Coffee into Bugs: Libc from Scratch

Part 1: Malloc and Free

Dave Allison

*“A software engineer is a device for converting coffee into bugs”*

I’m writing my own version of *libc* (the standard C library) from scratch for my 6502 project, a small 6502 based computer and with a C compiler. This is the first part of a series of articles describing how to write a simple version of *libc*.

All this code will be in C (the C99 standard) and the hope is that it is useful as a learning tool for those wanting to learn how to program in C, or those writing their own libc. This is all my own work and I have not copied anything from anywhere else. It’s also not related by my employer.

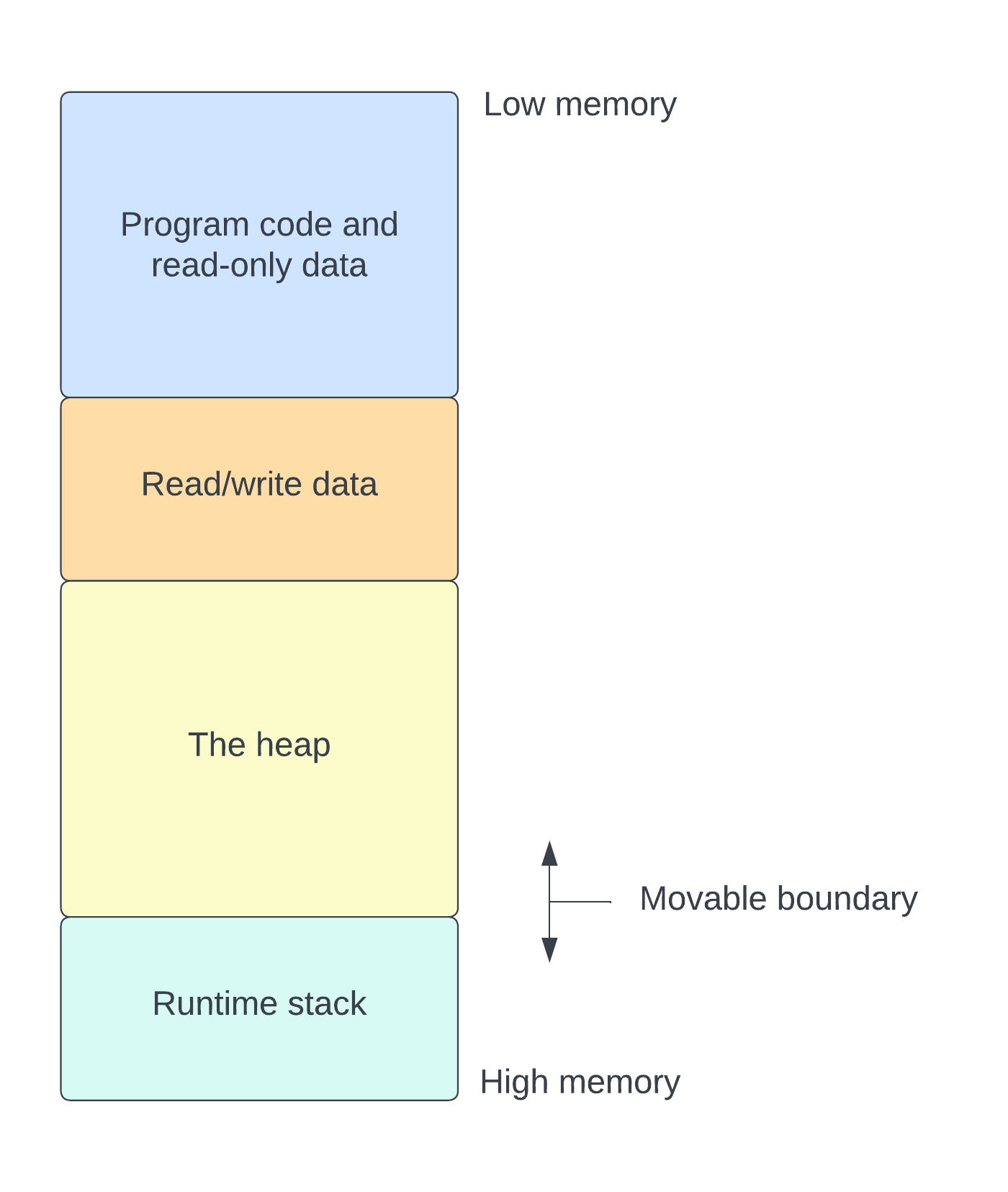
This article will present the functions ***malloc***, ***free***, ***realloc*** and ***calloc*** – the C library’s memory allocation facilities.

