values.

Constructive: A type is built-in or composite.

Abstraction-based: A type is defined by an interface, the set of operations it supports.

Constructive Classification of Types

Built-in types

- Integers, Booleans, characters, real numbers, ...

Enumeration and range types (neither built-in nor composite)

- C: `enum t { A, B }`
- Pascal: `0..100`

Composite types

- Records, arrays, files, lists, sets, pointers, ...

Records

A nested record definition in Pascal:

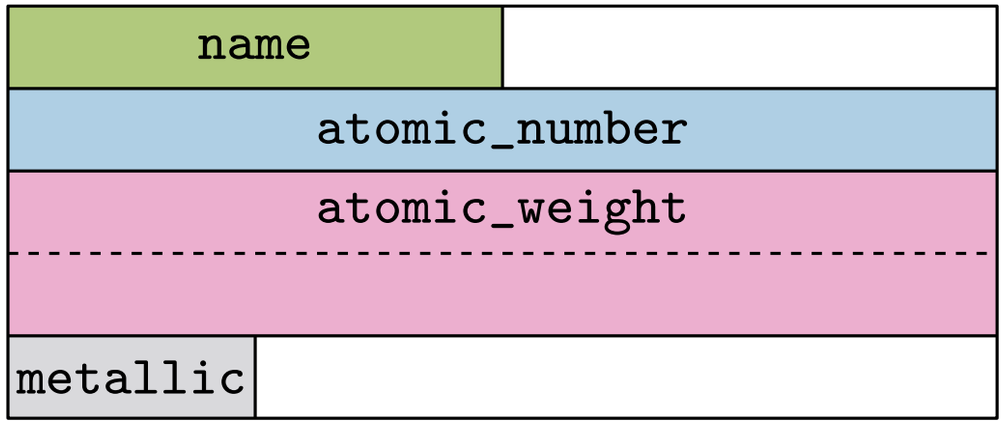
```
type ore = record
  name            : short_string;
  element_yielded : record
    name          : two_chars;
    atomic_n      : integer;
    atomic_weight : real;
    metallic      : Boolean
  end
end;
```

Accessing fields

- `ore.element_yielded.name`
- `name of element_yielded of ore`

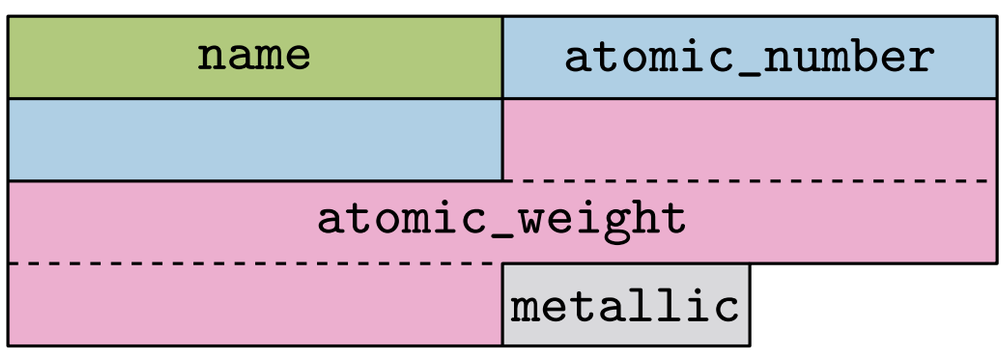
Memory Layout of Records

Aligned (fixed ordering)



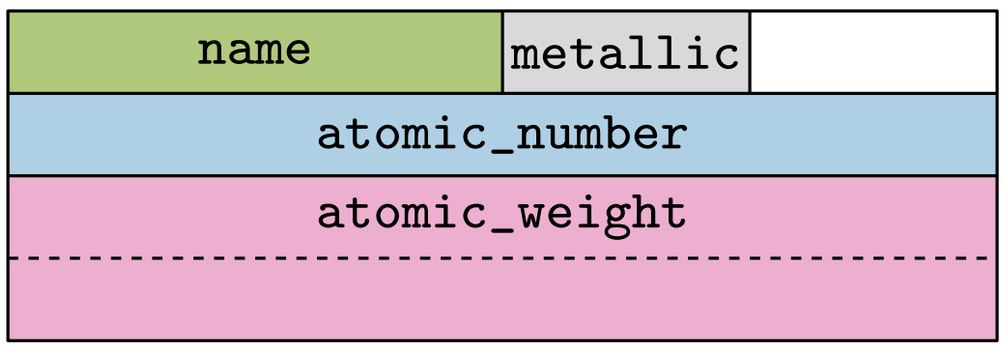
− Potential waste of space
+ One machine operation per element access
+ Guaranteed layout in memory (good for systems programming)

Packed



+ No waste of space
− Multiple machine operations per memory access
+ Guaranteed layout in memory (good for systems programming)

Aligned (optimized ordering)



± Reduced space overhead
+ One machine operation per memory access
− No guarantee of layout in memory (bad for systems programming)

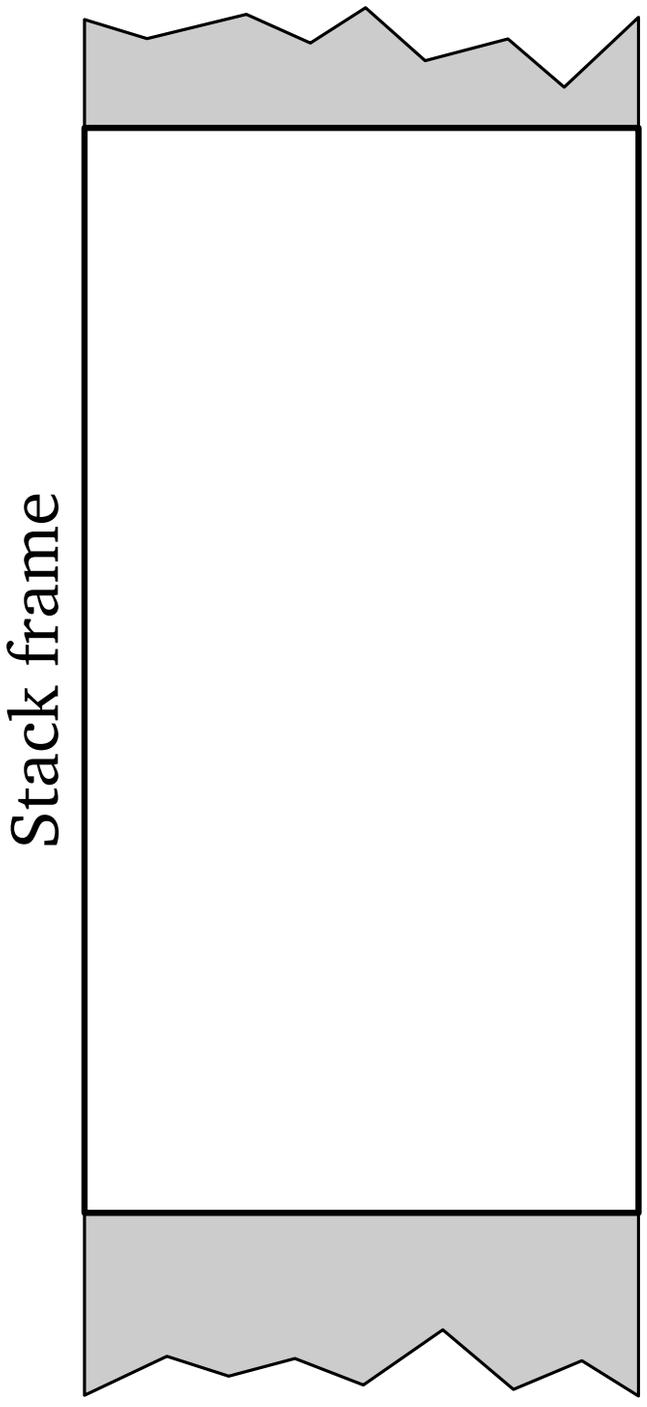
Arrays

	Shape fixed when?		
Stored where?	Compile time	Elaboration time	Dynamic
Static address	Static	—	—
Stack	Local	Local	—
Heap	Dynamic	Dynamic	Dynamic

Issues

- Memory allocation
- Bounds checks
- Index calculations (higher-dimensional arrays)

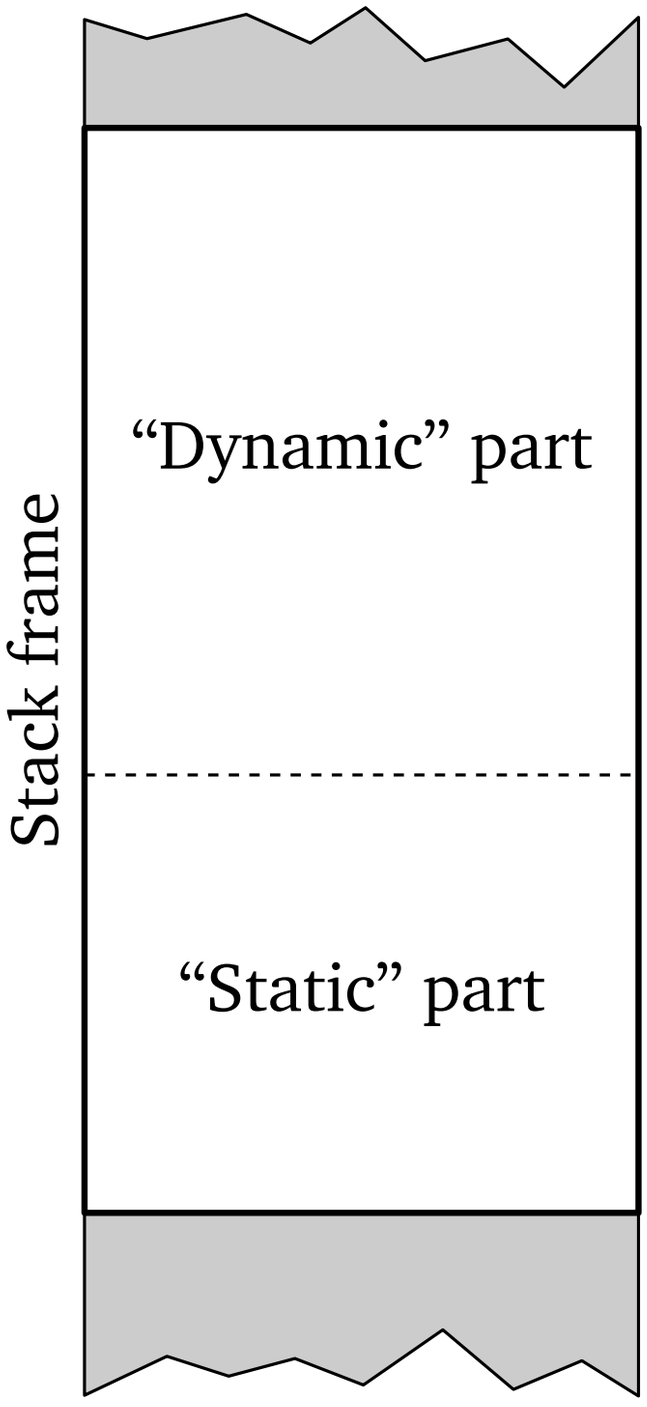
Elaboration-Time Shape Binding and Dope Vectors



Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

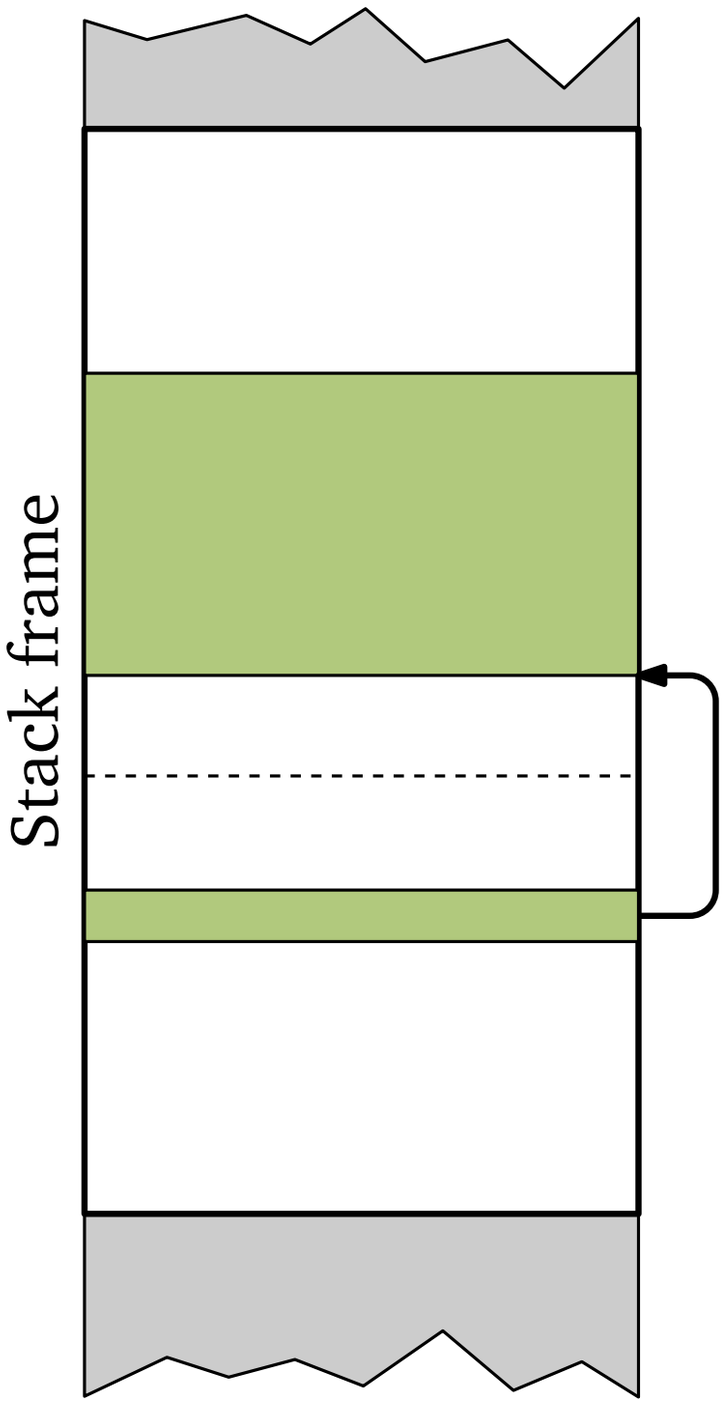
Elaboration-Time Shape Binding and Dope Vectors



Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

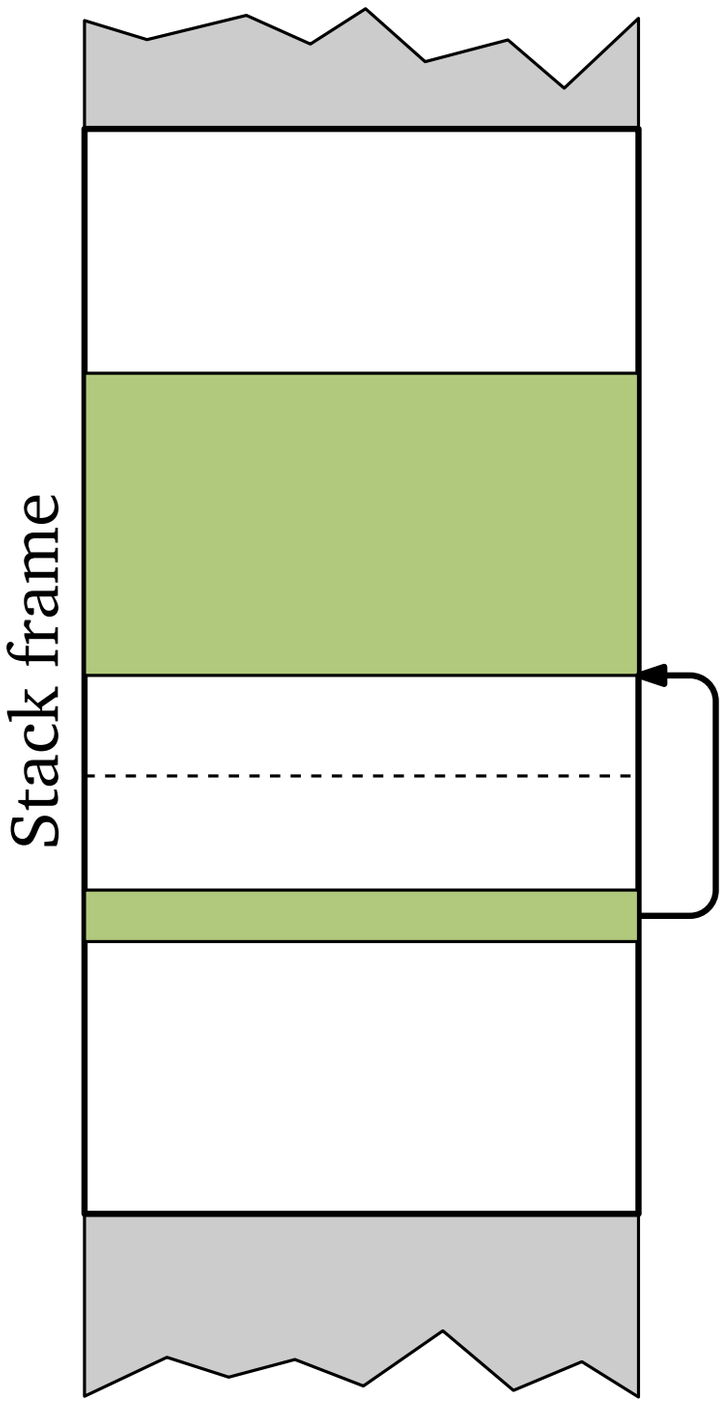
Elaboration-Time Shape Binding and Dope Vectors



Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

Elaboration-Time Shape Binding and Dope Vectors

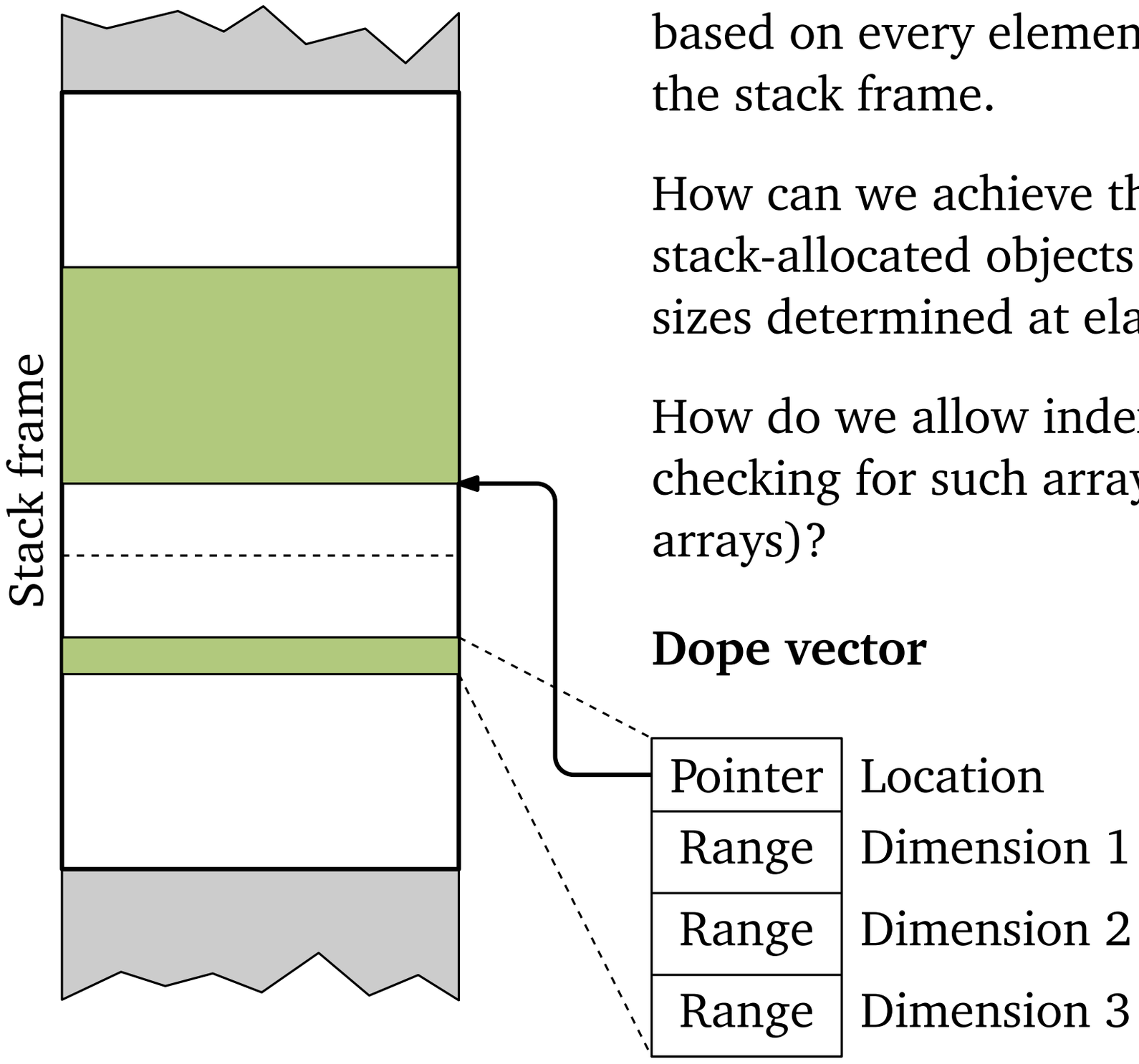


Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

How do we allow index calculation and bound checking for such arrays (and heap-allocated arrays)?

