Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection

Memory Management

Dynamic Memory Allocation

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection

- Need memory at runtime:
  - Activation records
  - Objects
  - Explicit allocations: **new**, **malloc**, etc.
  - Implicit allocations: strings, file buffers, arrays with dynamically varying size, etc.
- Language systems provide an important hidden player: runtime memory management

Memory Management
Memory Model
Memory Model
Declaring An Array
Creating An Array
Using An Array
Memory Managers In C#
Stacks
Heaps
Current Heap Links
Garbage Collection

Memory Model

Memory Management
Memory Model
Memory Model
Declaring An Array
Creating An Array
Using An Array
Memory Managers In C#
Stacks
Heaps
Current Heap Links
Garbage Collection

Memory Model

- For now, assume that the OS grants each running program one or more fixed-size regions of memory for dynamic allocation
- We will model these regions as C# arrays
  - To see examples of memory management code
  - And, for practice with C#

Memory Management
Memory Model
Memory Model
Declaring An Array
Creating An Array
Using An Array
Memory Managers In C#
Stacks
Heaps
Current Heap Links
Garbage Collection

Declaring An Array

Example (A C# array declaration)

```
int[] a = null;
```

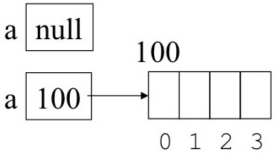
- Array types are reference types—an array is really an object, with a little special syntax
- The variable **a** above is initialized to **null**
- It can hold a reference to an array of **int** values, but does not yet

Memory Management
Memory Model
Memory Model
Declaring An Array
Creating An Array
Using An Array
Memory Managers In C#
Stacks
Heaps
Current Heap Links
Garbage Collection

Creating An Array

Example (Use new to create an array object)

```
int[] a = null;
a = new int[4];
```



Example (Single statement)

```
int[] a = new int[4];
```

Memory Management
Memory Model
Memory Model
Declaring An Array
Creating An Array
Using An Array
Memory Managers In C#
Stacks
Heaps
Current Heap Links
Garbage Collection

Using An Array

Example

```
int i = 0;
while (i < a.Length) {
  a[i] = 5;
  i++;
}
```

- Use **a[i]** to refer to an element as lvalue or rvalue: a[i] = 5;
  - **lvalue** is memory address: **a[i]**
  - **rvalue** is a value 5;
- **a** is an array reference expression and **i** is an **int** expression
- Use **a.Length** to access length
- Array indexes are 0..(**a.Length**-1)

Memory Management
Memory Model
Memory Model
Declaring An Array
Creating An Array
Using An Array
Memory Managers In C#
Stacks
Heaps
Current Heap Links
Garbage Collection

Memory Managers In C#

Example

```
public class MemoryManager {
  private int[] memory;
  /**
   * MemoryManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public MemoryManager(int[] initialMemory) {
    memory = initialMemory;
  }
  ...
}
```

- We will show C# implementations this way. The **initialMemory** array is the memory region provided by the operating system.

Memory Management
Memory Model
Stacks
Stacks Of Activation Records
A Stack Illustration
A C# Stack Implementation
Heaps
Current Heap Links
Garbage Collection

Stacks

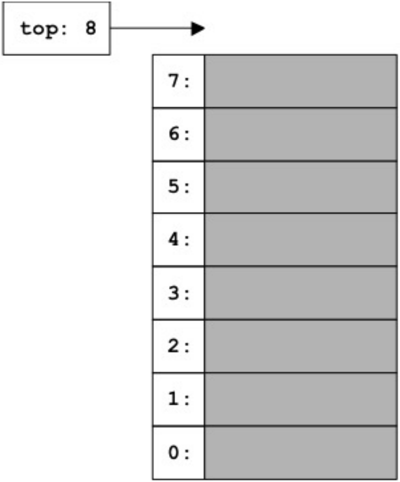
Memory Management
Memory Model
Stacks
Stacks Of Activation Records
A Stack Illustration
A C# Stack Implementation
Heaps
Current Heap Links
Garbage Collection

Stacks Of Activation Records

- **Recursion** requires multiple instances of function execution or activation
- Each instance requires parameters and local data held in memory defined as the activation record
- For recursive languages, activation records must be allocated dynamically
- Generally it suffices to **allocate** an activation record on call and **deallocate** on return
- This produces a stack of activation records: push on call, pop on return
- A simple memory management problem

