Dynamic Memory Allocation: Advanced Concepts

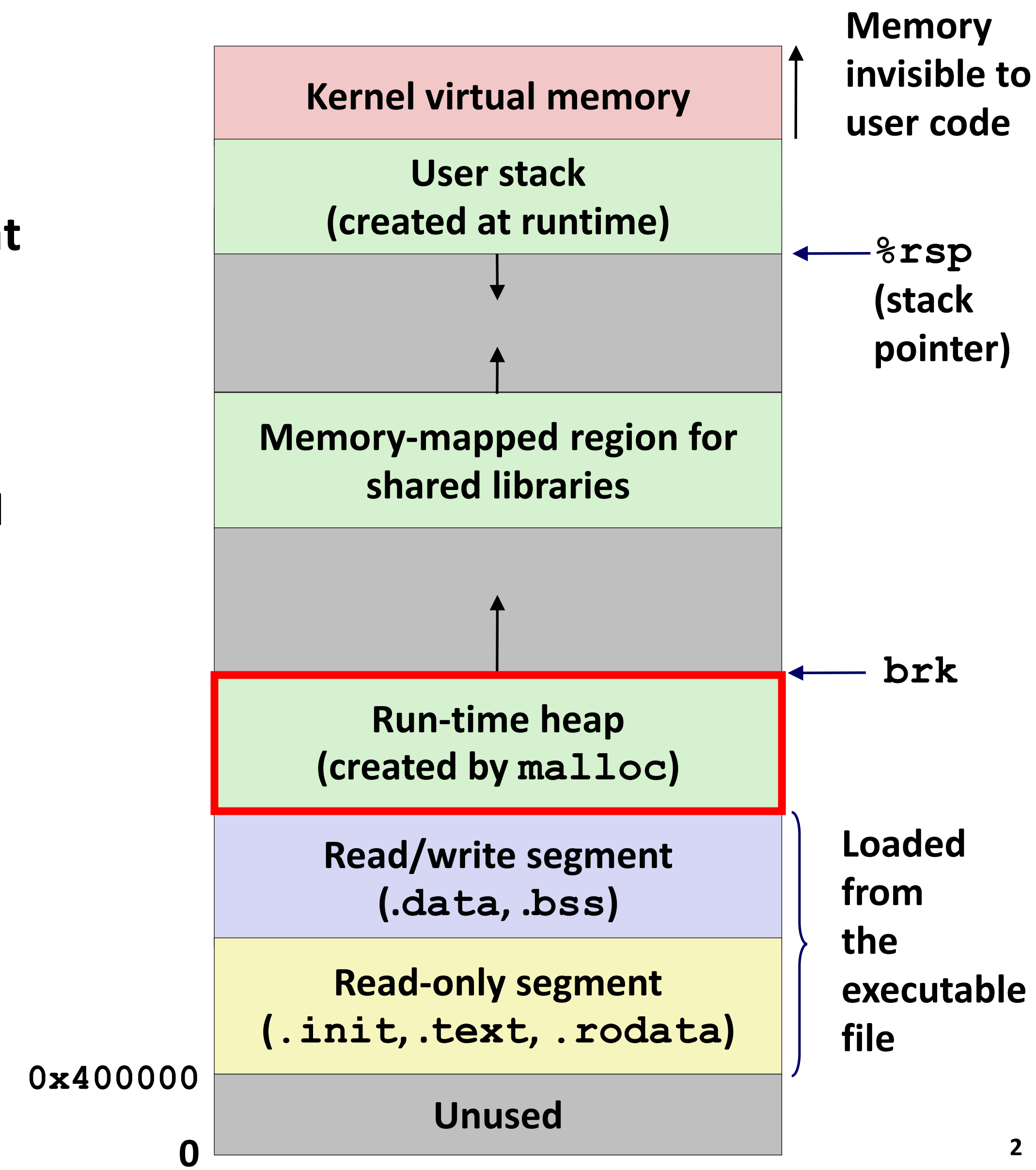
Textbook coverage:
Ch 9.10: Garbage collection
Ch 9.11: Common memory-related bugs in C programs

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors.
GC slides are based on materials of Alan Cox at Rice University.

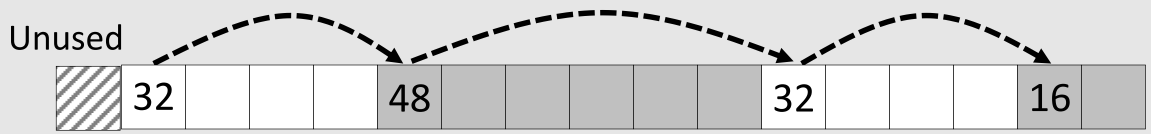
Review: Dynamic Memory Allocation

- Programmers use ***dynamic memory allocators*** (such as `malloc`) to acquire memory at runtime
- Dynamic memory allocators manage an area of process VM known as the ***heap***



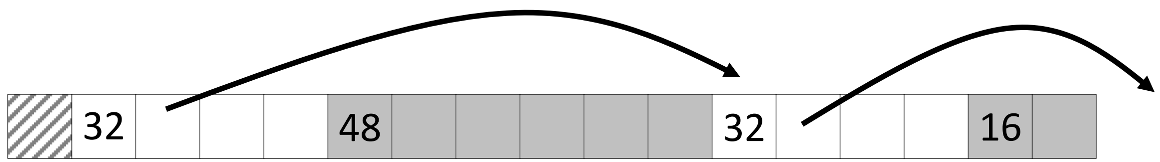
Review: Keeping Track of Free Blocks

- **Method 1:** ***Implicit list*** **using length—links all blocks**



Need to tag each block as allocated/free

- **Method 2:** ***Explicit list*** **among the free blocks using pointers**



Need space for pointers

- **Method 3:** ***Segregated free list***
  - Different free lists for different size classes

- **Method 4:** ***Blocks sorted by size***
  - Can use a balanced tree (e.g., Red-Black tree) with pointers within each free block, and the length used as a key

Review: Implicit Lists Summary

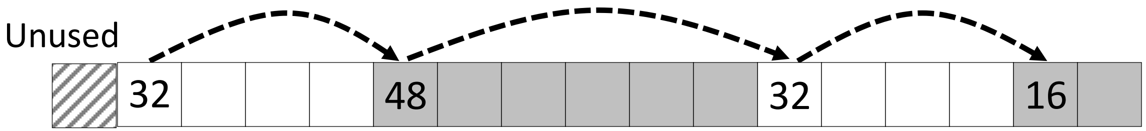
- **Implementation: very simple**
- **Allocate cost:**
  - linear time worst case
- **Free cost:**
  - constant time worst case
  - even with coalescing
- **Memory Overhead:**
  - Depends on placement policy
  - Strategies include first fit, next fit, and best fit
- **Not used in practice for `malloc/free` because of linear-time allocation**
- **However, the concepts of splitting and coalescing are general to *all* allocators**

Today

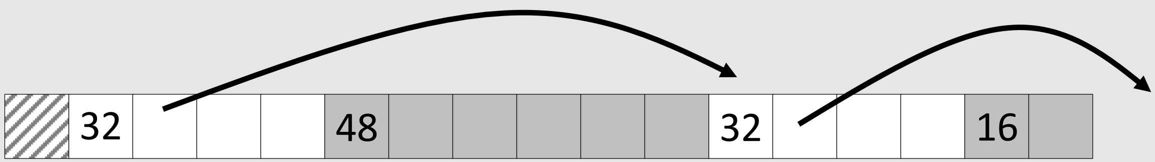
- **Explicit free lists**
- Segregated free lists
- Garbage collection
- Security issues related to dynamic memory

Keeping Track of Free Blocks

- **Method 1: *Implicit list* using length—links all blocks**



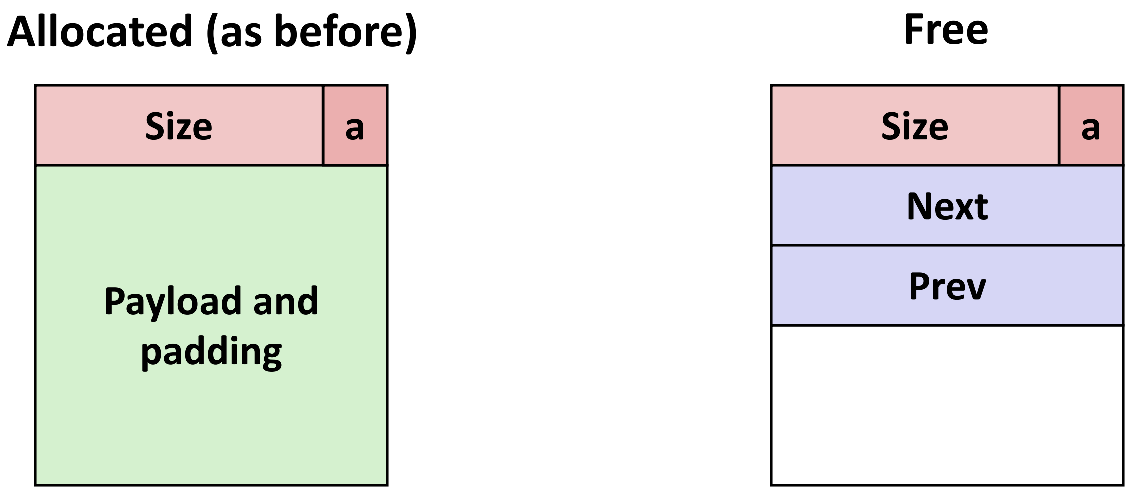
- **Method 2: *Explicit list* among the free blocks using pointers**



- **Method 3: *Segregated free list***
  - Different free lists for different size classes

- **Method 4: *Blocks sorted by size***
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Explicit Free Lists