Elaboration-Time Shape Binding and Dope Vectors

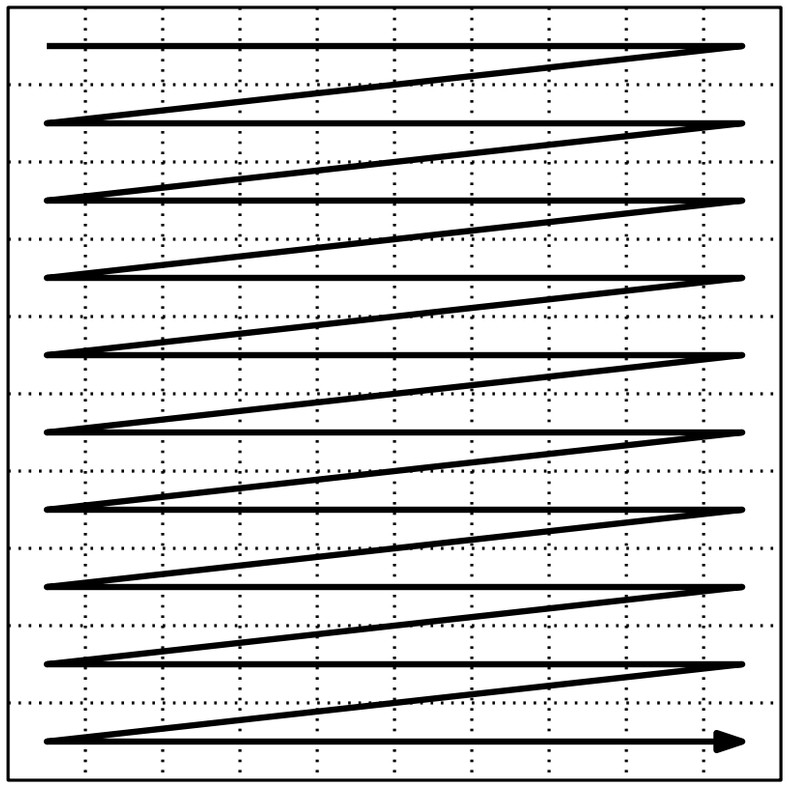


Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

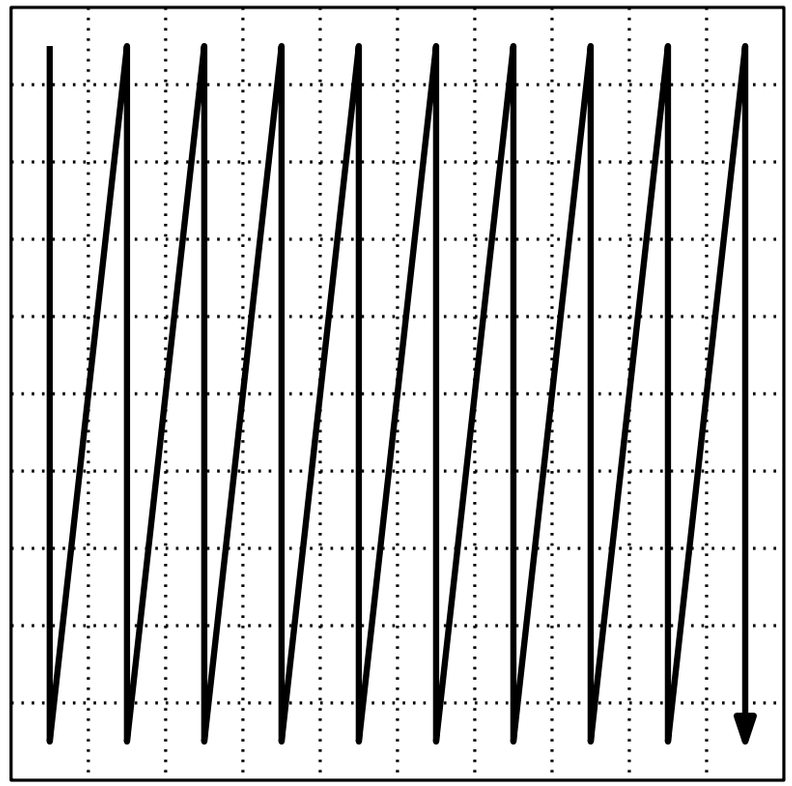
How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

How do we allow index calculation and bound checking for such arrays (and heap-allocated arrays)?

Contiguous Memory Layouts for 2-d Arrays

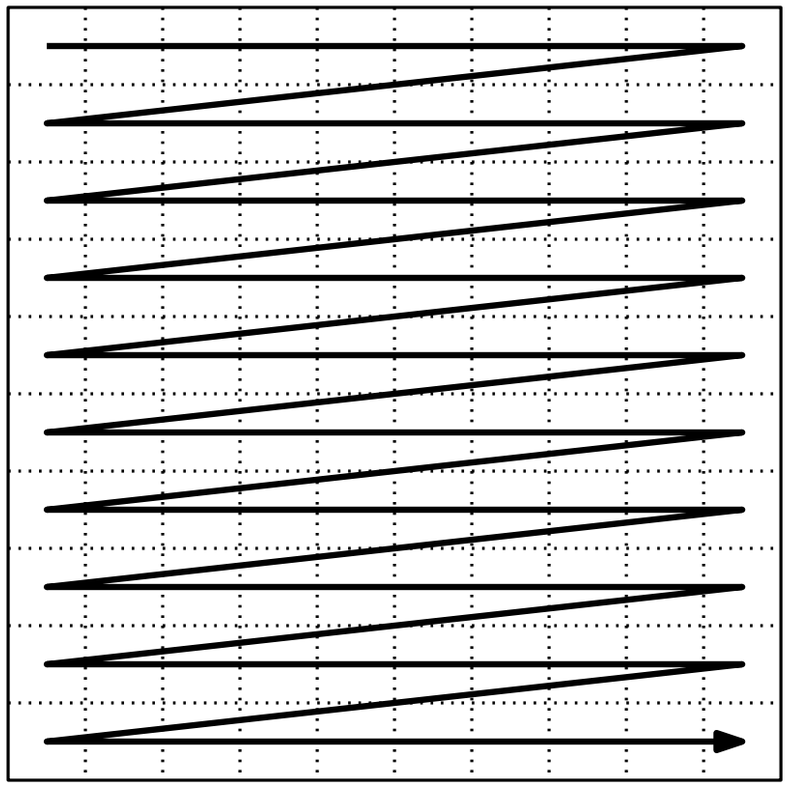


Row-major layout

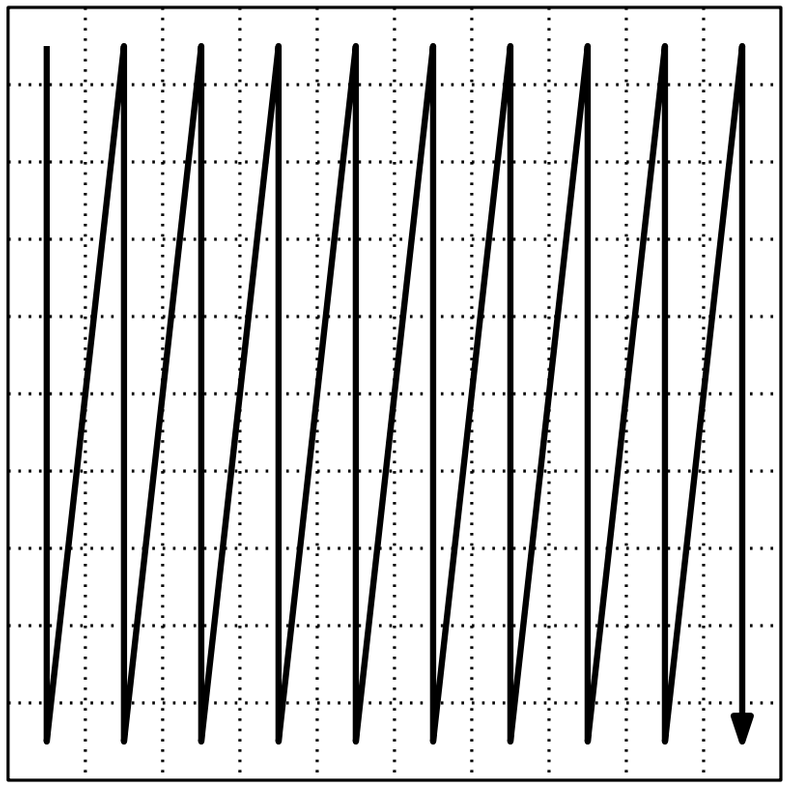


Column-major layout

Contiguous Memory Layouts for 2-d Arrays



Row-major layout



Column-major layout

There are more sophisticated block-recursive layouts which, combined with the right algorithms, achieve much better cache efficiency than the above.

Contiguous and Row-Pointer Layout

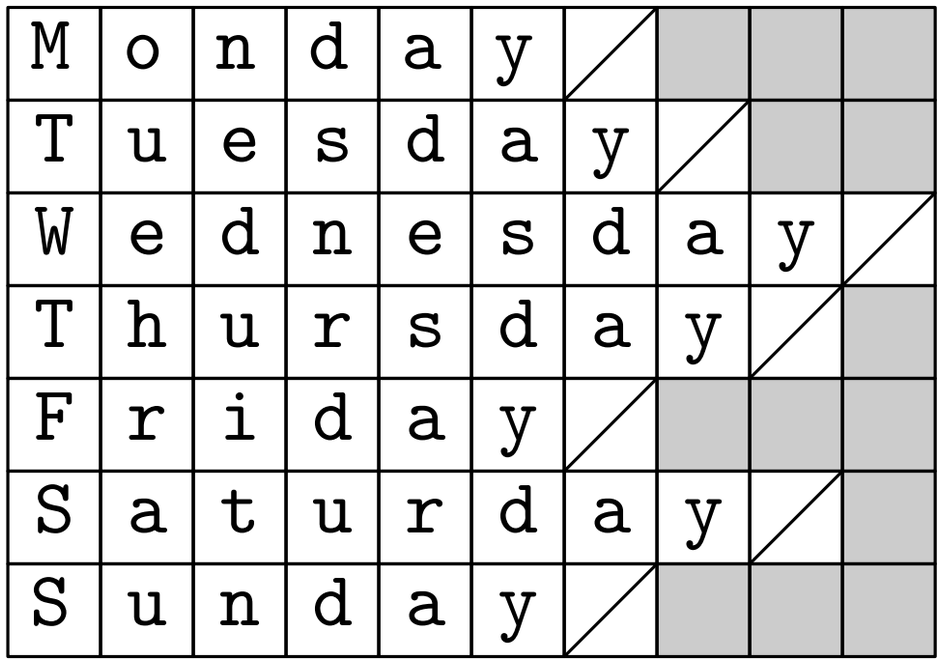
```
char days[][10] = {
  "Monday", "Tuesday",
  "Wednesday", "Thursday",
  "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```

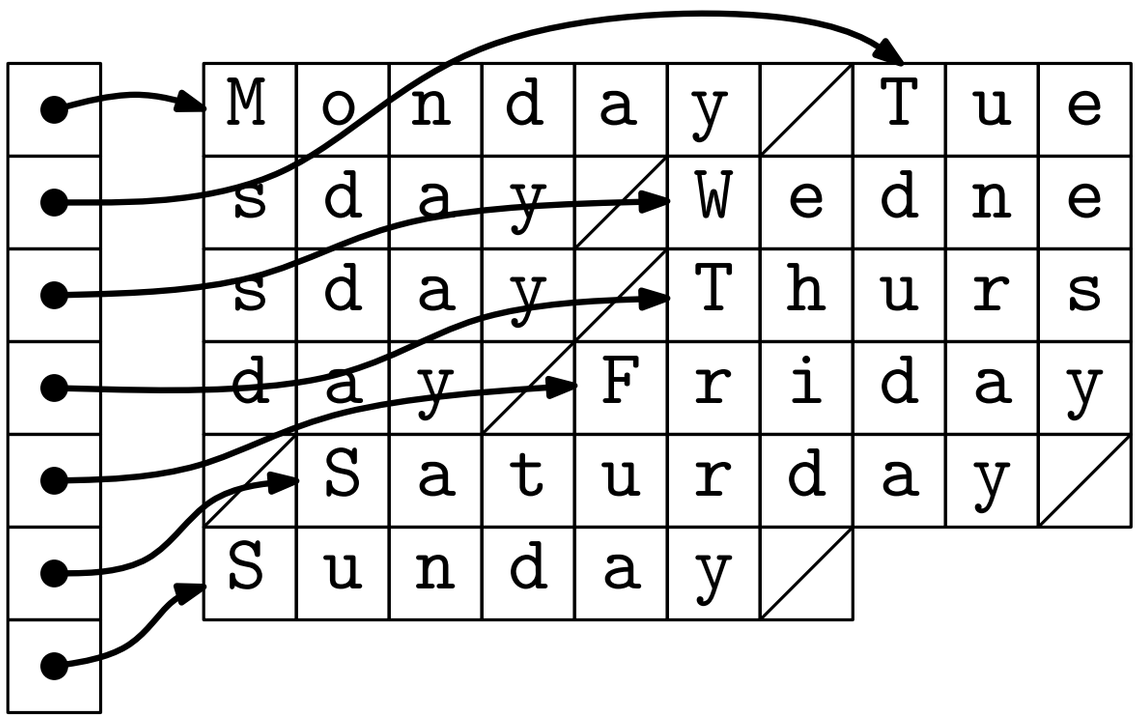
M	o	n	d	a	y			
T	u	e	s	d	a	y		
W	e	d	n	e	s	d	a	y
T	h	u	r	s	d	a	y	
F	r	i	d	a	y			
S	a	t	u	r	d	a	y	
S	u	n	d	a	y			

Contiguous and Row-Pointer Layout