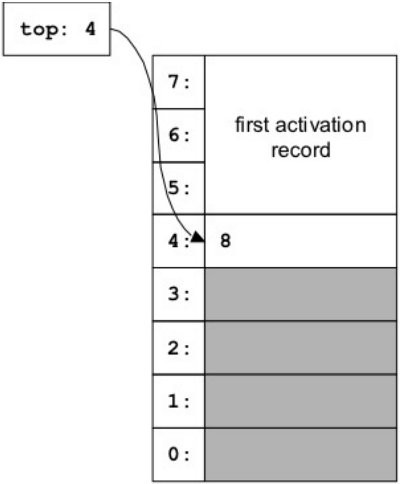
Memory Management
Memory Model
Stacks
Stacks Of Activation Records
A Stack Illustration
A C# Stack Implementation
Heaps
Current Heap Links
Garbage Collection

A Stack Illustration

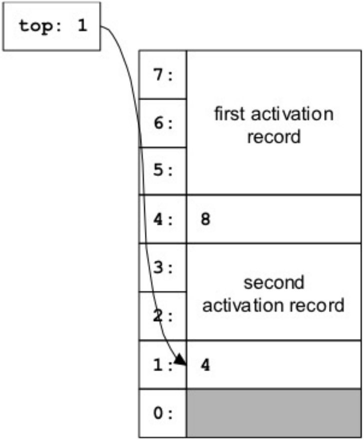


- An empty stack of 8 words. The stack will grow down, from high addresses to lower addresses.
- A reserved memory location (perhaps a register) records the address of the lowest allocated word.

A stack manager **m** with a memory array of 8 words, initially empty.



- The program calls **m.push(3)**, which returns 5: the address of the first of the 3 words allocated for an activation record.
- Memory management uses an extra word to record the previous value of top.



m.push(3);
m.push(2);

- The program calls **m.push(2)**, which returns 2: the address of the first of the 2 words allocated for an activation record. The stack is now full – there is not room even for **m.push(1)**.
- For m.pop(), just do **top = memory[top]** to return to previous configuration.

Memory Management
Memory Model
Stacks
Stacks Of Activation Records
A Stack Illustration
A C# Stack Implementation
Heaps
Current Heap Links
Garbage Collection

A C# Stack Implementation

Example

```
public class StackManager {
  private int[] memory; // the memory we manage
  private int top;      // index of top stack block
  /**
   * StackManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public StackManager(int[] initialMemory) {
    memory = initialMemory;
    top = memory.Length;
  }
  ...
```

push

Example

```
/**
 * Allocate a block and return its address.
 * @param requestSize int size of block, > 0
 * @return block address
 * @throws StackOverflowException if no stack space
 */
public int push(int requestSize) {
  int oldtop = top;
  top -= (requestSize+1); //extra word for oldtop
  if (top<0)
    throw new System.StackOverflowException();
  memory[top] = oldtop;
  return top+1;
}
```

Memory Management
Memory Model
Stacks
Stacks Of Activation Records
A Stack Illustration
A C# Stack Implementation
Heaps
Current Heap Links
Garbage Collection

pop

Example

```
  /**
   * Pop the top stack frame. This works only if
   * the stack is not empty.
   */
  public void pop() {
    top = memory[top];
  }
}
```

Example

```
public class StackManager {
  private int[] memory; // the memory we manage
  private int top;      // index of top stack block
  public StackManager(int[] initialMemory) {
    memory = initialMemory;
    top = memory.Length;
  }
  public int push(int requestSize) {
    int oldtop = top;
    top -= (requestSize+1); // oldtop extra word
    if (top<0)
      throw new System.StackOverflowException();
    memory[top] = oldtop;
    return top+1;
  }
  public void pop() {
    top = memory[top];
  }
}
```

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Heaps

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

The Heap Problem

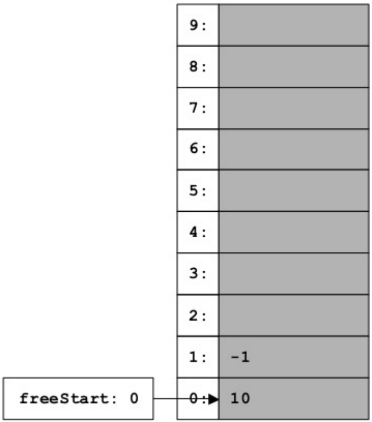
- Stack order allocation/deallocation makes implementation easy
- Not always possible: what if memory allocations and deallocations can come in any order?
- A **heap** is a pool of blocks of memory, with an interface for unordered **runtime** memory allocation and deallocation
- There are many mechanisms for this. . .

