Dynamic Memory Allocation: Garbage Collectors

Introduction to Computer Systems

Today

- **Garbage collection**
  - **Basic concepts**
  - Mark and Sweep
  - Reference Counting
  - Copying
  - Treadmill
  - Generational
- **Memory-related perils and pitfalls**

Implicit Memory Management: Garbage Collection

- ***Garbage collection:*** **automatic reclamation of heap-allocated storage—application never has to free**

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- **Common in many dynamic languages:**
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica

- **Variants ("conservative" garbage collectors) exist for C and C++**
  - However, cannot necessarily collect all garbage

Garbage Collection

- **How does the memory manager know when memory can be freed?**
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them

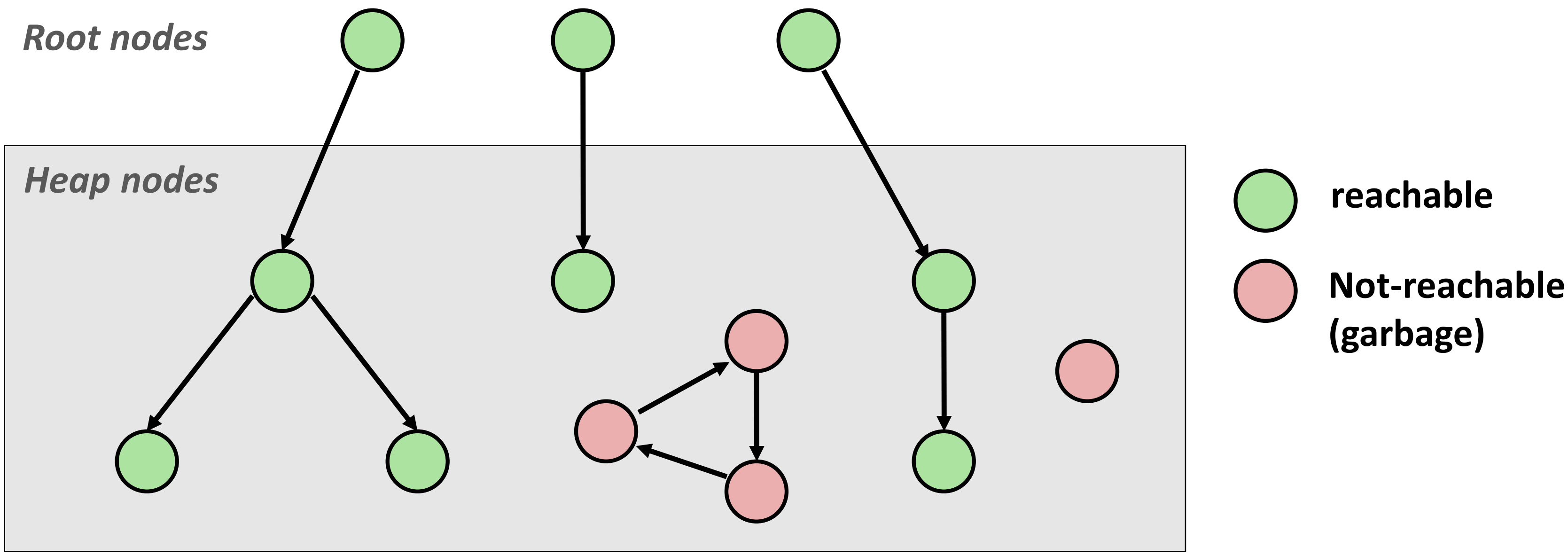
- **Must make certain assumptions about pointers**
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)

Classical GC Algorithms

- **Basic GC**
  - Mark-and-sweep collection (McCarthy, 1960)
    - Does not move blocks (unless you also "compact")
  - Reference counting (Collins, 1960)
    - Does not move blocks
  - Copying collection (Minsky, 1963)
    - Moves blocks
- **Advanced GC**
  - Tricolor Marking and The Treadmill
  - Generational Collectors (Lieberman and Hewitt, 1983)
    - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So focus reclamation work on zones of memory recently allocated
- **For more information: Jones and Lin, "*Garbage Collection: Algorithms for Automatic Dynamic Memory*", John Wiley & Sons, 1996.**

Memory as a Graph

- **We view memory as a directed graph**
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g. registers, locations on the stack, global variables)



A node (block) is *reachable* if there is a path from any root to that node.

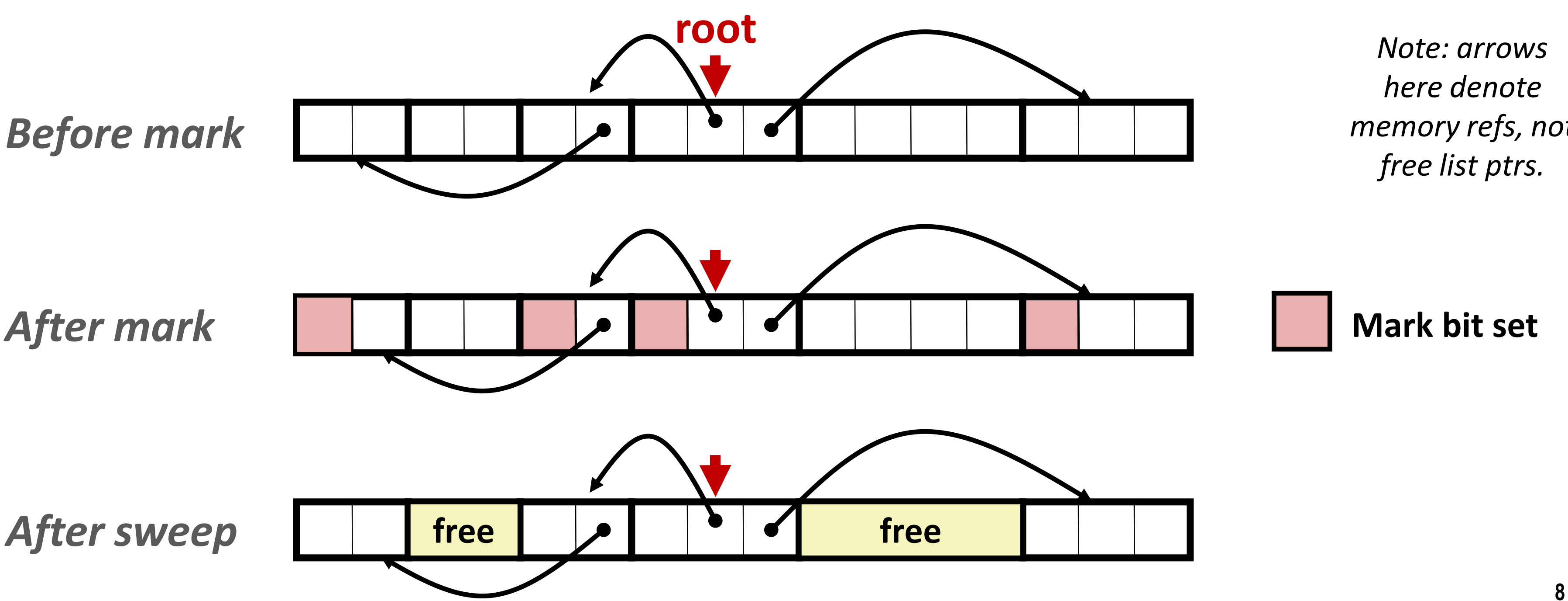
Non-reachable nodes are *garbage* (cannot be needed by the application)