```
char days[][10] = {
  "Monday", "Tuesday",
  "Wednesday", "Thursday",
  "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```

```
char *days[] = {
  "Monday", "Tuesday",
  "Wednesday", "Thursday",
  "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```

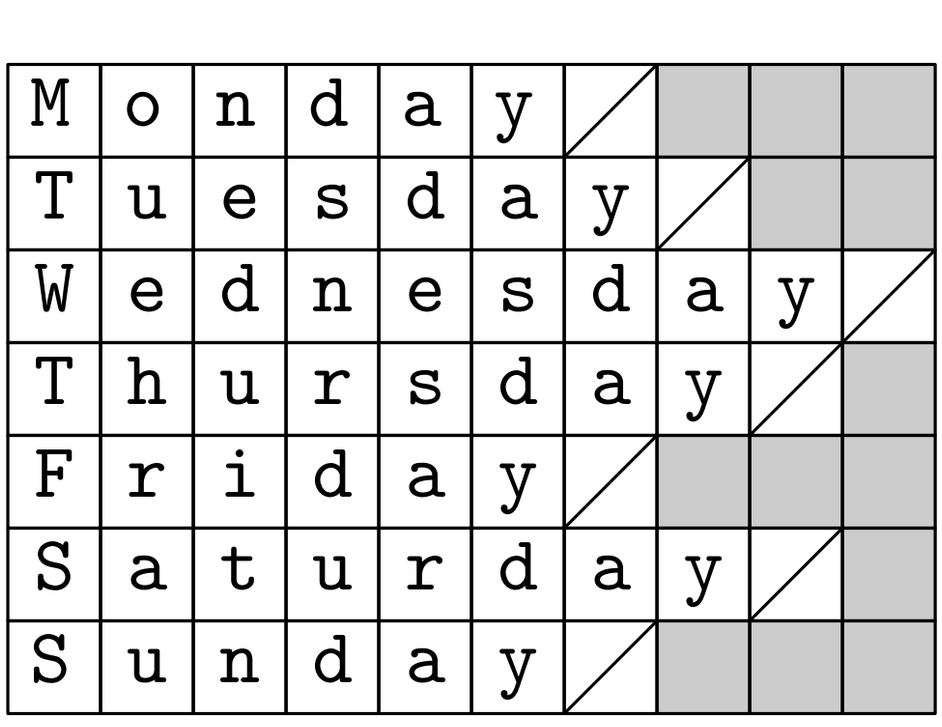


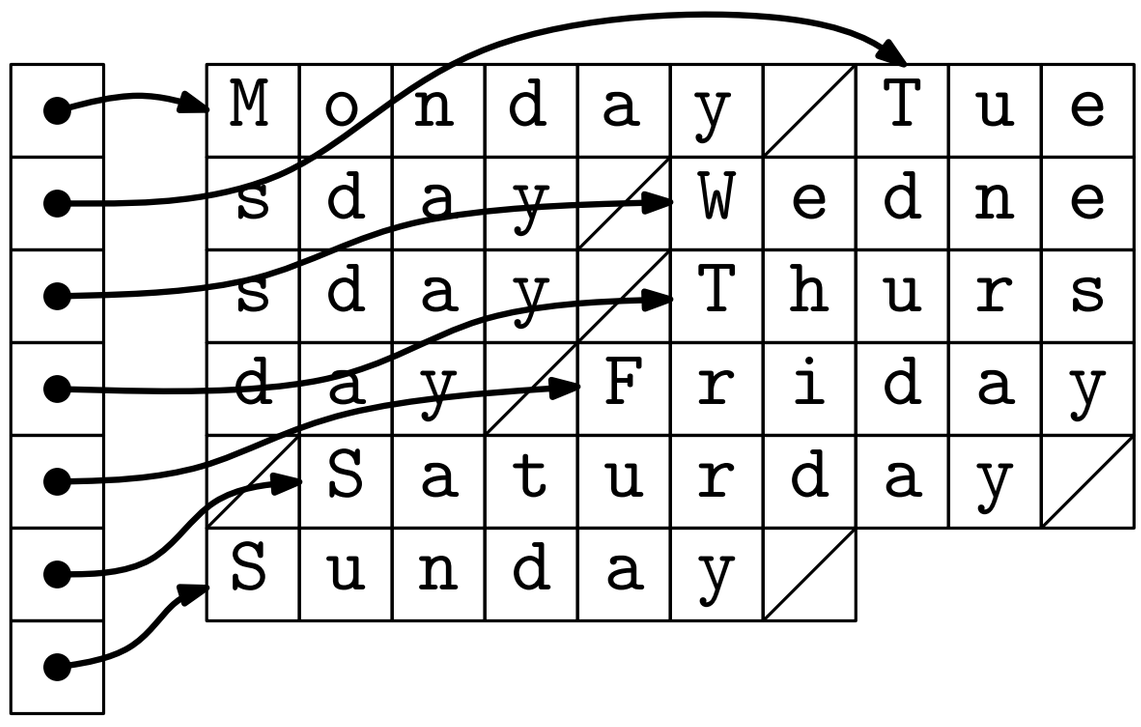


Contiguous and Row-Pointer Layout

```
char days[][10] = {
  "Monday", "Tuesday",
  "Wednesday", "Thursday",
  "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```

```
char *days[] = {
  "Monday", "Tuesday",
  "Wednesday", "Thursday",
  "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```





Memory layout determines space usage and nature and efficiency of address calculations.

Associative Arrays

- Allow arbitrary elements as indexes.
- Directly supported in Perl and many other scripting languages.
- C++ and Java call them **maps** (because they're not really arrays!)
- Scheme calls them **association lists** (A-lists).