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

First Fit

- A linked list of free blocks, initially containing one big free block
- To allocate:
  1. Search free list for first adequate block
  2. If there is extra space in the block, return the unused portion at the upper end to the free list
  3. Allocate requested portion (at the lower end)
- To free, just add to the front of the free list

Heap Illustration



A heap manager **m** with a memory array of 10 words, initially empty.

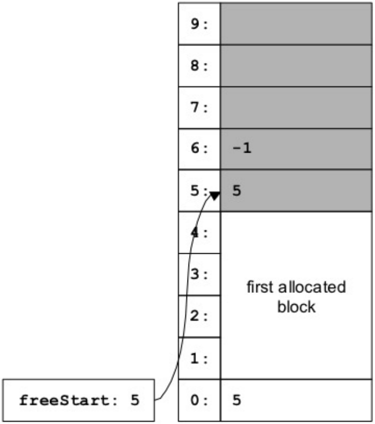
The link to the head of the free list is held in **freeStart**.

Every block, allocated or free, has its length in its first word.

Free blocks have free-list link in their second word, or -1 at the end of the free list.

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

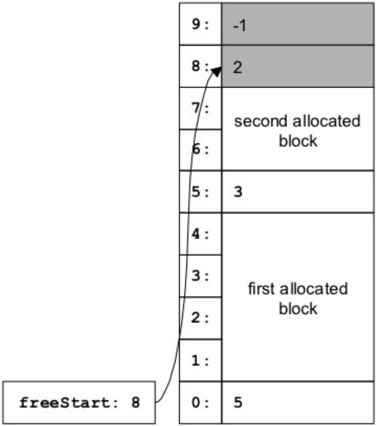
Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection



p1=m.allocate(4);

p1 = 1 – the address of the first of four allocated words.
An extra word holds the block length.
Remainder of the big free block was returned to the free list.
6: -1 End of free-list
5: 5 Free block length
0: 5 Block length

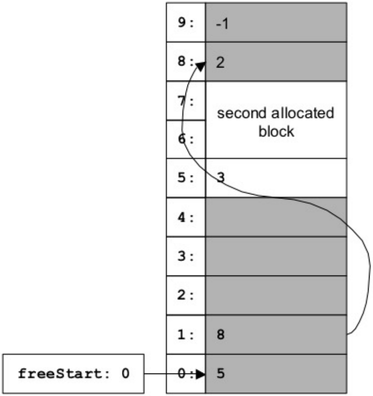
Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection



p1=m.allocate(4);
p2=m.allocate(2);

p1 = 1
p2 = 6 – address of first of two allocated words.
An extra word holds block length.
Remainder of the free block was returned to the free list.
9: −1 End of free-list
8: 2 Free block length
5: 3 Block length
0: 5 Block length

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection



p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);

Deallocates the first allocated block, returned to the head of the free list.

p2 = 6

9: –1 End of free-list

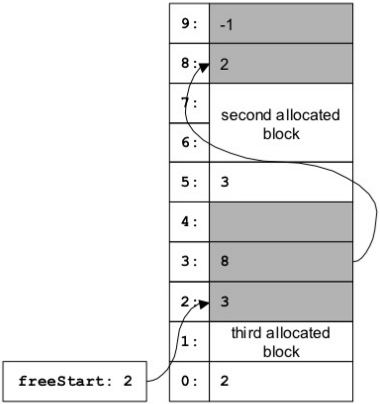
8: 2 Free block length

5: 3 Allocated block length

1: 8 Link to next free block

0: 5 Allocated block length

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

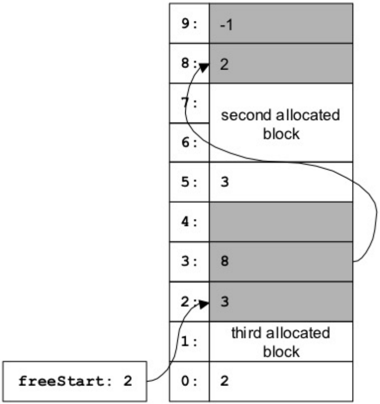


p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);
p3=m.allocate(1);

p2 = 6
p3 = 1 – address of allocated word.
Two suitable blocks. Other would have been an exact fit using Best Fit.
9: −1 End of free-list
8: 2 Free block length.
5: 3 Allocated block length.
3: 8 Link to next free block.
2: 3 Free block length.
0: 2 Block length.

