11 Heap memory management

In Module 3, we introduced *variables* as an abstraction of areas of memory and observed that variables can have multiple *instances* with different *extents* of time during which they are accessible. Global variables have a single instance whose extent is the entire program execution, so they can be implemented by choosing some fixed memory address. Local variable instances have last-in-first-out extents that correspond to calls and returns of procedures, so they can be implemented using a stack data structure. In Module 6, we observed that function values in functional languages and objects in object-oriented languages require more flexible extents. Such flexible extents can be implemented using a *heap*, a data structure for managing memory that allows blocks of memory to be allocated and freed for reuse at arbitrary times. Note that the word *heap* is used to mean different things in different contexts; in particular, there is no connection between a *heap* in the context of memory management and the *heap* tree-shaped data structure used to implement priority queues.

Since even low-level programming languages such as C provide a heap abstraction as part of the language, a widespread misconception is that heap allocation (e.g., `malloc` or `new`) and deallocation (e.g., `free` or `delete`) are primitive operations like arithmetic operations or loads and stores to memory. In fact, they are calls to intricate algorithms that manage interesting data structures. In this module, we will learn how a heap can be implemented.

We will first implement a heap with explicit allocation and deallocation operations, like in C and C++. We will observe that in some cases, the heap may become *fragmented* and prevent reuse of deallocated memory. We will then design a second heap that prevents fragmentation by moving allocated blocks in memory. As an added benefit, the type information needed for moving objects also makes it possible to automatically detect unreachable blocks, making the explicit deallocation unnecessary. In our attempt to eliminate fragmentation, we will arrive at an *automatic garbage collector*. It is this garbage collector that you will implement in Assignment 11 for use with your Lacs compiler.

11.1 A heap with explicit deallocation

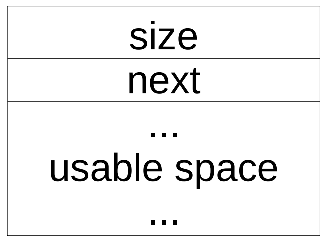
In Assignment 6, you implemented a simple heap allocator that allocates blocks one after another by repeatedly incrementing a pointer (`Reg.heapPointer`) that holds the address of the next unused memory location. In order to free such blocks for reuse, we need to create additional data structures to track which blocks of memory are in use (allocated) and which blocks have been freed for reuse (deallocated).

In order to manage memory as blocks rather than as individual words, we reserve a word of each block to store its size. The `Chunk` convention that we have followed throughout the course already maintains such a size word. If blocks are stored in the heap consecutively, one block after another, the sizes make it possible to navigate from one block to the one immediately after it, and thus to iterate through the blocks in the heap.

Once memory is organized into such blocks, we could reserve one additional bit in each block to indicate whether the block is allocated or free for reuse. To allocate a new block of a given size, we would iterate through the list of blocks in the heap until we found one that is large enough and whose bit indicates that it is free. This is a simple solution, but its downside is that finding a free block requires a linear search through all the blocks, possibly scanning the entire heap for each allocation in the worst case.

We can make allocation more efficient by organizing all the free blocks in the heap into a linked list. To allocate a block of a given size, we then need to search only the blocks in the list, which are all free, rather than the entire heap. This implementation of a heap is adapted from the implementation of `malloc` suggested in The C Programming Language by Brian Kernighan and Dennis Ritchie. This is the classic book that originally introduced the C language and described its implementation.

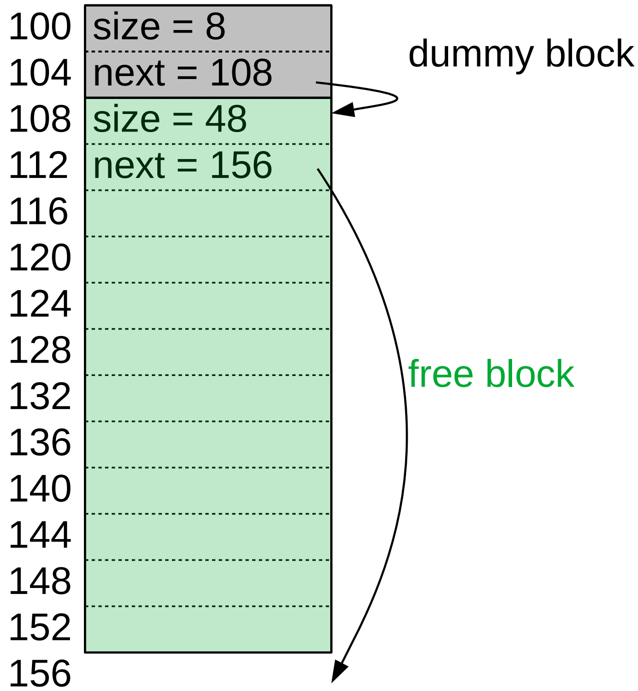
To implement the linked list of free blocks, we reserve a second word in each block to hold the address of the next free block in the list. Each block is then organized like this:



We begin implementing the heap as follows, using the abstractions that we have already developed in this course. First, we define utility procedures that, given the address of a block in the heap, read and write its size and next fields:

```
def size(block) = deref(block)
def next(block) = deref(block + 4)
def setSize(block, size) = assignToAddr(block, size)
def setNext(block, next) = assignToAddr(block + 4, next)
```

At the beginning of the program, we will initialize the heap as follows with two blocks. The first block will be a dummy block with no usable space, containing only the two words needed for its header. The purpose of the dummy block is to ensure that every real block in the heap has some block before it, to reduce the number of special cases we need to consider in the implementations of `malloc` and `free`. The second block will be a free block consisting of the entire remainder of the heap. If the example heap consists of 14 words starting at address 100, the initial state looks like this:



The *next* field of the dummy block always points to the beginning of the linked list of free blocks. The free blocks are linked using their *next* fields. The *next* field of the last free block in the list points to the memory location immediately after the end of the heap to indicate the end of the linked list. We set up this initial heap using the following initialization procedure:

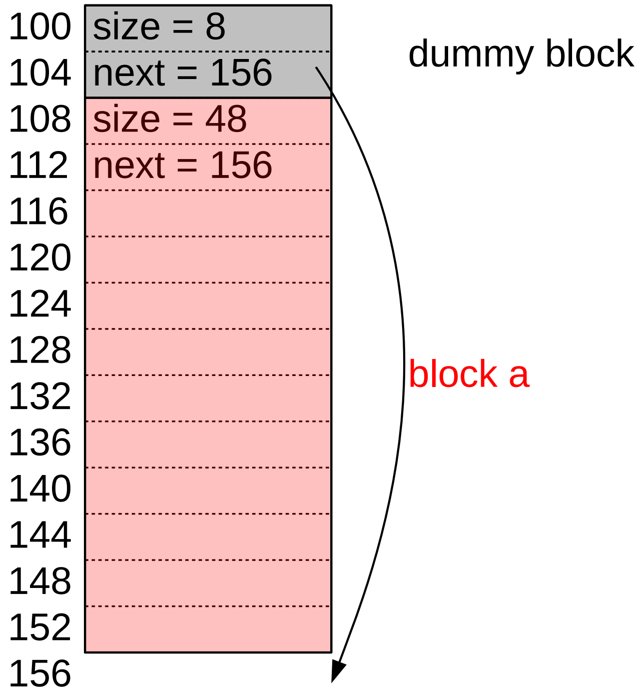
```
def init() = {
  val block = heapStart + 8
  setSize(heapStart, 8)
  setNext(heapStart, block)
  setSize(block, heapSize - 8)
  setNext(block, heapStart + heapSize)
}
```

Now that we have initialized the heap, we can allocate a block of `wanted` bytes of usable space (not including the two reserved words in the header) using an allocation procedure implemented as follows:

```
def malloc(wanted) = {
  def find(previous) = {
    val current = next(previous)
    if(size(current) < wanted + 8) find(current)
    else {
      if(size(current >= wanted + 16) { // split block
        val newBlock = current + wanted + 8
        setSize(newBlock, size(current) - (wanted + 8))
        setNext(newBlock, next(current))
        setSize(current, wanted + 8)
        setNext(previous, newBlock)
      } else {
        // remove current from free list
        setNext(previous, next(current))
      }
      current
    }
  }
  find(heapStart)
}
```

In the `find` procedure nested within `malloc`, the variable `current` holds the address of the free block that we will eventually return as allocated and the variable `previous` holds the address of the block before it in the linked list of free blocks. The `malloc` procedure initially calls `find` with `previous` being the address of the dummy block at the beginning of the heap. The `find` procedure first calculates `current` as the successor of `previous` in the linked list. If the `current` block is not big enough to hold `wanted` usable bytes, `find` does a tail call to itself passing `current` as the new `previous` to advance to the next block in the free list. (Instead of this tail call, we could have written an equivalent while loop.) If the `current` block is large enough, `find` unlinks it from the free list by setting the next field of the `previous` block to the address in the next field of the `current` block. It then returns the address of the `current` block, which is no longer linked in the free list, as the newly allocated block.

If we execute `a = malloc(8)` to allocate a block of 8 usable bytes in the initial heap, the heap will then look like this:

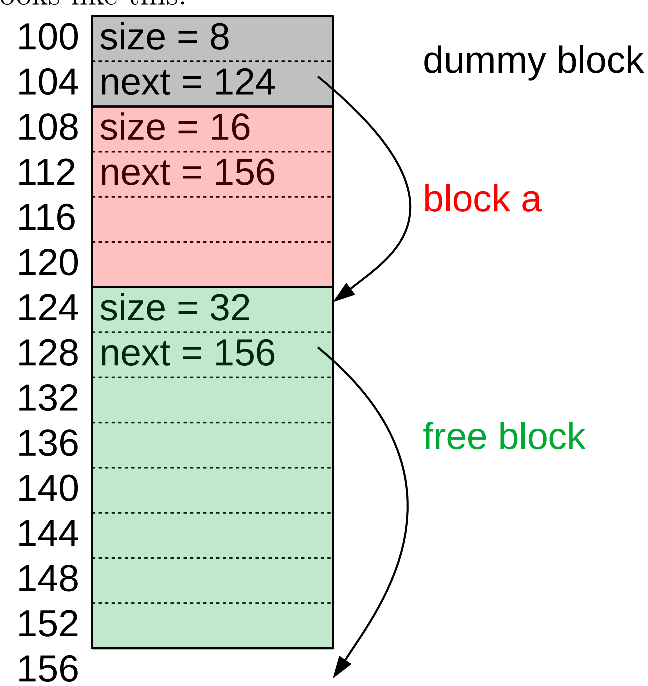


Although we asked for 8 usable bytes, so for a block of at least 16 bytes total including the two reserved words, the block with 48 total bytes was large enough, so it was unlinked from the free list and returned. The free list is now empty: the `next` field of the dummy block marking the beginning of the free list now points to the address after the end of the heap. A second allocation `b = malloc(8)` would fail because the free list is empty. A robust implementation of `malloc` should check for this and raise an error, but this is not shown to avoid cluttering the code.

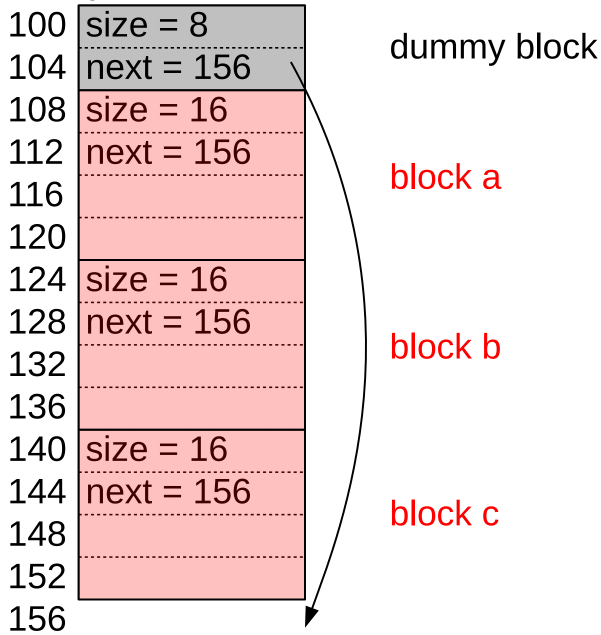
To enable the heap to allocate more than one block, it should *split* the block that it finds on the free list if it is big enough:

```
def malloc(wanted) = {
  def find(previous) = {
    val current = next(previous)
    if(size(current) < wanted + 8) find(current)
    else {
      if(size(current >= wanted + 16) { // split block
        val newBlock = current + wanted + 8
        setSize(newBlock, size(current) - (wanted + 8))
        setNext(newBlock, next(current))
        setSize(current, wanted + 8)
        setNext(previous, newBlock)
      } else {
        // remove current from free list
        setNext(previous, next(current))
      }
      current
    }
  }
  find(heapStart)
}
```

This time, if the `current` block is large enough to store `wanted` usable bytes and the headers of two blocks (16 bytes), we split it into two blocks. The first block (`current`) is exactly the size we requested: `wanted` bytes plus another 8 bytes for its header. The second block (`newBlock`) contains the remaining words of memory that used to be in the `current` block that was larger than necessary. The `current` block is unlinked from the list of free blocks and the `newBlock` is added to it. The heap after executing `a = malloc(8)` with this new implementation looks like this:



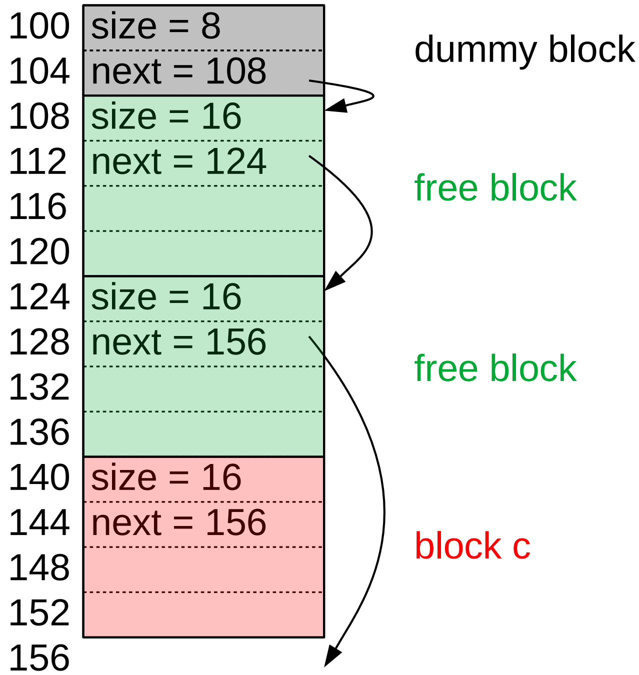
After additionally executing `b = malloc(8)` and `c = malloc(8)`, the heap looks like this:



Now suppose we wish to free blocks `a` and `b` for reuse. We could use the following implementation of `free` to free a block whose address is passed for the parameter `toFree`:

```
def free(toFree) = {
  def find(previous) = {
    val current = next(previous)
    if(current < toFree) find(current)
    else {
      val coalesceWithPrevious =
        previous + size(previous) == toFree &&
        previous > heapStart
      val coalesceWithCurrent =
        toFree + size(toFree) == current &&
        current < heapStart + heapSize
      if(!coalesceWithPrevious && !coalesceWithCurrent) {
        setNext(toFree, current)
        setNext(previous, toFree)
      } else if(coalesceWithPrevious && !coalesceWithCurrent) {
        setSize(previous, size(previous) + size(toFree))
      } else if(!coalesceWithPrevious && coalesceWithCurrent) {
        setSize(toFree, size(toFree) + size(current))
        setNext(toFree, next(current))
        setNext(previous, toFree)
      } else { // coalesceWithPrevious && coalesceWithCurrent
        setSize(previous, size(previous)+size(toFree)+size(current))
        setNext(previous, next(current))
      }
    }
  }
  find(heapStart)
}
```

Like in the implementation of `malloc`, the `find` procedure searches through the linked list of free blocks, this time to set `previous` to the last free block before the block `toFree` to be freed, and `current` to the block immediately after `previous` in the free list. This `current` block is the first block in the free list after the block `toFree` to be freed. The two calls to `setNext` link `toFree` into the free list in between blocks `previous` and `current`. By searching for the blocks `previous` and `current` before and after the block `toFree`, this implementation of `free` ensures that the blocks in the free list are always listed in order of their memory addresses. It would have been simpler to just link `toFree` to the beginning of the linked list, rather than to search the list for `previous` and `current`. To understand why it is valuable to maintain the free list in order of memory addresses, look at the state of the heap after both blocks `a` and `b` have been freed:



The free list now contains two free blocks, each of size 16, with 8 usable bytes. If we now decide to allocate `d = malloc(12)` in this heap, the allocation will fail because neither of the two free blocks is large enough to store 12 usable bytes, even though the total amount of free memory is sufficient. To make it possible to allocate block `d`, we will *coalesce* a freed block by combining it with the blocks immediately before and after it in memory, if those blocks are also free. To be able to do this, we need the addresses of the closest free blocks before and after the block `toFree`, and it is for this reason that we keep the list of free blocks ordered by memory address.

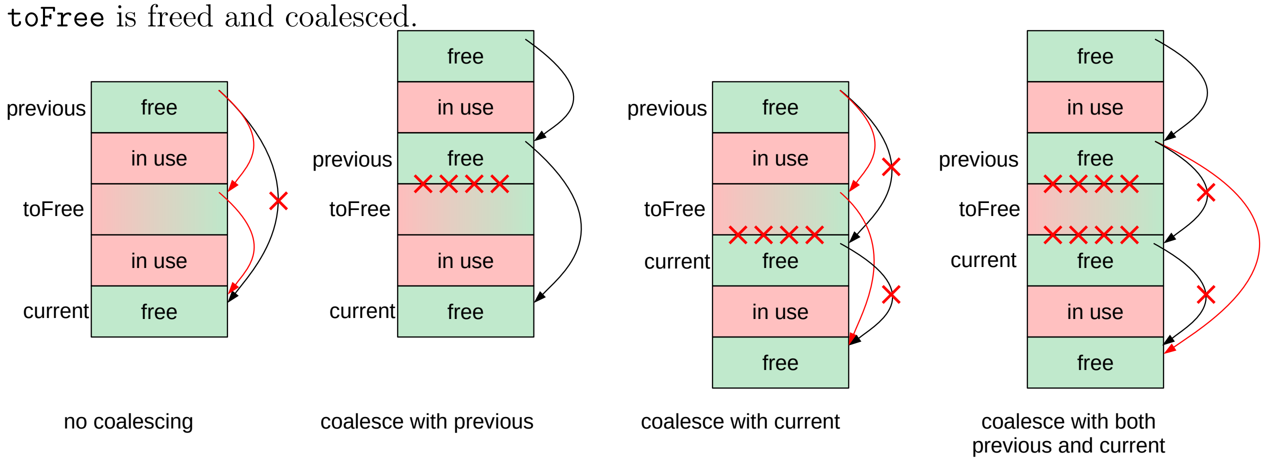
To implement coalescing, we first add code to decide whether to coalesce with the `previous` and `current` blocks. We coalesce `toFree` with the `previous` block if it is in memory immediately before `toFree` and if it is not the dummy block. Similarly, we coalesce `toFree` with the `current` block if it is in memory immediately after `toFree` and it is not the address after the end of the heap:

```
def free(toFree) = {
  def find(previous) = {
    val current = next(previous)
    if(current < toFree) find(current)
    else {
      val coalesceWithPrevious =
        previous + size(previous) == toFree &&
        previous > heapStart
      val coalesceWithCurrent =
        toFree + size(toFree) == current &&
        current < heapStart + heapSize
      if(!coalesceWithPrevious && !coalesceWithCurrent) {
        setNext(toFree, current)
        setNext(previous, toFree)
      } else if(coalesceWithPrevious && !coalesceWithCurrent) {
        setSize(previous, size(previous) + size(toFree))
      } else if(!coalesceWithPrevious && coalesceWithCurrent) {
        setSize(toFree, size(toFree) + size(current))
        setNext(toFree, next(current))
        setNext(previous, toFree)
      } else { // coalesceWithPrevious && coalesceWithCurrent
        setSize(previous, size(previous)+size(toFree)+size(current))
        setNext(previous, next(current))
      }
    }
  }
  find(heapStart)
}
```