```
(define e '((a 1) (b 2) (c 3))
```

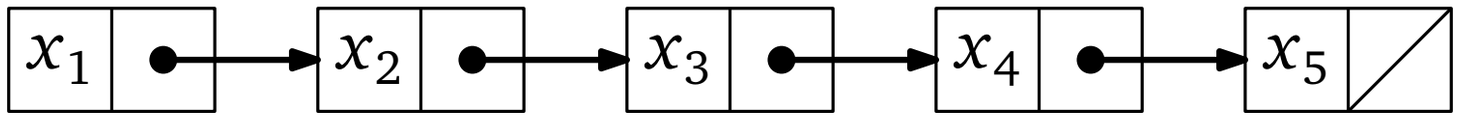
`(assoc 'a e)` returns `(a 1)`

Different lookup operations for different notions of equality

- `assq` uses `eq?`
- `assoc` uses `equal?`
- `assv` uses `eqv?`

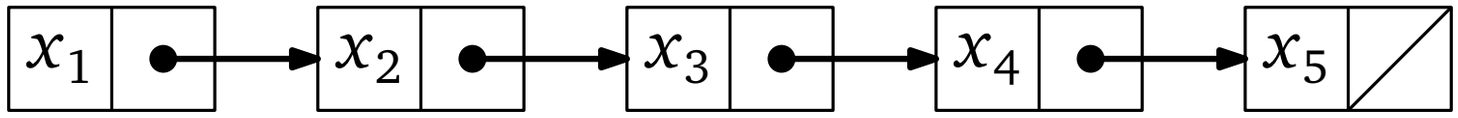
Arrays, Lists, and Strings

A list



Arrays, Lists, and Strings

A list

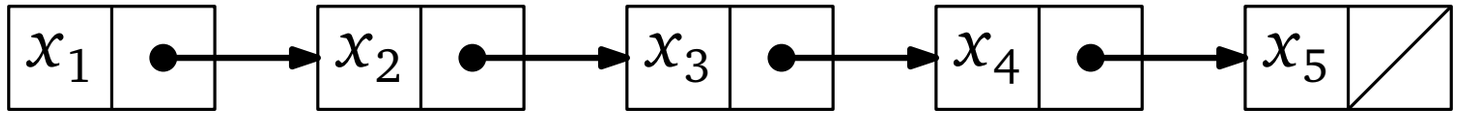


Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays, Lists, and Strings

A list



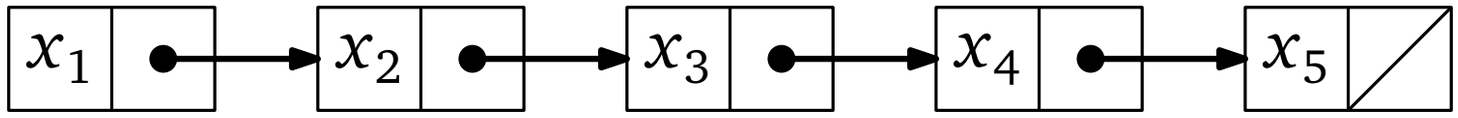
Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

Arrays, Lists, and Strings

A list



Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

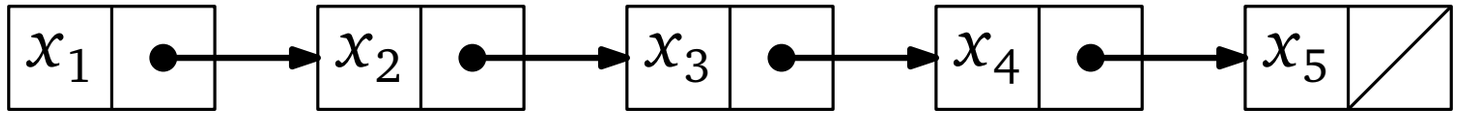
Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

Functional arrays that allow updates without copying the entire array are seriously non-trivial to implement.

Lists are naturally recursive and thus fit extremely well into the recursive approach taken to most problems in functional programming.

Arrays, Lists, and Strings

A list



Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

Functional arrays that allow updates without copying the entire array are seriously non-trivial to implement.

Lists are naturally recursive and thus fit extremely well into the recursive approach taken to most problems in functional programming.

Strings are arrays of characters in imperative languages and lists of characters in functional languages.

Pointers

- Point to memory locations that store data (often of a specified type, e.g., `int*`)
- Are not required in languages with reference model of variables (Lisp, ML, CLU, Java)
- Are required for recursive types in languages with value model of variables (C, Pascal, Ada)

Pointers

- Point to memory locations that store data (often of a specified type, e.g., `int*`)
- Are not required in languages with reference model of variables (Lisp, ML, CLU, Java)
- Are required for recursive types in languages with value model of variables (C, Pascal, Ada)

Storage reclamation

- Explicit (manual)
- Automatic (garbage collection)

Pointers

- Point to memory locations that store data (often of a specified type, e.g., `int*`)
- Are not required in languages with reference model of variables (Lisp, ML, CLU, Java)
- Are required for recursive types in languages with value model of variables (C, Pascal, Ada)

Storage reclamation

- Explicit (manual)
- Automatic (garbage collection)

Advantages and disadvantages of explicit reclamation

+ Garbage collection can incur serious run-time overhead
− Potential for memory leaks
− Potential for dangling pointers and segmentation faults

Pointer Allocation and Deallocation

C

- `p = (element *)malloc(sizeof(element))`
- `free(p)`
- Not type-safe, explicit deallocation

Pascal

- `new(p)`
- `dispose(p)`
- Type-safe, explicit deallocation

Java/C++

- `p = new element()` (semantics different between Java and C++, how?)
- `delete p` (in C++)
- Type-safe, explicit deallocation in C++, garbage collection in Java

Dangling References

A ***dangling reference*** is a pointer to an already reclaimed object.

Can only happen when reclamation of objects that are no longer needed is the responsibility of the programmer.

Dangling references are notoriously hard to debug and a major source of program misbehaviour and security holes.

Dangling References

A ***dangling reference*** is a pointer to an already reclaimed object.

Can only happen when reclamation of objects that are no longer needed is the responsibility of the programmer.

Dangling references are notoriously hard to debug and a major source of program misbehaviour and security holes.

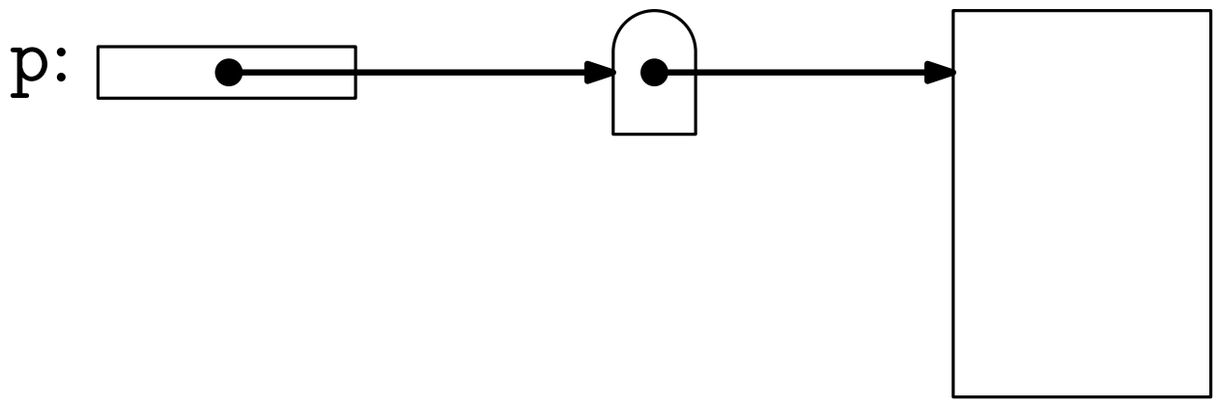
Techniques to catch them:

- Tombstones
- Keys and locks

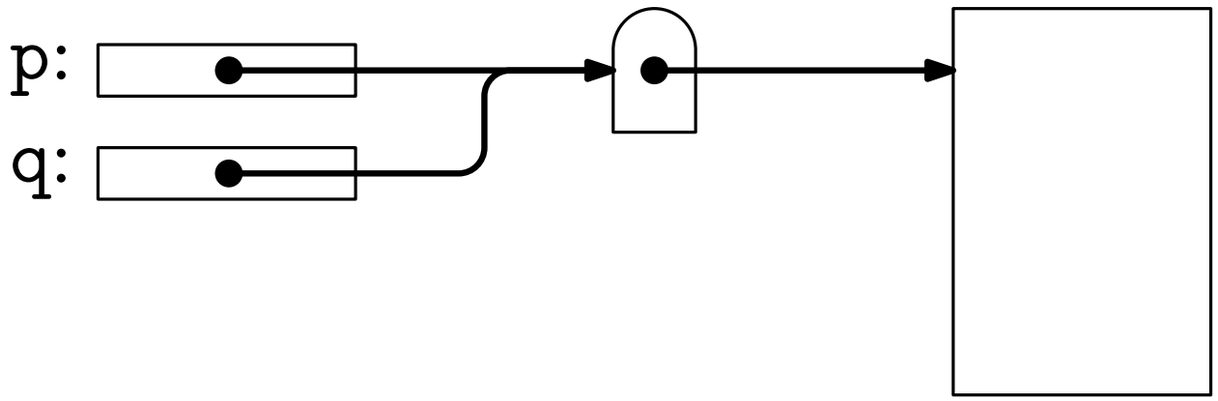
Idea: Better crash to expose the bug than to do the wrong thing.

Tombstones (1)

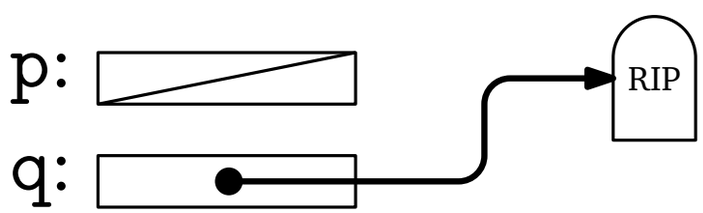
`new(p)`



`q := p`



`delete(p)`



Tombstones (2)

Issue:

- Space overhead
- Runtime overhead (two cache misses instead of one)
- Check for invalid tombstones = hardware interrupt (cheap):
  - RIP = null pointer
- How to allocate/deallocate the tombstones?
  - From separate heap (no fragmentation)
  - Need reference count or other garbage collection strategy to determine when I can delete a tombstone.
  - Need to track pointers to objects on the stack, in order to invalidate their tombstones.

Tombstones (2)

Issue:

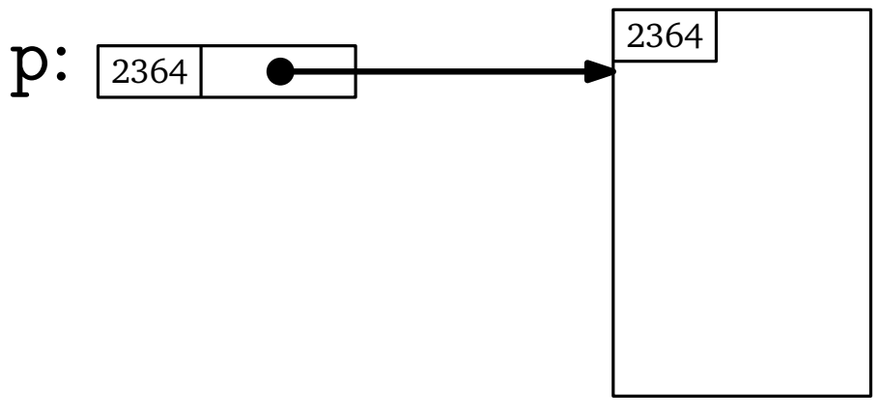
- Space overhead
- Runtime overhead (two cache misses instead of one)
- Check for invalid tombstones = hardware interrupt (cheap):
  - RIP = null pointer
- How to allocate/deallocate the tombstones?
  - From separate heap (no fragmentation)
  - Need reference count or other garbage collection strategy to determine when I can delete a tombstone.
  - Need to track pointers to objects on the stack, in order to invalidate their tombstones.

Interesting side effect:

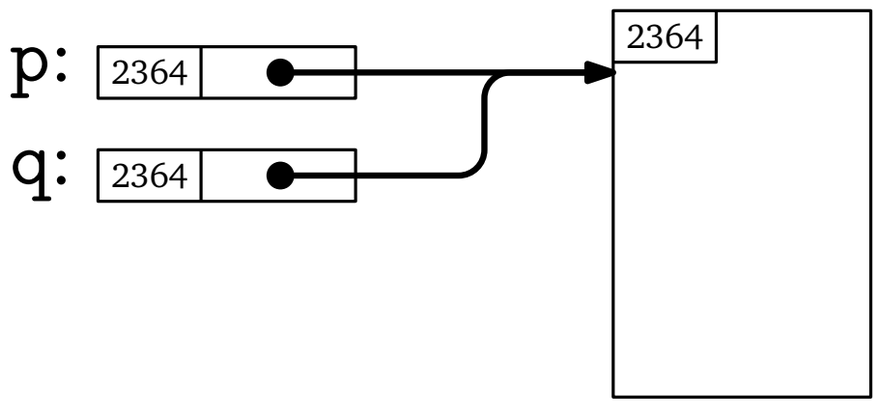
- The extra level of indirection allows for memory compaction.

Locks and Keys (1)

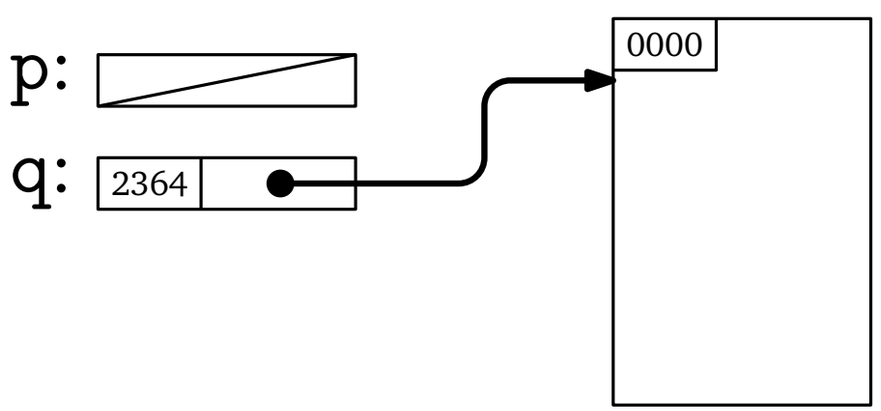
`new(p)`



`q := p`



`delete(p)`



Locks and Keys (2)

Comparison to tombstones:

- Can only be used for pointers to heap-allocated objects
- Provides only probabilistic protection
- Unclear which one has a higher run-time overhead
- Unclear which one has a higher space overhead

Locks and Keys (2)

Comparison to tombstones:

- Can only be used for pointers to heap-allocated objects
- Provides only probabilistic protection
- Unclear which one has a higher run-time overhead
- Unclear which one has a higher space overhead

Languages that provide for such checks for dangling references often allow them to be turned on/off using compile-time flags.

Garbage Collection

Automatic reclamation of space

- Essential for functional languages
- Popular in imperative languages (Clu, Ada, Modula-3, Java)
- Difficult to implement
- Slower than manual reclamation

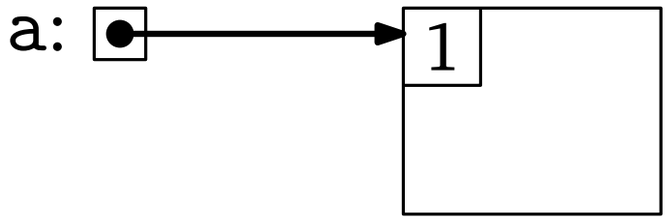
Garbage collection methods

- Reference counts
- Mark and sweep
- Mark and sweep variants
  - Stop and copy
  - Generational technique

Reference Counts

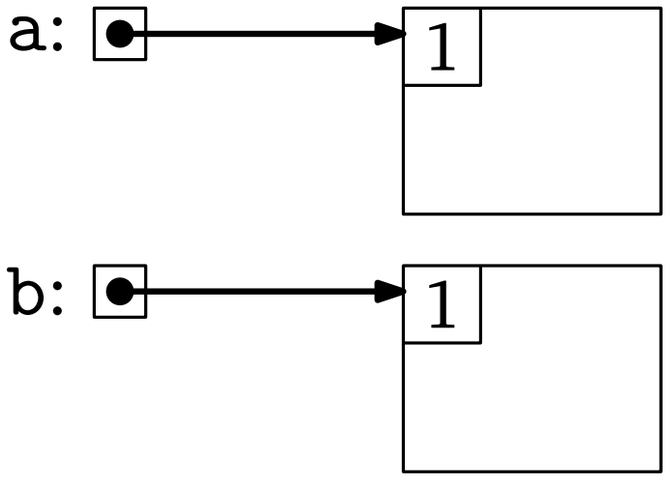
Reference Counts