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Exercise



p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);
p3=m.allocate(1);

1. How much memory is free?
2. What is the largest possible allocation?
3. Can this be allocated? **p4=m.allocate(2);**
4. Can this be allocated? **p4=m.allocate(3);**

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

A C# Heap Implementation

Example

```
public class HeapManager {
  static private final int NULL = -1; // null link
  public int[] memory;   // the memory we manage
  private int freeStart; // start of the free list
  /**
   * HeapManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public HeapManager(int[] initialMemory) {
    memory = initialMemory;
    memory[0] = memory.Length; // one big free block
    memory[1] = NULL; // free list ends with it
    freeStart = 0;    // free list starts with it
  }
```

Example

```
/**
 * Allocate a block and return its address.
 * @param requestSize int size of block, > 0
 * @return block address
 * @throws OutOfMemoryError if no block big enough
 */
public int allocate(int requestSize) {
  int size = requestSize + 1; // size with header
  // Do first-fit search: linear search of the free
  // list for the first block of sufficient size.
  int p = freeStart; // head of free list
  int lag = NULL;
  while (p!=NULL && memory[p]<size) {
    lag = p; // lag is previous p
    p = memory[p+1]; // link to next block
  }
  if (p==NULL) // no block large enough
    throw new System.OutOfMemoryException();
  int nextFree = memory[p+1]; // block after p
```

Example

```
    // Now p is the index of a block of sufficient size,
    // and lag is the index of p's predecessor in the
    // free list, or NULL, and nextFree is the index of
    // p's successor in the free list, or NULL.
    // If the block has more space than we need, carve
    // out what we need from the front and return the
    // unused end part to the free list.
    int unused = memory[p]-size; // extra space
    if (unused>1) { // if more than a header's worth
      nextFree = p+size; // index of the unused piece
      memory[nextFree] = unused; // fill in size
      memory[nextFree+1] = memory[p+1]; // fill in link
      memory[p] = size; // reduce p's size accordingly
    }
    // Link out the block we are allocating and done.
    if (lag==NULL) freeStart = nextFree;
    else memory[lag+1] = nextFree;
    return p+1; // index of useable word (after header)
  }
```

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Example

```
  /**
   * Deallocate an allocated block. This works only
   * if the block address is one that was returned
   * by allocate and has not yet been deallocated.
   * @param address int address of the block
   */
  public void deallocate(int address) {
    int addr = address-1;
    memory[addr+1] = freeStart;
    freeStart = addr;
  }
}
```

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

A Problem

Example

```
p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);
m.deallocate(p2);
p3=m.allocate(7);
```

- Final **allocate** will fail: we are breaking up large blocks into smaller blocks but never reversing the process
- Need to *coalesce* adjacent free blocks

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

A Solution

- We can implement a smarter **deallocate** method
  - Maintain the free list sorted in address order
  - When freeing, look at the previous free block and the next free block
  - If adjacent, coalesce
- This is a lot more work than just returning the block to the head of the free list

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Example

```
/**
  * Deallocate an allocated block. This works only
  * if the block address is one that was returned
  * by allocate and has not yet been deallocated.
  * @param address int address of the block
  */
 public void deallocate(int address) {
   int addr = address-1; // real start of the block

   // Find the insertion point in the sorted free
   // list for this block.
   int p = freeStart;
   int lag = NULL;
   while (p!=NULL && p<addr) {
     lag = p;
     p = memory[p+1];
   }
```

Example

```
// Now p is the index of the block to come after
// ours in the free list, or NULL, and lag is the
// index of the block to come before ours in the
// free list, or NULL.

// If the one to come after ours is adjacent to it,
// merge it into ours and restore the property
// described above.

if (addr+memory[addr]==p) {
  memory[addr] += memory[p]; // add its size to ours
  p = memory[p+1]; //
}
```