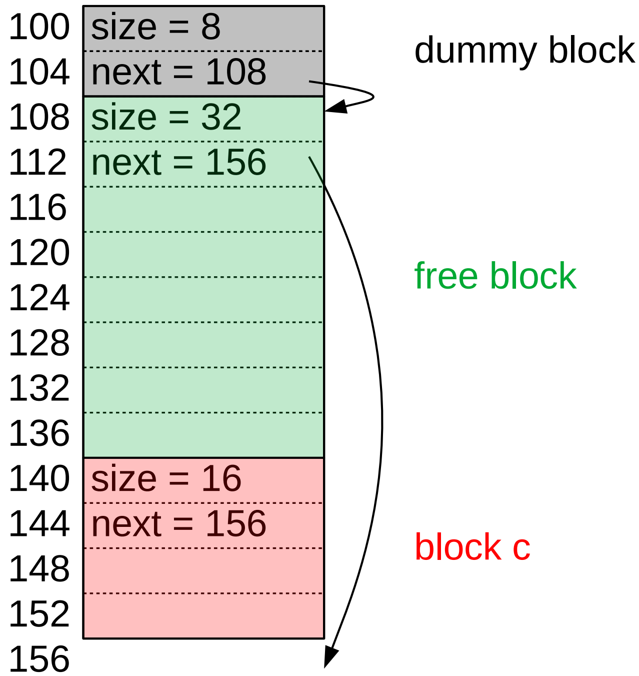
We then implement each of the four possible cases in which `toFree` is not coalesced at all, coalesced with `previous` but not `current`, coalesced with `current` but not `previous`, or coalesced with both `previous` and `current`. In the last case, the three blocks `previous`, `toFree`, and `current` are coalesced to form a single large block:

```
def free(toFree) = {
  def find(previous) = {
    val current = next(previous)
    if(current < toFree) find(current)
    else {
      val coalesceWithPrevious =
        previous + size(previous) == toFree &&
        previous > heapStart
      val coalesceWithCurrent =
        toFree + size(toFree) == current &&
        current < heapStart + heapSize
      if(!coalesceWithPrevious && !coalesceWithCurrent) {
        setNext(toFree, current)
        setNext(previous, toFree)
      } else if(coalesceWithPrevious && !coalesceWithCurrent) {
        setSize(previous, size(previous) + size(toFree))
      } else if(!coalesceWithPrevious && coalesceWithCurrent) {
        setSize(toFree, size(toFree) + size(current))
        setNext(toFree, next(current))
        setNext(previous, toFree)
      } else { // coalesceWithPrevious && coalesceWithCurrent
        setSize(previous, size(previous)+size(toFree)+size(current))
        setNext(previous, next(current))
      }
    }
  }
  find(heapStart)
}
```

The four cases of coalescing are summarized in the following diagram. In each case, the black arrows and block boundaries indicate the state *before* block `toFree` is freed and coalesced and the red arrows and crossed-out block boundaries indicate the state *after* block `toFree` is freed and coalesced.



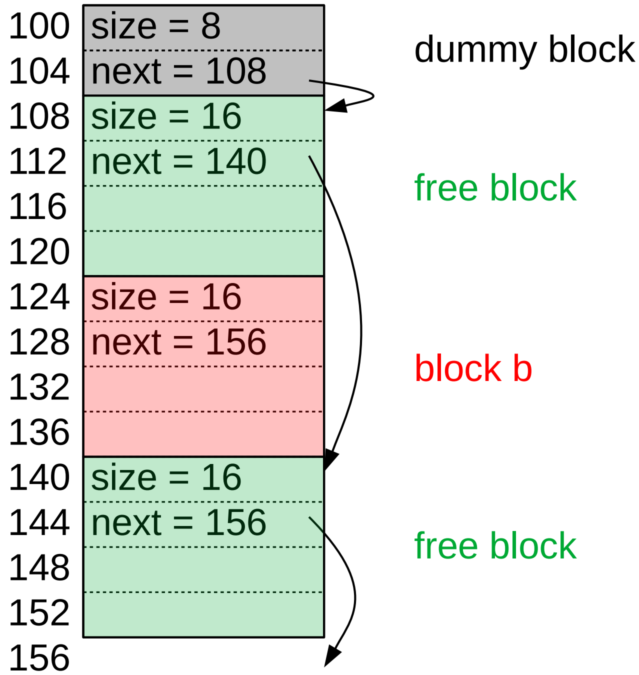
With coalescing, the heap after freeing blocks `a` and `b` looks as follows. The two freed blocks are coalesced into one large block of size 32, which is large enough to allocate `d = malloc(12)`.



In the worst case, both `malloc` and `free` take time linear in the size of the heap because the `find` procedure in both cases does a linear search through the list of free blocks, and the maximum possible number of free blocks is linear in the size of the heap.

11.2 Fragmentation and compaction

Suppose that instead of freeing block `a` and `b`, we free blocks `a` and `c`. There is now enough total free memory space to allocate `d = malloc(12)`, but the space is *fragmented*: it is split into two small free blocks that cannot be coalesced because they are separated by block `b`, which is in use.



In the general worst case, fragmentation can be arbitrarily bad in that it can prevent allocation of even a small block even when a large amount of memory is free. The allocation of the 12 byte block `d = malloc(12)` could fail even if there were 12 megabytes or 12 gigabytes of free memory, if those 12 gigabytes were fragmented into many blocks of size smaller than 12 each, separated by blocks that are in use.

To reliably avoid fragmentation in all cases, we need to be able to *move* blocks in memory. In the example, the free space is separated by the block `b` that is in use. We could combine the two free blocks into one large one if we could move block `b` to the beginning or end of the heap. The process of moving blocks that are in use together, so they are adjacent in the heap with no fragmented free blocks between them, is called *compaction*.

If compaction moves a block, the address of the block changes. Any pointers in the heap that hold the original address need to be *updated* to the new address to maintain the structure of the data in the heap. This is possible only if we can identify all the pointers in memory. Specifically, for each word in memory, we need to know whether its bits represent an address (and thus should be updated) or some other value (and thus should not be changed), much like we did for relocation in Module 2. Therefore, compaction requires *sound type information*. Furthermore, the information about which words in memory hold addresses must be communicated to the implementation of the heap: the type information must be made available at run time.