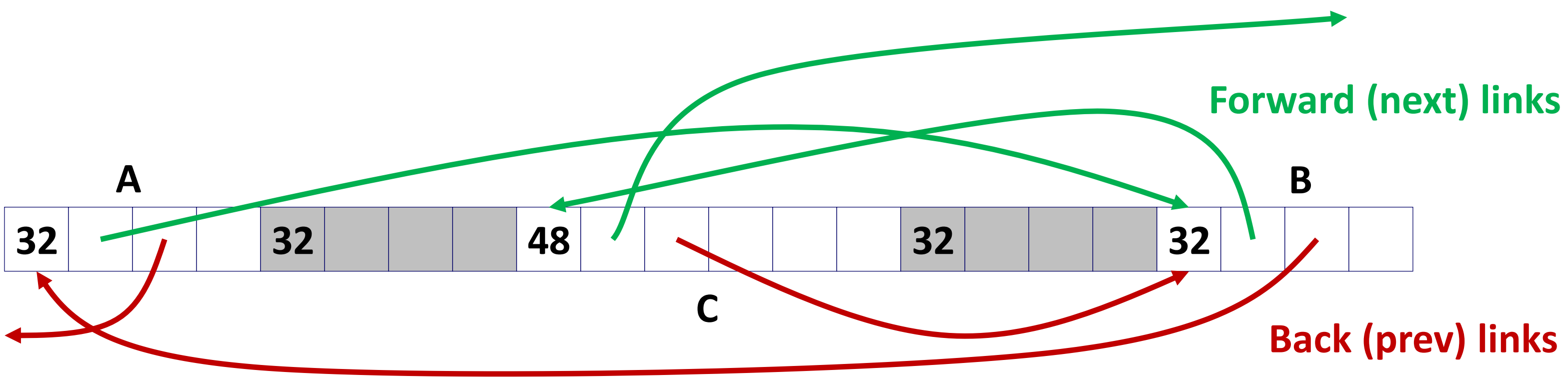
- **Maintain list(s) of *free* blocks, not *all* blocks**
  - Luckily we track only free blocks, so we can use payload area

    ➔ Reduce internal fragmentation
  - Store forward/back pointers
    - A pointer points to a free block, allowing to only traverse free blocks

Explicit Free Lists

- **Logically:**

- **Physically: blocks can be in any order**



Freeing With Explicit Free Lists

- ***Policy: inserting a freed block back to the free list***
  - Where should you put a newly freed block in the free list?

- **List-friendly policy**
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - FIFO (first-in-first-out) policy
    - Insert freed block at the end of the free list

- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    addr(prev) < addr(curr) < addr(next)

Explicit List Summary

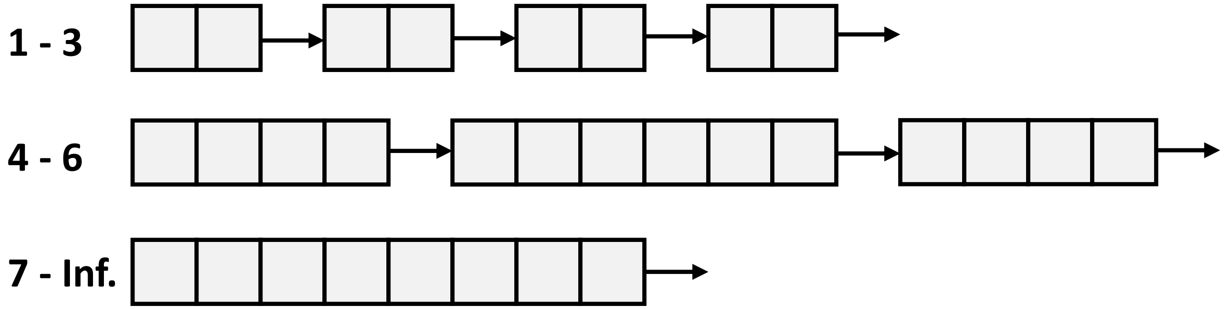
- **Comparison to implicit list:**
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free
    - because need to splice blocks in and out of the list

Today

- Explicit free lists
- **Segregated free lists**
- Garbage collection
- Security issues related to dynamic memory

Segregated List (Seglist) Allocators

- Each ***size class*** of blocks has its own free list



- Often have separate classes for each small size

Seglist Allocator

- **Given an array of free lists, each one for some size class**

- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size $m > n$ (i.e., first fit)
  - If an appropriate block is found:
    - Split block and then allocate
      - Insert the free-fragment in the appropriate free list
  - If no block is found, try next larger class

- **If no block is found in the end:**
  - Request additional heap memory from OS (using **`sbrk()`**)
  - Allocate block of *n* bytes from this new memory
  - Insert remainder as a single free block in appropriate size class.

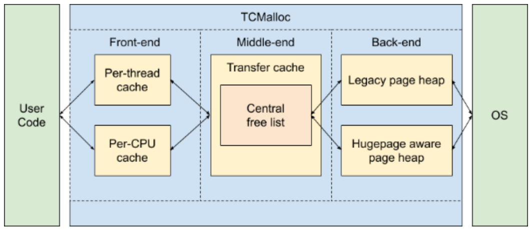
Seglist Allocator (cont.)

- **To free a block:**
  - Coalesce and place on appropriate list

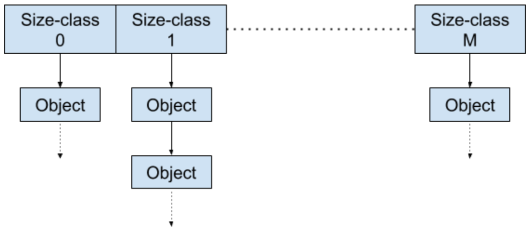
- **Advantages of seglist allocators vs. non-seglist allocators (both with first-fit)**
  - Higher throughput
    - Constant-time vs. Linear time
  - Better memory utilization
    - Seglist allocator (first-fit) avoids unnecessary splits
    - First-fit search of segregated free list approximates a best-fit search of entire heap.

Memory Allocators in Real-world

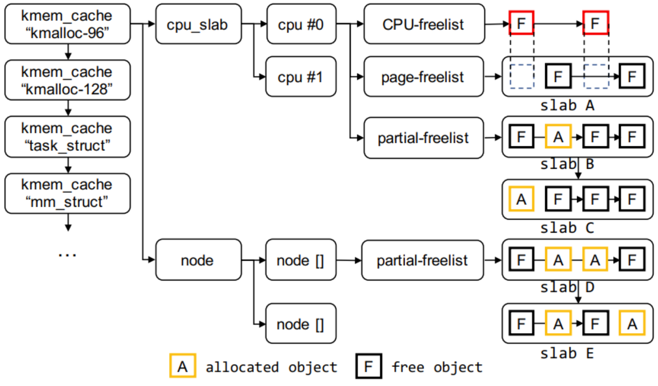
- Real-world allocator designs are more complex than you may imagine!



- tcmalloc (by Google)
  - https://google.github.io/tcmalloc/design.html



- SLUB allocator (Linux Kernel)
  - https://compsec.snu.ac.kr/buildtex/pspray-e807cf40.pdf



More Info on Allocators

- **D. Knuth, *The Art of Computer Programming,* vol 1, 3rd edition, Addison Wesley, 1997**
  - The classic reference on dynamic storage allocation
- **Wilson et al, "*Dynamic Storage Allocation: A Survey and Critical Review*", Proc. 1995 Int'l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.**
  - Comprehensive survey

Today

- Explicit free lists
- Segregated free lists
- **Garbage collection**
- Security issues related to dynamic memory

Explicit Memory Allocation/Deallocation

- **Explicit Memory Allocation/Deallocation**
  - + Usually low time- and space-overhead
  - - Challenging for developers to use correctly
    - e.g., Lead to crashes, memory leaks, etc.

Implicit Memory Deallocation

- **Implicit Memory Deallocation!**
  - + Easy to use
    - Programmers don’t need to free data explicitly
  - - Price to pay
    - Depends on implementation
- **Q. HOW could a memory manager know**
  when to deallocate data
  without instruction from programmer?

Implicit Memory Management: Garbage Collection

- **Garbage collection**
  - Automatic reclamation of heap-allocated storage
  - Applications never have to free

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

- **Common in functional languages and modern object oriented languages:**
  - C#, Go, Java, Lisp, Python, Scala, Swift

- **Variants: Conservative garbage collectors**
  - Cannot collect all garbage
  - e.g., V8 JavaScript engine (Chrome)

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
  - But, we can tell that certain blocks cannot be used
    - if there are no pointers to them

- **Need to 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 casting them to an int, and then back again

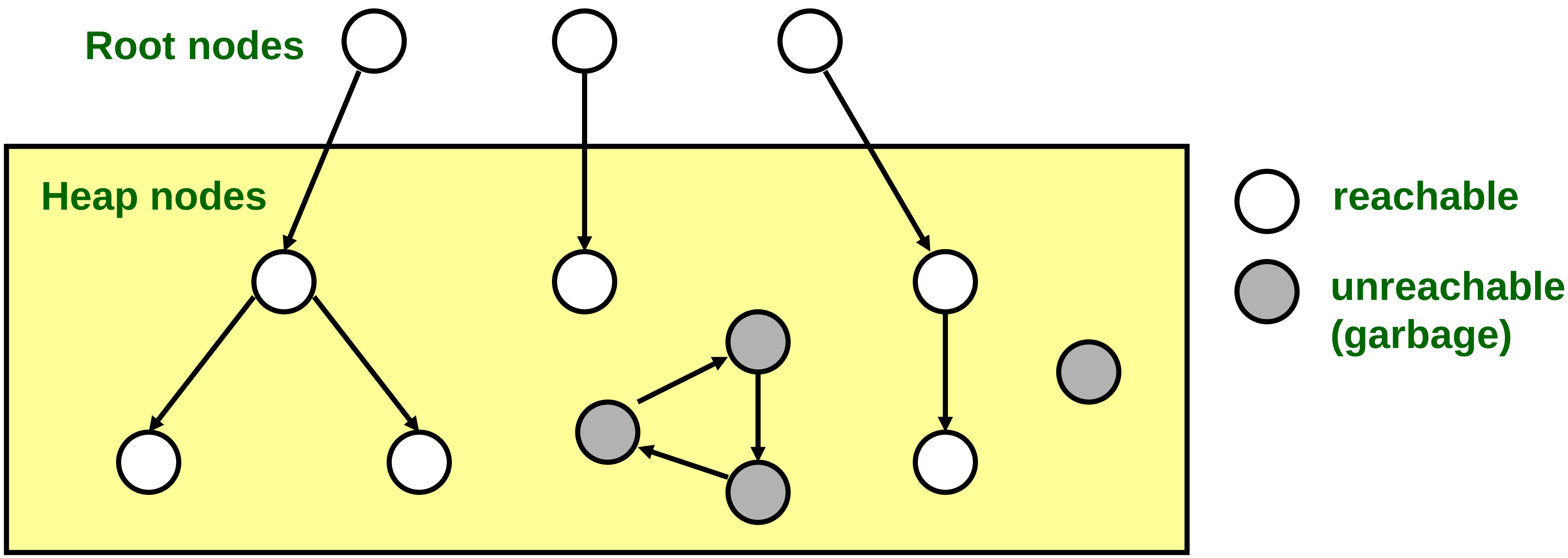
Classical GC algorithms

- Reference counting (Collins, 1960)
- Mark and sweep collection (McCarthy, 1960)

- For more information, see Jones and Lin, “Garbage Collection: Algorithms for Automatic Dynamic Memory”, John Wiley & Sons, 1996.

