Lecture 13: Complex Types and Garbage Collection

COMP 524 Programming Language Concepts
Stephen Olivier
March 17, 2009

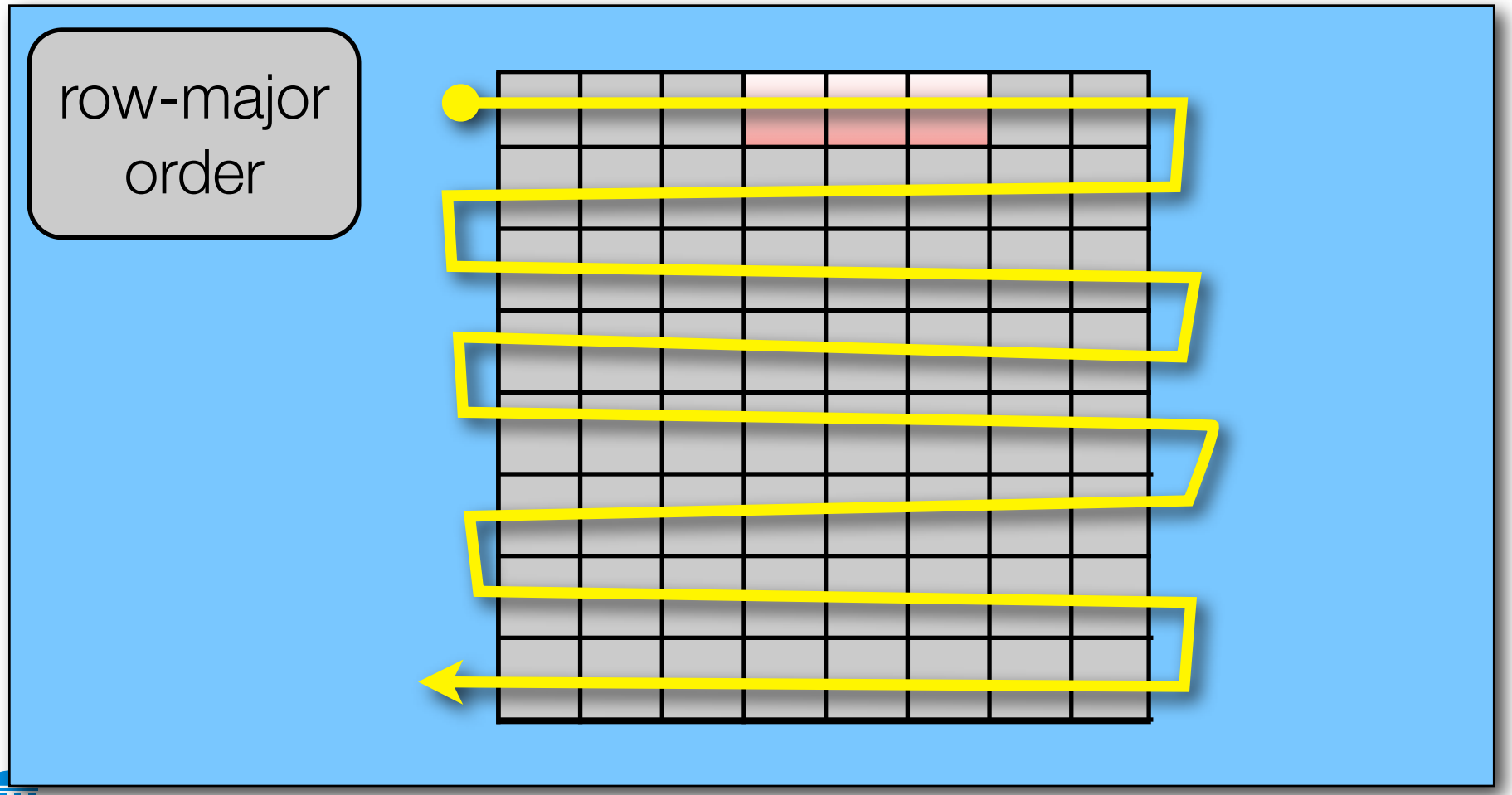
Based on slides by A. Block, notes by N. Fisher, F. Hernandez-Campos, and D. Stotts

Goals

- Discuss Arrays, Pointers, Recursive Types, Strings, Sets, Lists, Equality, and Garbage Collection