Today

- **Garbage collection**
  - Basic concepts
  - **Mark and Sweep**
  - Reference Counting
  - Copying
  - Treadmill
  - Generational
- **Memory-related perils and pitfalls**

Mark and Sweep Collecting

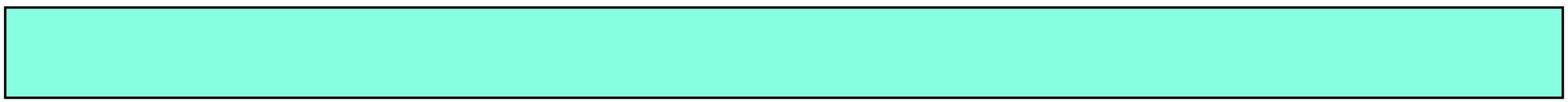
- **Can build on top of malloc/free package**
  - Allocate using `malloc` until you "run out of space"
- **When out of space:**
  - Use extra ***mark bit*** in the head of each block
  - ***Mark:*** Start at roots and set mark bit on each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



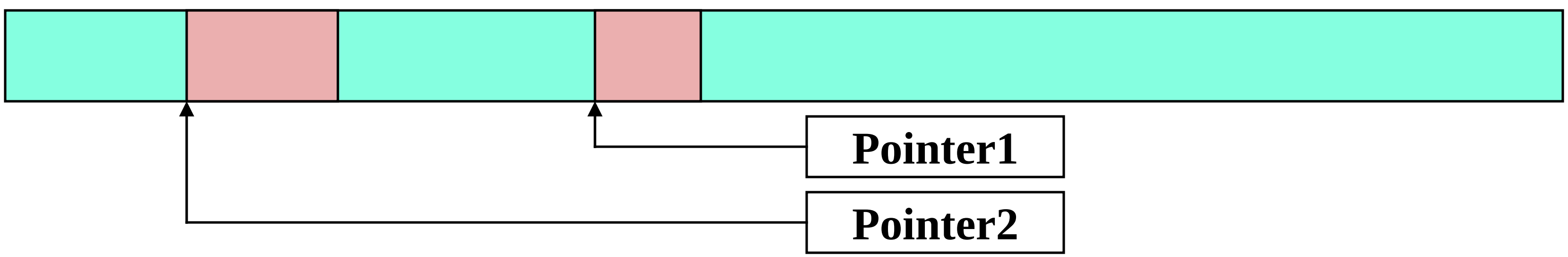
Note: arrows here denote memory refs, not free list ptrs.

Mark-phase

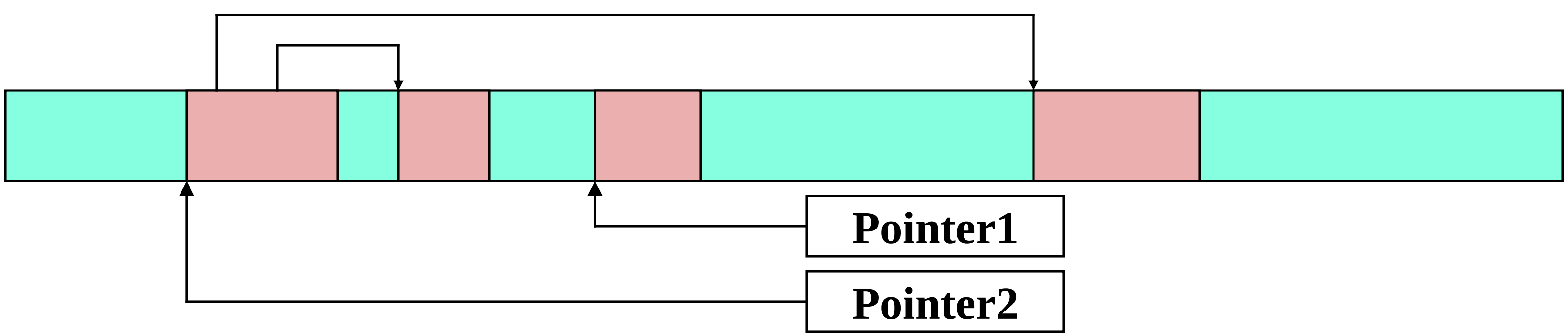
Mark all objects as "unused"



Mark the objects pointed by root set as "used"

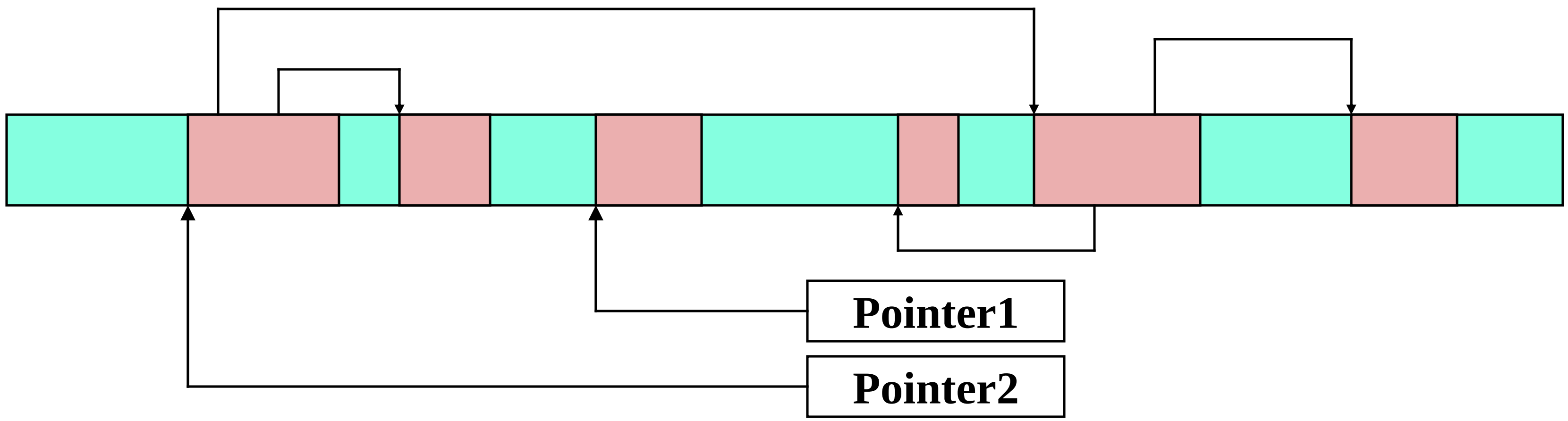


Mark the objects reached by root set as "used"