Memory as a Graph

- **Node**: Each data block is a node in the graph
- **Edge**: Each pointer is an edge in the graph
- **Root nodes**: locations not in the heap that contain pointers into the heap
  - You never know the heap address at the program loading time
  - So your initial reference should begin with the pointers in non-heap space
    - e.g., registers, stack variables, global variables

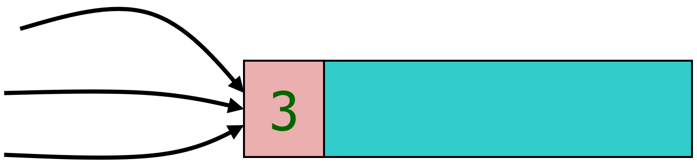


Reference Counting

- **Overall idea**
  - Maintain a free list of unallocated blocks
  - Maintain a count of the number of references in each allocated block
  - To allocate, grab a sufficiently large block from the free list
  - When a count goes to zero, deallocate it

Reference Counting: More Details

- **Each allocated block keeps a count of references to the block**
  - Reachable → count is positive
  - Compiler inserts counter increments and decrements as necessary
  - Deallocate when count goes to zero



- **Typically built on top of an explicit deallocation memory manager**
  - All the same implementation decisions as before
  - E.g., splitting (during allocation) & coalescing (during free)

Reference Counting: Example

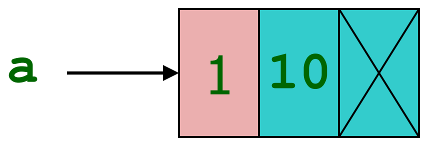
```
struct node {
  int value;
  struct node *next;
};
typedef struct node node_t;

node_t *gen_node(int v, node_t *next) {
  node_t *p = malloc(sizeof(node_t));
  p->value = v;
  p->next = next;
  return p;
}
```

```
node_t *a = gen_node(10,NULL);
node_t *b = gen_node(20,a);
a = b;
b = …
a = …
```

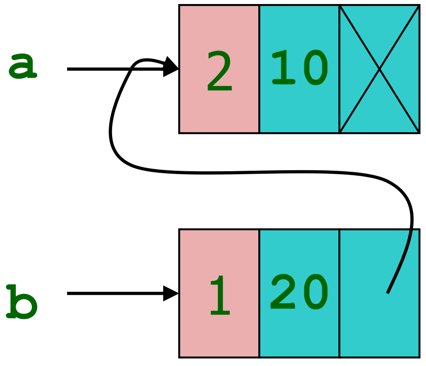
Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```



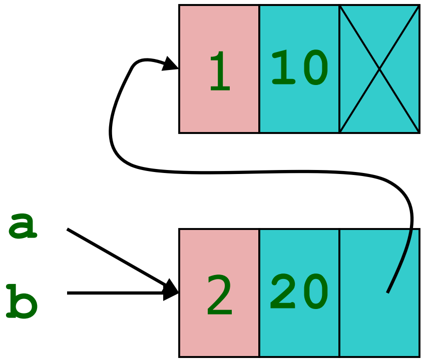
Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```



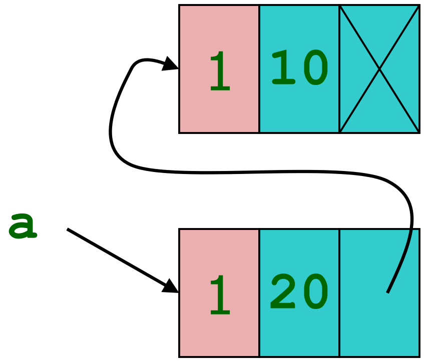
Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```



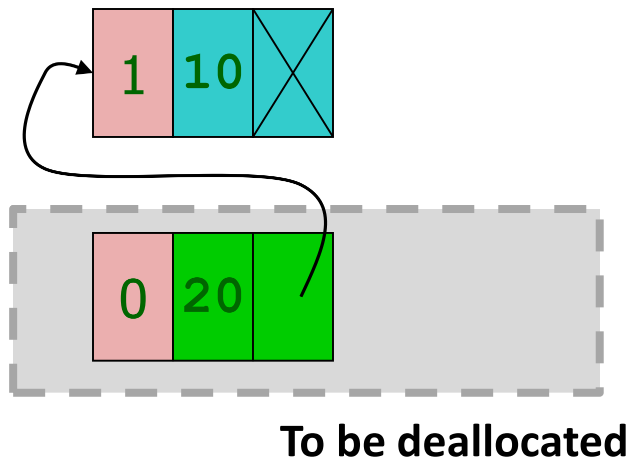
Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```



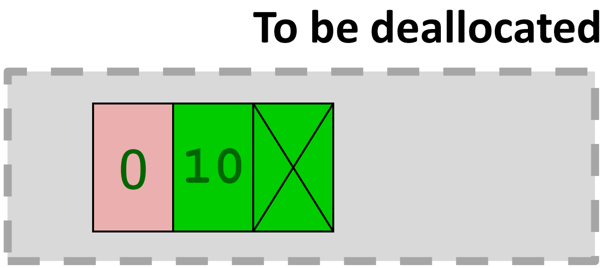
Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```



Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```



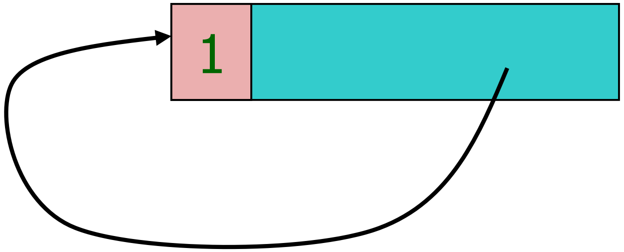
Reference Counting: Example

```
node_t *a = gen_node(10,NULL)
node_t *b = gen_node(20,a)
a = b
b = …
a = …
```

Good.
All deallocated in the end!

Reference Counting: Problem

- What's the problem?



No other pointer to this data, so can't be referenced.

But count is not zero, so never deallocated

Following does NOT hold:

Count is positive → reachable

Can occur with any cycle

Reference Counting: Summary

- **Disadvantages:**
  - Managing & testing counts is generally expensive
    - Can optimize
  - Doesn’t work with cycles!
    - Approach can be modified to work, with difficulty
    - All web browsers including Chrome/Firefox heavily rely on the reference counting with C++ (or smart pointers)

- **Advantage:**
  - Simple
    - Easily adapted, e.g., for parallel or distributed GC

GC Without Reference Counts

- **If don’t have counts, how to deallocate?**

- **Determine reachability by traversing pointer graph directly**
  - Stop user’s computation (stop the world) periodically to compute reachability
  - Deallocate anything unreachable

Mark & Sweep

- **Overall idea**
  - Maintain a free list of unallocated blocks
  - To allocate, grab a sufficiently large block from free list
  - When no such block exists, GC
    - Should find blocks & put them on free list