# The Program Heap

Although it’s called a *heap*, the name’s not entirely accurate. Rather than adhering to the algorithmic definition of a heap (<https://en.wikipedia.org/wiki/Heap_(data_structure)>), the program heap is just a bunch of memory available to the program that may or may not be arranged in some kind of data structure. In a typical memory layout, the heap is located between the top of the program’s data segment and the bottom of the program stack. With the advent of randomized virtual addresses this may not always be true though.

On a 6502, the heap will be located just above program’s variables and extend up to the end of RAM. Typically, a 6502 computer will have some ROM at the high addresses (above 0xc000 for example) so the heap ends just below that.

Here’s what a typical running program would look like in memory.



The linker[[1]](#footnote-1) arranges the program code and data at the lower addresses with the data being located above the code. Normally, the code will be in read-only memory. The program read/write data contains the static variables and is located above the code. The program runtime stack starts at the top of memory and usually grows downwards towards the top of the read/write data. The size of the stack is determined by how many function stack frames are pushed onto it at any time.[[2]](#footnote-2)

Lying between the top of the stack (the lower address in memory) and the top of the read/write data is available memory and this is where the heap can be found. The upper boundary of the heap can be fixed or variable. On an operating system with virtual memory, it’s usually allocated in chunks and expanded as necessary. On a 6502 it is fixed size.

A C program provides a couple of functions that can be used to allocate memory from the heap and return it when it’s no longer needed. The most basic of them are called ***malloc*** and ***free***. They work by arranging the heap into a data structure that keeps track of free blocks of memory. When we need some memory, the *malloc* function will look for a free block and allocate it. When done with the memory, the *free* function will put the block back into the data structure of free blocks.

# Malloc and free

Let’s look at how to write the *malloc* and *free* functions. We’ll cover *realloc* and *calloc* later.

Rather than attempt to write a highly optimized heap-allocator, let’s keep it as simple as possible. The first time I saw *malloc* implemented was in the first edition of Kernighan and Ritchie’s “*The C Programming Language*” back in 1980’s. I remember spending hours poring over the source code trying to understand it. I eventually got there.

The first thing we need is a data structure to hold free *blocks*. A simple singly linked list is a good way to do this. Each free block is a contiguous sequence of bytes with a header at its lowest address. The header tells us how many bytes are in the block and has a pointer to the next block in the list.

The header for a free block looks like this:

typedef struct FreeBlockHeader {

size\_t length; // Length including header.

struct FreeBlockHeader\* next;

} FreeBlockHeader;

The *FreeBlockHeader* contains a *length* indicator that has type *size\_t*. I’m sure you are aware that *size\_t* is a type that varies depending on the architecture of the processor being used. For an x86\_64 processor, it’s 64 bits long, but for a 6502 it will be only 16 bits. After the length comes a pointer to another *FreeBlockHeader* called *next*. This will either be NULL or will contain the address of the next free block in the linked list.

Free blocks are ordered by ascending address, meaning that the *next* pointer always points to a block that has a higher address than the current one. In the following diagram, there are 3 free blocks, with each one linking to one at a higher address. The length field contains the length of the entire free block’s data, including the length of the *FreeBlockHeader* itself. The space between free blocks is occupied by allocated blocks.