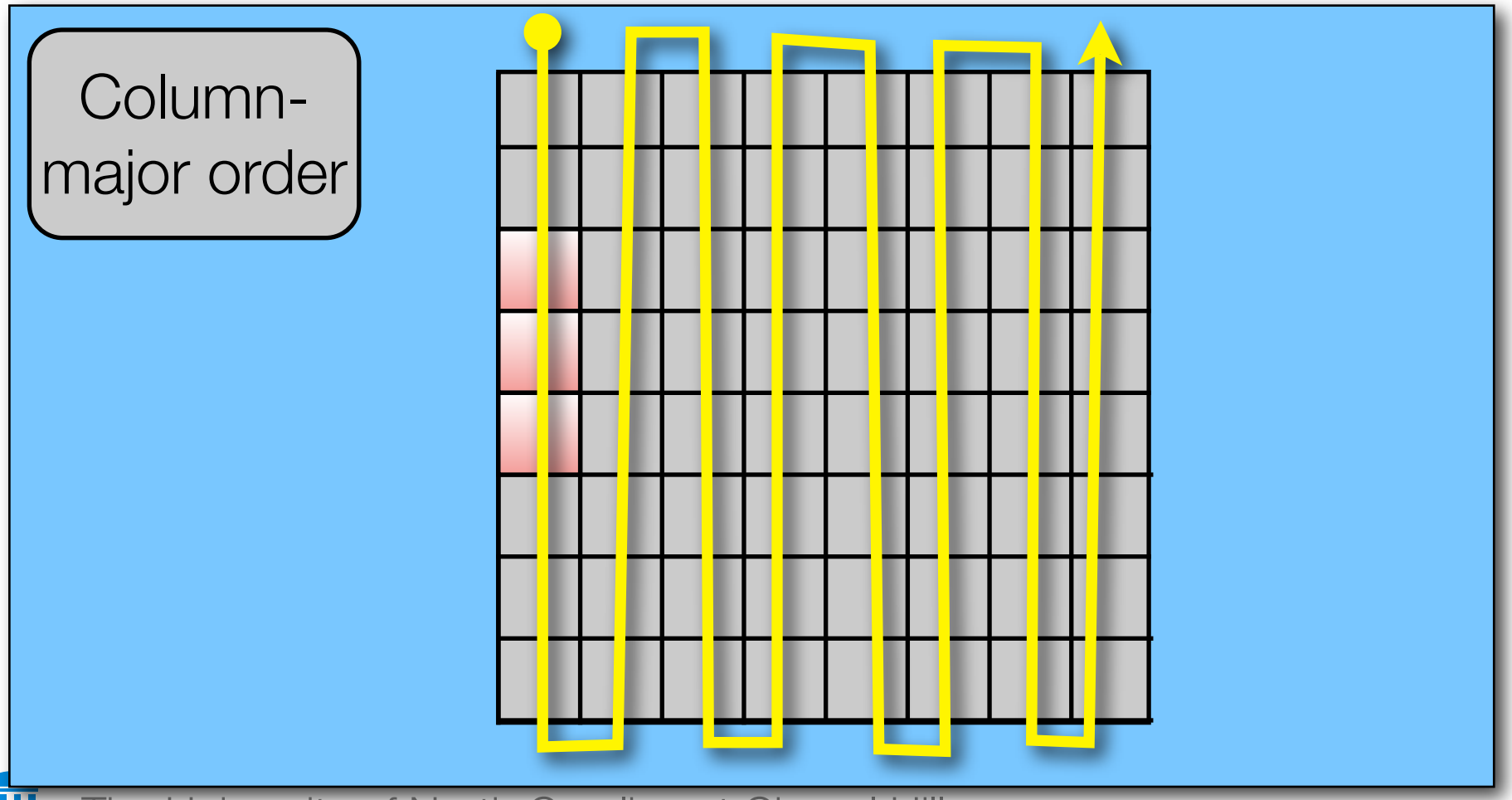
Arrays

- Arrays are usually stored in contiguous locations

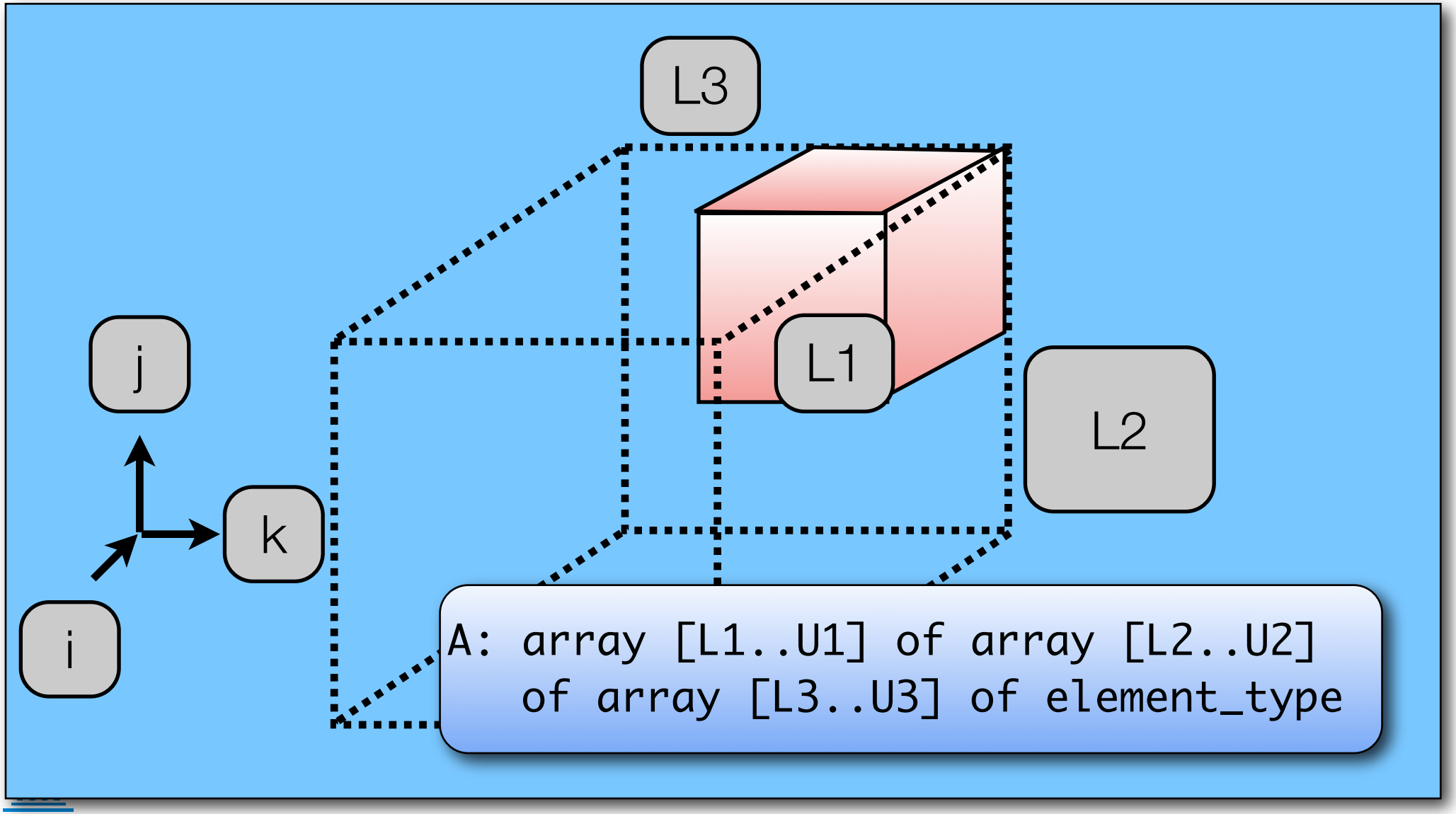


Arrays

- Arrays are usually stored in contiguous locations



Multidimensional Arrays: Address Calculations



Multic

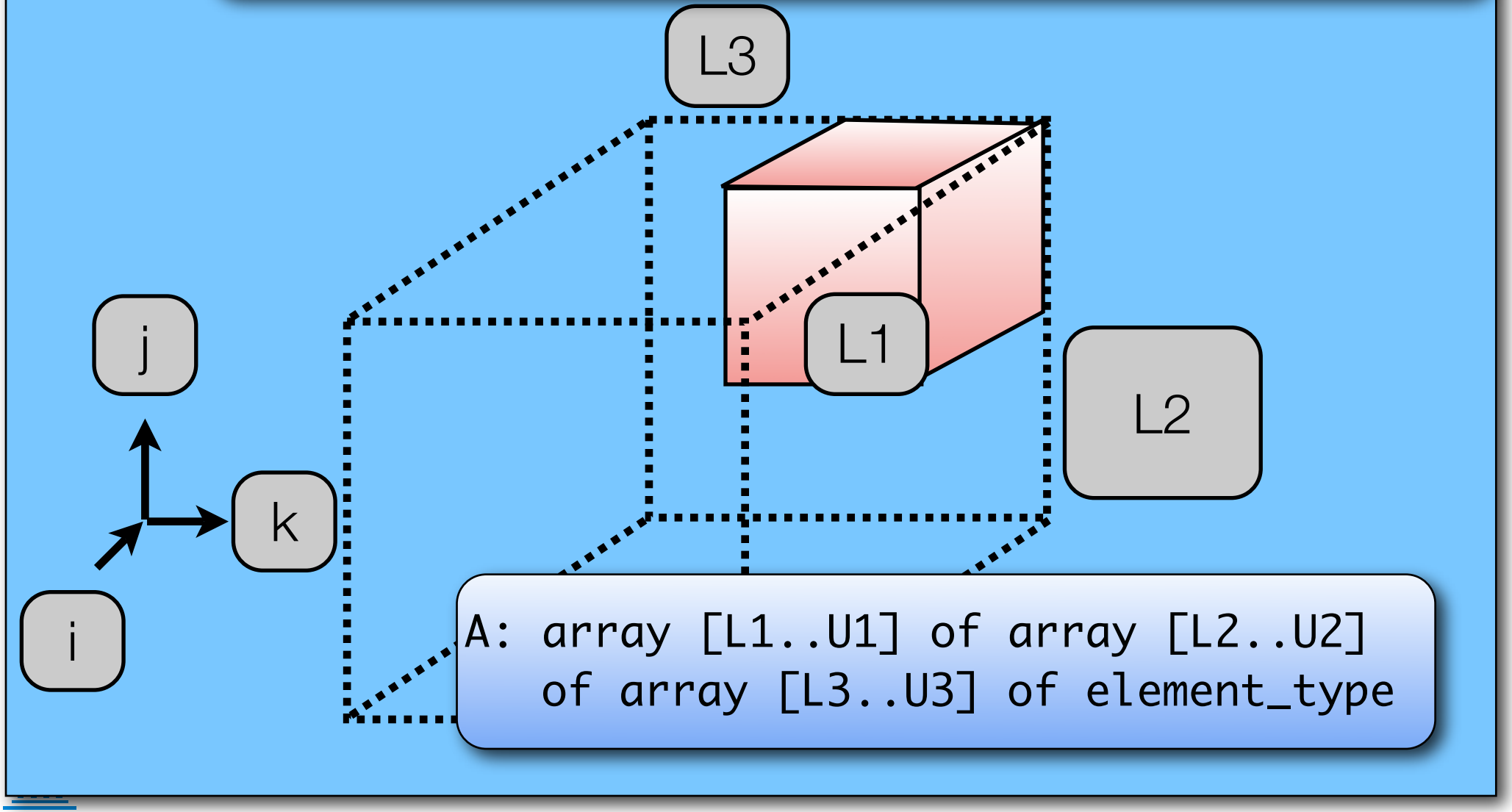
S3 = size of element

S2 = (U3 - L3 + 1) x S3

S1 = (U2 - L2 + 1) x S2

Address of A[i,j,k]

Address of A + (i-L1)xS1 + (j-L2)xS2 + (k-L3)xS3

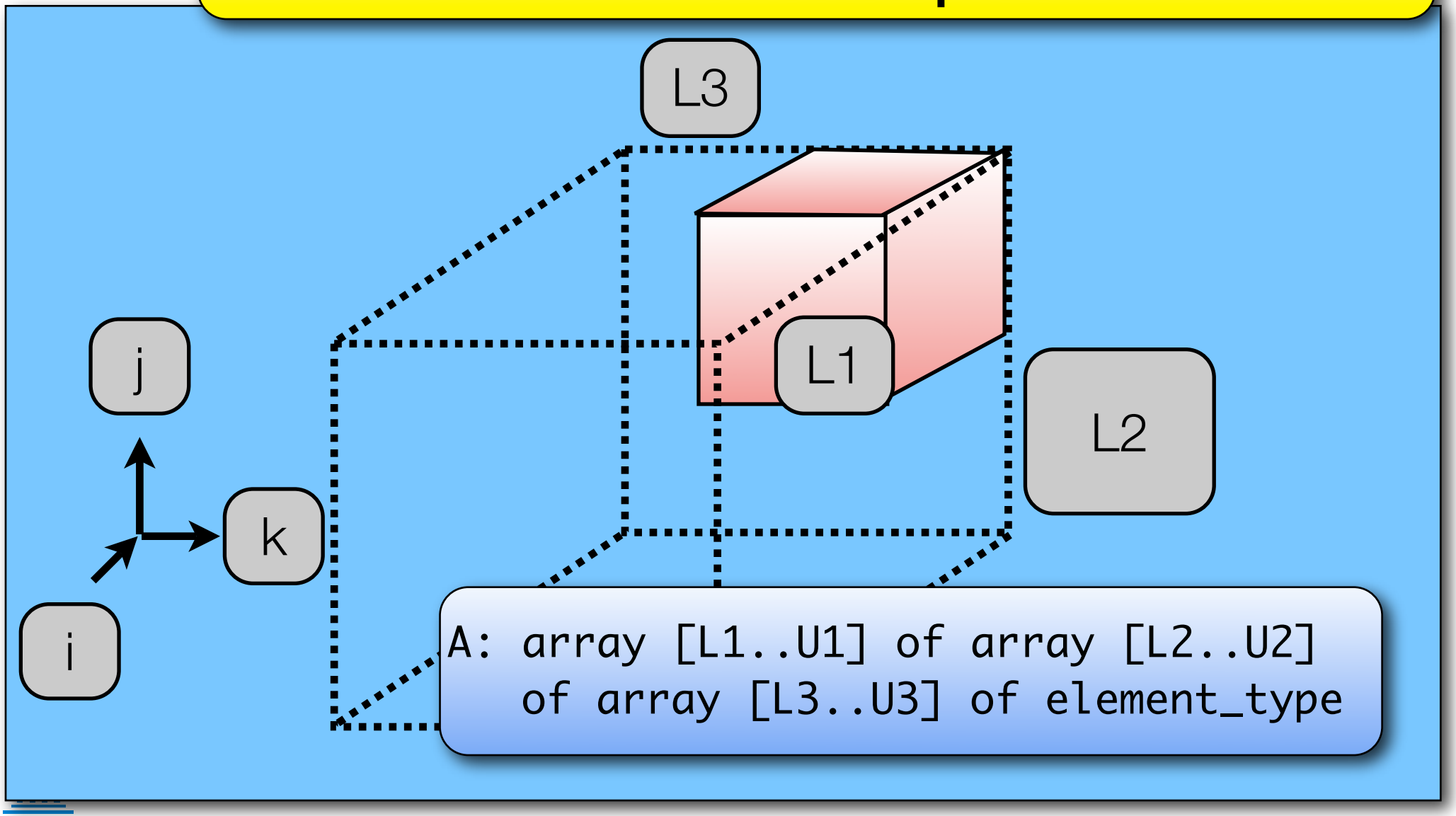


Multic

Optimized

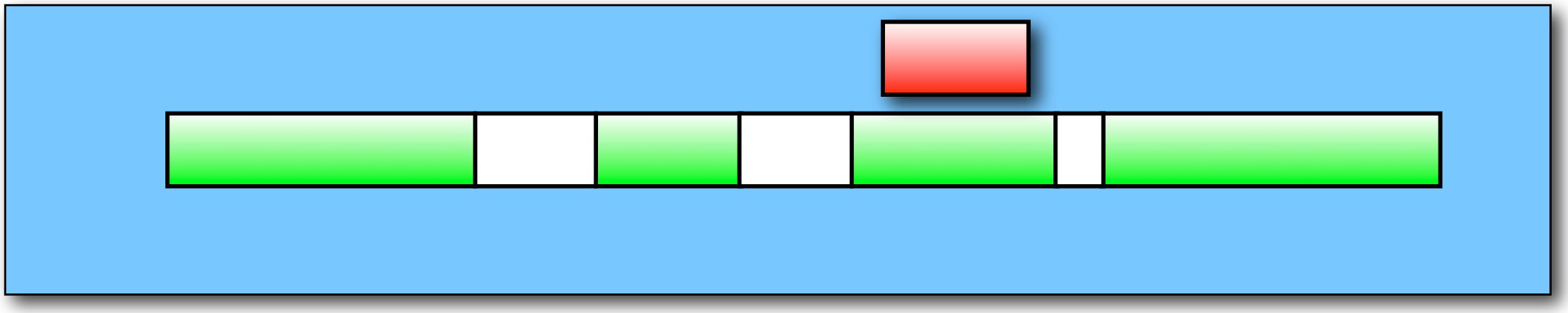
(i x S1) + (j x S2) + (k x S3) + address A -
[(L1 x S1) + (L2 x S2) + (L3 x S3)].

the phrase [(L1 x S1) + (L2 x S2) + (L3 x S3)] can be determined at compile-time



Heap-based Allocation

- The **heap** is a region of storage in which sub-blocks can be allocated and deallocated.