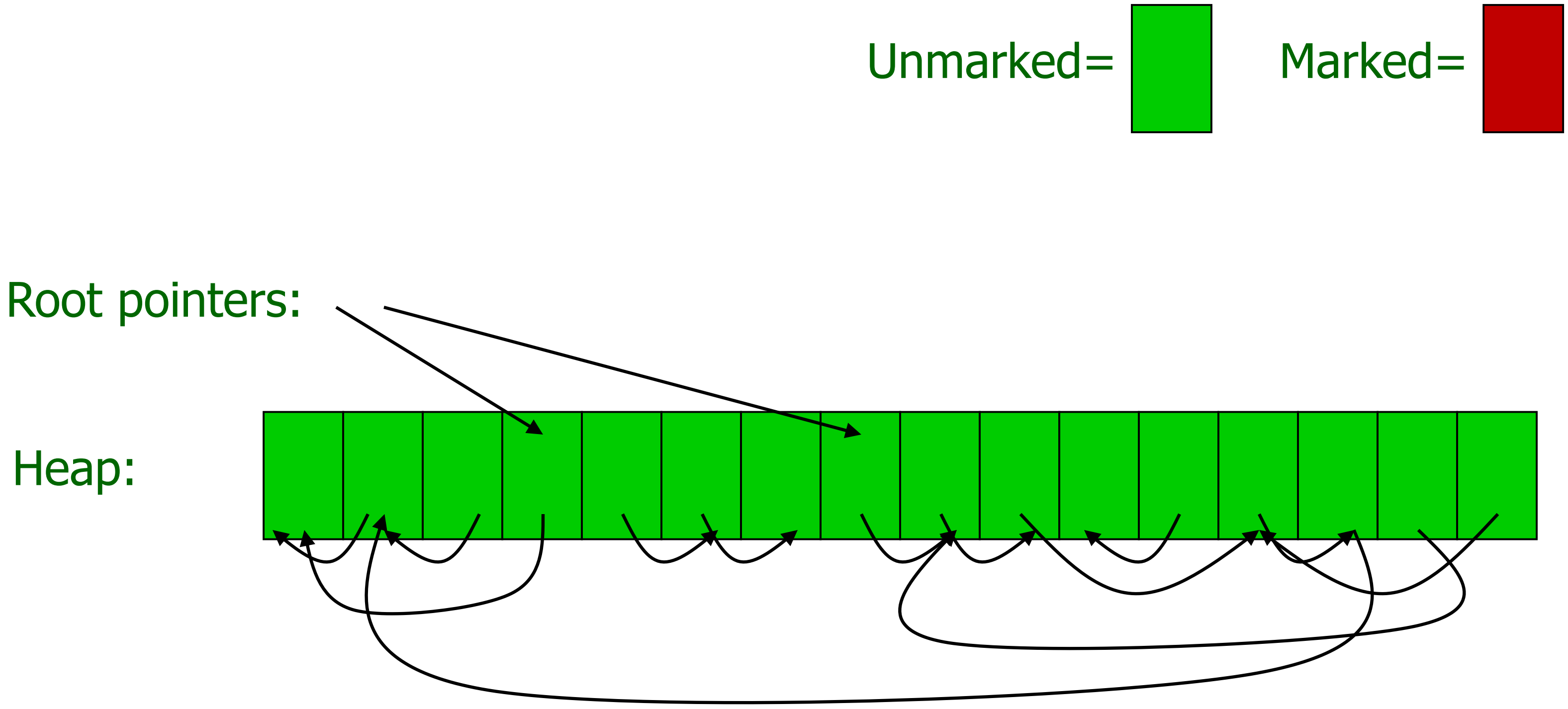
Mark & Sweep: GC

- **Follow all pointers, marking all reachable data**
  - Use depth-first search (or breadth-first search)
  - Data must be tagged with its type information, so GC knows its size and can identify pointers
    - So you shouldn’t have integer-pointer casting.
  - Each piece of data must have a mark bit

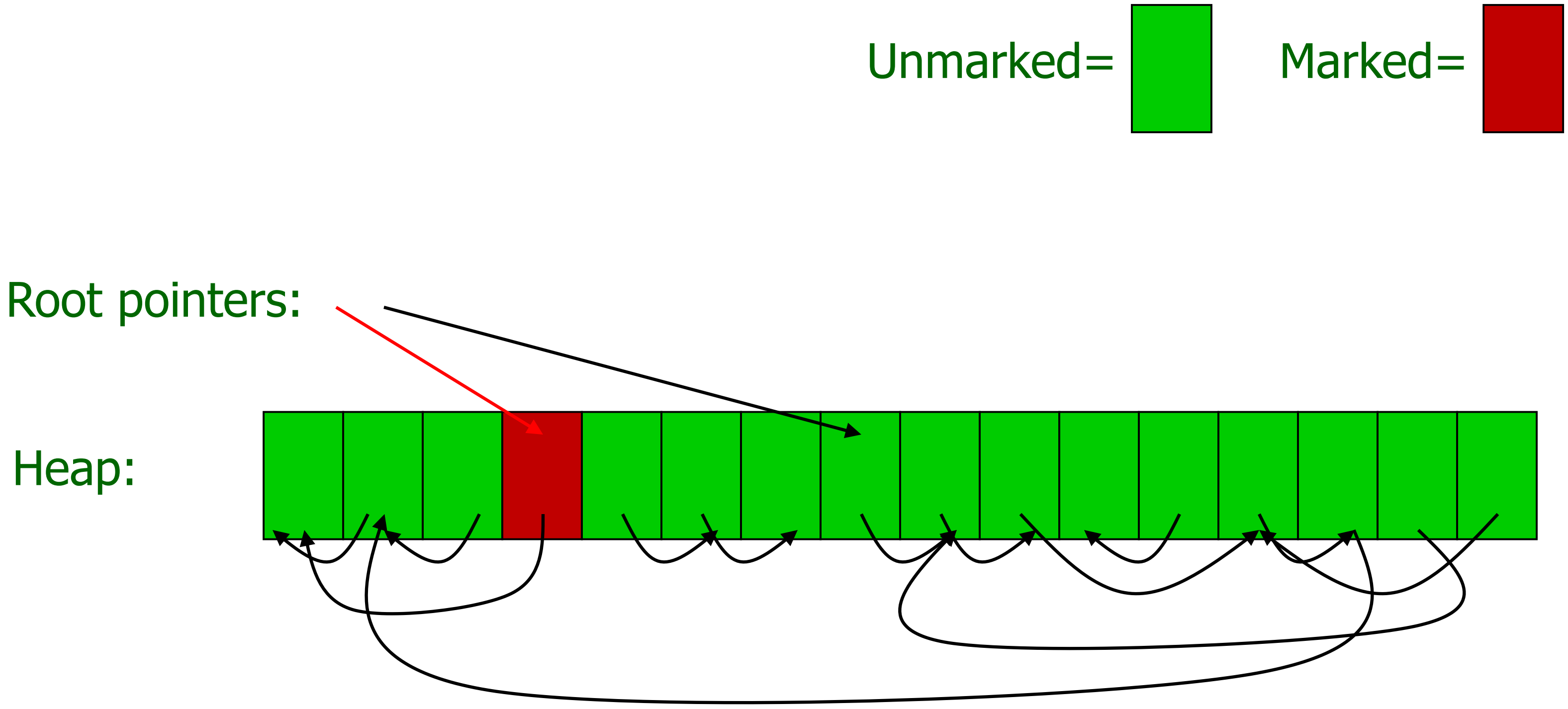
- **Sweep over all heap, putting all unmarked data into a free list**

Mark & Sweep: GC Example

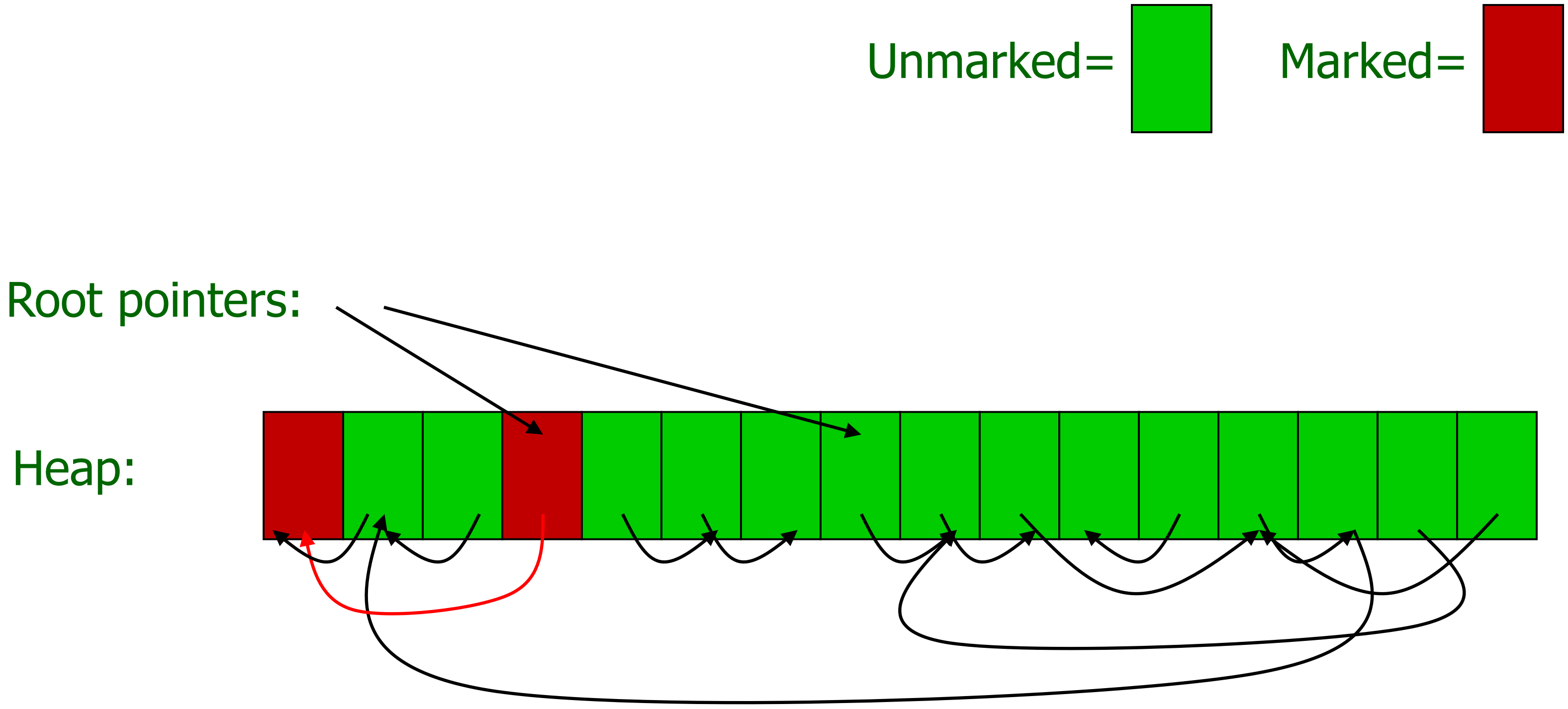
Assume fixed-sized, single-pointer data blocks, for simplicity.