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Example

```
  if (lag==NULL) { // ours will be first free
    freeStart = addr;
    memory[addr+1] = p;
  }
  else if (lag+memory[lag]==addr) { // block before is
                                    // adjacent to ours
    memory[lag] += memory[addr]; // merge ours into it
    memory[lag+1] = p;
  }
  else { // neither: just a simple insertion
    memory[lag+1] = addr;
    memory[addr+1] = p;
  }
}
```

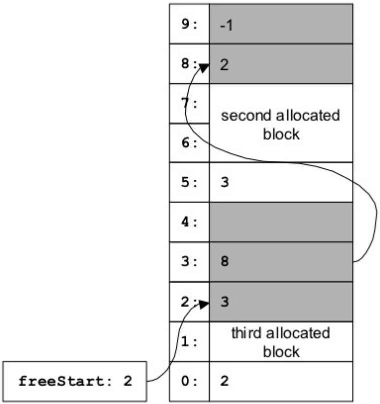
Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Quick Lists

- Small blocks tend to be allocated and deallocated much more frequently
- A common optimization: keep separate free lists for popular (small) block sizes
- On these *quick lists*, blocks are one size
- *Delayed coalescing*: free blocks on quick lists are not coalesced right away (but may have to be coalesced eventually)

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Fragmentation



- When free regions are separated by allocated blocks, so that it is not possible to allocate all of free memory as one block
- More generally: any time a heap manager is unable to allocate memory even though enough is free
  - If it allocated more than requested
  - If it does not coalesce adjacent free blocks

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Other Heap Mechanisms

- An amazing variety
- Three major issues:
  - Placement – where to allocate a block
  - Splitting – when and how to split large blocks
  - Coalescing – when and how to recombine
- Many other refinements

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Placement

- Where to allocate a block
- Our mechanism: *first fit* from FIFO free list
- Some mechanisms use a similar linked list of free blocks: *first fit*, *best fit*, *next fit*, etc.
- Some mechanisms use a more scalable data structure like a balanced binary tree

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Splitting

- When and how to split large blocks
- Our mechanism: split to requested size
- Sometimes you get better results with less splitting – just allocate more than requested
- A common example: rounding up allocation size to some multiple

Memory Management
Memory Model
Stacks
Heaps
The Heap Problem
First Fit
Heap Illustration
Exercise
A C# Heap Implementation
A Problem
A Solution
Quick Lists
Fragmentation
Other Heap Mechanisms
Placement
Splitting
Coalescing
Current Heap Links
Garbage Collection

Coalescing

- When and how to recombine adjacent free blocks
- We saw several varieties:
  - No coalescing
  - Eager coalescing
  - Delayed coalescing (as with quick lists)

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Current Heap Links

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Current Heap Links

- So far, the running program is a black box: a source of allocations and deallocations
- What does the **running program** do with addresses allocated to it?
- Some systems track current heap links
- A *current heap link* is a memory location where a value is stored that the running program will use as a heap address

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Problem: Find Current Heap Links

- Basic problem is to find heap memory to be freed.
- Start with the root set: memory locations outside of the heap with links into the heap
  - Active activation records (if on the stack)
  - Static variables, etc.
  - Dynamic allocations, using keyword **new**
- For each memory location in the set, look at the allocated block it points to, and add all the memory locations in that block
- Repeat until no new locations are found

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Discarding Impossible Links

- Depending on the language and implementation, we may be able to discard (ignore) some locations from the set:
  - If they do not point into allocated heap blocks
  - If they do not point to allocated heap blocks (C#, but not C), for example: **Intlist a = null;**
  - If their static type rules out use as heap links (C#, but not C) and cannot be freed: **int a = 5**

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Errors In Current Heap Links

- *Exclusion errors*: a memory location that actually is a current heap link is left out
- *Unused inclusion errors*: a memory location is included, but the program never actually uses the value stored there
- *Used inclusion errors*: a memory location is included, but the program uses the value stored there as something other than a heap address – as an integer in C, for example

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Errors Are Unavoidable

- For heap manager purposes, exclusion errors are unacceptable
- We must include a location if it might be used as a heap link (e.g. aliased reference undetectable)
- This makes unused inclusion errors unavoidable
- Depending on the language, used inclusions may also be unavoidable (e.g. if aliases are allowed)

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Used Inclusion Errors In C

- Static type and runtime value may be of no use in telling how a value will be used
- Variable **x** may be used either as a pointer or as an int.