Issues with Heap Allocation

- Pointers
  - Used in value model of variables
  - Not necessary for reference model
- Dangling References
- Garbage Collection

Pointers

- Pointers serve two purposes:
  - Efficient (and sometimes intuitive) access to elaborated objets (as in C)
  - Dynamic creation of linked data structures, in conjunction with a heap storage manger.
- Several Languages (e.g., Pascal) restrict pointers to accessing things in the heap
- Pointers are used with a value model of variables
  - They aren't needed with a reference model (already implicit)

C Pointers and Arrays

- C Pointers and arrays

```
int *a == int a[]
int **a == int *a[]
```

C Pointers and Arrays

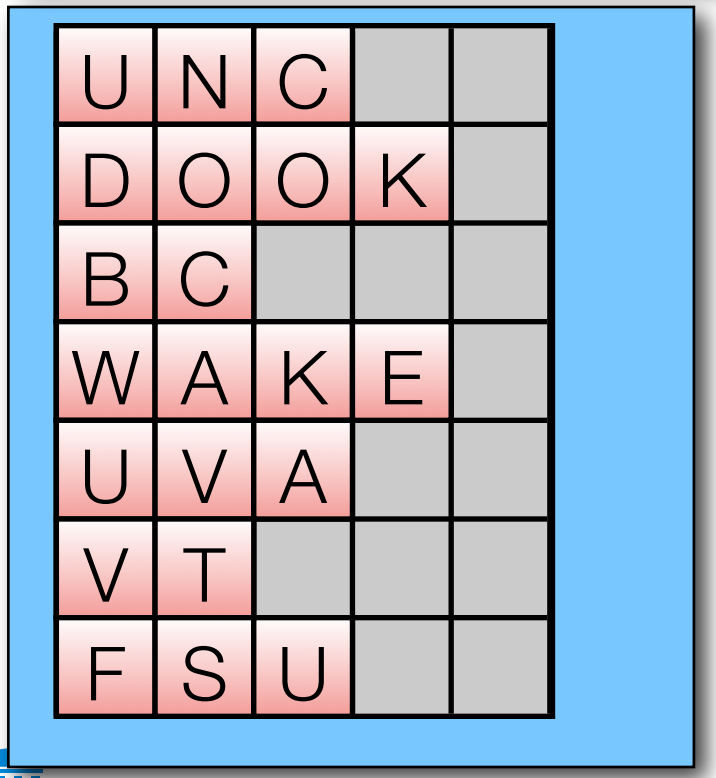
- But equivalencies don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension
  - otherwise it allocates a pointer

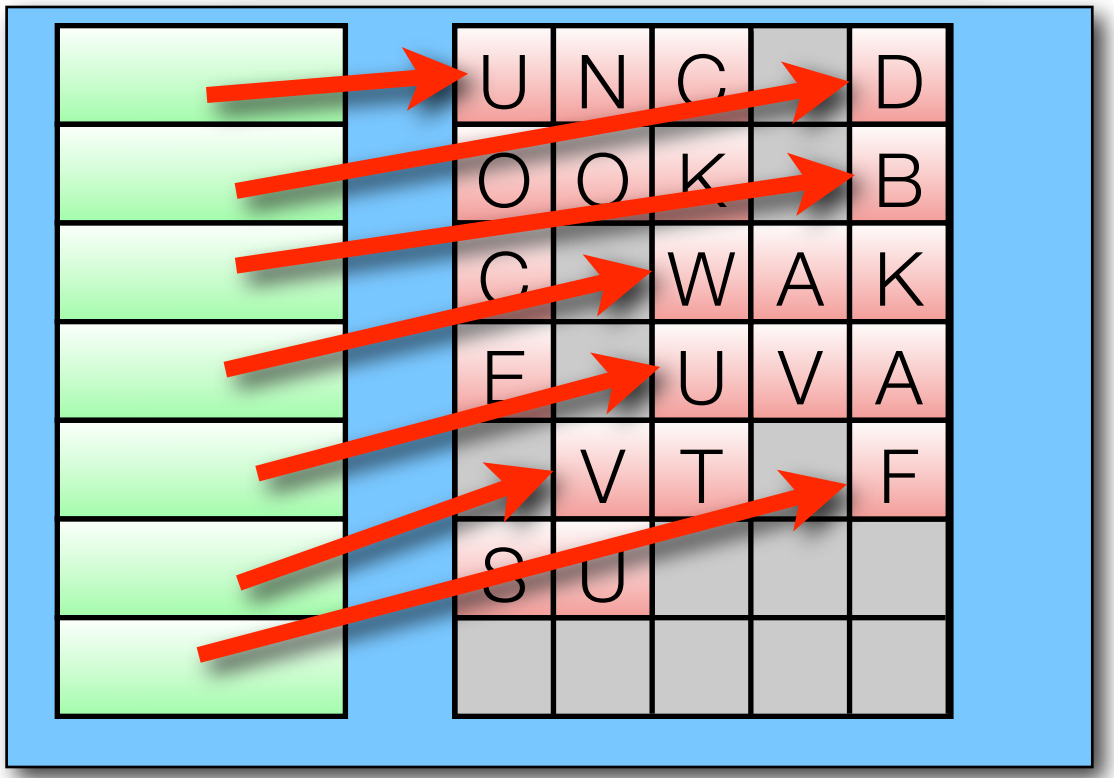
```
int **a, int *a[] // Pointer to pointer to int
int *a[n] //n-element array of row pointers
int a[n][m] // 2-D array
```

Contiguous 2D Arrays vs. Row Pointers (C)

```
char schools[][5] = {
  “UNC”, “DOOK”, “BC”,
  “WAKE”, “UVA”, “VT”,
  “FSU”
};        /* 35B data */
```

```
char *schools[] = {
  “UNC”, “DOOK”, “BC”,
  “WAKE”, “UVA”, “VT”,
  “FSU”
}; /* 28B data+28B ptr */
```





Recursive Types (now with Pointers!): Binary Tree

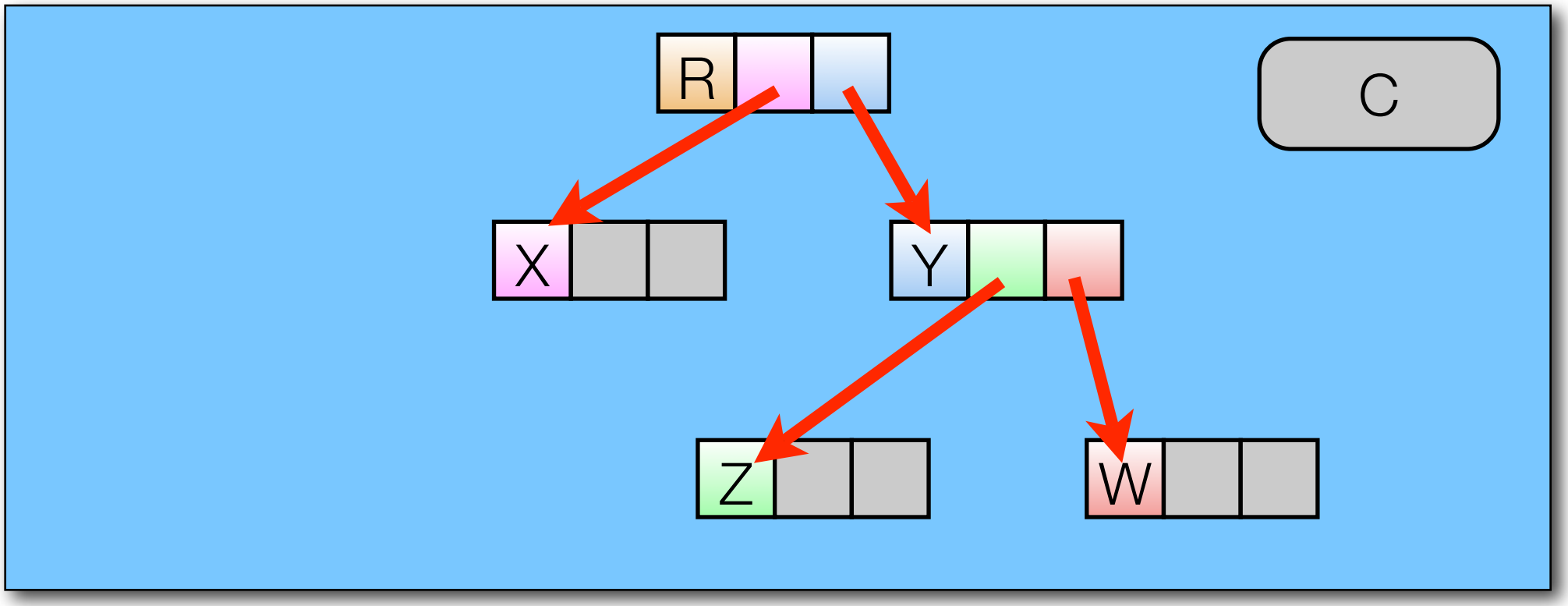
C

```
struct chr_tree{
  struct chr_tree *l, *r;
  char var;
}
```

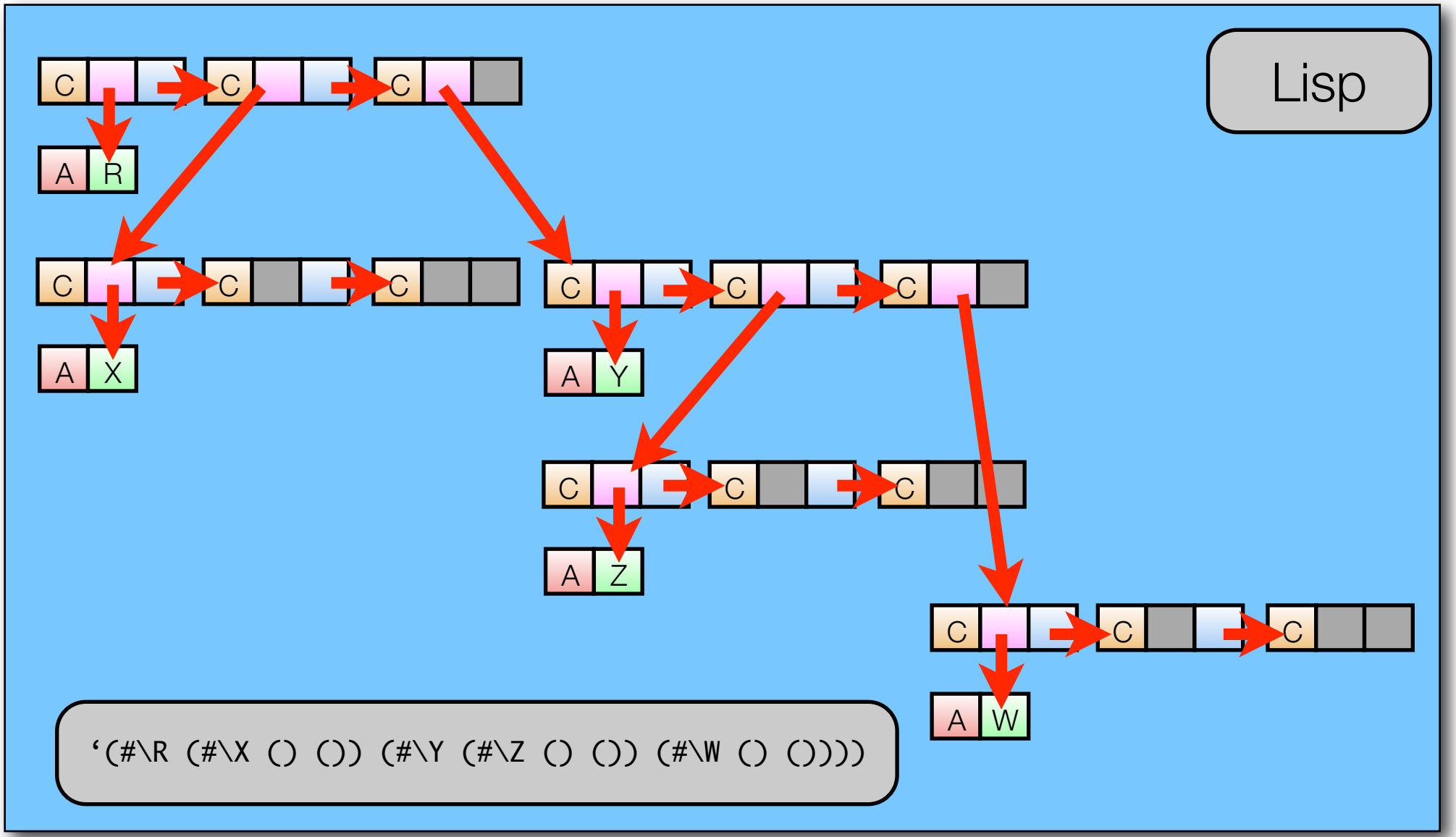
Ada

```
type chr_tree;
type chr_tree_ptr is access chr_tree;
type chr_tree is record
  left,right:chr_tree_ptr;
  val:character;
end record;
```