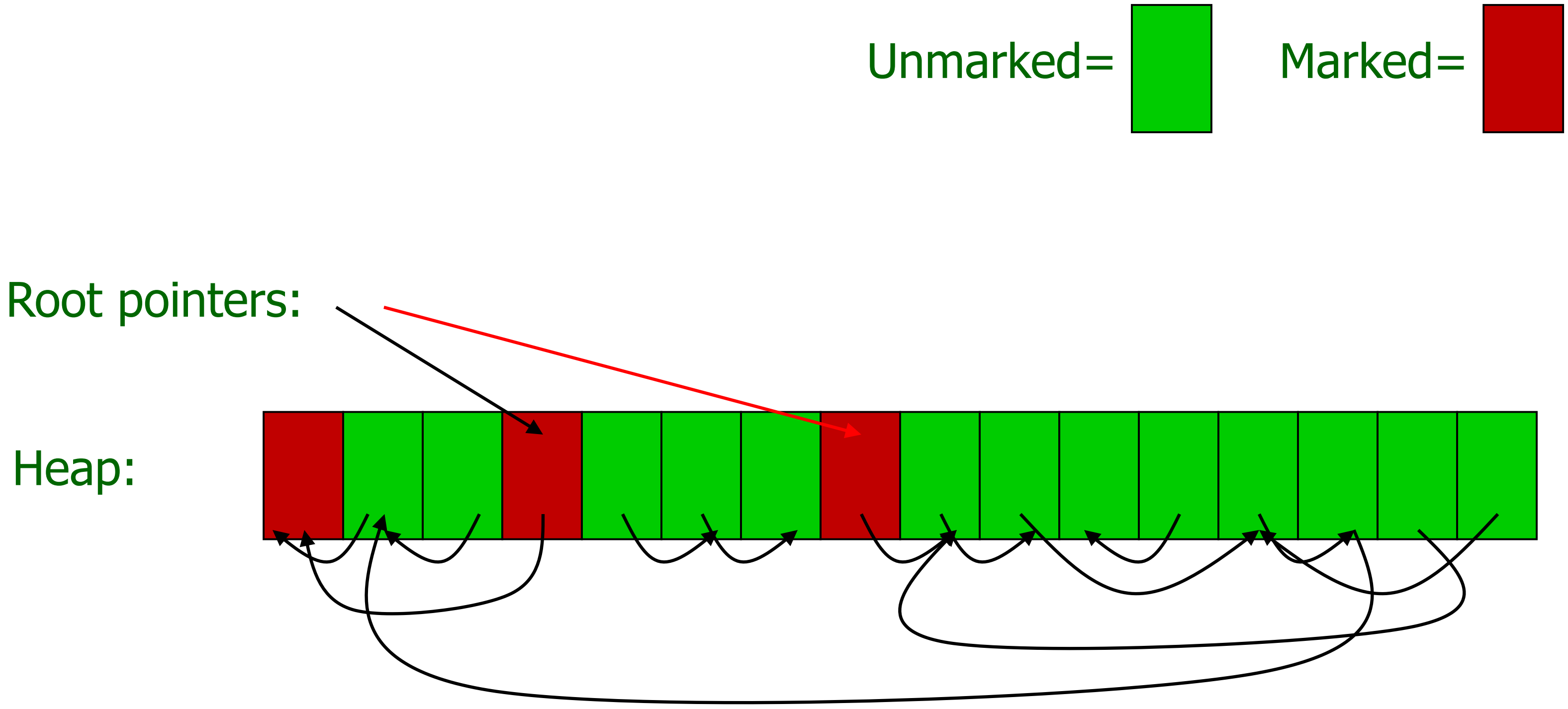
Mark & Sweep: GC Example



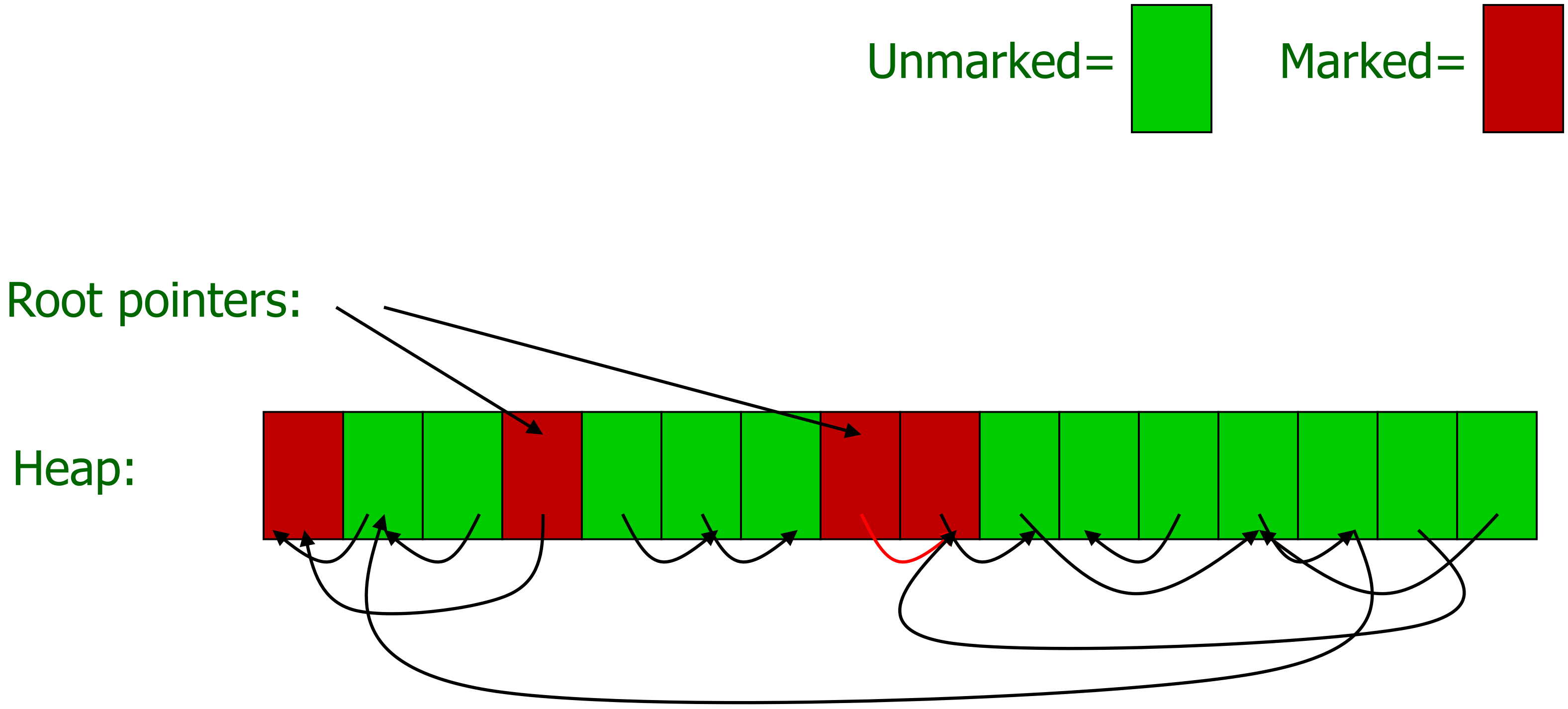
Mark & Sweep: GC Example



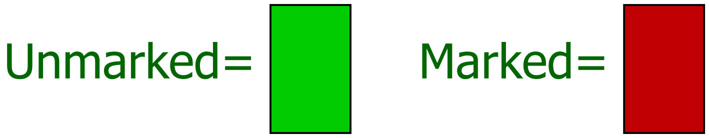
Mark & Sweep: GC Example

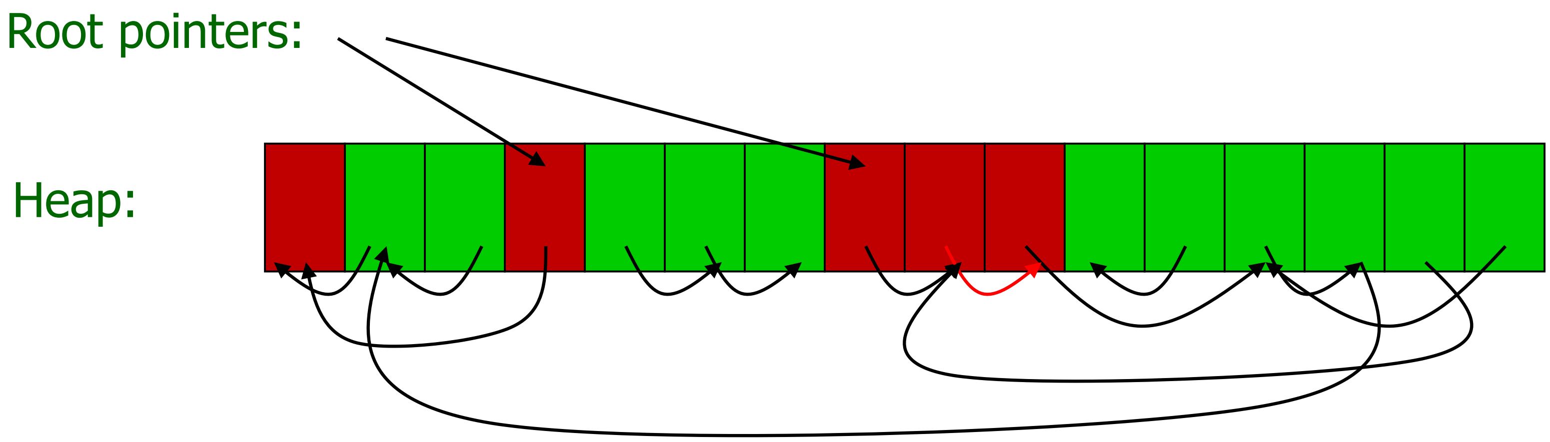


Mark & Sweep: GC Example

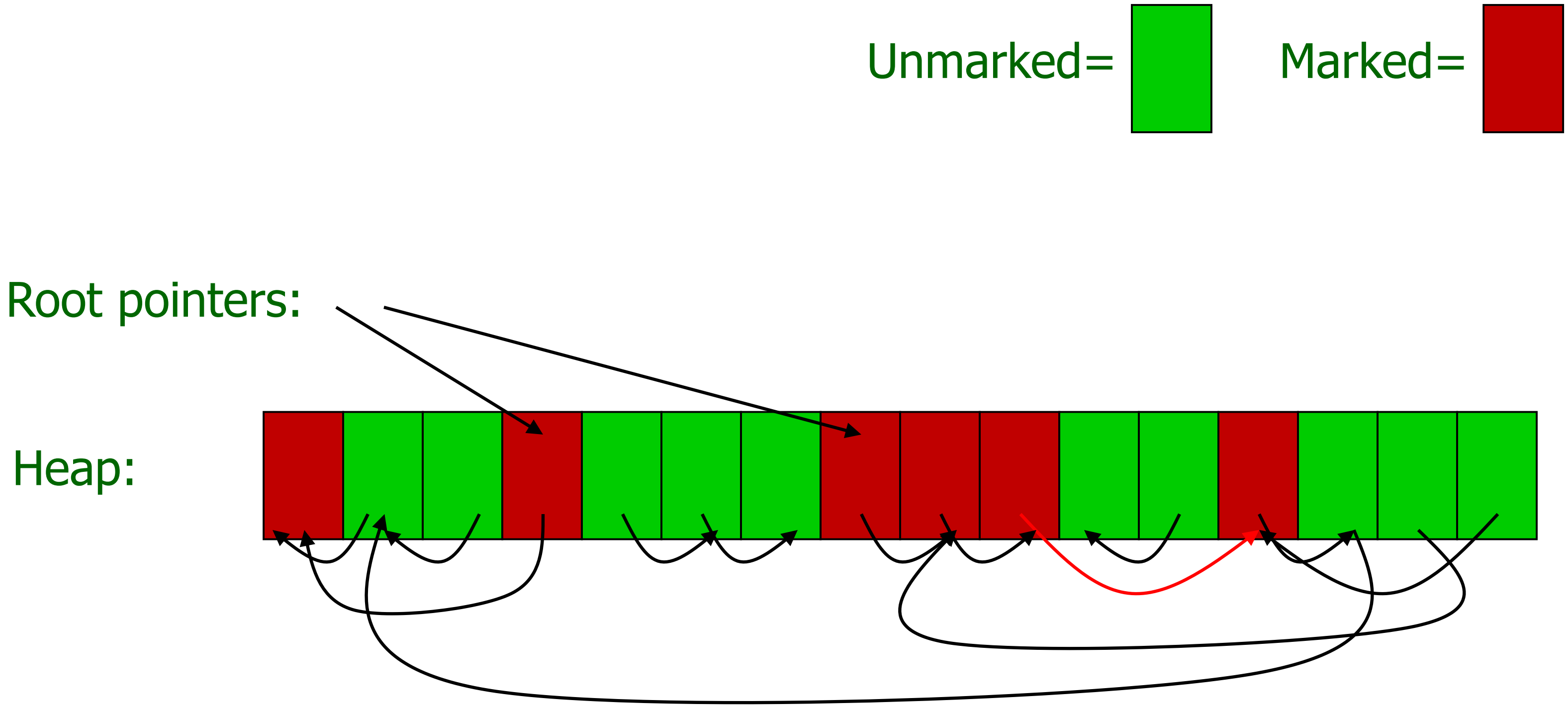


Mark & Sweep: GC Example

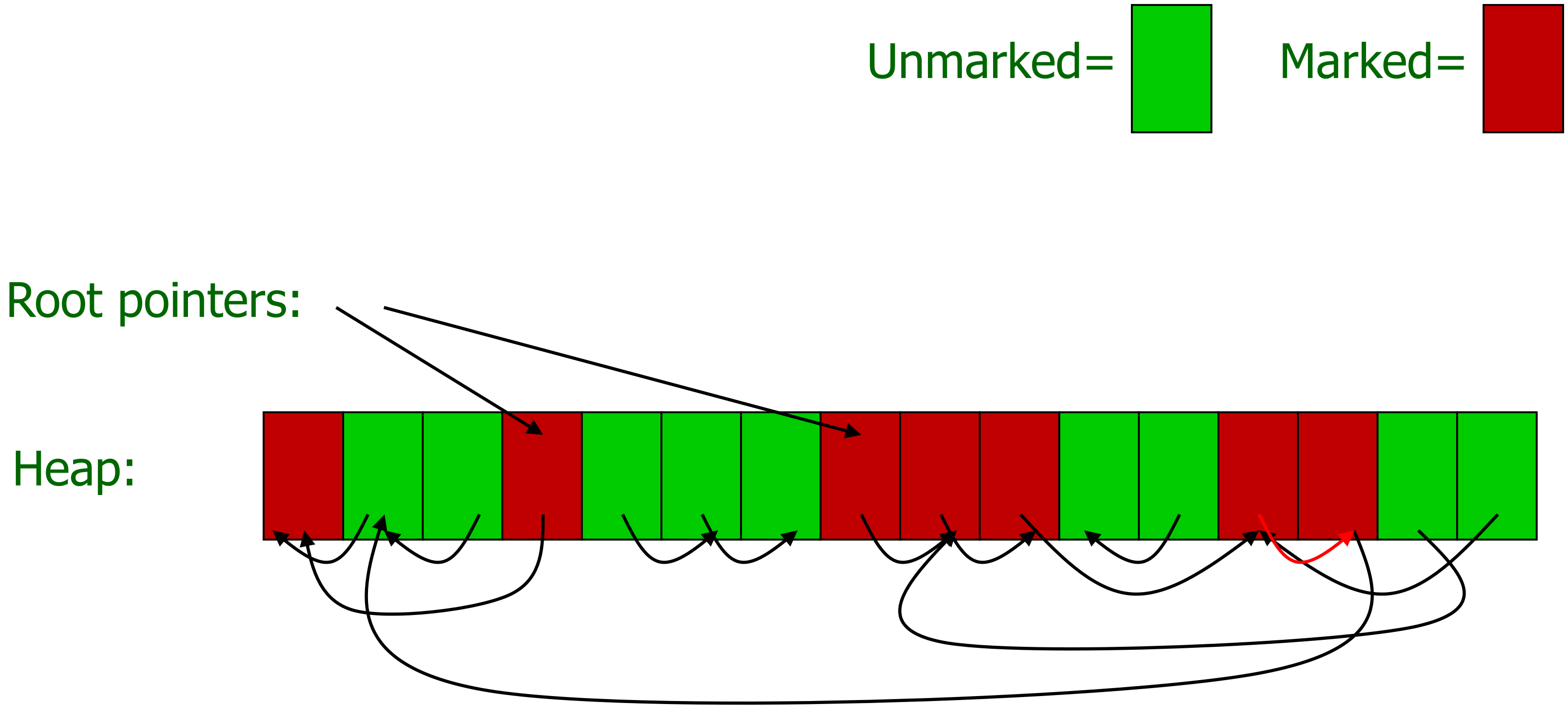




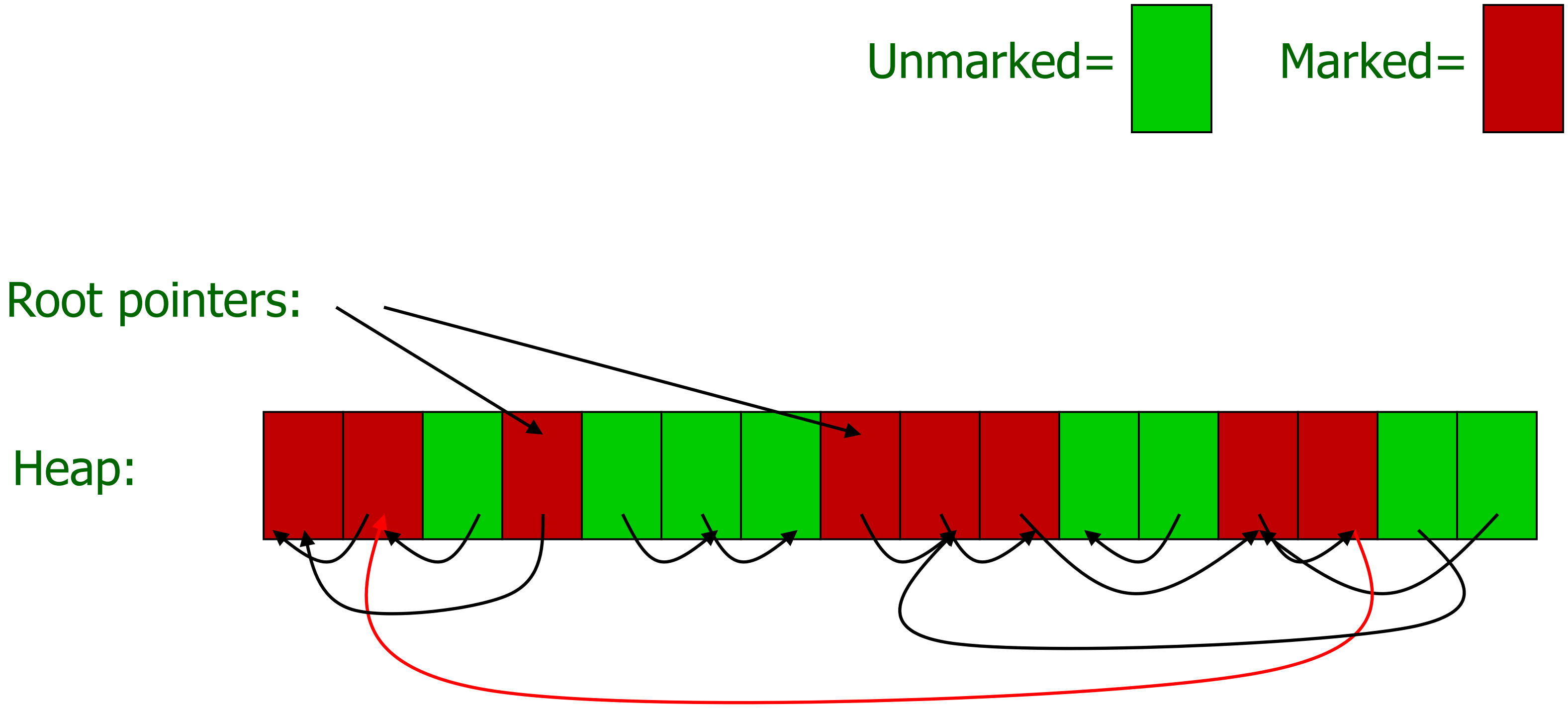
Mark & Sweep: GC Example



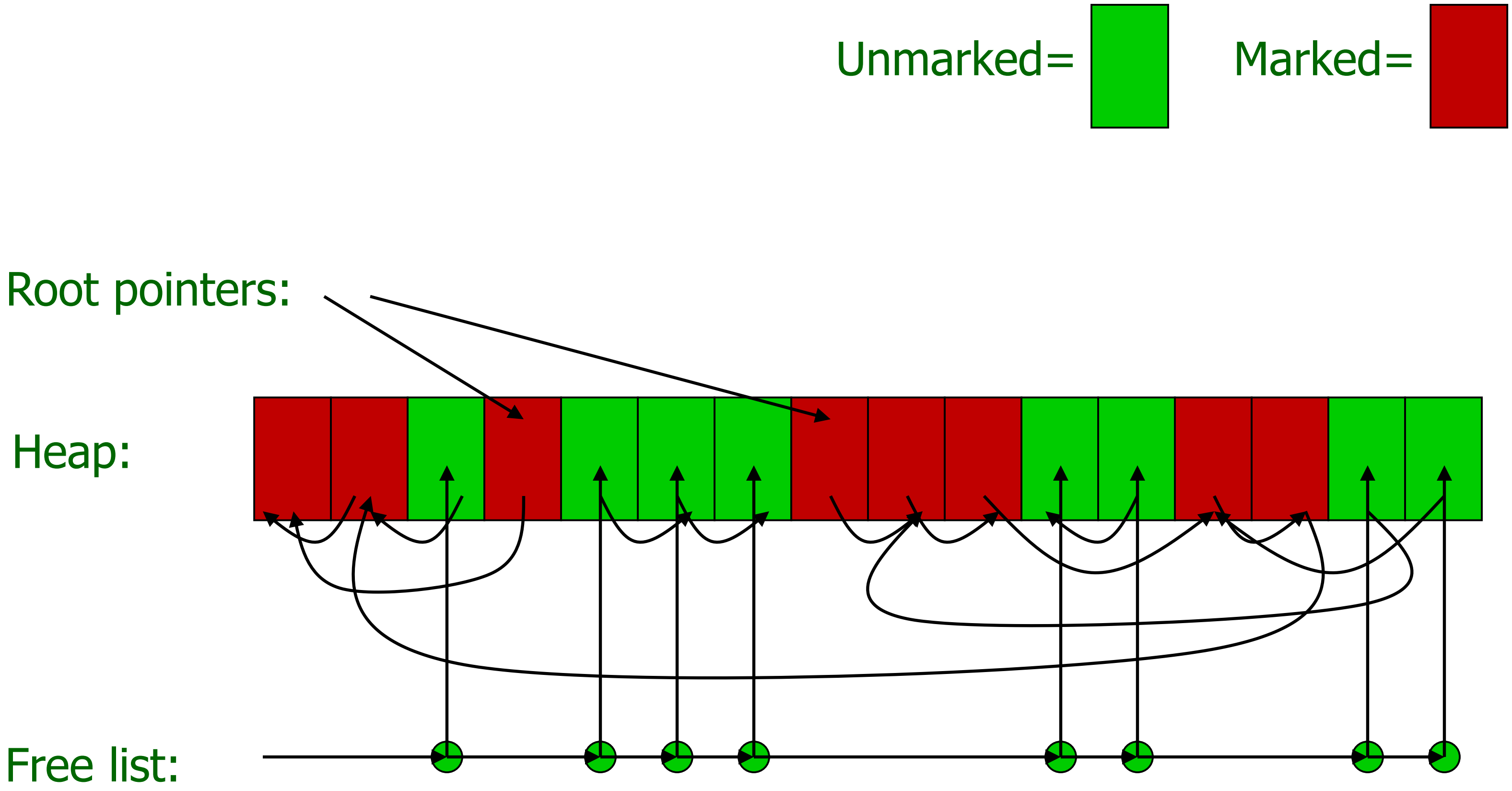
Mark & Sweep: GC Example



Mark & Sweep: GC Example



Mark & Sweep: GC Example



Mark & Sweep: Summary

- **Advantages:**
  - No time/space overhead for reference counts
  - Handles cycle references

- **Disadvantage:**
  - Noticeable pauses for GC