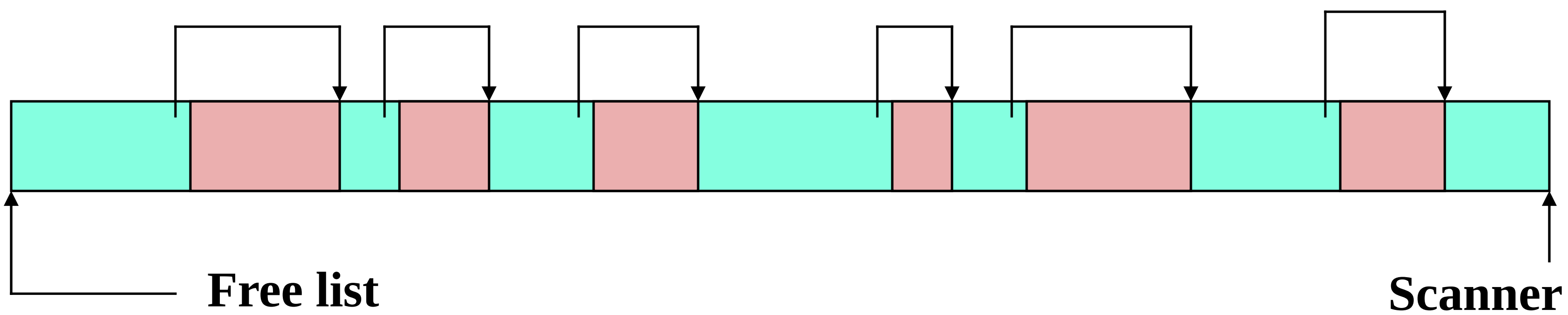


Mark-phase and Sweep-phase

Repeat this marking process until no new objects be marked



- Scan all heap space
- Reclaim the objects marked as “unused”



Assumptions For a Simple Implementation

- **Application**
  - `new(n)`: returns pointer to new block with all locations cleared
  - `read(b,i)`: read location `i` of block `b` into register
  - `write(b,i,v)`: write `v` into location `i` of block `b`

- **Each block will have a header word**
  - addressed as `b[-1]`, for a block `b`
  - Used for different purposes in different collectors

- **Instructions used by the Garbage Collector**
  - `is_ptr(p)`: determines whether `p` is a pointer
  - `length(b)`: returns the length of block `b`, not including the header
  - `get_roots()`: returns all the roots

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

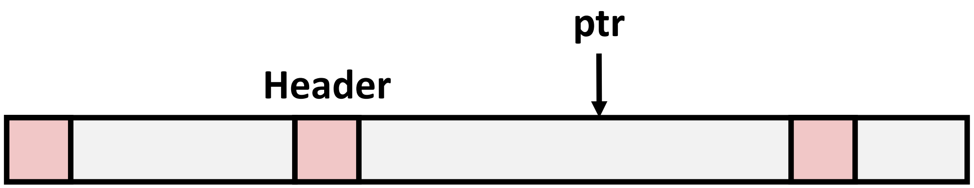
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // call mark on all words
     mark(p[i]);                    //   in the block
   return;
}
```

Sweep using lengths to find next block

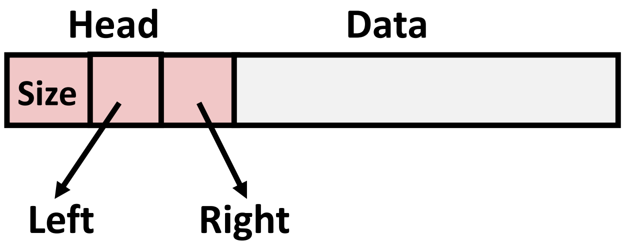
```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Conservative Mark & Sweep in C

- A "conservative garbage collector" for C programs
  - `is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, in C pointers can point to the middle of a block



- So how to find the beginning of the block?
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced-tree pointers can be stored in header (use two additional words)



Left: smaller addresses
Right: larger addresses

Mark-compact & Incremental Mark-sweep

- **Mark-compact**
  - Two phases
    - Mark-phase: just like the mark-sweep collector
    - Compact-phase: move all the live objects to one end and leave all the garbage to the other
  - Eliminate the external fragmentation and improver spatial locality

- **Incremental Mark-sweep**
  - Application processing may continue while garbage collection is taking place

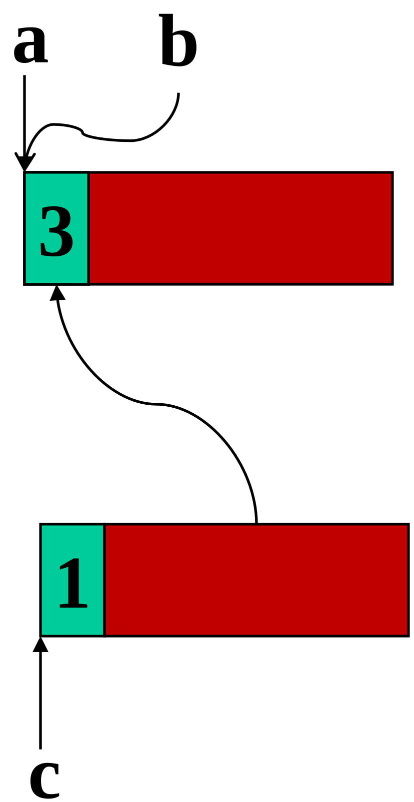
Today

- **Garbage collection**
  - Basic concepts
  - Mark and Sweep
  - **Reference Counting**
  - Copying
  - Treadmill
  - Generational
- **Memory-related perils and pitfalls**

Reference Counting

- **Each object has an associated counter of the references**
  - When the object is first created, the counter is set to one
  - Whenever a reference is copied, the counter is incremented
  - Whenever a reference is overwritten, the counter is decremented
  - When the counter reaches zero, the object becomes garbage

Example of RC



- Create a new object

  `a = new Object();`

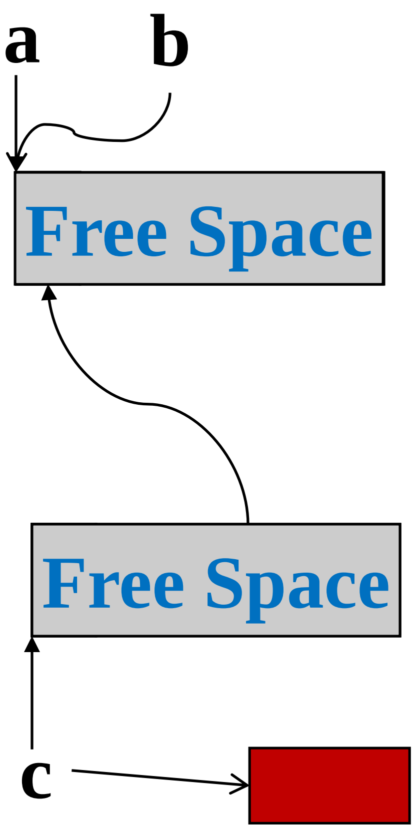
- The counter is incremented when reference is copied

  `b = a;`

- The counter is incremented when another pointer points to the object

  `c.something = a;`

Example of RC (cont.)