Binary Tree with Explicit Pointers

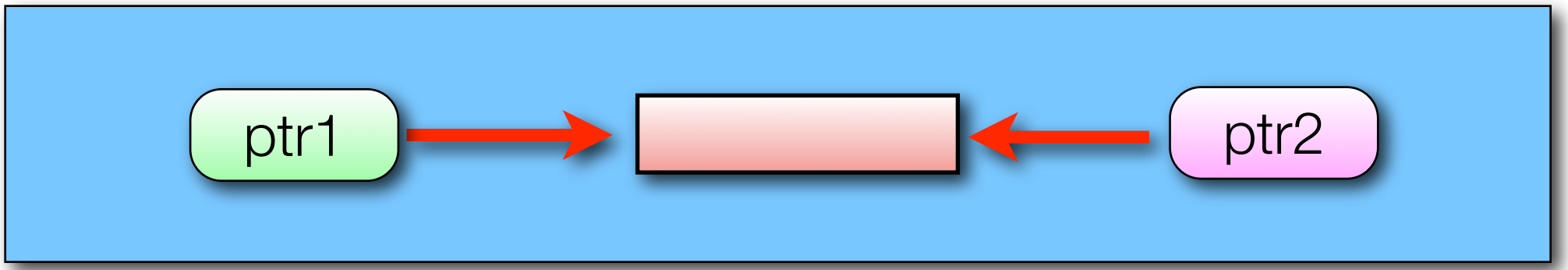


Binary Tree in Reference Model



Problems with Explicit Reclamation

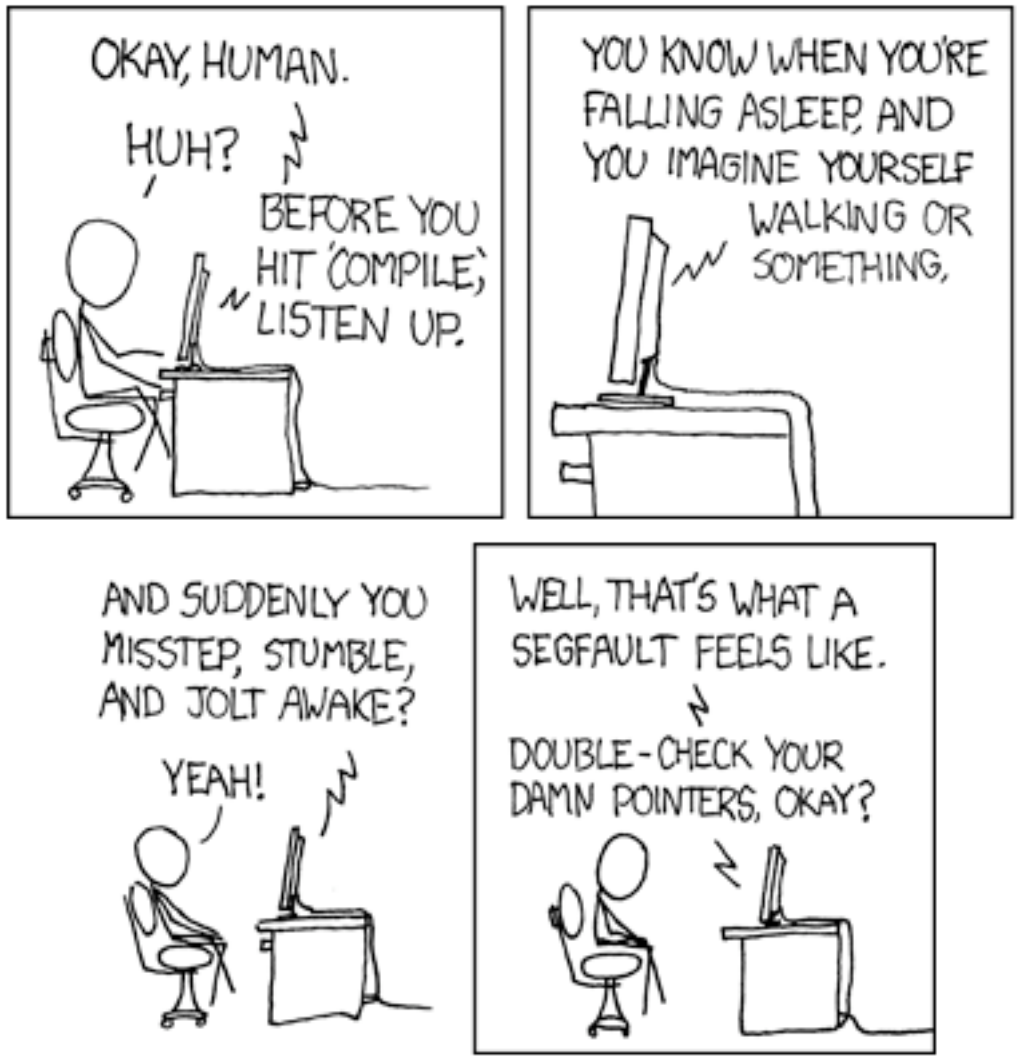
- Explicit reclamation of heap objects is problematic
  - The programmer may forget to deallocate some objects
    - Causing memory leaks
    - For example, in the previous example, the programmer may forget to include the delete statement
  - References to deallocated objets may not be reset
    - Creating dangling references



Problems with Explicit Reclamation

- Explicit reclamation of heap objects is problematic
  - The programmer may forget to deallocate some objects
    - Causing memory leaks
    - For example, in the previous example, the programmer may forget to include the delete statement
  - References to deallocated objets may not be reset
    - Creating dangling references

Issues with Heap Allocation

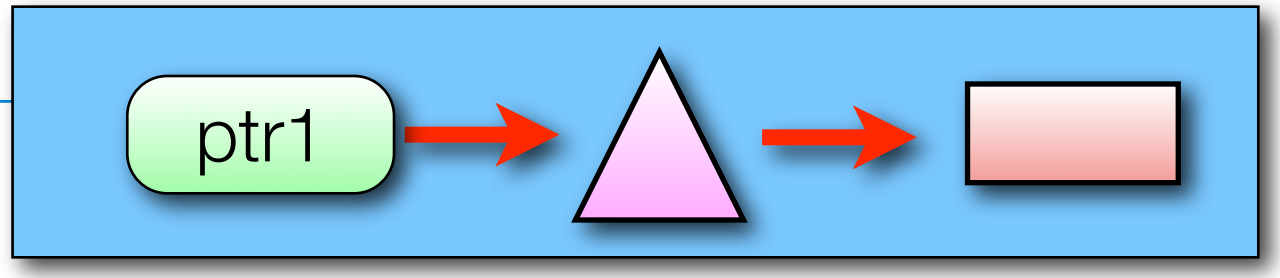


Dealing with Dangling References

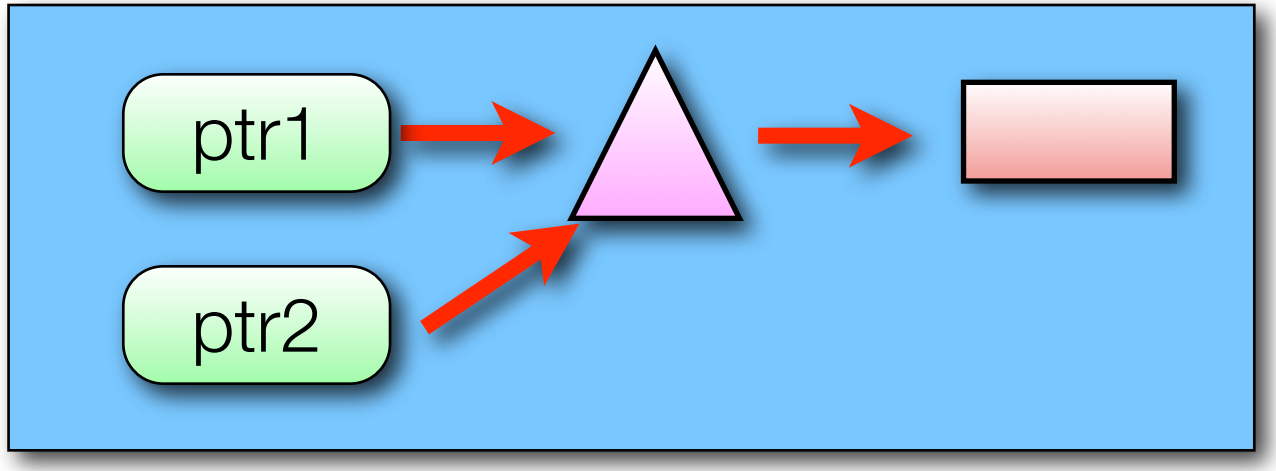
- **Tombstones**: Use an intermediary device
  - Pointers refer to the tombstone rather than the actual object
    - Indirection costs
  - Tombstones invalidated on deallocation
    - Problem: when do we reclaim them
- **Lock and Keys**: Use a universal key to per-object lock
  - Check key against lock each access (costly)
  - Reset lock to zero on deallocation and it should get a different value upon reuse