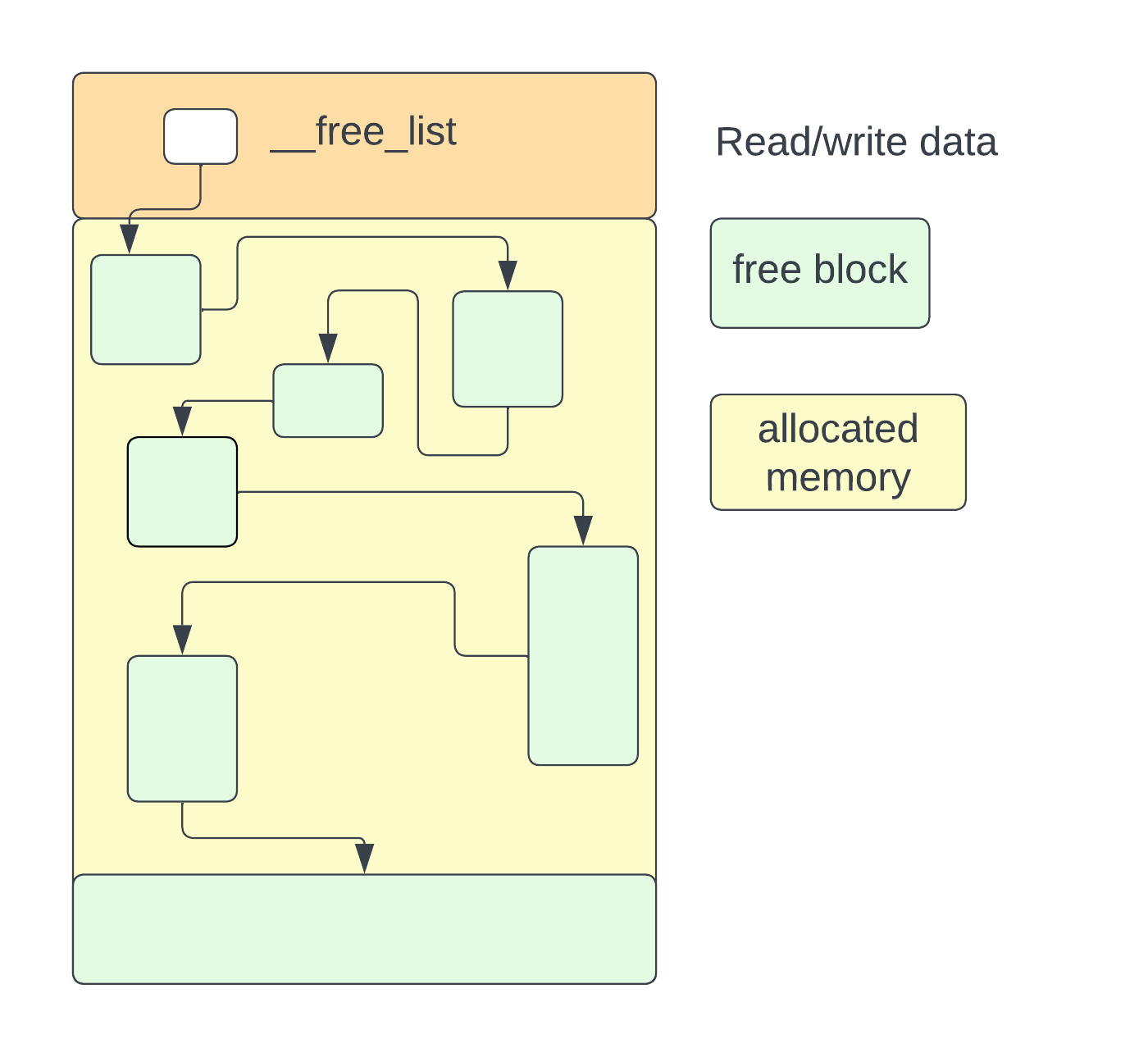
Diagram

Description automatically generated

Initially, when the heap is empty and no blocks have been allocated, there will be one free block, with its length set to the entire length of the heap. We need a place to store the address of the first free block. Let’s put this in the program’s data segment and call it *\_\_free\_list.* Initially this will be NULL but we will initialize it lazily the first time *malloc* is used.

FreeBlockHeader\* \_\_free\_list;