- The counter is decremented when one reference is dead

```
{
  Object a, b;
  ……
  a = new Object();
  b = a;
  ……
}   // when program executed to here
```

- The counter is decremented when the pointer points to another object
  `c = some_thing_else;`
- The object will be reclaimed when its counter reaches zero
- The pointers in the reclaimed objects should be taken into account
  - Recursive reclamation

Discussion and Deferred RC

- **Discussion**
  - Incremental nature
  - The problem with cycles
  - Usually used between distributed nodes
- **Deferred Reference Counting**
  - Short-lived objects make considerable cost

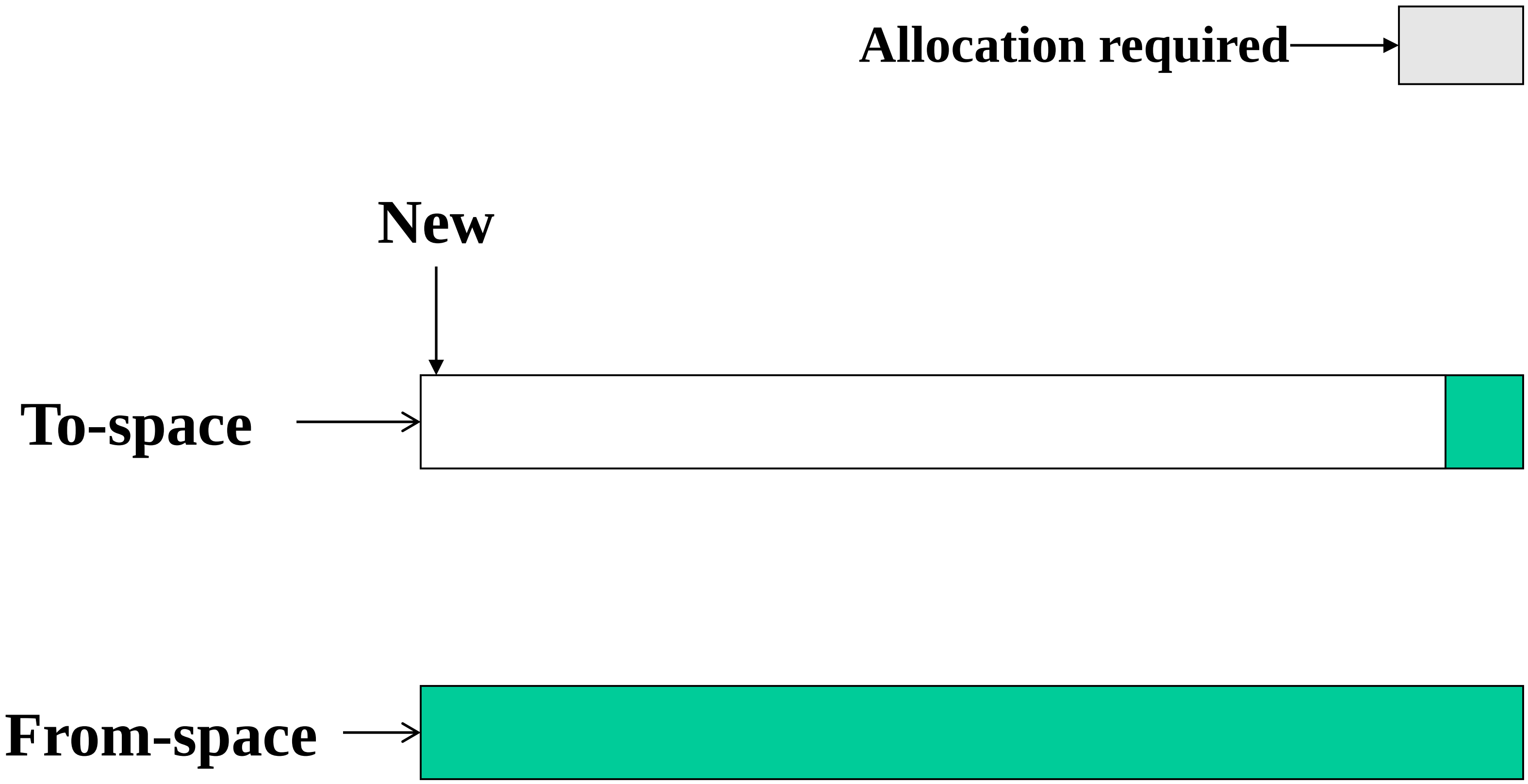
Today

- **Garbage collection**
  - Basic concepts
  - Mark and Sweep
  - Reference Counting
  - **Copying**
  - Treadmill
  - Generational
- **Memory-related perils and pitfalls**