Tombstones

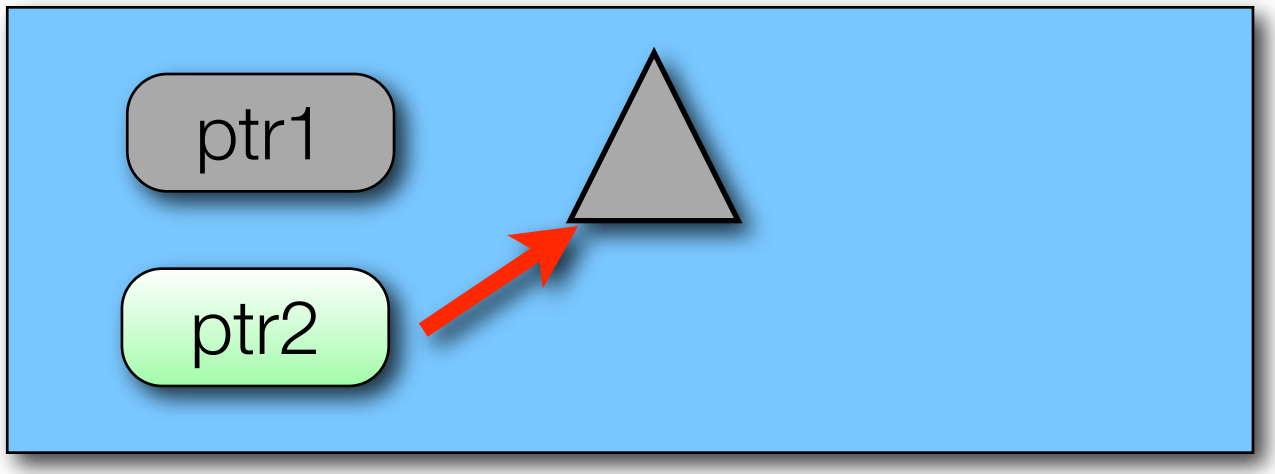
```
new(ptr1);
```



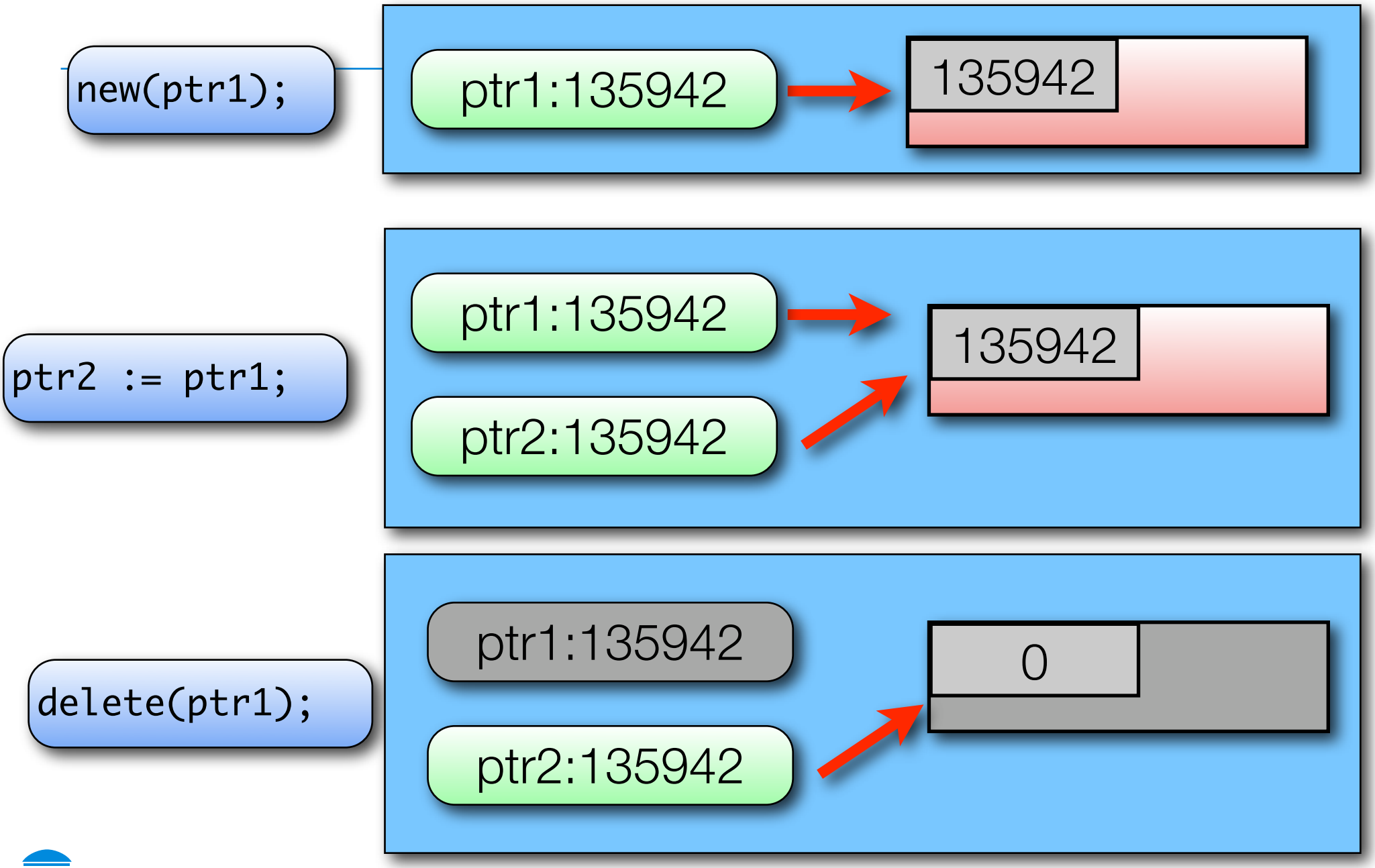

```
ptr2 := ptr1;
```




```
delete(ptr1);
```



Locks and Keys



Garbage Collection

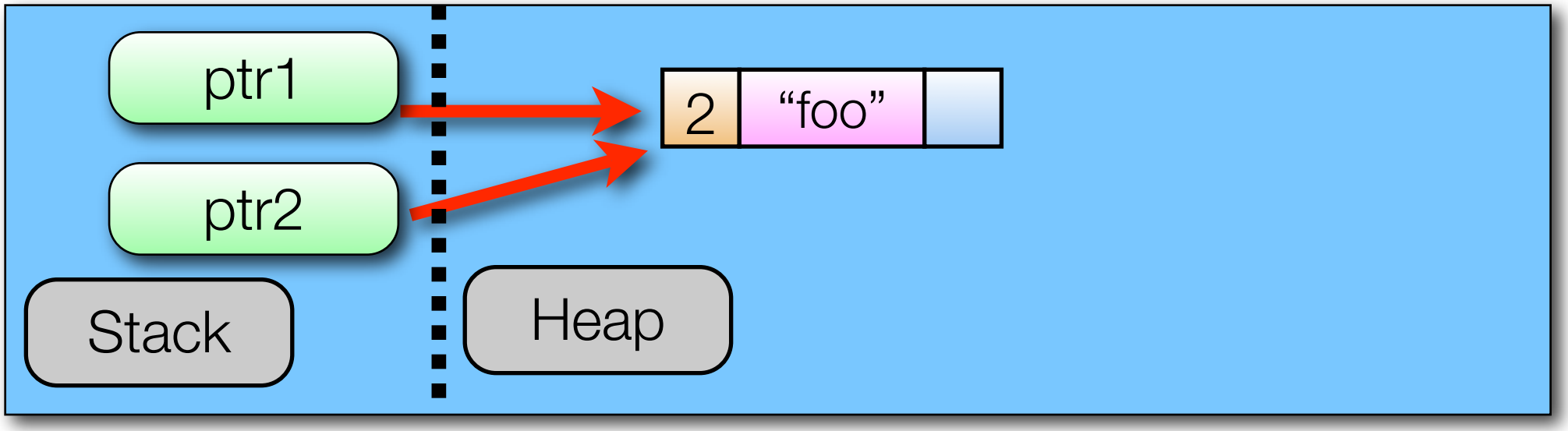
- Automatic reclamation of the **space used by objects that are no longer useful**:
  - **Developed for functional languages**
    - Essential in this programming paradigm.
  - More and more popular in imperative languages
    - Java, C#, Python
- **Generally slower than manual reclamation, but it eliminates a very frequent programming error**

Garbage Collection: Techniques

- When is an object no longer useful?
- There are several garbage collection techniques that answer this question in a different manner
  - Reference counting
  - Mark-and-sweep collection
  - store-and-copy
  - generational collection

Reference Counting

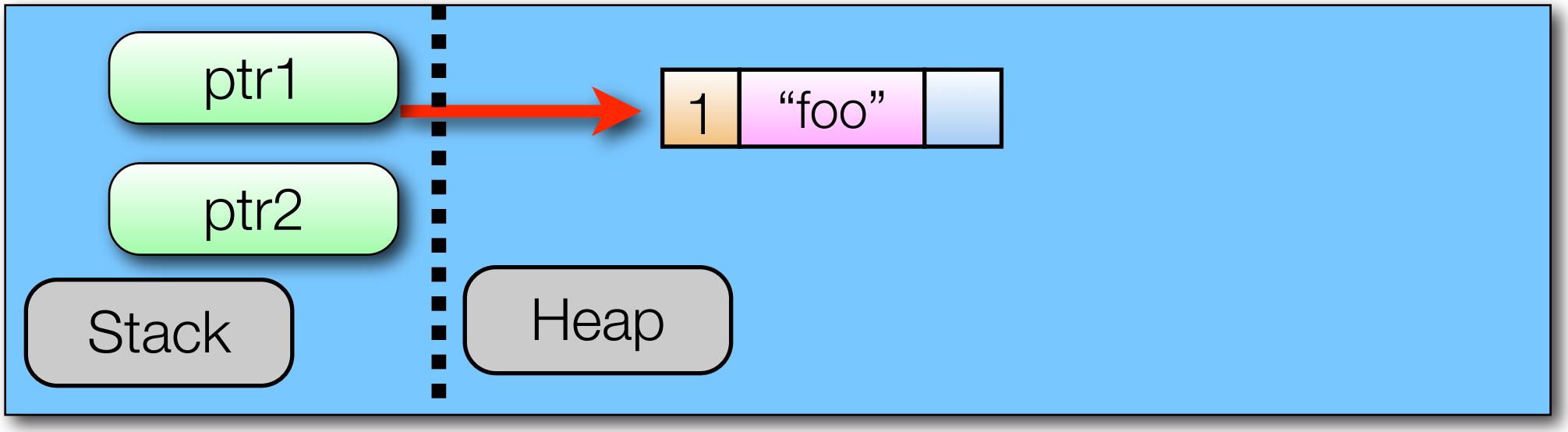
- Each object has an **associated reference counter**



- Keeps reference counter up to date, and **deallocates objects when the counter is zero**

Reference Counting

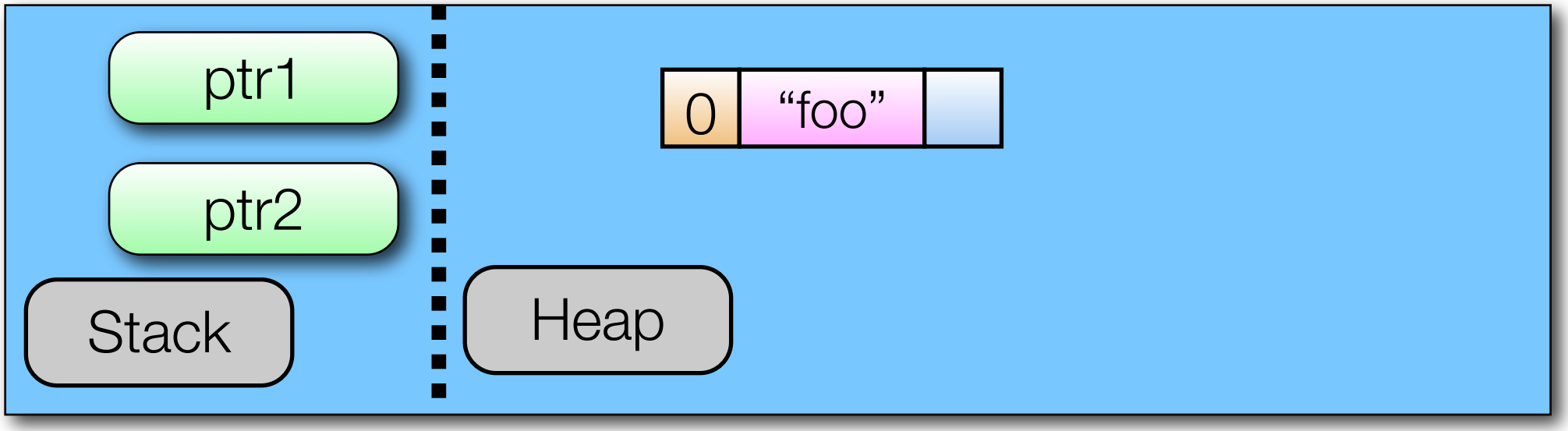
- Each object has an **associated reference counter**



- Keeps reference counter up to date, and **deallocates objects when the counter is zero**