In the compiler that we have been developing in this course, each `Variable` has an `isPointer` flag that indicates whether the variable will store an address or an integer. Each `Chunk` associates a `Variable` with each word in the `Chunk`. Our compiler thus knows which words in memory hold addresses, and it must make that information available at run time. One way to do this is with a convention that within each `Chunk`, we place all the addresses first, at smaller offsets, and all non-address words aftewards. This is possible because the `Chunk` class controls the offset that it assigns to each `Variable`, so it can place the variables in memory in any order. Then, in the second word of each `Chunk`, which we have been reserving for memory management, we store the *number of addresses* in the `Chunk`. If that word specifies n addresses, then at run time, the implementation of the heap can determine that the first n words after the two-word header of the `Chunk` are addresses and the words after the first n are not addresses. The following diagram summarizes the layout of a `Chunk`.

size
number of address words
... address words ...
... non-address words ...

Once we make a decision to implement compaction in the heap, allocation and deallocation can be simplified considerably:

- In allocation, we no longer need to maintain and search a list of free blocks to be reused. Instead, allocation can always return the block after the last one that was allocated, like in the simple heap allocator that you implemented in Assignment 6. When allocation reaches the end of the heap, it can trigger a compaction, which will move all the blocks that are in use together, leaving one large area of free memory after them that can again be used to allocate more blocks simply by incrementing a pointer like in the simple heap allocator.

- Deallocation no longer needs to link freed blocks into a linked list or to coalesce them. It needs to only mark blocks as free for a later compaction pass.

Although each compaction may take a long time because it moves all allocated objects in the heap, a compaction needs to be done only rarely, after many allocations and deallocations have occurred and we have reached the end of the heap. When the cost of compaction is spread over these many allocations, the cost per allocation can be very low. In many cases, the cost per allocation is even lower than the cost of work done by `malloc` and `free`.

11.3 Automatic garbage collection

Once we know which words in the heap are addresses, which we need to know to make compaction possible, we can take one further step and remove the need for the program to explicitly deallocate blocks for reuse. Instead, we design the compaction algorithm to compact all blocks that are *reachable* by a path of pointers from the stack. If a block is not reachable by such a path, then the executing program can never find its address, and will therefore never access it, so it is safe to consider such a block to be free and reuse its memory. By compacting all *reachable* blocks, we ensure that we include all blocks that may ever be accessed in the future. A memory manager that finds free blocks automatically in this way is called an automatic *garbage collector*.

More precisely, we say that a block in the heap is *reachable* if:

1. the address of the block is in a block on the stack, or

2. the address of the block is in an already *reachable* block in the heap.

We will look in detail at one specific algorithm, *Cheney's copying garbage collector,* which you will implement in Assignment 11. One benefit of this algorithm is its simplicity. Despite that simplicity, many modern real-world garbage collection systems are based on such a copying collector (together with various other algorithms).

Cheney's algorithm conceptually splits the heap into two halves, called *semispaces*. The two semispaces are called the *from-space* and the *to-space*.

New blocks are allocated in the from-space. A pointer (`heapPointer`) keeps track of the first free address in the from-space, so allocation requires only incrementing this pointer, much like in the simple heap allocator from Assignment 6.

When the `heapPointer` reaches the end of the from-space, a compaction is triggered, which copies all reachable blocks from the from-space to the beginning of the to-space. Since these blocks are placed one immediately after another in the to-space, they are compact with no free space between them; all the free space is together in one piece after the blocks that have been copied. After all reachable blocks have been copied to the to-space, the blocks in the from-space are no longer needed and are abandoned. The designations of the from-space and the to-space are then *switched*: the original to-space becomes the from-space and vice versa. The `heapPointer` is set to the address in the new from-space after all the blocks that have been copied, and allocation can continue there.

Eventually, the `heapPointer` reaches the end of the new from-space and another compaction occurs, switching the from-space and the to-space back to their original designations. This process continues with the from-space and to-space switching places back and forth with each compaction.

To make this general description more concrete, let us first look at how we initialize the heap. We will make use of three constants:

- `heapStart`, the address of the first word in the heap,

- `heapEnd`, the address after the last word in the heap, and

- `heapMiddle`, the address halfway between `heapStart` and `heapEnd`.

We assume that the heap contains an even number of words so that we can split it into two equal semispaces. In `MemoryManagement.scala` in the handout code, these constants are defined at $\frac{1}{4}$ of memory, $\frac{3}{4}$ of memory, and $\frac{1}{2}$ of memory, respectively. Thus, the first $\frac{1}{4}$ of memory is reserved for the code of the MIPS program, the second $\frac{1}{4}$ of memory is the first semispace of the heap, the third $\frac{1}{4}$ of memory is the second semispace of the heap, and the last $\frac{1}{4}$ of memory is reserved for the stack. We will designate two registers to implement the heap:

- `heapPointer`, as already mentioned, holds the address of the first free word in the from-space, and

- `fromSpaceEnd` holds the address after the last word in the from-space.

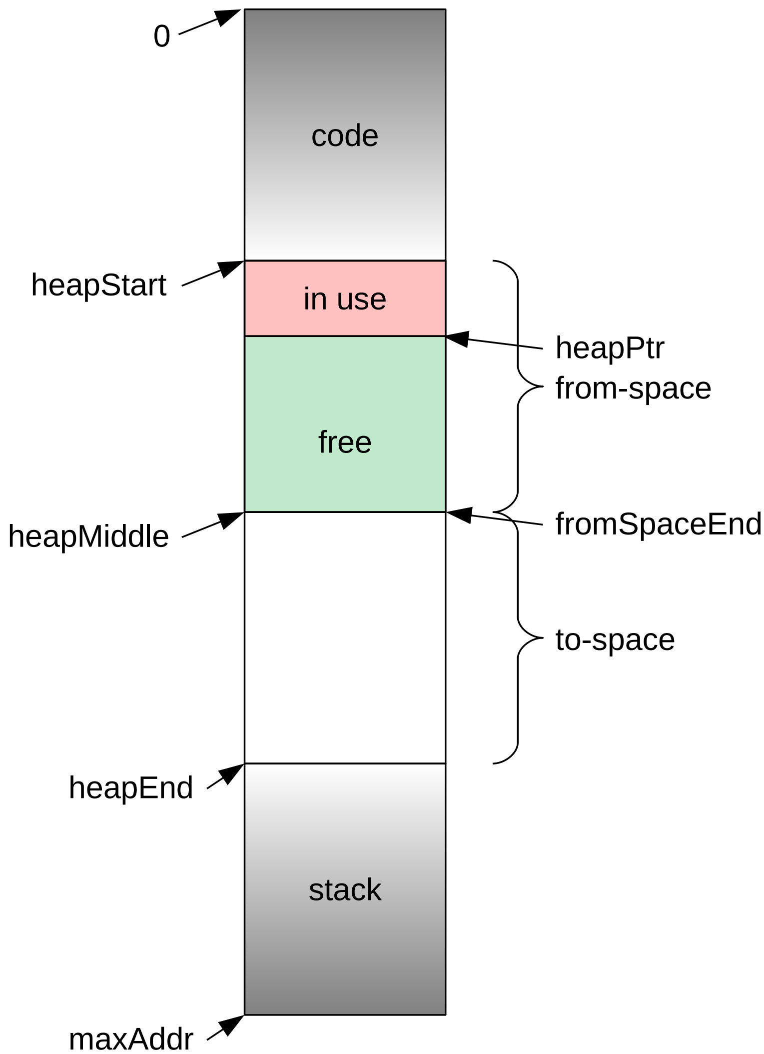
The fromSpaceEnd register serves both to identify which of the two semispaces is currently the from-space and to enable us to check when allocation has reached the end of the from-space. These two registers are initialized as follows, making the first semispace the from-space and the second semispace the to-space:

```
def init() = {
  heapPointer = heapStart
  fromSpaceEnd = heapMiddle
}
```