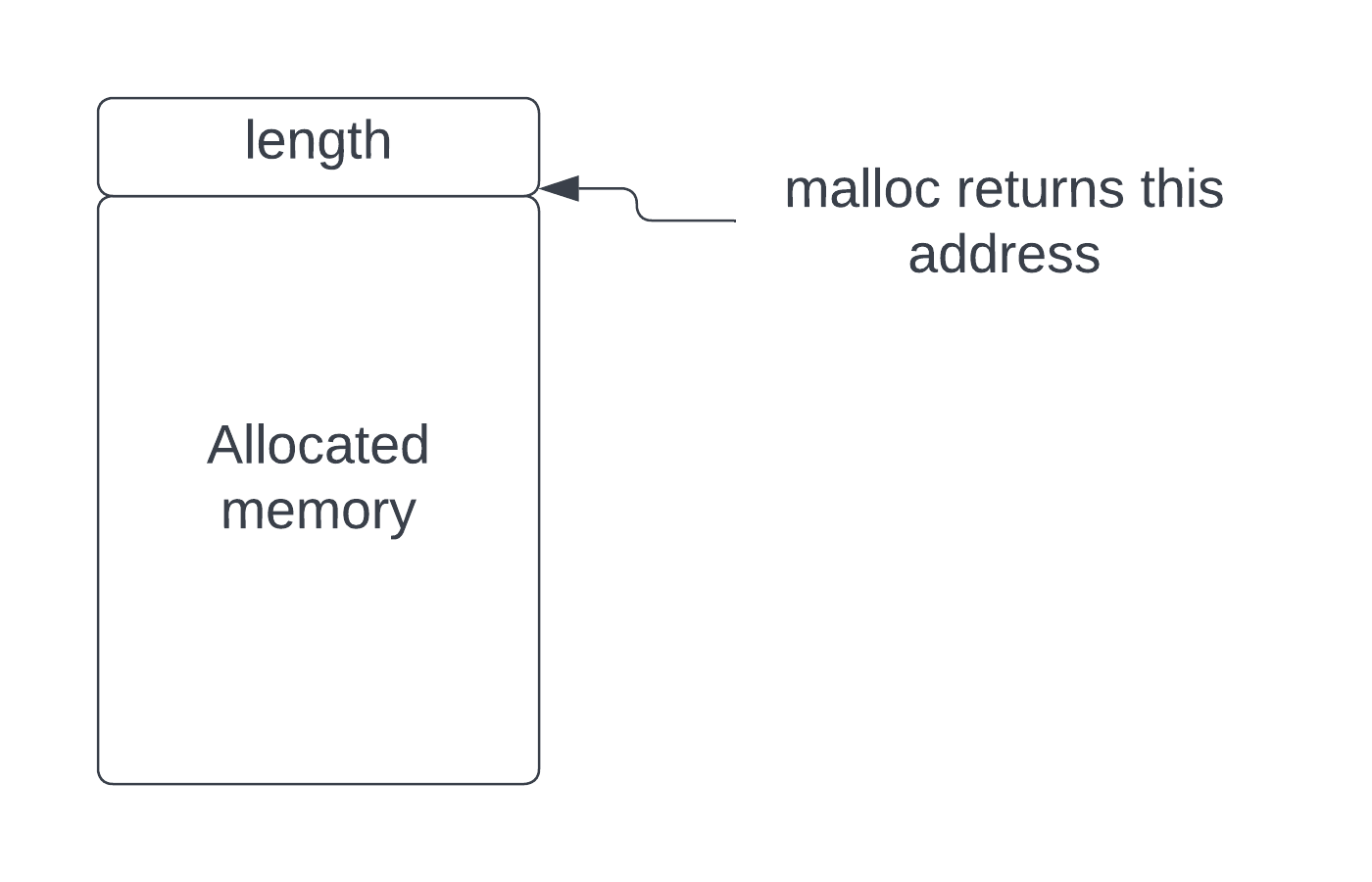
After the program has been running for a while, the following illustration shows the potential state of the program heap. The light green blocks are free and everything else, in yellow, has been allocated. The variable *\_\_free\_list*, in the program’s read/write data points to the first free block with each block linking to the next via its *next* pointer. The free blocks are arranged in order of increasing address.



## Allocating blocks with *malloc*

The *malloc* function will search this linked list looking for a free block that has sufficient space available to accommodate the size it’s being asked for. When it finds a block, it will take some (or all) of the memory from the block, adjusting the linked list to include the unneeded bytes.

When a block is allocated from a free block, the allocation needs to keep track of how many bytes were allocated. This is done by using a couple of extra bytes at the beginning of the block to hold the length of the block. The address returned by *malloc* is just beyond this length marker.



The *length* marker contains the length of the block requested by *malloc* and does not include its own length.

Let’s take a look at the *malloc* function.

void\* malloc(size\_t n) {

if (\_\_free\_list == NULL) {

InitFreeList();

}

n = AlignSize(n); // Aligned.

size\_t full\_length = n + sizeof(size\_t);

FreeBlockHeader\* free\_block = \_\_free\_list;

FreeBlockHeader\* prev = NULL;

while (free\_block != NULL) {

if (free\_block->length >= full\_length) {

// Free block is big enough. If there's enough room for the free block

// header, take the lower part of the free block and keep the remainder

// in the free list.

n = TakeStartOfFreeBlock(free\_block, n, full\_length, prev);

size\_t\* newblock = (size\_t\*)free\_block; // Start of new block.