```
a = new Obj();
```



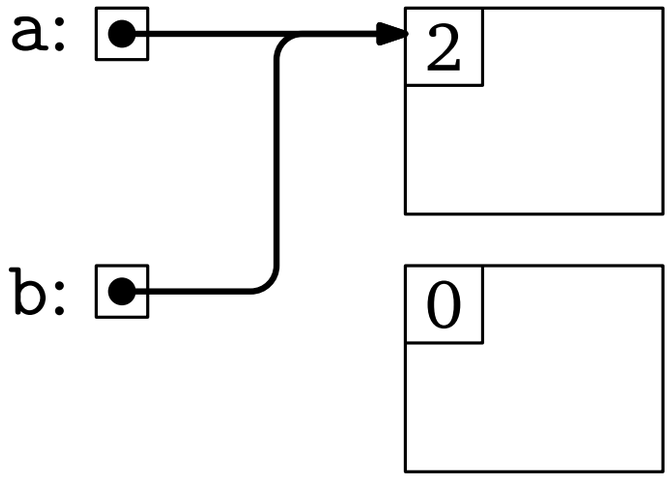
Reference Counts

```
a = new Obj();
b = new Obj();
```



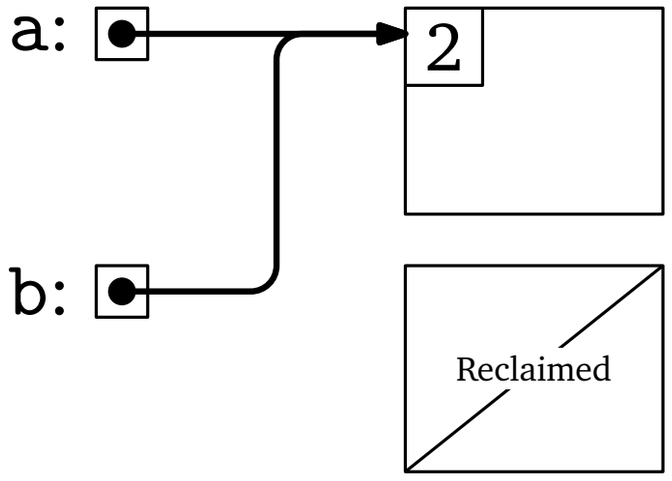
Reference Counts

```
a = new Obj();
b = new Obj();
b = a;
```



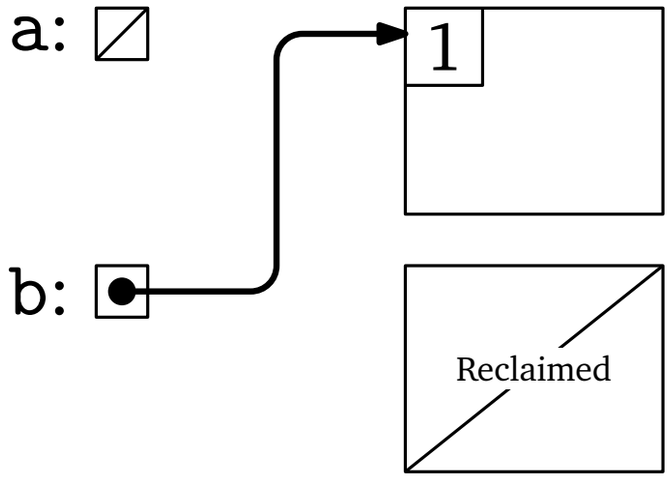
Reference Counts

```
a = new Obj();
b = new Obj();
b = a;
```



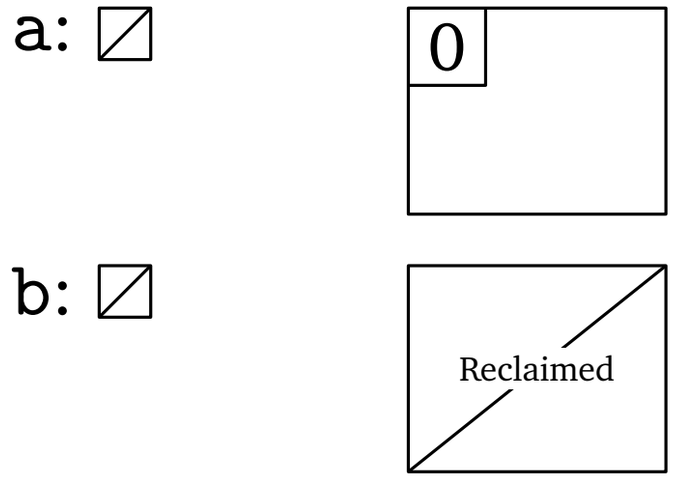
Reference Counts

```
a = new Obj();
b = new Obj();
b = a;
a = null;
```



Reference Counts

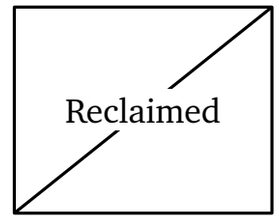
```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```



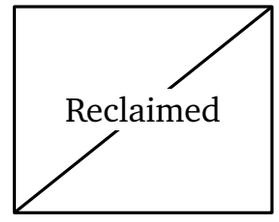
Reference Counts

```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```

a: ☒

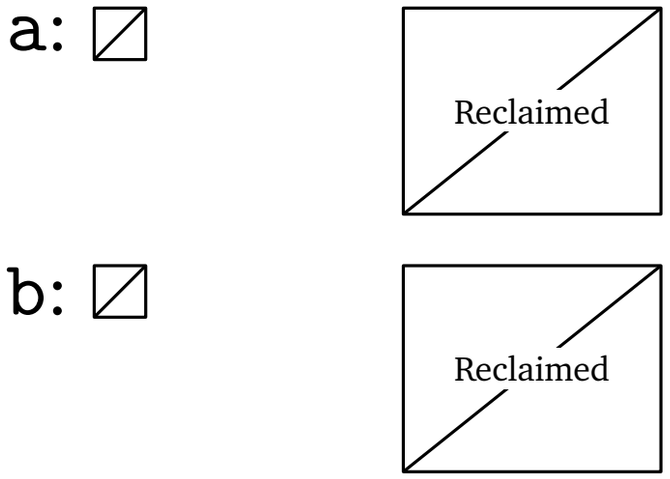


b: ☒



Reference Counts

```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```



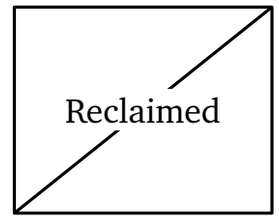
Subroutine return

- Decrease reference counts of all objects referenced by variables in subroutine's stack frame.
- Requires us to keep track of which entries in a stack frame are pointers.