To allocate a block of wanted bytes, this time including the header, we increment the heapPointer by wanted, as you did in the simple heap allocator in Assignment 6:

```
def allocate(wanted) = {
  if(heapPointer + wanted > fromSpaceEnd) {
    gc()
  }
  val oldHeapPointer = heapPointer
  heapPointer = heapPointer + wanted
  oldHeapPointer
}
```

If we have called init, followed by several calls to allocate to allocate some blocks, the overall structure of memory looks like this:



Compaction happens in the `gc` procedure, which `allocate` calls when the from-space becomes full and the new block to be allocated would reach past the end of the from-space. Let us look at the implementation of the `gc` procedure in steps.

```
def gc() = {
  val toSpaceStart = if(fromSpaceEnd == heapMiddle) heapMiddle
                     else heapStart
  var free = toSpaceStart
  var scan = dynamicLink // top of stack, not incl. frame of gc
  while(scan < maxAddr) {
    forwardPtrs(scan)
    scan = scan + size(scan)
  }
  scan = toSpaceStart
  while(scan < free) {
    forwardPtrs(scan)
    scan = scan + size(scan)
  }
  fromSpaceEnd = toSpaceStart + semiSpaceSize
  heapPointer = free

  def forwardPtrs(block) = {
    for each offset o in block that holds an address {
      val newAddr = copy(deref(block + o))
      assignToAddr(block + o, newAddr)
    }
  }

  // copy block from from-space to to-space, return new address
  def copy(block) = {
    if(block is not in from-space) { block }
    else {
      if(size(block) >= 0) { // not yet copied
        copyChunk(free, block)
        setNext(block, free) // leave forwarding address
        setSize(block, 0 - size(block)) // mark block as copied
        free = free + size(free)
      }
      next(block) // return forwarding address
    }
  }
}
```

First, the `gc` procedure computes the address of the beginning of the to-space, `toSpaceStart`, using `fromSpaceEnd` to identify the current from-space and to-space. The variable `free` is used to keep track of the first free address in the to-space, much like the `heapPointer`. It is initialized to `toSpaceStart`.

The `gc` procedure then uses the `scan` variable to loop through all blocks on the stack, looking for addresses in those blocks. By the definition of a *reachable* block, any block in the heap whose address appears in a block on the stack is reachable and therefore needs to be copied from the from-space to the to-space. The `forwardPtrs` procedure looks through a block on the stack for all words that represent addresses and calls the `copy` procedure on each such address. The `copy` procedure copies a block from the from-space to the to-space and returns its new address. The `forwardPtrs` procedure then replaces the old from-space address in the block on the stack with the new to-space address (`newAddr`). The loop that calls `forwardPtrs` on each block in the stack starts with the `dynamicLink`, the frame of the procedure that called `gc`, typically `allocate`. This avoids the garbage collector considering blocks in the heap to be reachable if they are reached only from the variables in the frame of the `gc` procedure, such as `toSpaceStart`, `free`, and `scan`, but includes all other variables in blocks on the stack that hold addresses. In order for `dynamicLink` to correctly point to the block on the stack below the frame of `gc`, the frame of the procedure that calls `gc` must be on the stack, not on the heap. This is the case for `allocate` and all procedures that Marmoset uses to test the implementation of `gc`.

Addresses on the stack may point into the heap, but they may also point to other blocks on the stack, or they may be the addresses of machine language instructions in the code of the program. Therefore, the `copy` procedure leaves any block that is not in the from-space of the heap alone, just returning its address unchanged. If everything is working correctly, it should never happen that `copy` is called with an address in the to-space.

If a block is in the from-space, `copy` copies it to the free part of the to-space and increments the `free` pointer. It also marks the original block as copied, for example by setting its size word to a negative number, and leaves the address of the copy in the to-space in the block, for example in the second, `next` word. This is done so that if there is more than one pointer in the stack to the same block in the heap, the block will not be copied multiple times. Instead, `copy` will notice that the block has already been copied because its size is negative; instead of copying the block, `copy` will read the forwarding address from its `next` word to determine the address in the to-space to which the block was copied the first time it was encountered. When `copy` returns the address of the copy of the block in the to-space to `forwardPtrs`, `forwardPtrs` replaces the address of the original block in the stack with the new address.

At the end of the while loop, all from-space blocks whose addresses are on the stack have been copied to the to-space and the addresses on the stack have been replaced with the addresses of the copies. However, this does not account for all reachable blocks in the from-space, since a block can be reachable indirectly by an address in another reachable block, rather than directly by an address on the stack. In addition, the blocks that have been copied to the to-space may contain addresses inside them, and these addresses still point into the from-space. Both of these issues are solved by the second while loop:

```
def gc() = {
  val toSpaceStart = if(fromSpaceEnd == heapMiddle) heapMiddle
                     else heapStart
  var free = toSpaceStart
  var scan = dynamicLink // top of stack, not incl. frame of gc
  while(scan < maxAddr) {
    forwardPtrs(scan)
    scan = scan + size(scan)
  }
  scan = toSpaceStart
  while(scan < free) {
    forwardPtrs(scan)
    scan = scan + size(scan)
  }
  fromSpaceEnd = toSpaceStart + semiSpaceSize
  heapPointer = free

  def forwardPtrs(block) = {
    for each offset o in block that holds an address {
      val newAddr = copy(deref(block + o))
      assignToAddr(block + o, newAddr)
    }
  }

  // copy block from from-space to to-space, return new address
  def copy(block) = {
    if(block is not in from-space) { block }
    else {
      if(size(block) >= 0) { // not yet copied
        copyChunk(free, block)
        setNext(block, free) // leave forwarding address
        setSize(block, 0 - size(block)) // mark block as copied
        free = free + size(free)
      }
      next(block) // return forwarding address
    }
  }
}
```

The second while loop looks similar to the first, but instead of traversing the stack, it traverses the to-space. Like the first while loop, the second while loop calls `forwardPtrs` on each block in the to-space to look for addresses in it. For each such address, `forwardPtrs` again uses `copy` to copy the block designated by the address from the from-space to the to-space if it has not been copied already, and replaces the address by the address of the copy.