Example

```
union {
   char *p;   /* May hold heap address */
   int i;     /* int                   */
} x;
x.p = (char *) malloc(5);
x.i++;
```

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Current Heap Links
Problem
Discarding Links
Errors In Links
Errors Are Unavoidable
Used Inclusion Errors In C
Heap Compaction
Garbage Collection

Heap Compaction

- One approach based on current heap links
- Memory manager follows links and moves allocated blocks:
  - Copy the block to a new location
  - Update all links referencing that block
- So it can compact the heap, moving all allocated blocks to one end, leaving one big free block and no fragmentation
- When to compact?
  - After every deallocation but may not be necessary
  - When there's no free heap memory (execution suspended while memory manager executes). Bad for time sensitive operations.

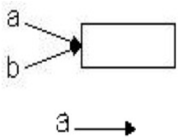
Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Garbage Collection

Common Human Managed Pointer Errors

Example

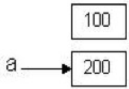
```
int *a = new int();
int *b = a;
delete(b);
*a = 21;
```



Dangling pointer: uses a reference after the memory it pointed to has been deallocated

Example

```
int *a = new int();
a = new int();
```



Memory leak: first allocation (100) not deallocated and not now accessible

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Garbage Collection

- Since so many errors are caused by improper deallocation…
- …and since it is a burden on the programmer to have to worry about it…
- …why not have the language system reclaim blocks automatically?

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

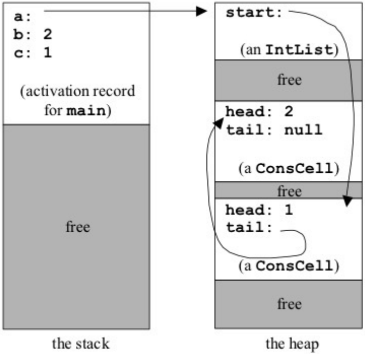
Three Major Approaches

- Mark and sweep
- Copying
- Reference counting

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Mark And Sweep

- A mark-and-sweep collector uses current heap links in a two-stage process:
  - *Mark*: find the live heap links and mark all the heap blocks linked to by them
  - *Sweep*: make a pass over the heap and return unmarked blocks to the free pool
- Blocks are not moved, so *used* and *unused* inclusion errors are tolerated



Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Mark and Sweep Implementation

- Performed only when memory is nearly exhausted.
- *Mark phase*: follow roots of all lists, marking each as visited.
- *Sweep phase*: examine all memory, if not marked reclaim, adding to free-list, set all memory to unmarked.

This was the first GC algorithm, devised by John McCarthy for Lisp.

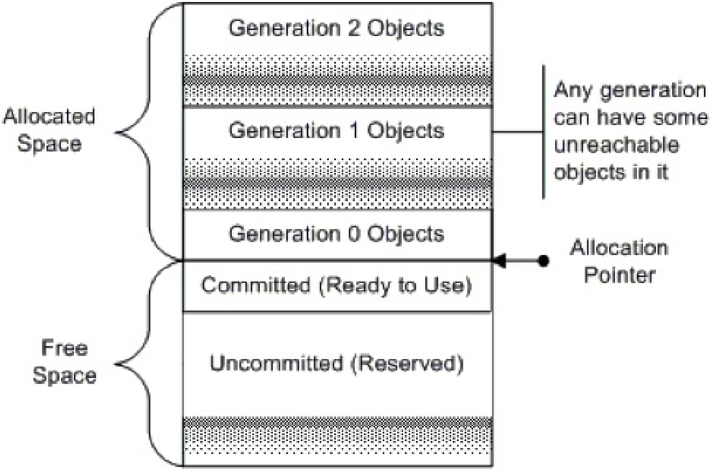
Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

.NET CLR Garbage Collection Rules

- All garbage-collectable objects are allocated from one contiguous range of address space.
- Heap divided into generations so possible to eliminate most of the garbage by looking at only a small fraction of the heap.
- Objects within a generation are all roughly the same age.
- Higher-numbered generations indicate areas of the heap with older objects – those objects are much more likely to be stable.
- The oldest objects are at the lowest addresses, while new objects are created at increasing addresses.
- The allocation pointer for new objects marks the boundary between the used (allocated) and unused (free) areas of memory.
- Periodically the heap is compacted by removing dead objects and sliding the live objects up toward the low-address end of the heap. This expands the unused area at the bottom of the diagram in which new objects are created.
- Order of objects in memory remains the order in which created, for good locality.
- There are never any gaps between objects in the heap.
- Only part of the free space is committed. When necessary, more memory is acquired from the operating system in the reserved address range.