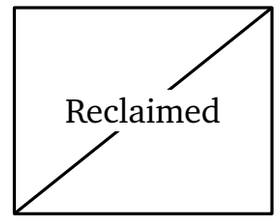
Reference Counts

```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```

a: ⊠



b: ⊠



Subroutine return

- Decrease reference counts of all objects referenced by variables in subroutine's stack frame.
- Requires us to keep track of which entries in a stack frame are pointers.

Pros/cons

+ Fairly simple to implement
+ Fairly low cost
− Does not work when there are circular references. (Not a problem in purely functional languages. Why?)

Mark and Sweep

Algorithm:

1. Mark every allocated memory block as ***useless***.
2. For every pointer in the static address space and on the stack, mark the block it points to as ***useful***.
3. For every block whose status changes from useless to useful, mark the blocks referenced by pointers in this block as useful. Apply this rule recursively.
4. Reclaim all blocks marked as useless at the end.

Mark and Sweep

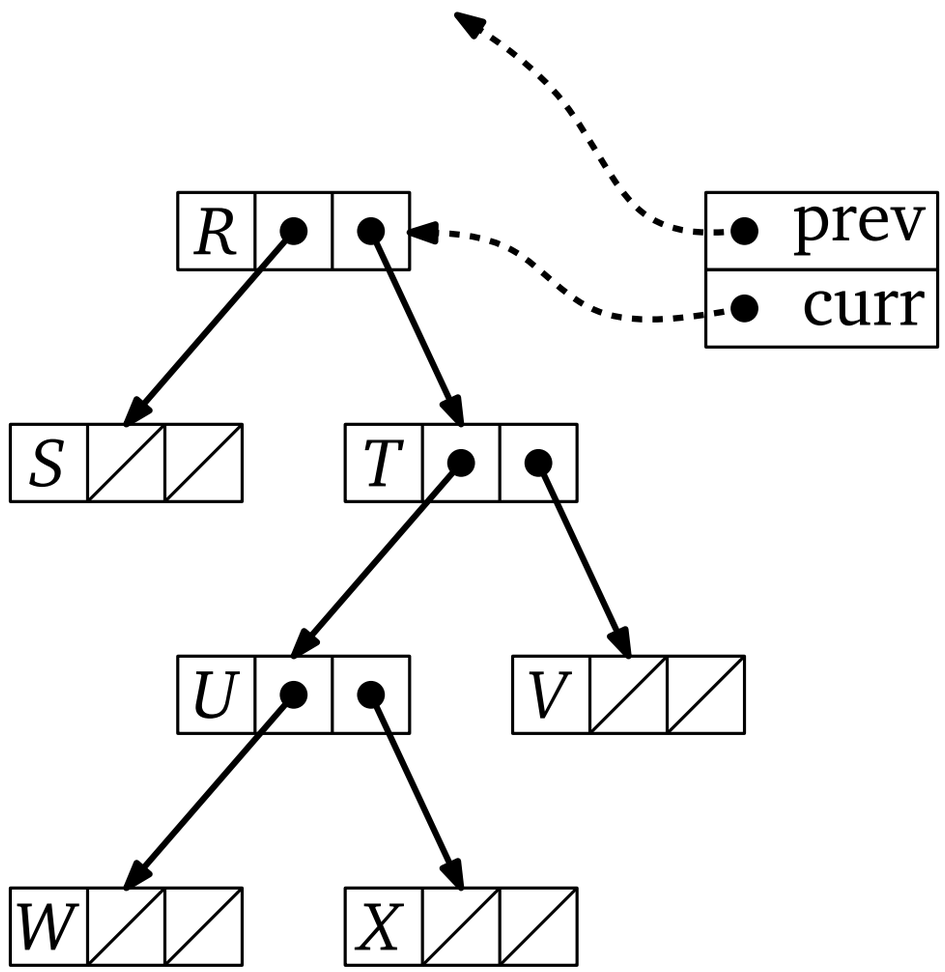
Algorithm:

1. Mark every allocated memory block as ***useless***.
2. For every pointer in the static address space and on the stack, mark the block it points to as ***useful***.
3. For every block whose status changes from useless to useful, mark the blocks referenced by pointers in this block as useful. Apply this rule recursively.
4. Reclaim all blocks marked as useless at the end.

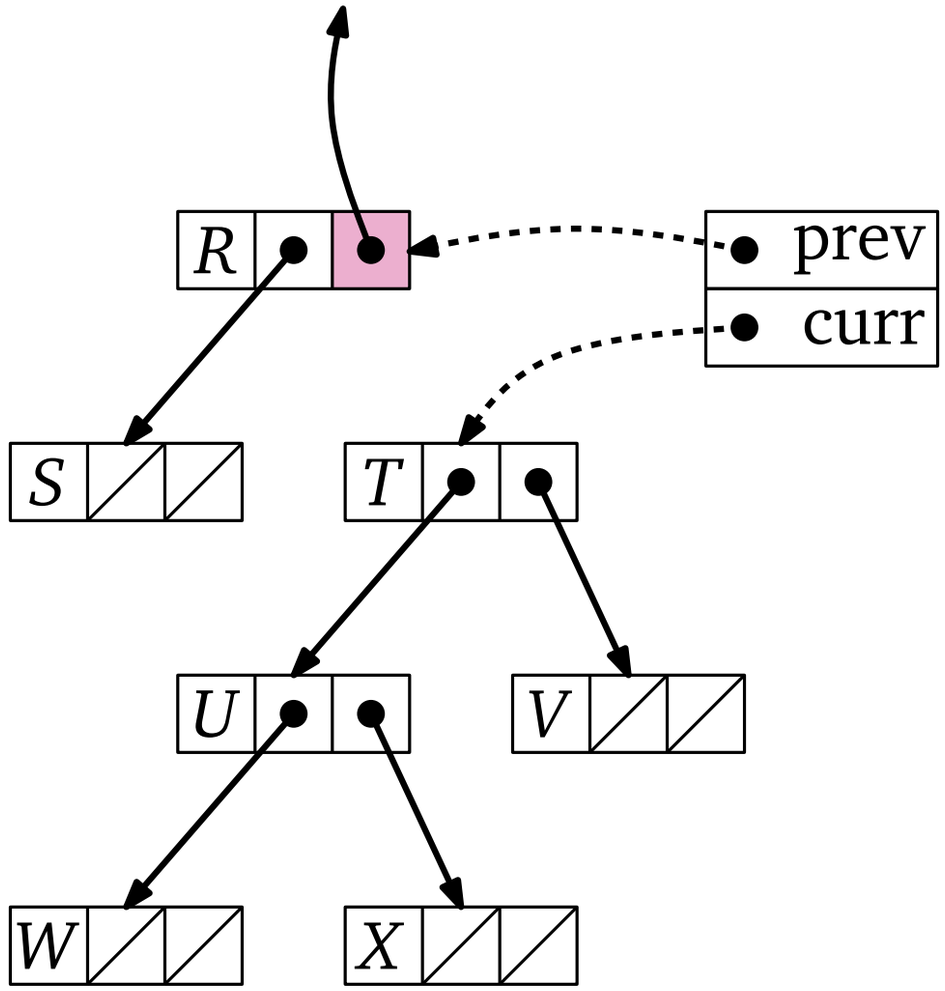
Pros/cons:

- − More complicated to implement
- − Requires inspection of all allocated blocks in a sweep: costly.
- − High space usage if the recursion is deep.
- − Requires type descriptor at the beginning of each block to know the size of the block and to find the pointers in the block.
- \+ Works with circular data structures.

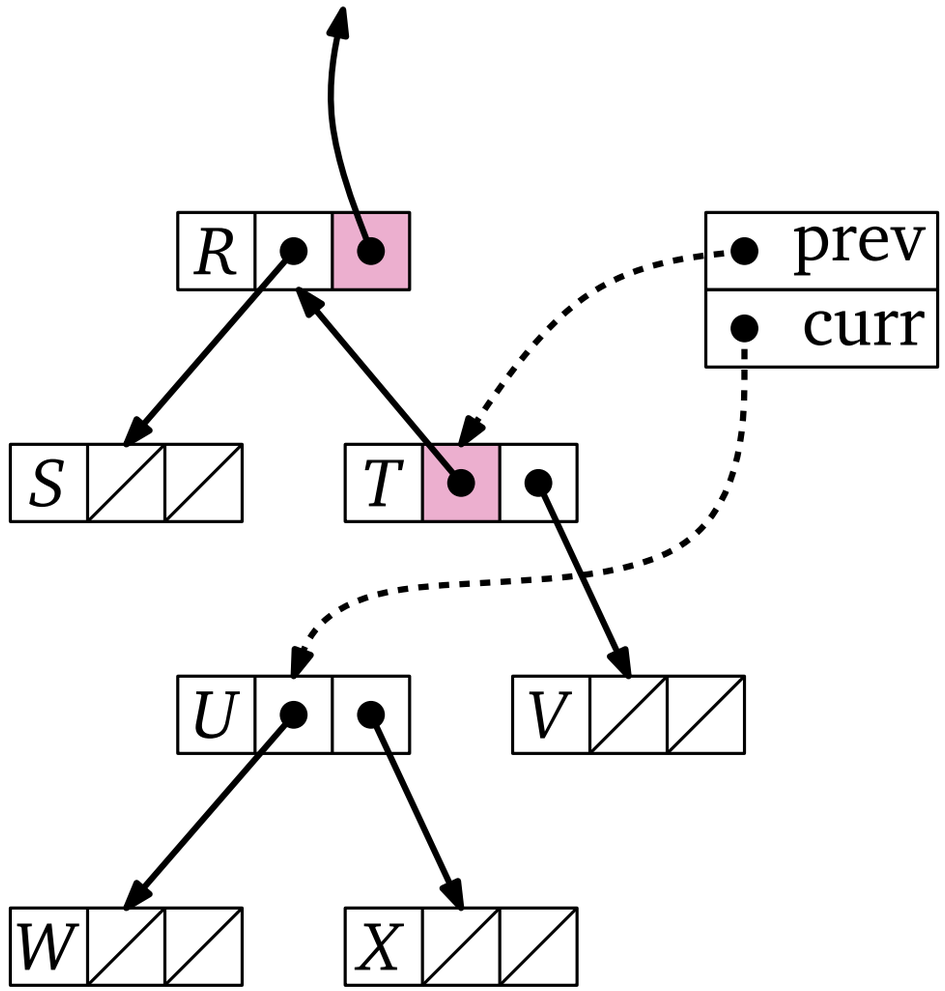
Mark and Sweep in Constant Space



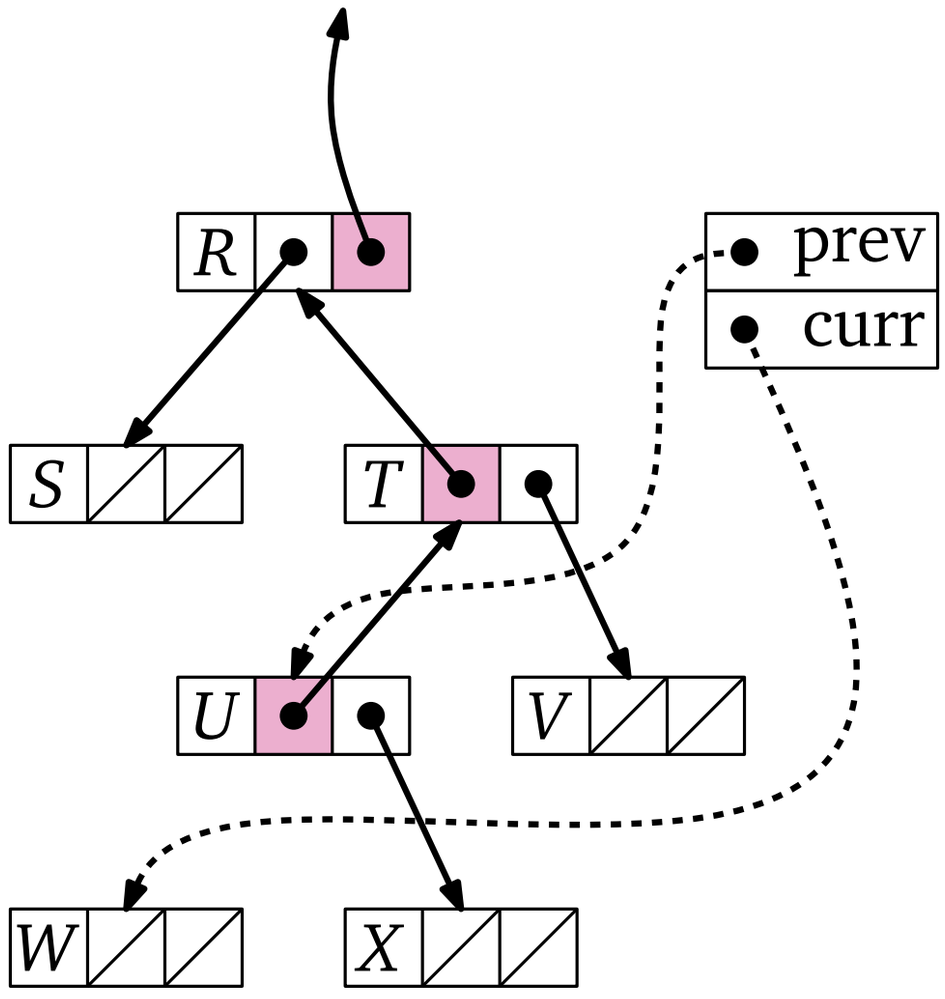
Mark and Sweep in Constant Space



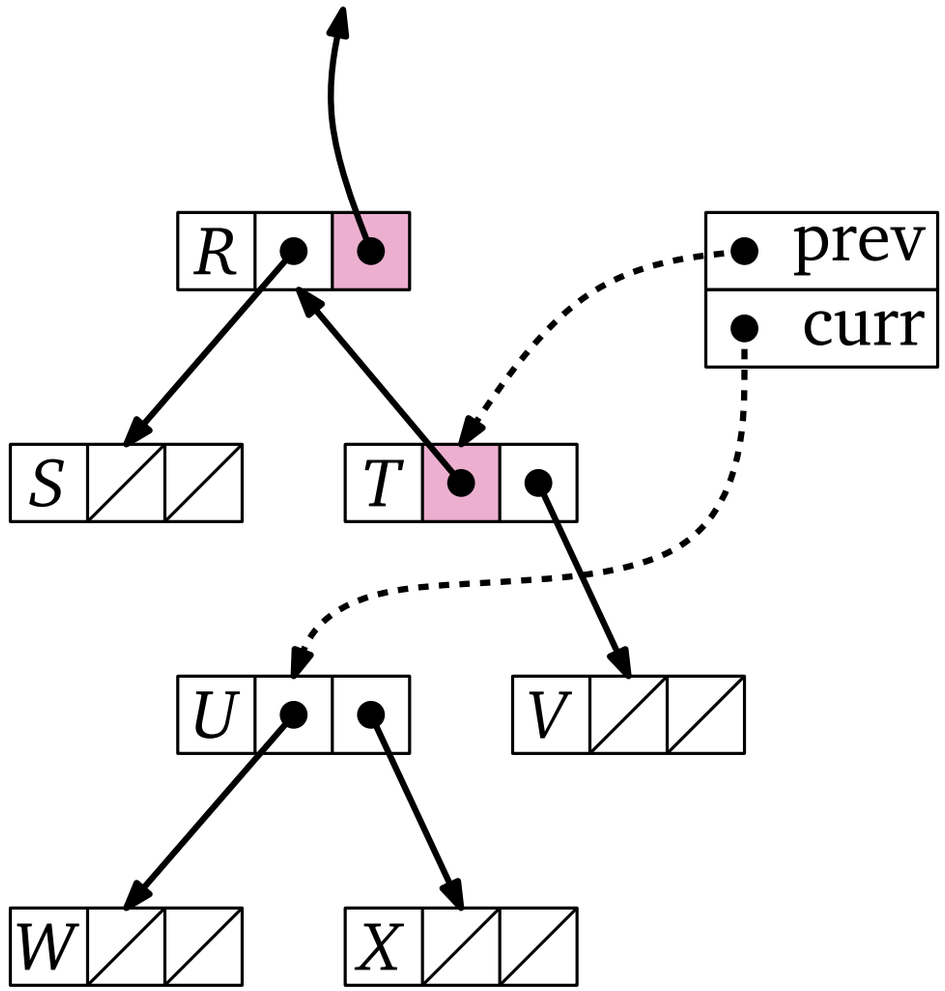
Mark and Sweep in Constant Space



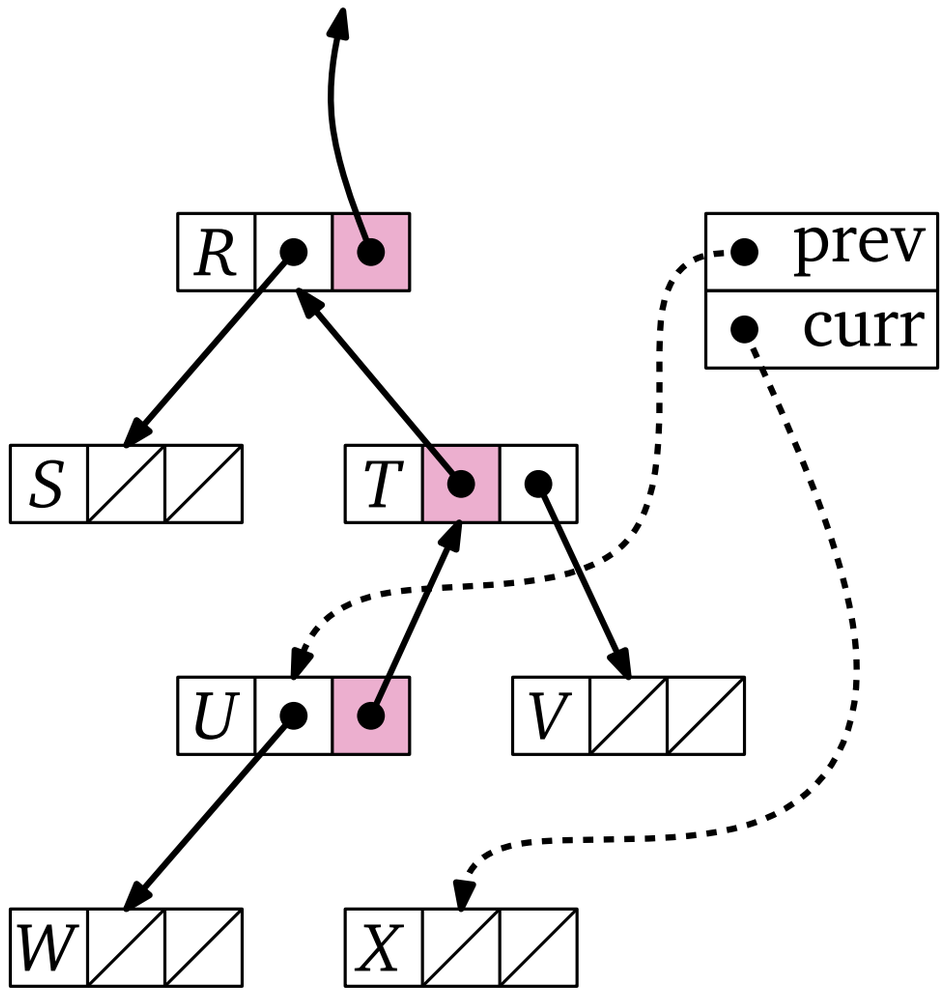
Mark and Sweep in Constant Space



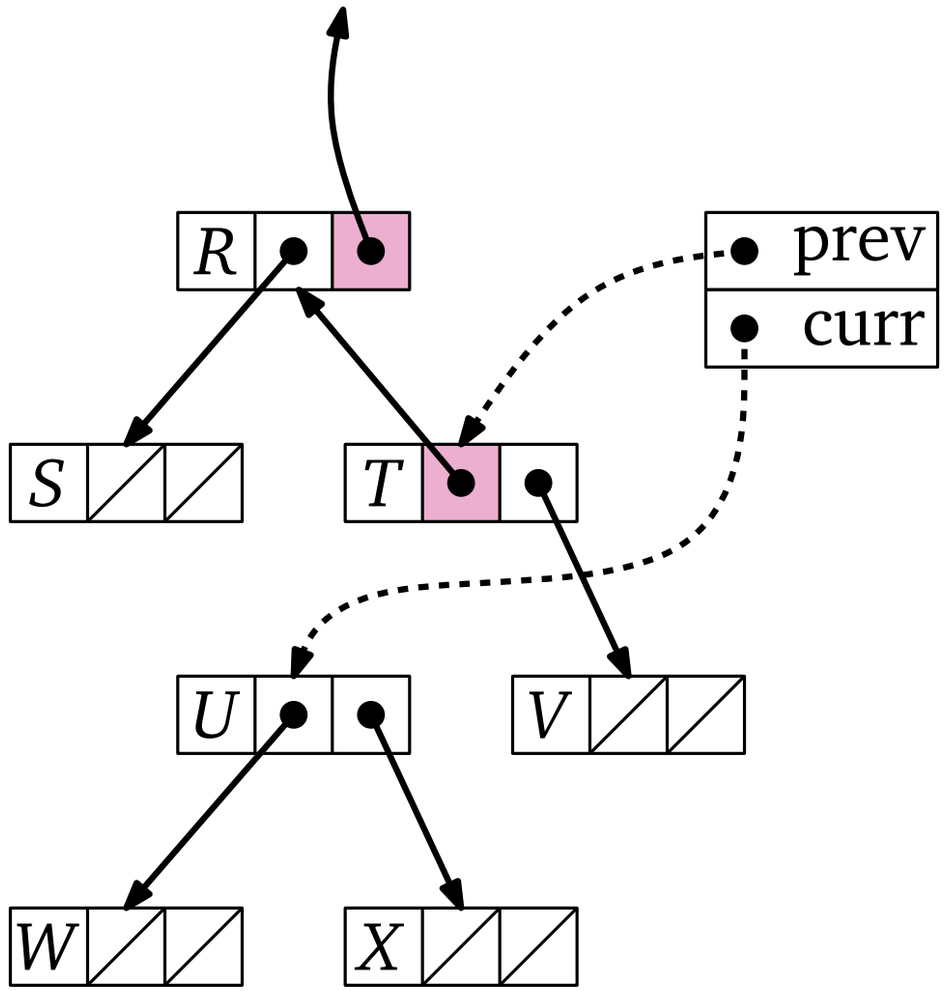
Mark and Sweep in Constant Space



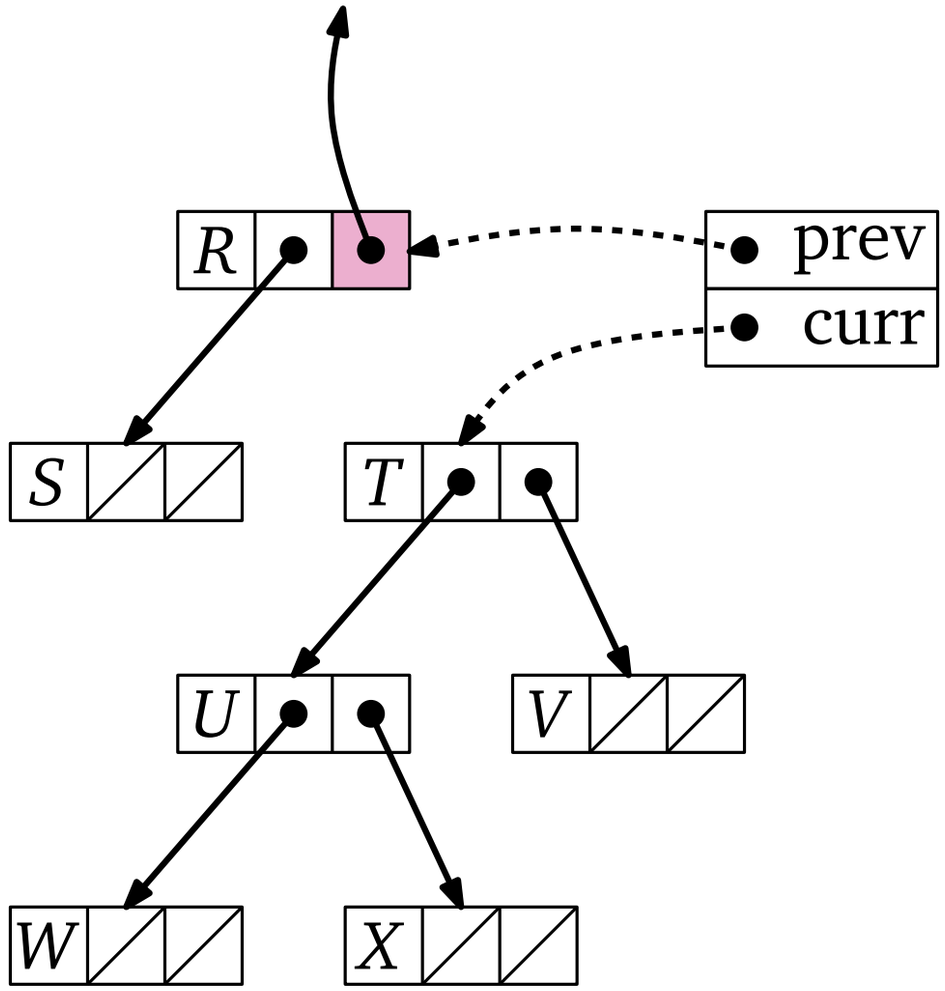
Mark and Sweep in Constant Space



Mark and Sweep in Constant Space



Mark and Sweep in Constant Space



Mark and Sweep Variant: Stop and Copy

Algorithm:

- Heap is divided into two halves. Allocation happens in the first half.
- Once the first half is full, start garbage collection ***and compaction***:
  - Find useful objects by following pointers as in standard mark-and-sweep but without first marking objects as useless.
  - Every useful object that is found is copied to the second half of the heap and replaced with a tag pointing to the new location.
  - Every subsequent pointer to this object finds the tag and gets replaced with a pointer to the new copy.
  - At the end, the roles of the two halves are swapped.

Mark and Sweep Variant: Stop and Copy

Algorithm:

- Heap is divided into two halves. Allocation happens in the first half.