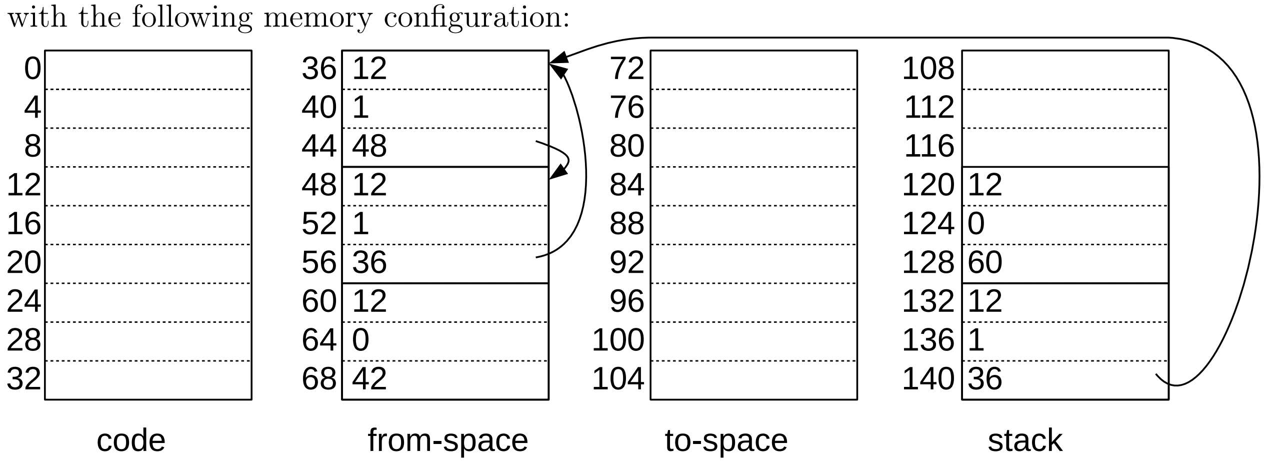
A subtle aspect of this second while loop is that in its condition `scan < free`, *both* `scan` and `free` are changing. The `scan` pointer points to the block currently being scanned and `free` indicates the end of all the blocks in the to-space. During the call to `forwardPtrs(scan)`, if newly reachable blocks are discovered and copied to the to-space, the `free` pointer is incremented accordingly. Therefore, as long as newly reachable blocks are being found, both pointers are increasing, with `scan` chasing `free`. Eventually, when all the reachable blocks in the from-space have been copied to the to-space, `free` stops increasing and `scan` catches up to it, ending the loop. When that happens, we can be sure that we have called `forwardPtrs` on all the blocks in the to-space, not only those that were copied there in the first while loop, but also those that were copied in the second while loop due to being referenced in some other reachable block. Thus, by the time the `scan` catches up to `free`, all reachable blocks in the from-space have been copied to the to-space and all addresses of from-space blocks in the to-space have been updated to the to-space copies of those from-space blocks. The compaction is complete.

This loop with two pointers avoids the need for any additional data structures to traverse the reachable part of the from-space. Normally, to find all the reachable nodes in a graph such as the heap, we might use a depth-first search or breadth-first search algorithm, which would require separate stack or queue data structures. In Cheney's algorithm, however, the to-space doubles as the queue in a breadth-first search, avoiding the implementation complexity and the memory requirements of an explicit queue data structure. This makes the algorithm simpler than a graph traversal with an explicit data structure.

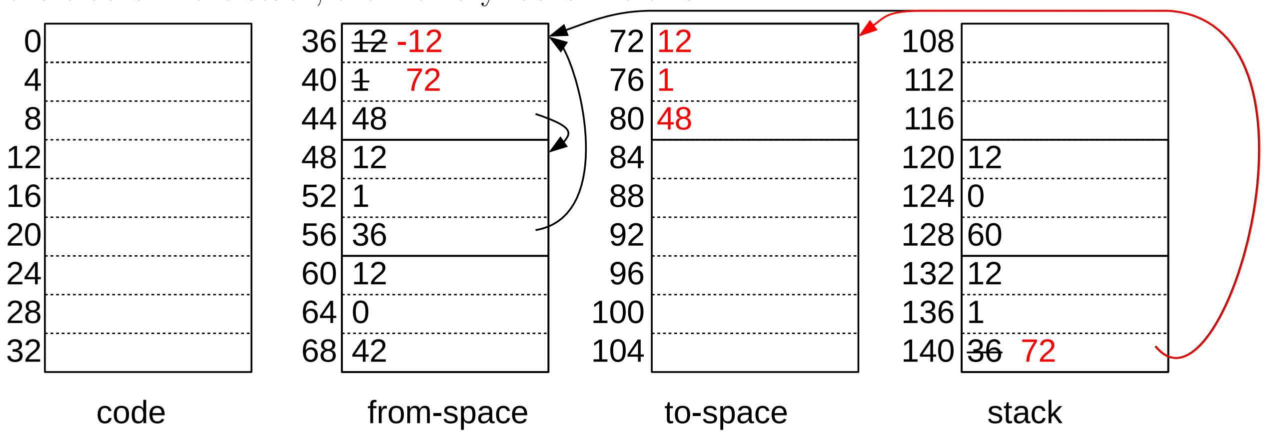
After the end of the second while loop, the roles of the from-space and to-space are swapped by setting `fromSpaceEnd` to the end of the former to-space. The value of `free` is copied to the `heapPointer` so that future allocations can occur in the new from-space after the blocks that have been copied there by the compaction.