\*newblock = n; // Size of allocated block.

return (void\*)((uintptr\_t)free\_block + sizeof(size\_t));

}

prev = free\_block;

free\_block = free\_block->next;

if (free\_block == NULL) {

// Expand the heap if we can.

free\_block = ExpandHeap();

}

}

return NULL;

}

The first thing done is to lazily initialize the *\_\_free\_list* pointer if it’s NULL. This is done through a call to *InitFreeList*, defined, for the 6502, as:

#define MEMTOP 0xc000

extern char \_end[];

static void InitFreeList() {

\_\_free\_list = (FreeBlockHeader\*)\_end;

\_\_free\_list->length = MEMTOP - (int)\_end;

\_\_free\_list->next = NULL;

}

This implementation of this is for a 6502 processor where the free memory extends from the linker-defined *\_end* variable to the fixed address 0xc000. The linker defines *\_end* as the address just beyond the top of the program’s read/write data area. For other processors with an operating system, a call to *mmap* may be used to get some memory from the OS.

You can see that the *InitFreeList* function puts a *FreeBlockHeader* at the address specified by *\_end* and initializes it to the full size of the memory. It also sets *next* to NULL.

Diagram

Description automatically generated

After possibly initializing the free list, the *malloc* function calculates the actual number of bytes needed to be allocated. It is passed a parameter specifying how many bytes are needed but there might be alignment requirements for the memory (aligned to a 4 byte boundary on an ARM processor, for example) and we will need space for the allocated block header (the length marker).

The *AlignSize* function aligns the length to a multiple of *sizeof(size\_t)* elements as follows:

static size\_t AlignSize(size\_t s) {

#ifdef ALIGN

return (s + (sizeof(size\_t) - 1)) & ~(sizeof(size\_t) - 1);

#else

return s;

#endif

}

For a 6502, the ALIGN macro should not be defined so we won’t do any alignment.

After calculating the required length, *malloc* then begins a search of the free list looking for a free block that has sufficient space available to accommodate its needs. It keeps track of the current block in ***free\_block*** and the previous block in ***prev***. For each block it examines, it looks at its length and tests it against the size it needs. For a block that has enough space we look to see how much space will be left after we take the amount we need. If we reach the end of the free list without finding a block of sufficient size, we have two options:

1. Give up and return NULL
2. Try to allocate more memory for the heap from the operating system.

For a 6502 we have a fixed size heap, so option 1 is the only option we have. On another processor with a real operating system, we could call the ***sbrk***() or ***mmap***() functions to allocate more, not necessarily contiguous, memory for the heap. The function *ExpandHeap* will accomplish the additional memory allocation if it can, adding it to the end of the free list.

If we have more than *sizeof(FreeBlockHeader)* bytes available after allocation, we have enough to keep the remainder as a free block, otherwise we need to take the whole block. If we decide to remove the block, the previous block’s *next* pointer is modified to point to the chosen block’s *next* pointer. If there was no previous block, we are looking at the first block, so we modify *\_\_free\_list* to point to the next block.

If the free block is big enough to accommodate *malloc’s* needs and also have enough space left to remain free, we call ***TakeStartOfFreeBlock*** to take the bottom part (lower address) of the block for *malloc* and place a *FreeBlockHeader* at the beginning of the unneeded part of the block.

The *TakeStartOfFreeBlock* function is fairly simple and is defined as follows.

static size\_t TakeStartOfFreeBlock(FreeBlockHeader\* block, size\_t num\_bytes,

size\_t full\_length, FreeBlockHeader\* prev) {

assert(block->length > full\_length);

size\_t rem = block->length - full\_length;

if (rem >= sizeof(FreeBlockHeader)) {

FreeBlockHeader\* next = (FreeBlockHeader\*)((uintptr\_t)block +

full\_length);

next->length = rem;

next->next = block->next;

// Remove from free list.

if (prev == NULL) {

// No previous free block, this becomes the first in the list.