- Once the first half is full, start garbage collection ***and compaction***:
  - Find useful objects by following pointers as in standard mark-and-sweep but without first marking objects as useless.
  - Every useful object that is found is copied to the second half of the heap and replaced with a tag pointing to the new location.
  - Every subsequent pointer to this object finds the tag and gets replaced with a pointer to the new copy.
  - At the end, the roles of the two halves are swapped.

Pros/cons:

\+ Time proportional to number of useful objects, not total number of objects.

\+ Eliminates external fragmentation.

± Only half the heap is available for allocation. Not really an issue if we have virtual memory.

Mark and Sweep Variant: Generational Technique (1)

Algorithm:

- Heap divided into several (often two) regions
- Allocation happens in first region
- Garbage collection:
  - Apply mark and sweep to first region
  - Every object that survives a small number (often one) of rounds of garbage collection in a region gets promoted to the next region in a manner reminiscent of stop-and-copy.
  - Inspect subsequent regions only if collection in the regions inspected so far did not free up enough space.

Mark and Sweep Variant: Generational Technique (1)

Algorithm:

- Heap divided into several (often two) regions
- Allocation happens in first region
- Garbage collection:
  - Apply mark and sweep to first region
  - Every object that survives a small number (often one) of rounds of garbage collection in a region gets promoted to the next region in a manner reminiscent of stop-and-copy.
  - Inspect subsequent regions only if collection in the regions inspected so far did not free up enough space.

Idea:

- Most objects are short-lived. Thus, collection inspects only the first region most of the time and thus is cheaper.

Mark and Sweep Variant: Generational Technique (2)

Two main issues:

- How to discover useful objects in a region without searching the other regions?
- How to update pointers from older regions to younger regions when promoting objects?

Mark and Sweep Variant: Generational Technique (2)

Two main issues:

- How to discover useful objects in a region without searching the other regions?
- How to update pointers from older regions to younger regions when promoting objects?

Techniques:

- Disallow pointers from old regions to young regions by moving objects around appropriately.
- Keep list of old-to-new pointers. (Requires instrumentation → runtime overhead.)

Conservative Garbage Collection

Normally, garbage collectors need to know which positions in an object are pointers. This requires type descriptors to be stored with each object.

A more conservative approach that does not need type descriptors:

- Treat each word as a pointer.

This may fail to reclaim useless objects because some integer or other value happens to equal the address of the object.

Statistically this is rare.