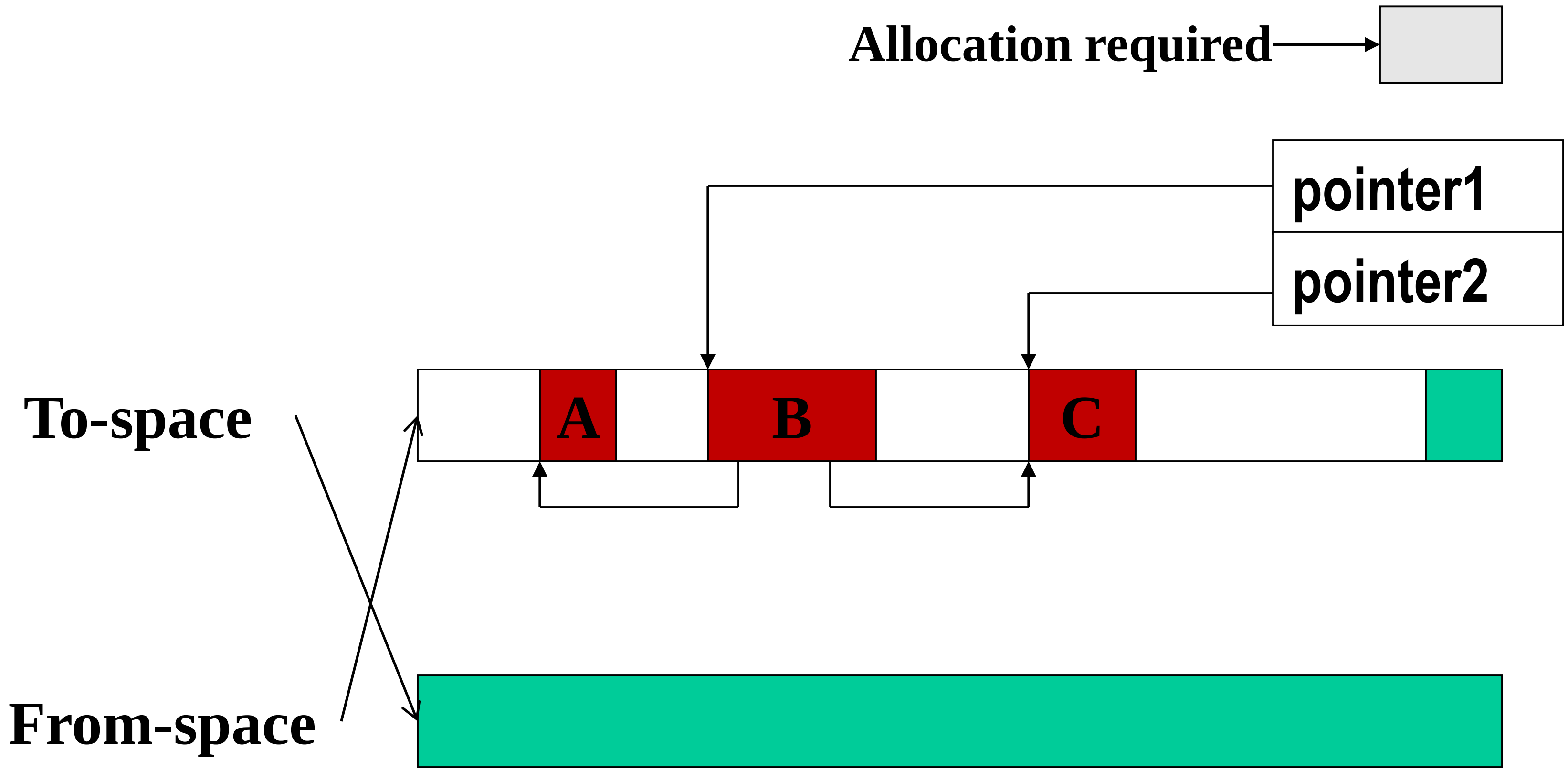
Copying

- **Originally "stop-and-copy" scheme**
- **The heap is divided into two contiguous semi-spaces:**
  - From-space: initially empty
  - To-space: allocation area
- **If the allocation requirement can't be satisfied**
  - Exchange from-space and to-space
  - Copy all reachable objects from from-space into one end of to-space
  - Begin new allocations in new to-space