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Diagram



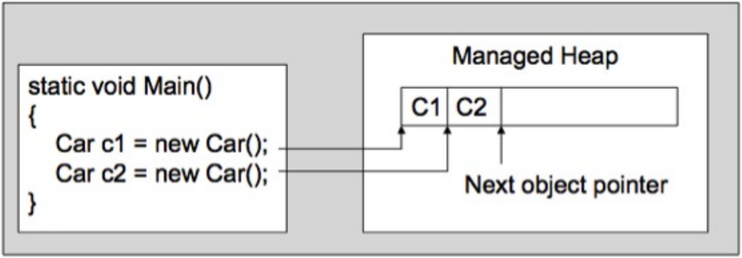
Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Tasks of new instruction

- Calculate total amount of memory required for object
- Examine managed heap to ensure room for object
- Return the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Allocate objects sequentially on heap

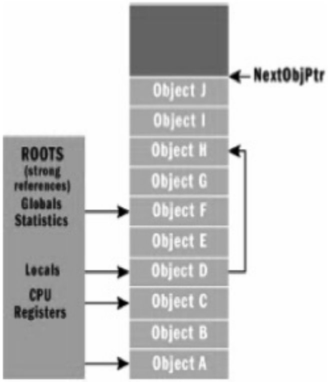


- Rule: If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will occur.

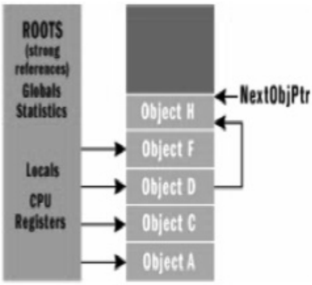
Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Reachable Objects

Follow stack object pointers into heap



Allocated objects on the heap



Managed heap after collection

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Optimize Decision

- Object generations:
  - Each object assigned generation (e.g. 0 to 2)
    - Generation 0: newly allocated objects
    - Generation 1: objects GCed once
    - Generation 2: objects GCed twice
  - Generation 0 most active (temporary objects)
  - GC Generation 0, if allocate fails, GC older generations.

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Garbage Collection Steps

- Mark: Garbage collector searches for managed objects referenced in managed code
- Sweep: Garbage collector attempts to finalize objects that are unreachable
- Sweep: Garbage collector frees objects that are unmarked and reclaims their memory

Finalize in C#

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

- Allows an Object to attempt to free resources and perform other cleanup operations before the Object is reclaimed by garbage collection.
- Example: Write object to a file.
- Implement a *finalizer* only if there are *unmanaged* resources to dispose, such as files, network connections, etc.
- *Finalize* is not called directly or overridden but is implicitly called when a destructor executes.
- In following example, **~ExampleClass()** destructor executed when **ExampleClass** object has no references (to deallocate).

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Example (Destructor)

```
using System;
using System.Diagnostics;
public class ExampleClass {
  Stopwatch sw;
  public ExampleClass() {
    sw = Stopwatch.StartNew();
    Console.WriteLine("Instantiated object");
  }
  ~ExampleClass() {
    sw.Stop();
    Console.WriteLine("Finalizing instance {0}."
      + " Existed {1}", this, sw.Elapsed);
  }
}
public class Demo {
  public static void Main() {
    ExampleClass ex = new ExampleClass();
    ex.ShowDuration();
  }
}
```

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Reference Count Steps

Rather than mark and sweep, keeping track of a count of references on each object can aid with garbage collection.

- Set count to 1 on allocation,
- Increment by 1 with each new reference,
- Decrement by 1 whenever a name no longer references,
- Reclaim memory when the reference count becomes 0.

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Allocating a Reference Count Cell

- The following assumes that generic memory cells are allocated for use in representing the universal data structure of a linked list.
- Used in languages such as Objective-C, Scheme, Lisp, etc.