  - Time/space overhead for keeping track of pointers/references

NOTE: Conservative GC

- **Goal**
  - Allow GC in C-like languages

- **Usually a variation of Mark & Sweep**

- **Must conservatively assume that integers and other data can be cast to pointers**
  - Compile-time analysis to see when this is definitely not the case
  - Coding style heavily influences effectiveness

GC Summary

- **Safety**
  - **GC**: not programmer-dependent
  - **Explicit malloc/free**: programmer-dependent
- **Time overhead**
  - **GC**: Higher time overhead
    - Generally less predictable time overhead
  - **Explicit malloc/free**: lower time overhead
- **Space overhead**
  - **GC**: Generally higher space overhead (for extra metadata)
  - **Explicit malloc/free**: less space overhead

Today

- Explicit free lists
- Segregated free lists
- Garbage collection
- **Security issues related to dynamic memory**

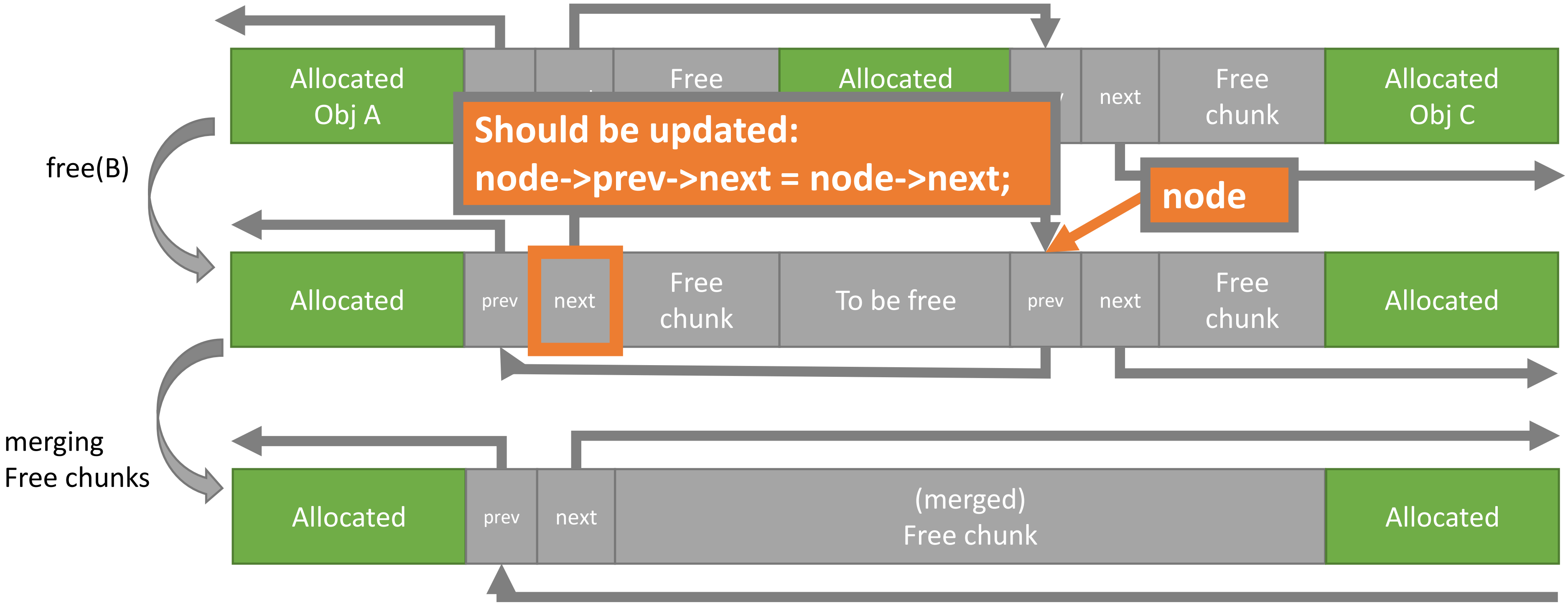
Security issues related to dynamic memory

- **Uninitialized memory use**
- **Heap overflow**
  - Use the memory beyond the block boundary
- **Double-free**
  - Freeing blocks multiple times
- **Use-after-free**
  - Using a (dangling) pointer after the pointed block is freed

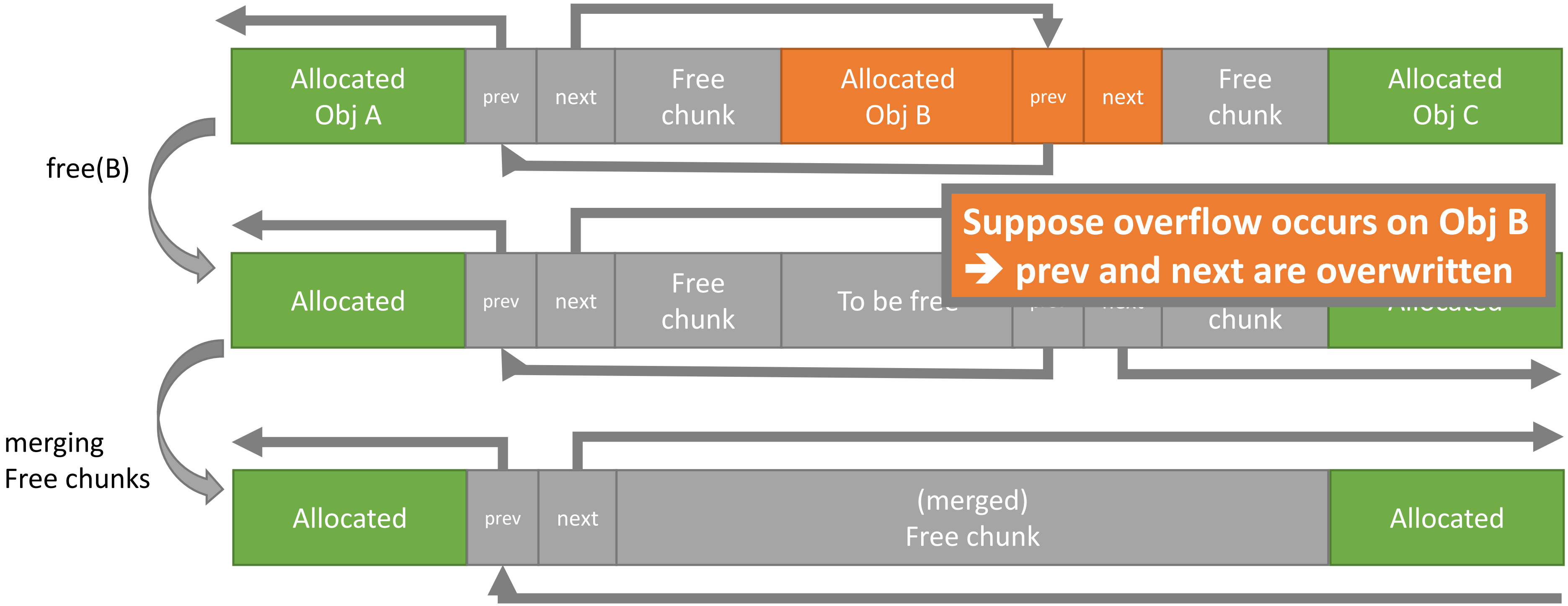
Heap Overflow: K&R Malloc

- Maintain a free list
  - A linked list of free chunks
  - prev/next pointers per free chunk
- An object is allocated by splitting up the free chunk
- Free chunks are merged if possible