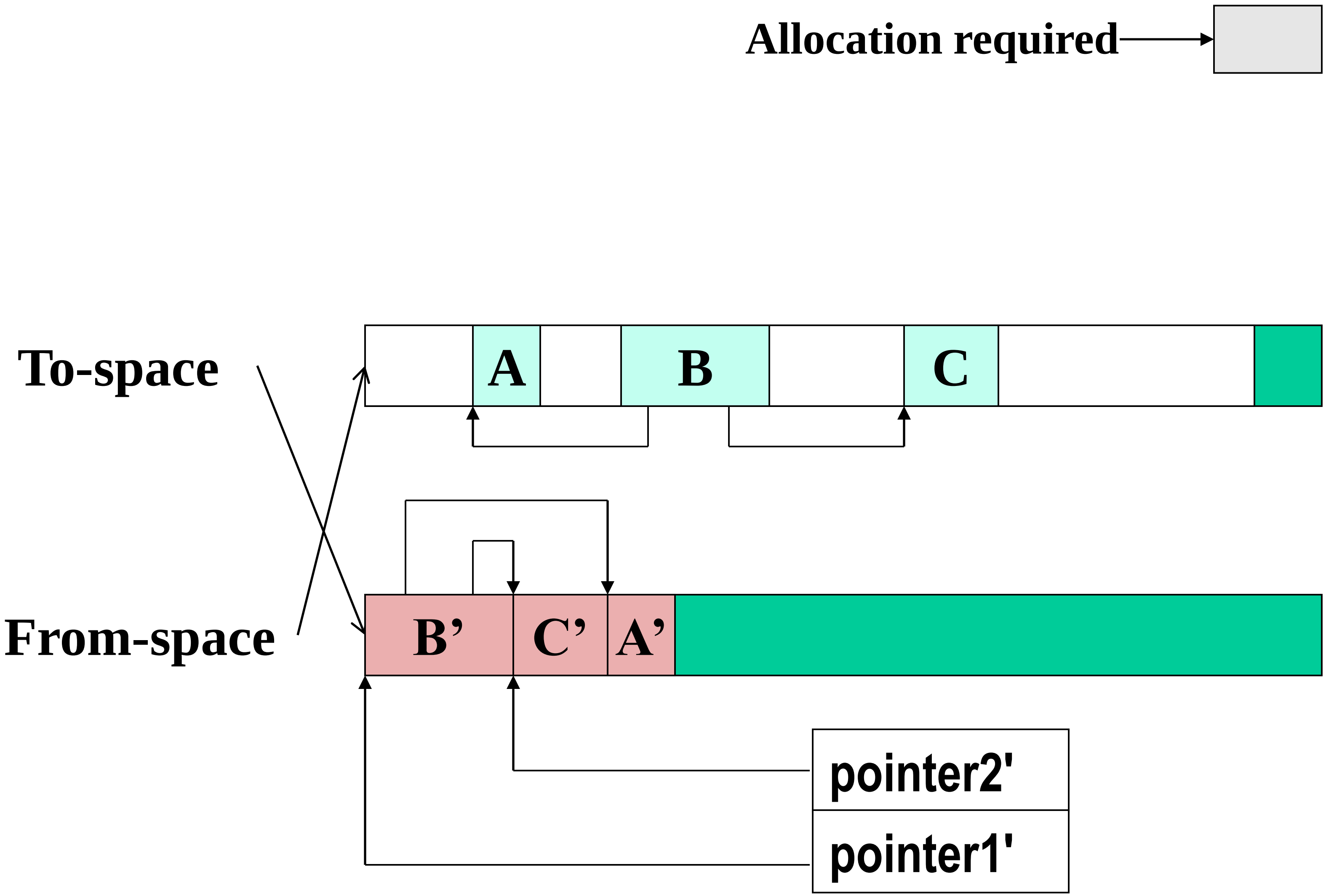
Example of Copying



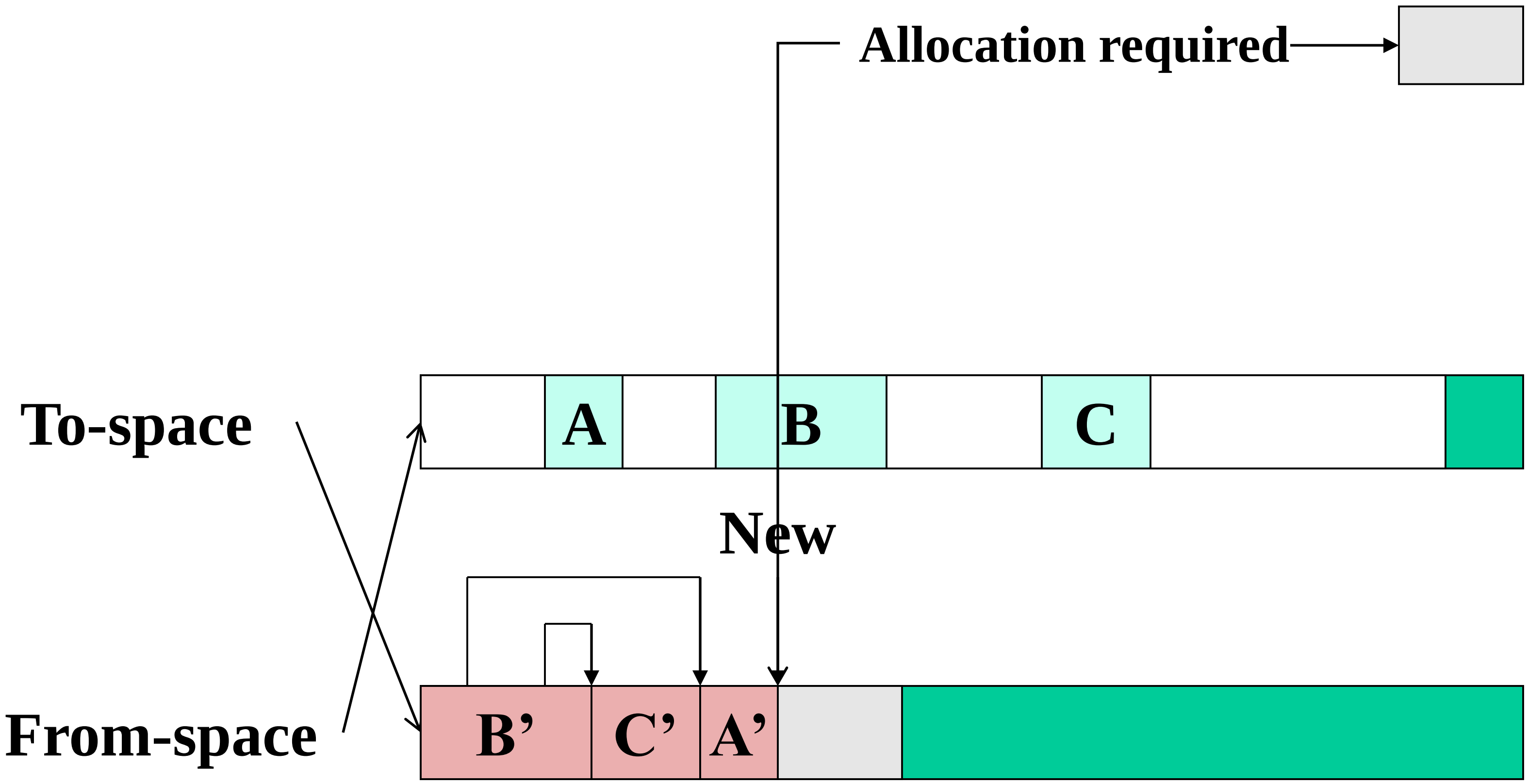
Example of Copying



Example of Copying

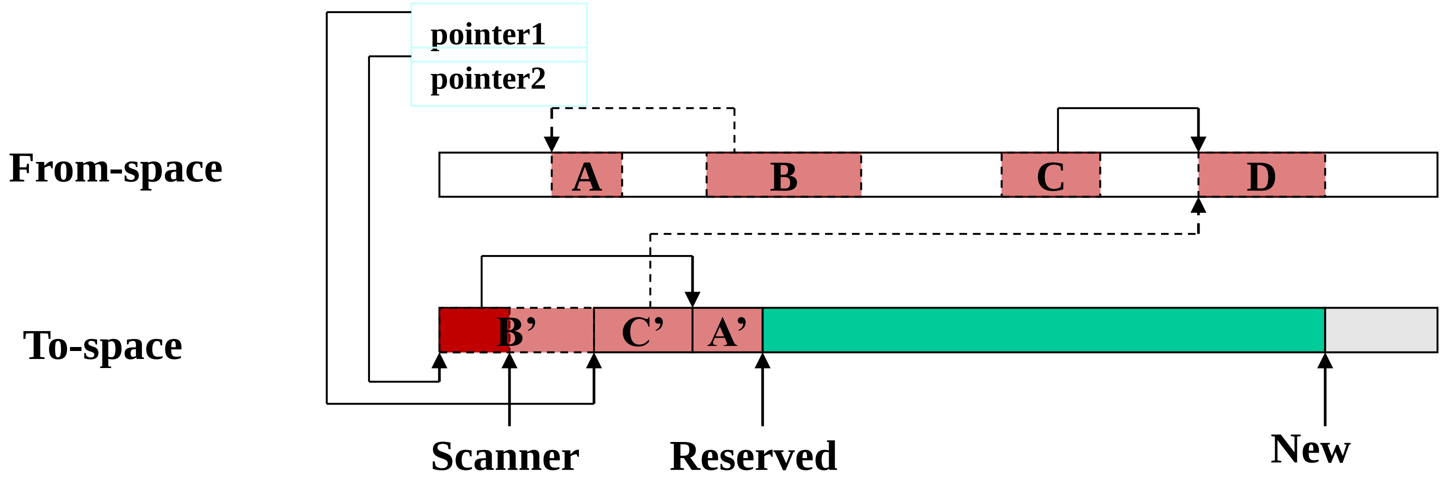


Example of Copying



Discussion and Variants

- **Discussion about Copying Collector**
  - Stop-and-wait method suspends applications
  - Difficult support conservative collection
  - Half used, half wasted
- **Baker's Real-time Copying**
  - Incrementally copying all live objects out of from-space into to-space while new memory is allocated from to-space



Today

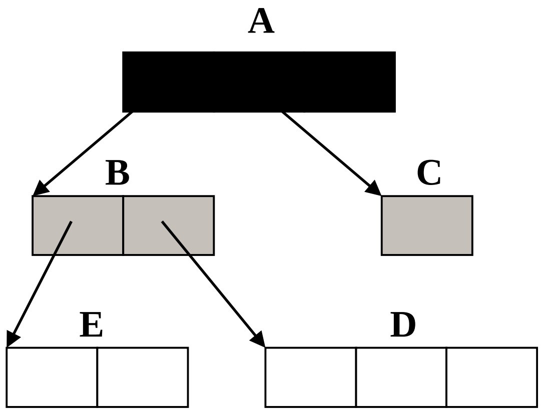
- **Garbage collection**
  - Basic concepts
  - Mark and Sweep
  - Reference Counting
  - Copying
  - **Treadmill**
  - Generational
- **Memory-related perils and pitfalls**