Example

```
class Cell {
  Object left, right;
  int count=1;
}
...  Cell Q = new Cell();
```

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Deallocate – Decrement Count

- Each time another name references the same memory (aliases) the count is incremented by 1.
- Whenever a name no longer references a location the count is decremented using the following code:

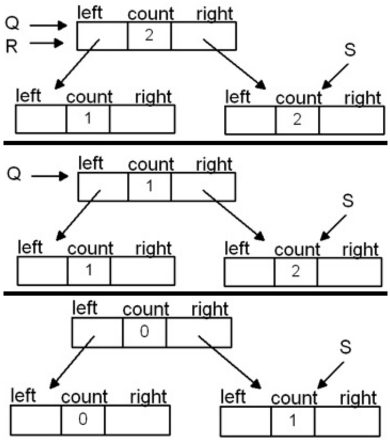
Example

```
void decrement(Cell c) {
  c.count--;
  if(c.count == 0) {
    decrement(c.right);
    decrement(c.left);
    delete c;
  }
}
```

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Deallocate Example

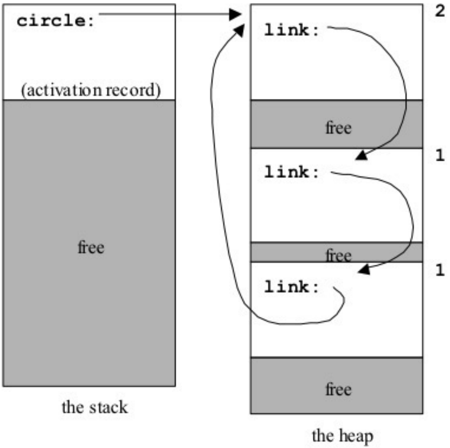
- Before decrement(R) on the memory references
- After decrement(R) on the memory references
- After decrement(Q) on the memory references, cell having count of 0 are reclaimed.



Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Reference Counting Problem

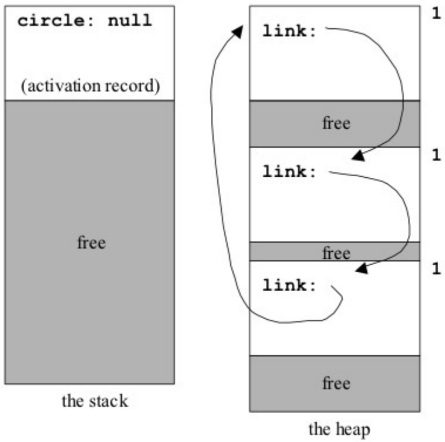
- One problem with reference counting: it misses cycles of garbage.
- Here, a circularly linked list is pointed to by circle.



Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Reference Counting Problem

- When circle is set to null, the reference counter is decremented.
- No reference counter is zero, though all blocks are garbage.



Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Reference Counting

- Problem with cycles of garbage
- Problem with performance generally, since the overhead of updating reference counters is high, must follow links
- One advantage: naturally incremental, with no big pause as collecting occurs constantly, when reference counter = 0

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Garbage Collecting Refinements

- *Generational* collectors
  - Divide block into generations according to age
  - Garbage collect in younger generations more often (using previous methods)
- *Incremental* collectors
  - Collect garbage a little at a time
  - Avoid the uneven performance of ordinary mark-and-sweep and copying collectors

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Garbage Collecting Languages

- Some require it: C#, ML, Scala, Scheme, Lisp
- Some encourage it: Ada
- Some make it difficult: Objective-C, C, C++
  - Objective-C has GC but only recently
  - Even for C and C++ it is possible
  - STL and other libraries that replace the usual **malloc/free** with a garbage-collecting manager

Memory Management
Memory Model
Stacks
Heaps
Current Heap Links
Garbage Collection
Pointer Errors
Garbage Collection
3 Approaches
Mark And Sweep
Implementation
.NET GC Rules
Allocate Sequentially
Optimize Decision
GC Steps
Finalize in C#
Reference Counting
Allocate a Cell
Deallocate a Cell
GC Refinements
GC Languages

Conclusion

- Memory management is an important hidden player in language systems
- Performance and reliability are critical
- Different techniques are difficult to compare, since every run of every program makes different memory demands
- An active area of language systems research and experimentation