Reference Counting

- Each object has an **associated reference counter**

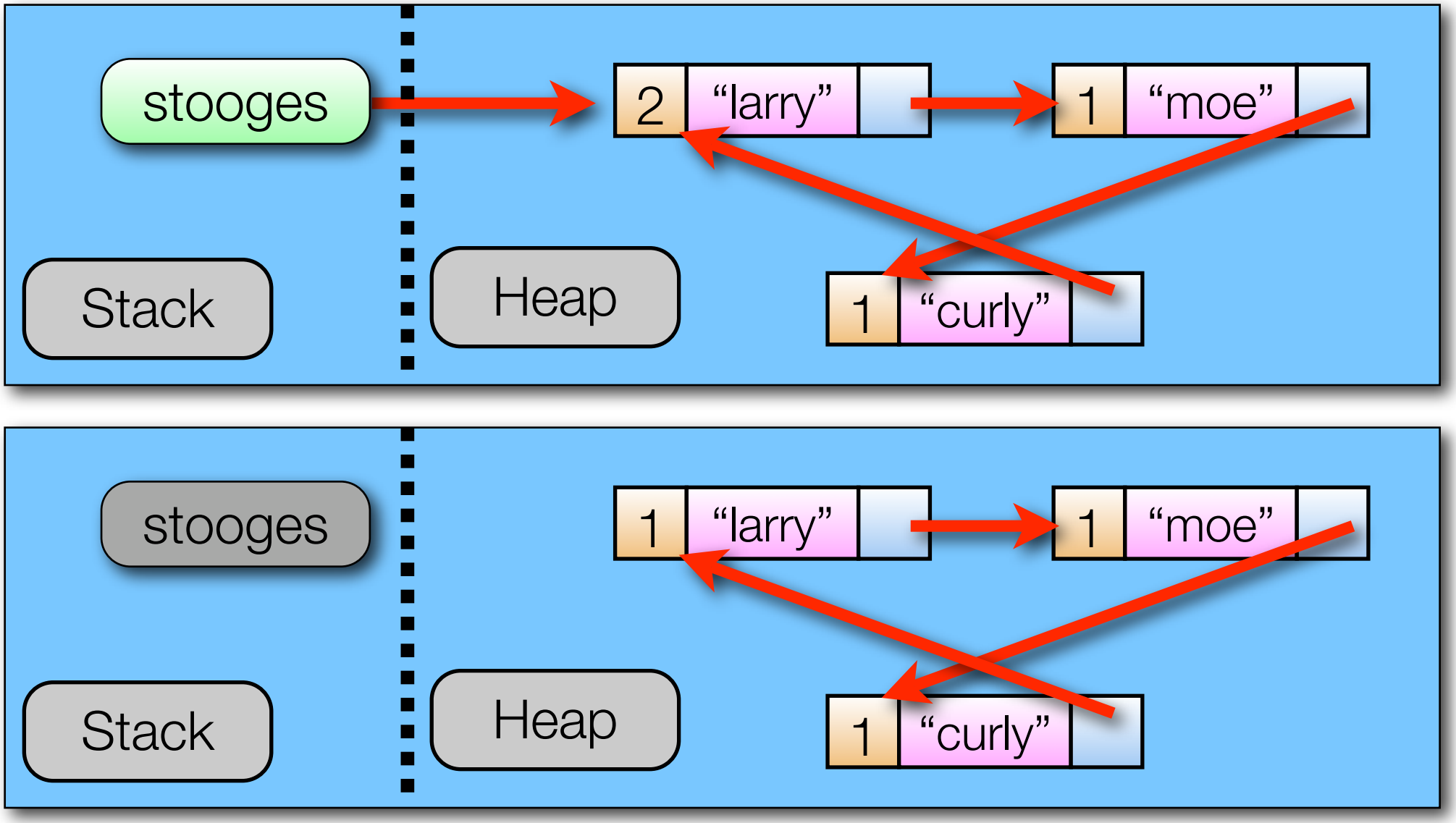


- Keeps reference counter up to date, and **deallocates objects when the counter is zero**

Reference Counting: Problems

- **Extra overhead** of storing and updating reference counts
- **Strong Typing required**
  - Impossible in language like C
  - It cannot be used for variant records
- It doesn't work **with circular data structures**
  - This is a problem with this definition of useful object as an object with one or more references

Reference Counting: Circular Data Structures



Mark-and-Sweep Collection

- A better definition of **useless** is **one that cannot be reached by following a chain of valid pointers starting from outside the heap**.
- Mark-and-Sweep GC applies this definition

Mark-and-Sweep

- Algorithm:
  - Mark every block in the heap as **useless**
  - Starting with all pointers outside the heap, **recursively explore all linked data structures**
  - Add **every block that remain marked to the free list**.
- Run whenever free space is low

Mark-and-Sweep Collection: Problems

- Block must begin with an **indication of its size**
- A **stack of depth proportional to the longest reference chain is required**
  - AND! We are **usually running low when running the GC**
- Must **"stop the world"** to run
  - Much work in parallel garbage collection to address this
  - Also can use periodically in combination with reference counting

Mark-and-Sweep Collection: Problems

- Block must begin with an **indication of its size**
- A **stack of depth proportional to the longest reference chain is required**
  - AND! We
- Must **"sto**
  - Much work in parallel garbage collection to address this
  - Also can use periodically in combination with reference counting

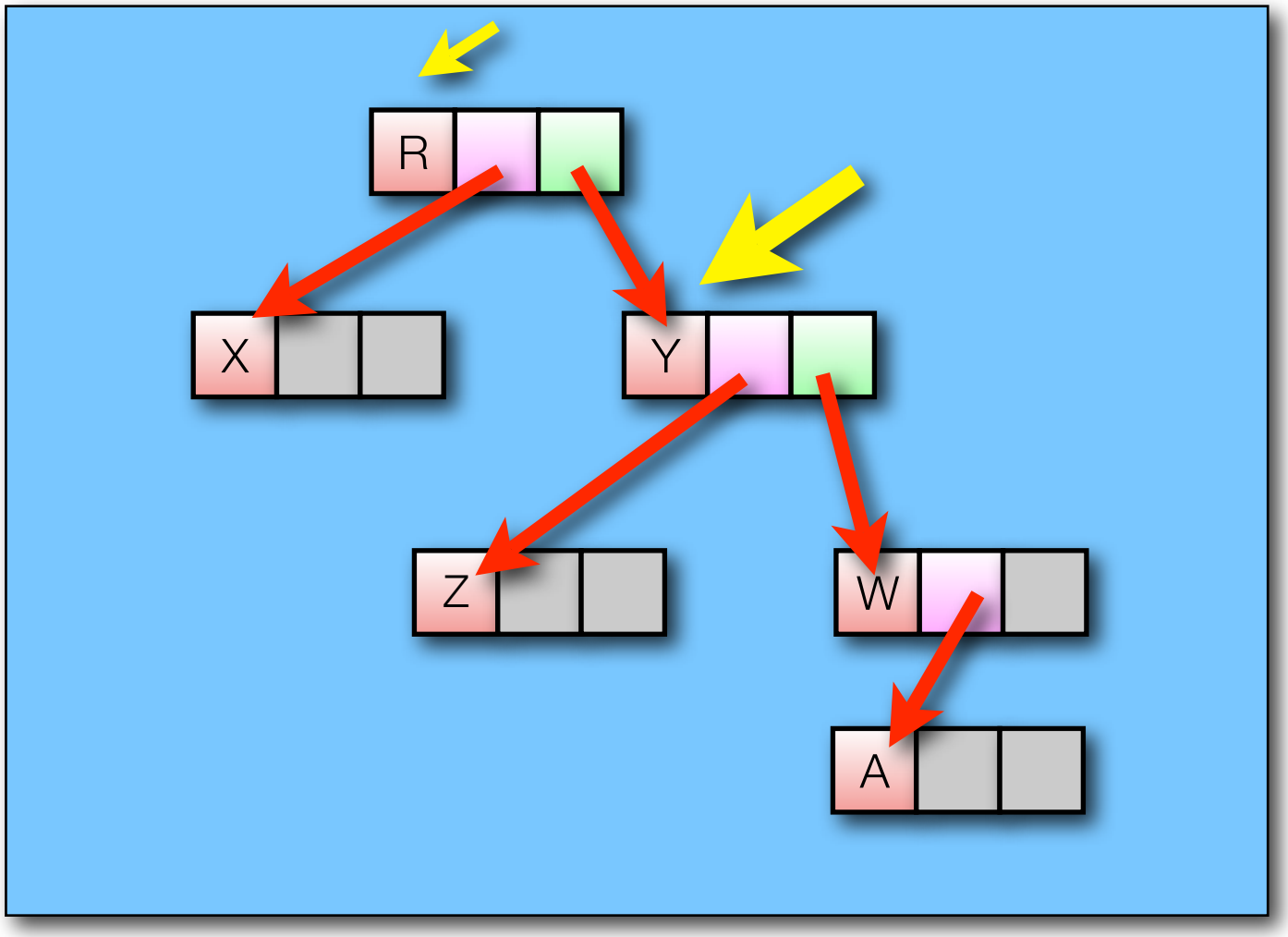
If we use **type descriptors**, that indicate their size, then we don't need to do this.

Mark-and-Sweep Collection: Problems

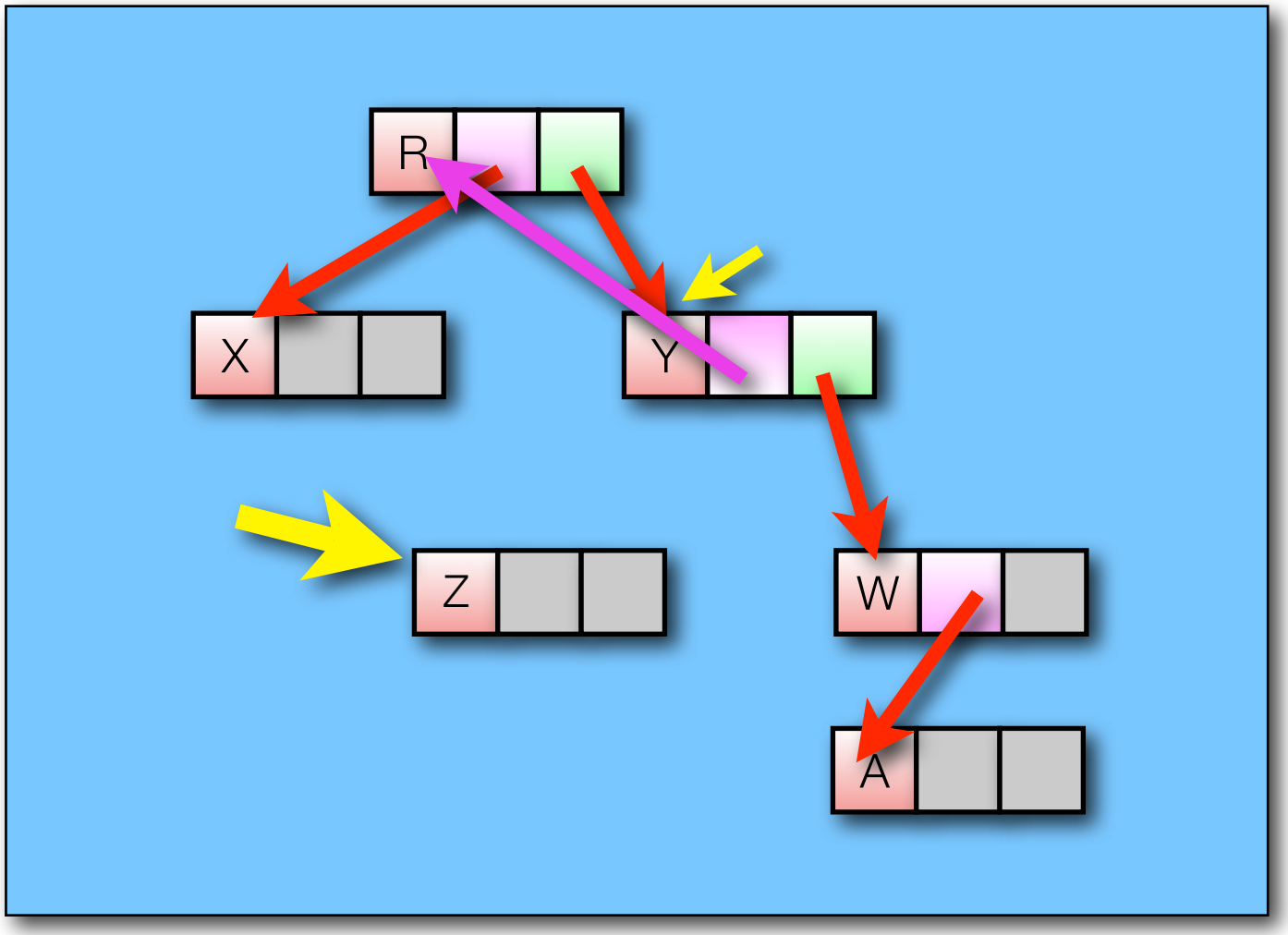
- Block must begin with an **indication of its size**
- A **stack of depth proportional to the longest reference chain is required**
  - AND! We are **usually running out when running the GC**
- Must **"stop the world"** to run
  - Much wo
  - Also can counting

Possible to implement **without a stack**!