Tricolor Marking

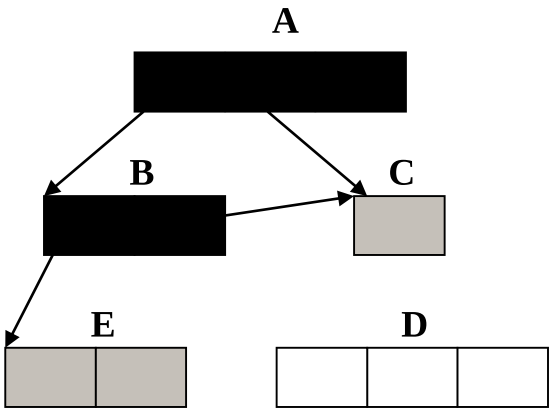
- **Meaning of Incrementality:**
  - Incremental garbage collectors allow application processing to continue while garbage collection is performed
- **Abstraction of incremental GC: Tricolor scheme**
  - White objects -- garbage
  - Black objects -- reachable objects that already processed
  - Gray objects -- reachable objects that haven’t been processed
- **For the incremental reason**
  - the black objects and gray objects may not be reachable objects
  - the white objects may not be garbage

Principle of Tricolor Marking

- **Initially, all objects are assumed to be garbage and are colored white**
  - The purpose: identify all living objects and color them black
- **The traversal phase proceeds in a wave-front of grey objects which separates the white objects from the black ones.**
- **The abstraction of tricolor marking is helpful in understanding incremental tracing phase of garbage collector.**

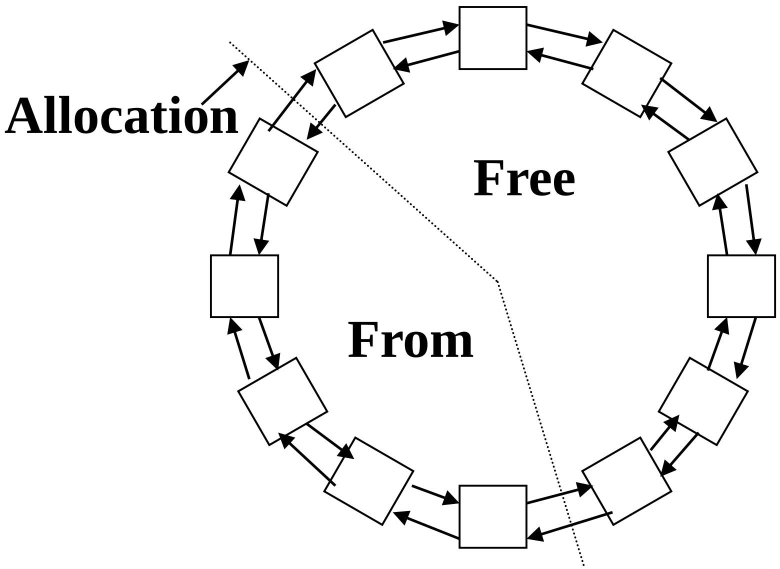


Before

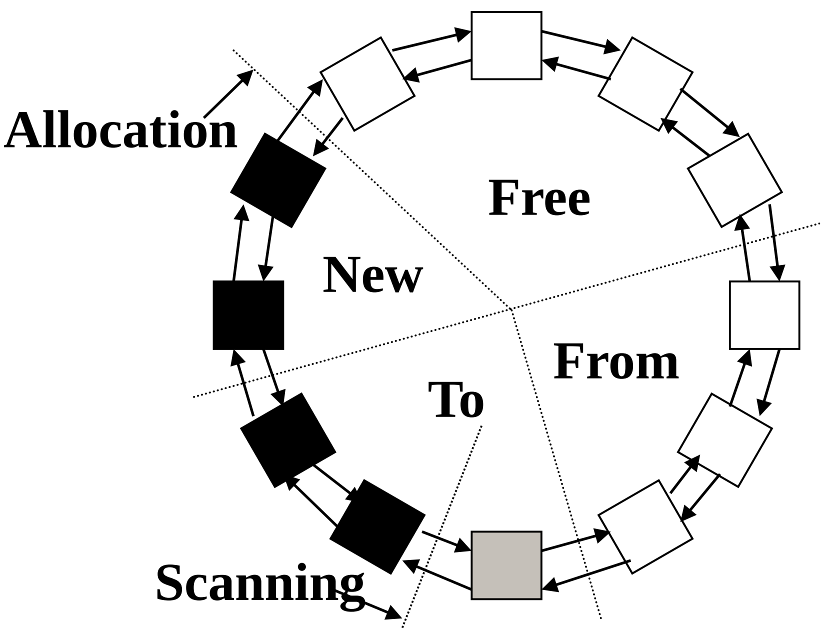


After

The Treadmill

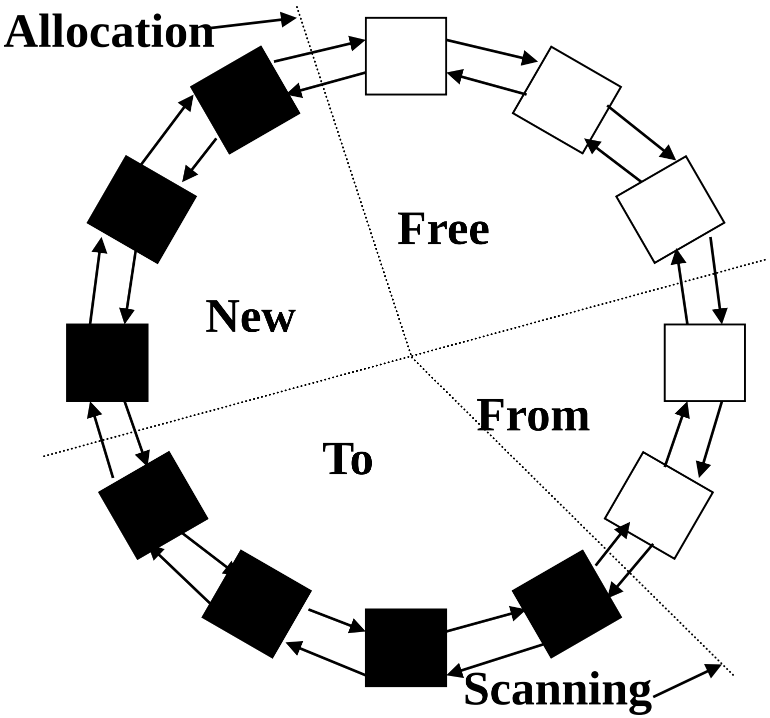


- **A New Scan: All are white**
  - Free-list: free blocks
    - Available for allocation
  - From-list: allocated blocks
    - All objects are white (garbage)
- **During Scanning:**
  - New-list: All black
    - Newly allocated during GC
  - To-list: Black or grey
    - Reachable objects
  - From-list: All white
    - May not be garbage
  - Free-list: All white



The Treadmill

- **When to complete:** ***No grey objects in To-list***
- **All are white again**
  - New-From-list = Old-To-list + Old-New-list
  - New-Free-list = Old-Free-list + Old-From-list
  - New New-list: empty
  - New To-list: empty