Heap Overflow: Free chunks in K&R Malloc



Heap Overflow: Overflowing Metadata

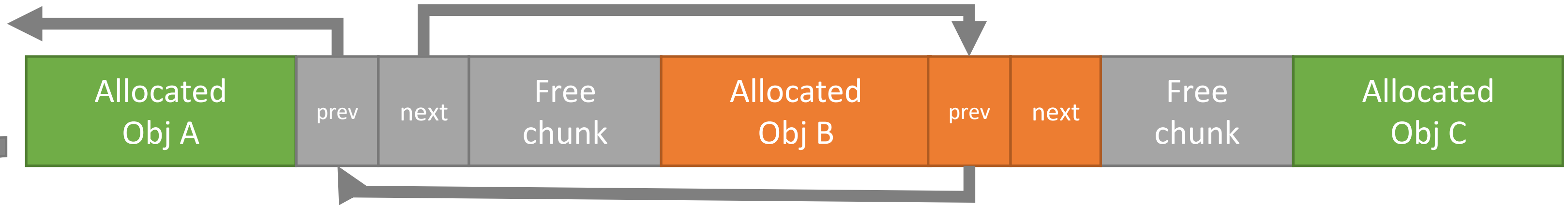


Heap Overflow: Free chunks in K&R Malloc

- Suppose overflow occurs on Obj B

```
char *p =malloc(16);
// …
memcpy(p, src, 32);
```

Heap Overflow: Overflowing Metadata



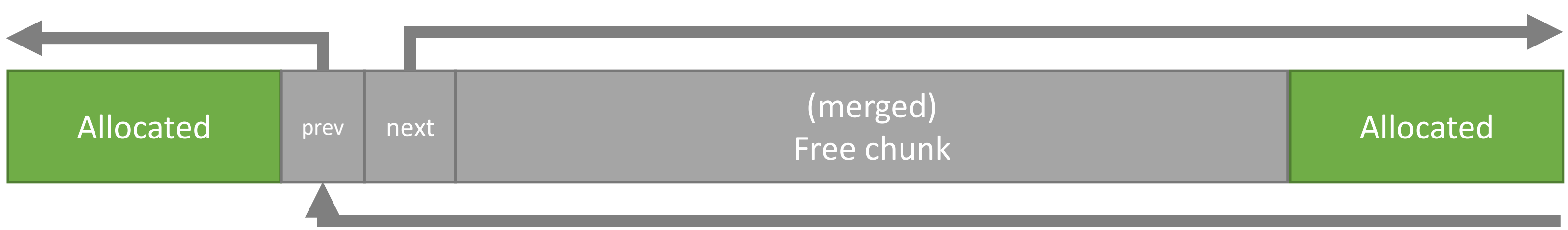
free(B)

Updating the linked list for merging

node.**prev-**>next = node.**next**;

➔ Transform heap overflow into arbitrary memory write vulnerability

merging Free chunks



Use-after-free

- **Root cause: a dangling pointer**
  - A pointer points to a freed memory region
- **Exploitation step:**
  - 1) Trigger **free** (dangling pointer is created)
  - **2) Overwrite** the freed region with the object having a different type
  - **3) Use** a dangling pointer

Use-after-free: An example from Chromium

```
class Doc : public Element {
    // ...
    Element *child;
};

class Body : public Element {
    // ...
    Element *child;
};
```

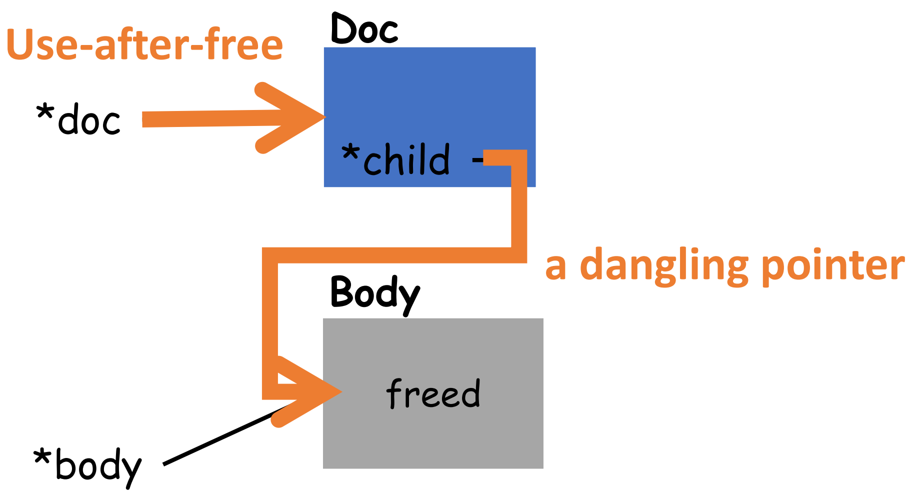
```
Doc *doc = new Doc();
Body *body = new Body();

doc->child = body;

delete body;

doc->child->getAlign();
```

An example from Chromium



Allocate objects

```
Doc *doc = new Doc();
Body *body = new Body();
```

Propagate pointers

```
doc->child = body;
```

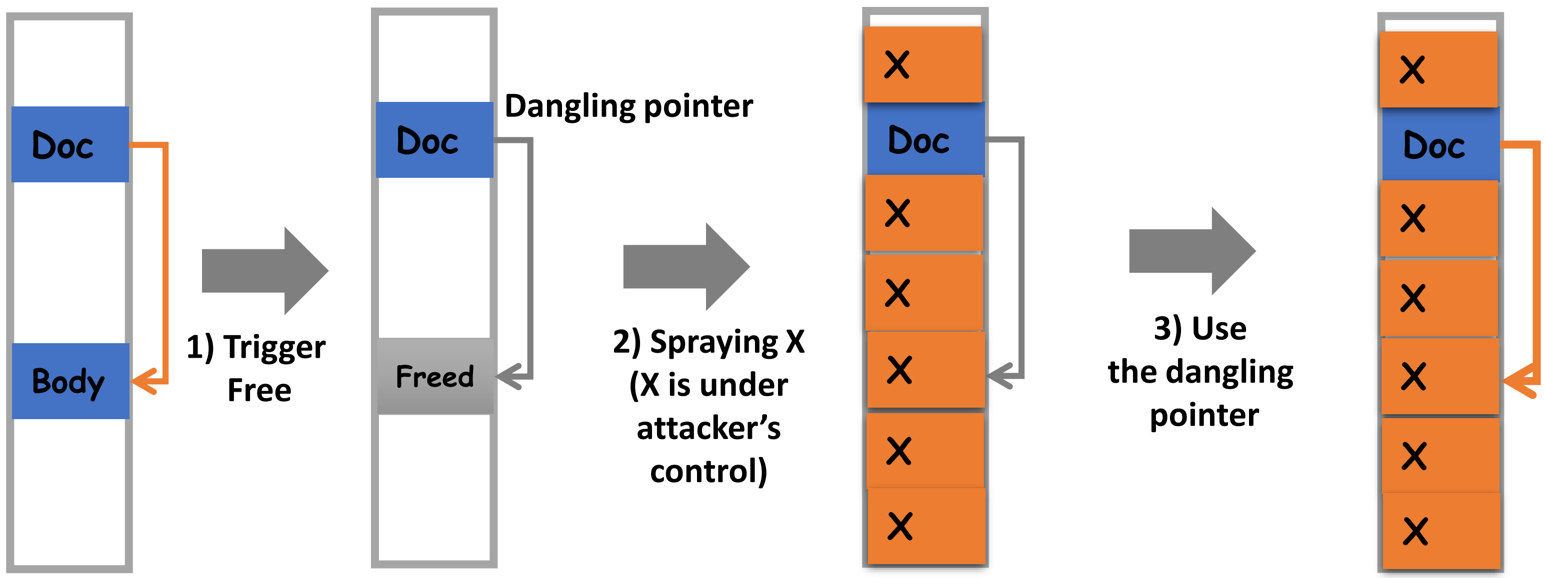
Free an object

```
delete body;
```

Use a dangling pointer

```
doc->child->getAlign();
```

Exploitation with Heap Spray



Using the dangling pointer leads to control-flow hijacks

➔Most C++ objects have virtual function pointer table (polymorphic classes)

How to Spray Heap

- Heap Spray: Attacker somehow needs to **control memory allocators**
- Different heap spray methods depending on target platforms
  - Web Browsers
    - Input: HTML
    - A long list of specific HTML tag blocks
      - Browser (renderer) executes a dedicated allocation routine per HTML tag
  - JavaScript
    - Input: JavaScript
    - Directly allocate from JavaScript (e.g., new[])
    - JS engine will allocate the object when interpreting the attacker-provided script
  - Kernel
    - Input: syscalls
    - Keep invoking a specific syscall (with well-crafted parameters)
      - Kernel executes a dedicated allocation for each syscall