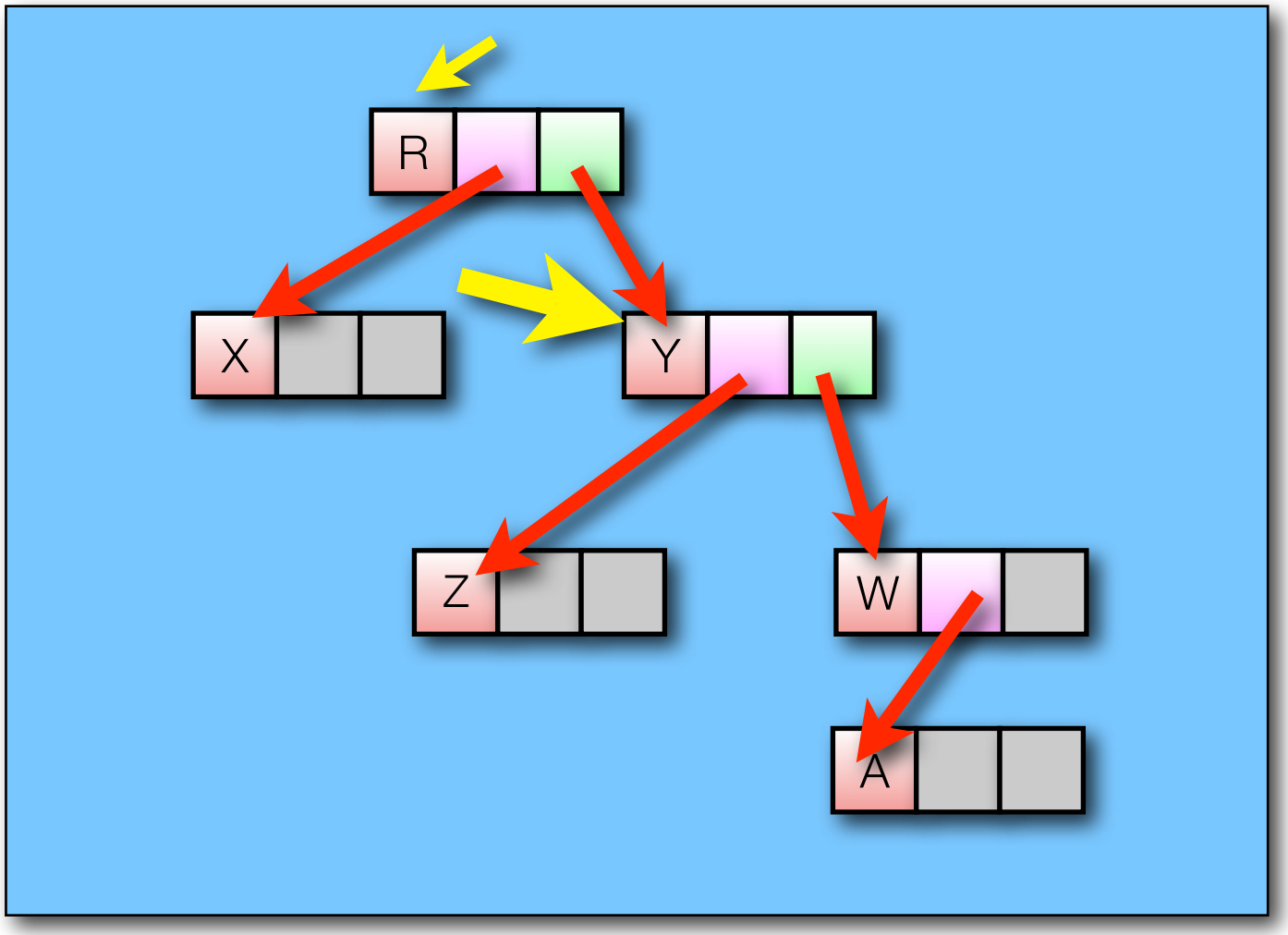
Mark-and-Sweep Collection: Pointer Reversal



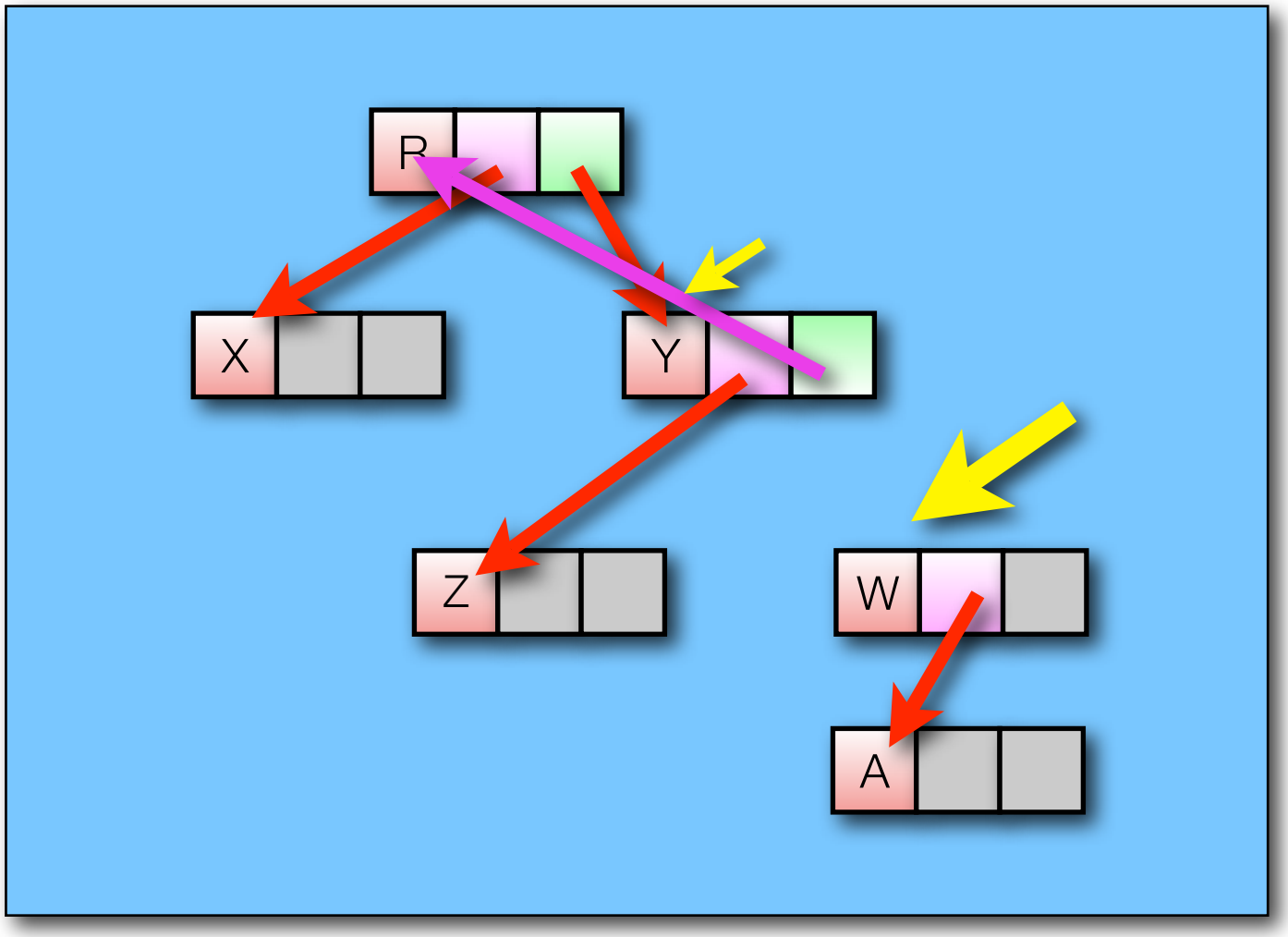
Mark-and-Sweep Collection: Pointer Reversal



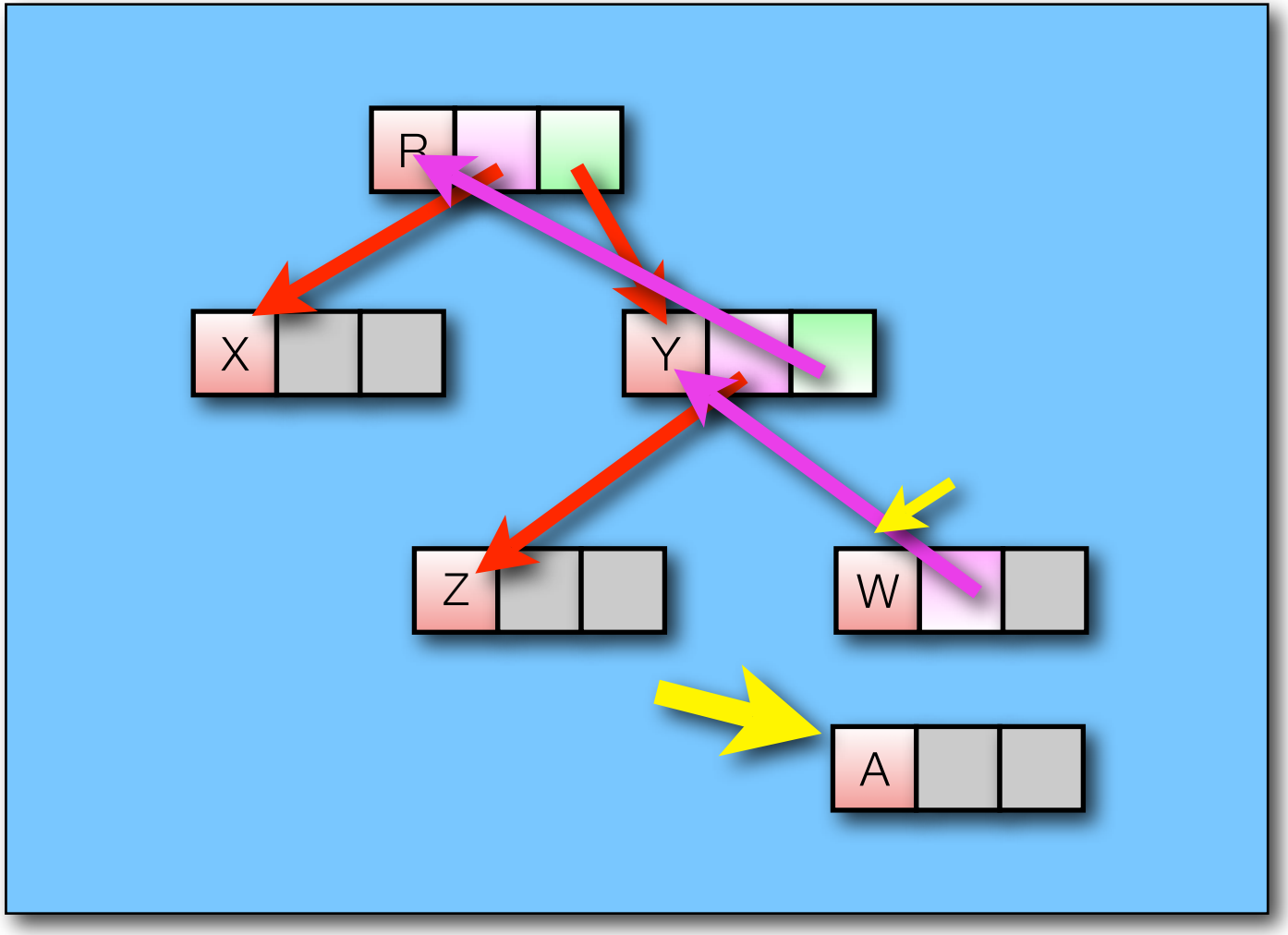
Mark-and-Sweep Collection: Pointer Reversal