Today

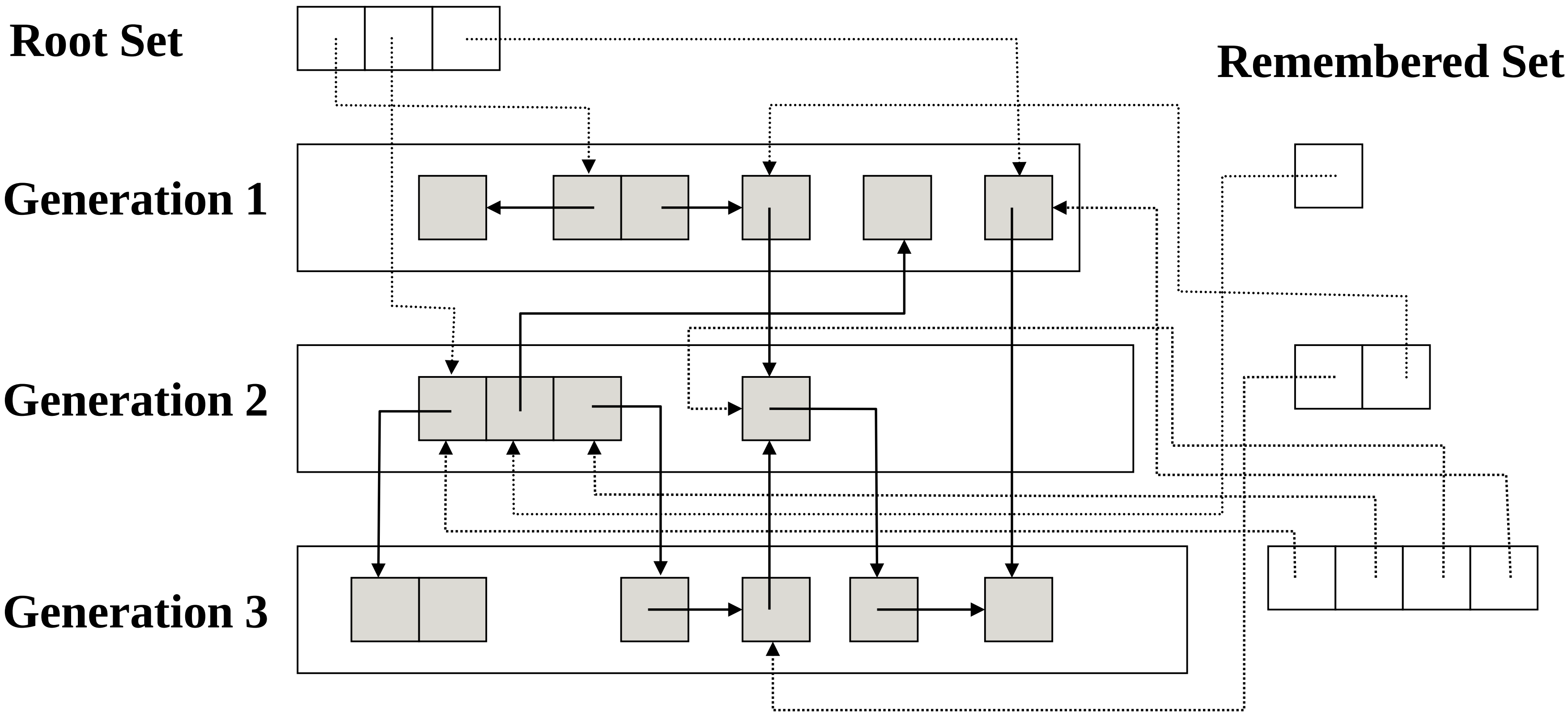
- **Garbage collection**
  - Basic concepts
  - Mark and Sweep
  - Reference Counting
  - Copying
  - Treadmill
  - **Generational**
- **Memory-related perils and pitfalls**

Generational GC

- **Infant mortality**
  - Most objects live a very short time, while a small percentage of them tend to live much longer
  - 80% to 98% of newly-allocated objects die within a few million instructions
- **Generations**
  - Each holding objects of different age categories
    - New objects are allocated in the youngest generation
    - Whenever an object has lived for some period of time, it is moved to the next generation
  - Each designed to be collected individually

Example of Generational GC

- **Inter-generational references**
  - *Remembered set:* containing the identity of all references into the generation from heap objects residing in other generations
  - *Directionality of reference*: Newer objects tend to point to Older objects



Train Algorithm

- **An incremental garbage collection scheme for achieving non-disruptive reclamation of the oldest generational area, the mature object space.**
- **The algorithm achieves its incrementality by dividing mature object space into a number of fixed-sized blocks and collecting one block at each invocation.**
  - The blocks are referred as cars
  - The set of blocks to which a car belongs as its train
  - Mature object space can then be thought of as a giant railway station with trains lined up on its tracks

Today

- **Garbage collection**
  - Basic concepts
  - Mark and Sweep
  - Reference Counting
  - Copying
  - Treadmill
  - Generational
- **Memory-related perils and pitfalls**

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

C operators

Operators	*Associativity*
`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= != <<= >>=`	right to left
`,`	left to right

- **`->`, `()`, and `[]` have high precedence, with `*` and `&` just below**
- **Unary `+`, `-`, and `*` have higher precedence than binary forms**

Source: K&R page 53

C Pointer Declarations: Test Yourself!

`int *p`	p is a pointer to int
`int *p[13]`	p is an array[13] of pointer to int
`int *(p[13])`	p is an array[13] of pointer to int
`int **p`	p is a pointer to a pointer to an int
`int (*p)[13]`	p is a pointer to an array[13] of int
`int *f()`	f is a function returning a pointer to int
`int (*f)()`	f is a pointer to a function returning int
`int (*(*f())[13])()`	f is a function returning ptr to an array[13] of pointers to functions returning int
`int (*(*x[3])())[5]`	x is an array[3] of pointers to functions returning pointers to array[5] of ints

Source: K&R Sec 5.12

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;

...

scanf(“%d”, val);
```

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

Overwriting Memory

- **Allocating the (possibly) wrong sized object**

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Overwriting Memory

- Off-by-one error

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

Overwriting Memory

- **Not checking the max string size**

```
char s[8];
int i;

gets(s);  /* reads “123456789” from stdin */
```

- **Basis for classic buffer overflow attacks**

Overwriting Memory

- **Misunderstanding pointer arithmetic**

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

Overwriting Memory

- **Referencing a pointer instead of the object it points to**

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Referencing Nonexistent Variables

- **Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```

Freeing Blocks Multiple Times

- **Nasty!**

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

Referencing Freed Blocks

- **Evil!**

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
   ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
     ...
   free(head);
   return;
}
```

Dealing With Memory Bugs

- **Debugger: `gdb`**
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- **Data structure consistency checker**
  - Runs silently, prints message only on error
  - Use as a probe to zero in on error
- **Binary translator: `valgrind`**
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block
- **glibc malloc contains checking code**
  - **`setenv MALLOC_CHECK_ 3`**