11.3.1 An example

Let us illustrate the execution of the copying garbage collector on an example heap. We start with the following memory configuration:

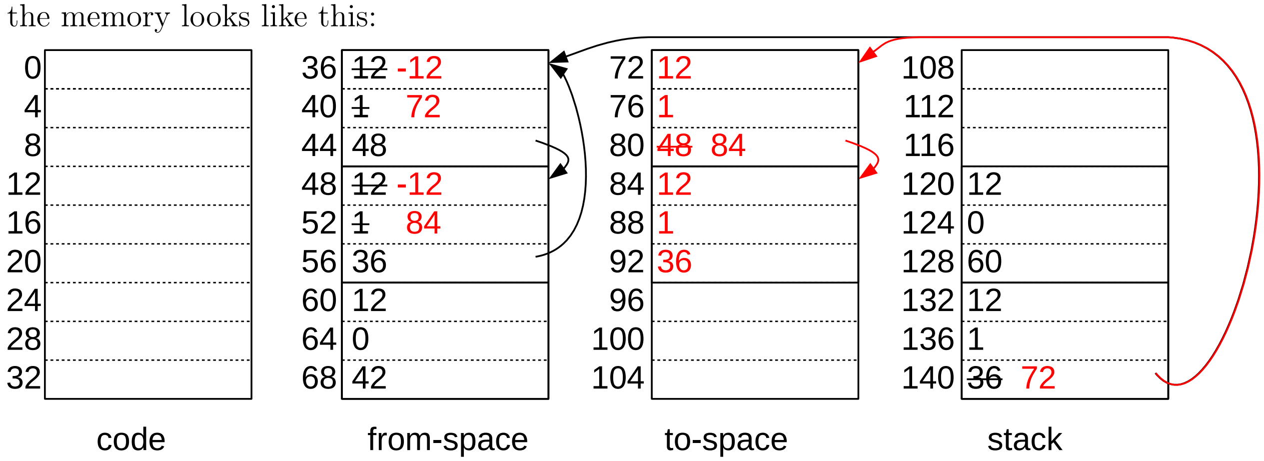


There are two blocks on the stack. Although the first block contains the value 60, which could be a memory address, it is not interpreted as an address but as an integer because the number of addresses in the block, indicated by the second word of the block, is 0. The second block on the stack contains one address, the address 36. After the first while loop processes the blocks in the stack, the memory looks like this:



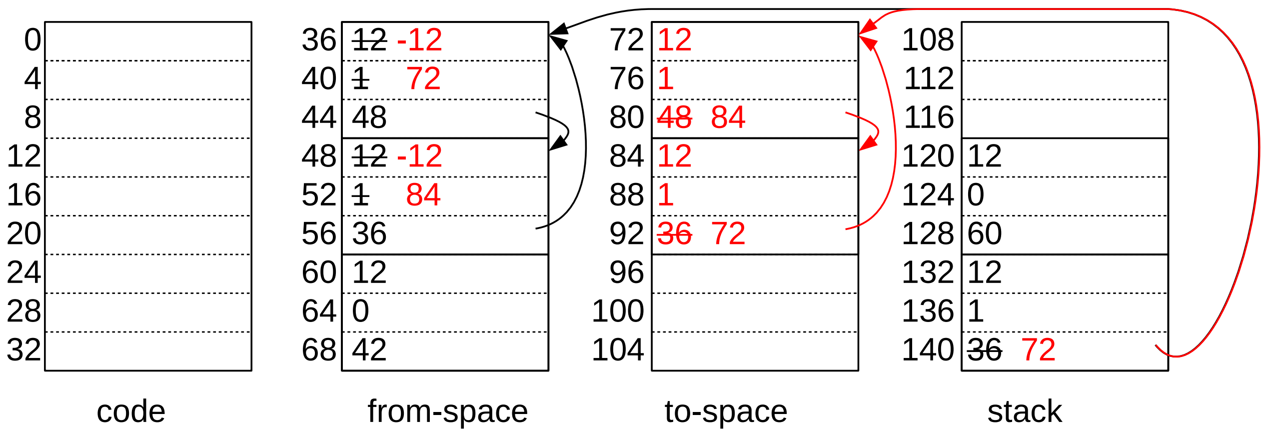
The block from address 36 in the from-space has been copied to address 72 in the to-space. In the from-space, the size of the block has been made negative and the forwarding address 72 has been written into the second word of the original block. Finally, in the stack, the address 36 has been replaced with the new address 72. Notice that the pointer in the copied block, 48, still points into the from-space.

The second loop then begins to scan the to-space. After it has scanned the copied block, the memory looks like this:



Since the block in the to-space contained the address 48, the block at address 48 in the from-space has been copied to a new address in the to-space, 84. In the from-space, the size has been made negative and the forwarding address 84 has been stored in the original block.

Although the `scan` pointer has advanced to where the `free` pointer used to be, the `free` pointer also advanced when the second block was copied, so the second loop is not yet finished. It still needs to scan the second block in the to-space. After it does so, the memory looks like this:



When scanning the second block, `forwardPtrs` calls `copy` on the address 36. The `copy` procedure notices that the block at address 36 has a negative size, indicating that it has already been copied, so it does not copy it again, but instead just returns the forwarding address from its second word, 72. The address 36 in the second block in the to-space is thus changed to the forwarding address 72. The cycle of the two blocks that was in the from-space is reproduced in the to-space.

At this point, both the `scan` and `free` pointers are at address 96, so the collection finishes. The block at address 60, which was not reachable in the original heap, has not been copied and is abandoned in the old from-space. The from-space and to-space are then switched and allocation can continue at the `free` pointer, address 96.

11.3.2 Properties of the garbage collector

In the copying collector, allocation usually takes constant time and the constant is small, since allocation needs to only increment the `heapPointer`. The exception is the rare case in which a garbage collection is triggered. The cost of a collection is proportional to the size of the reachable blocks in the heap, since the copying collector copies only those blocks and never even touches the unreachable blocks. In many practical scenarios, there are significanly fewer reachable blocks than unreachable blocks, so this cost can be significantly lower than searching the whole heap. The overall cost depends on how full the memory is. When a small fraction of the total memory is in use, garbage collections occur less frequently and copy only a small fraction of the total heap. When a large fraction of memory is in use, allocation reaches the end of the from-space sooner, so more garbage collections take place, and each collection has more blocks to copy. There is a tradeoff between time and space: a garbage collector can be very fast if the heap is significantly larger than the space actually in use, and becomes much slower when the heap is close to full. For those interested in additional reading about the performance tradeoffs of many different memory management strategies, see the classic paper A unified theory of garbage collection by David Bacon, Perry Cheng, and V. T. Rajan.

One problem with the algorithm that we have discussed is that it allows only half the heap to be used for allocating blocks, wasting half the memory. Although real-world collectors are based on the same basic principles, they are much less wasteful. Memory is generally divided into more than two spaces of various sizes and a collection needs only one of those many spaces to be empty. The use of more than two spaces requires more complicated algorithms that need to keep track of pointers that cross from one space to another.

The use of multiple spaces makes garbage collection even more efficient because blocks that stay reachable for a long time can migrate to spaces that are garbage collected less often, while in frequently collected spaces, only a small fraction of the blocks are still reachable by the time of a collection, making those collections fast. This division of spaces into old, rarely collected ones and young, frequently collected ones is called *generational garbage collection*.