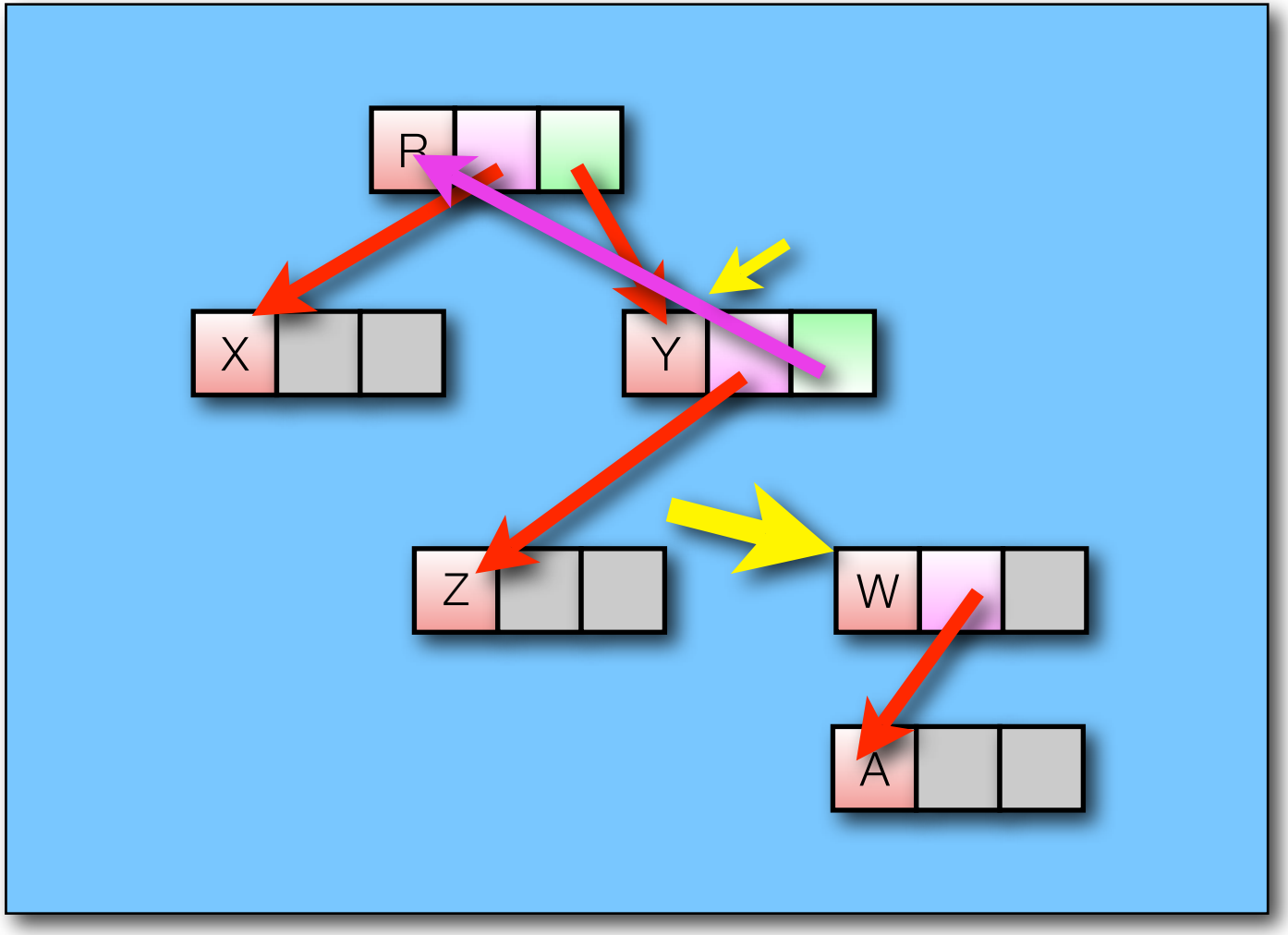
Mark-and-Sweep Collection: Pointer Reversal



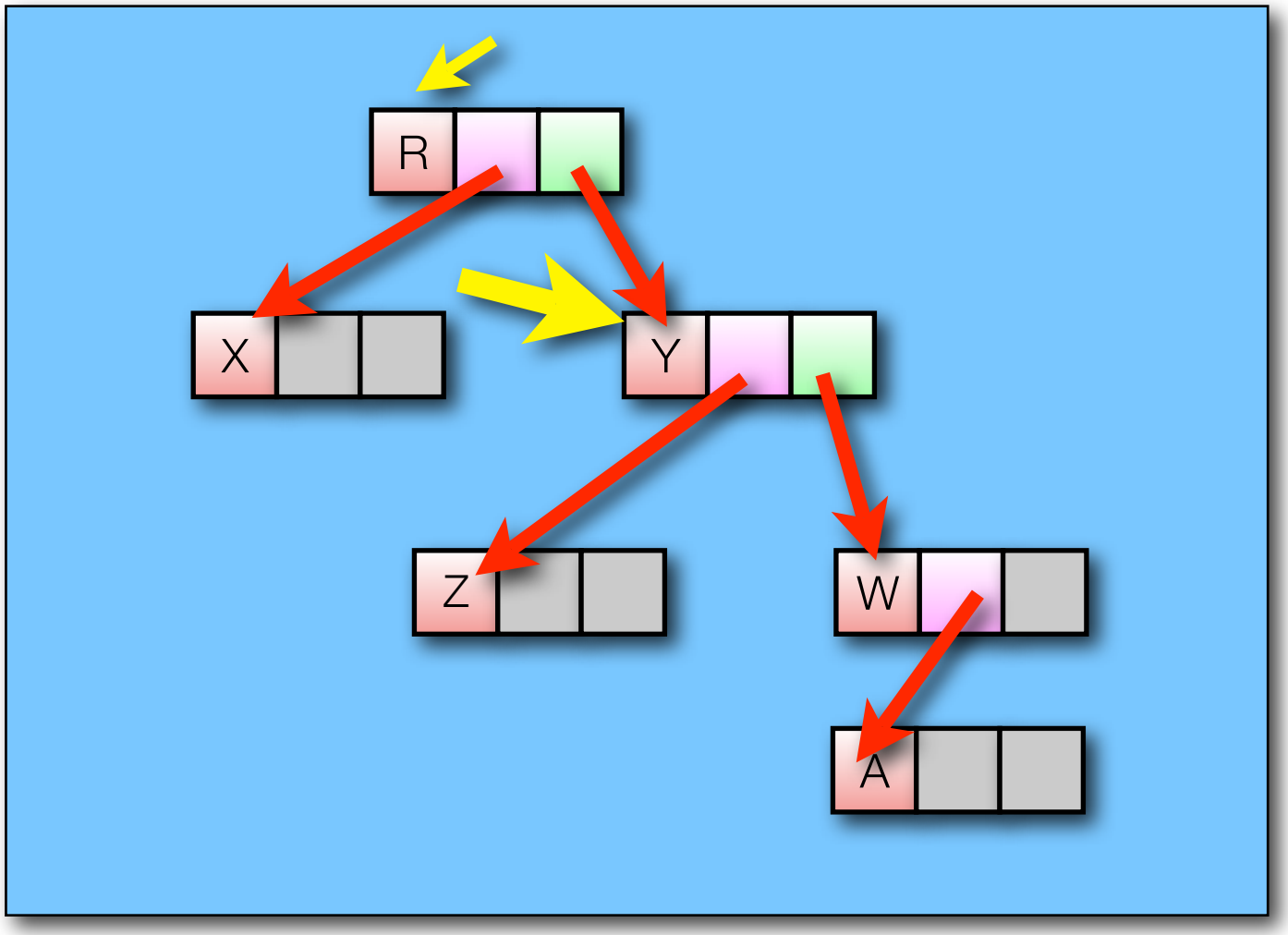
Mark-and-Sweep Collection: Pointer Reversal



Mark-and-Sweep Collection: Pointer Reversal

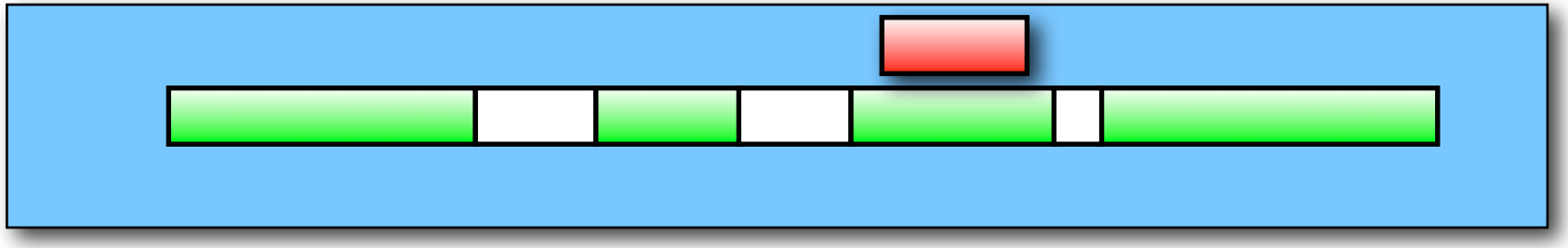


Mark-and-Sweep Collection: Pointer Reversal



Store-and-Copy

- Use to reduce external fragmentation



- S-C divides the **available space in half** and **allocates everything in that half until its full**
- When that happens, **copy each useful block to the other half, clean up the remaining block, and switch the roles of each half**.
- Drawback: only get to use half of heap at a time

Generational Collection

- Based on the idea that objects are often short-lived
- Heap space divided into regions based on how many times objects have been seen by the garbage collector
- Region of objects allocated since last GC (called “nursery”) explored for reclamation before the regions containing older objects

Lists

- We've seen lists in ML
- In LISP, homogeneity is not required
  - A list is technically defined recursively as either the empty list or a pair consisting of an object (which may be either a list or an atom) and another (shorter) list
  - In Lisp, in fact, a program is a list, and can extend itself at run time by constructing a list and executing it.
- Lists are also supported in some imperative programs

List Comprehensions

- Specify expression, enumerator, and filter(s)
- Example: squares of odd numbers less than 100:

```
[i*i | i <- [1..100], i `mod` 2 == 1]
```

- Provided in Haskell, Miranda, Python

Strings

- A string may be regarded as merely an **array of characters** or as a **distinct data type**
- Some languages that consider them character arrays nonetheless provide some syntactic sugar for them
  - For example, C: *char str[ ] = "hello world";*
    - Orthogonality problem: can't assign string literal after declaration
- Support for **variable-length strings**
  - Built-in type in functional languages (Lisp, Scheme, ML)
  - String class in object-oriented languages (C++, Java, C#)

Sets

- A Set is an **unordered collection** of an arbitrary number of **distinct values** of a common type.
- The **universe** comprises all the possible values that could be in the set
  - e.g. The 256 ASCII characters
- Often represented as a bit vector for fast set operations
  - Each bit represents one possible member of the set
  - This is not feasable for a large universe, i.e. the integers
    - Use a hash table or tree instead

Equality and Assignment

- What does equality mean for complex types?
  - **Shallow comparison**: refer to the same object
  - **Deep comparison**: identical structure with identical values throughout all data members
- Can also have shallow or deep assignment
  - Tricky with pointers: In a deep copy, pointers in both copies point to the same objects