\_\_free\_list = next;

} else {

// Chain to previous free block.

prev->next = next;

}

} else {

// We have less than sizeof(FreeBlockHeader)

// Take whole block.

if (prev == NULL) {

\_\_free\_list = block->next;

} else {

prev->next = block->next;

}

// Allocate whole block.

num\_bytes = block->length - sizeof(size\_t);

}

return num\_bytes;

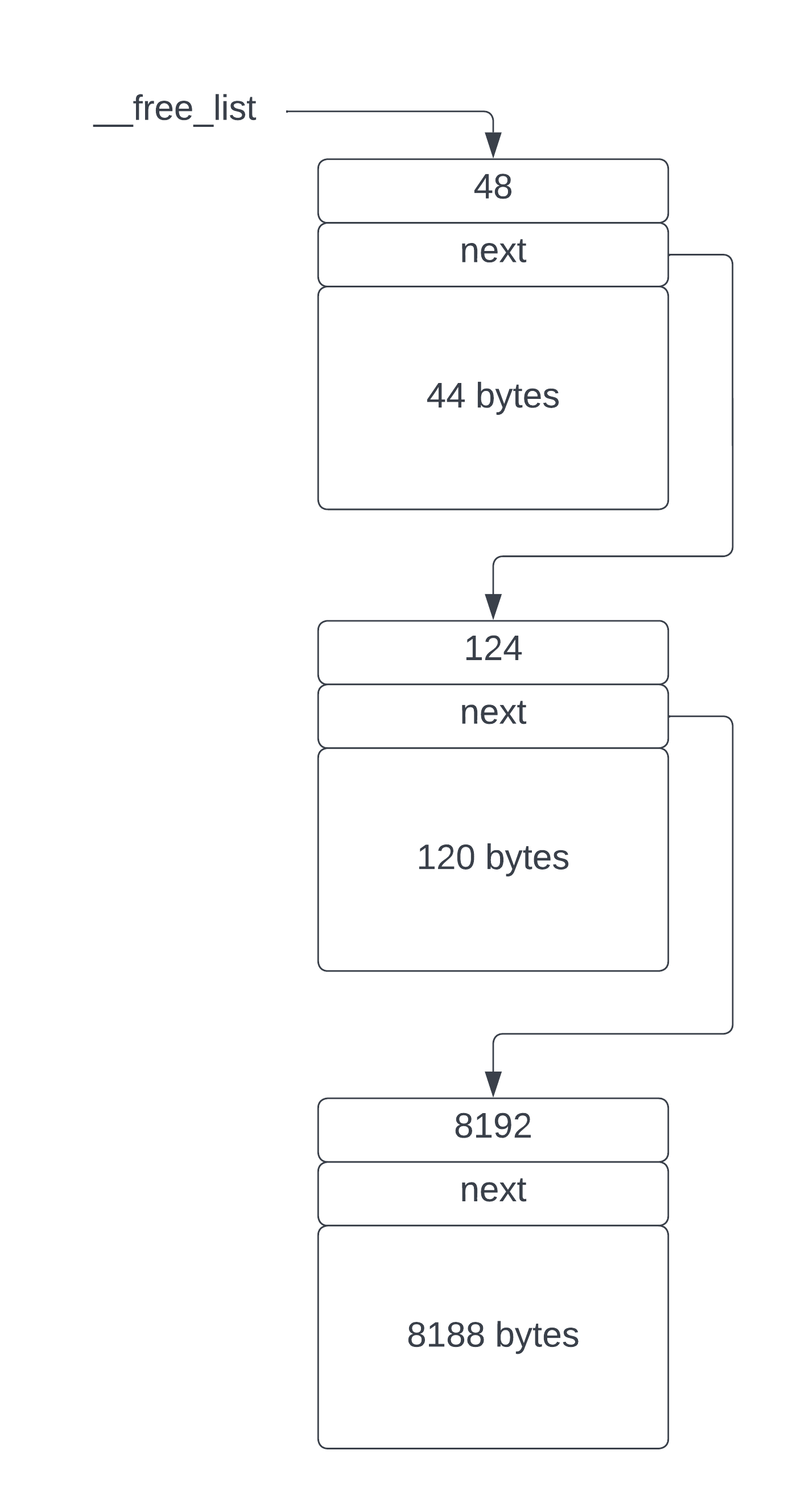
}

After first asserting that the free block is indeed big enough, we calculate the number of bytes that will remain after we take the allocated block from it, in the variable ***rem***. If there will sufficient remaining bytes free to accommodate a *FreeBlockHeader*, it places a new *FreeBlockHeader* at an offset into the free block that leaves ***full\_length*** bytes before it that will be used for the allocated block.

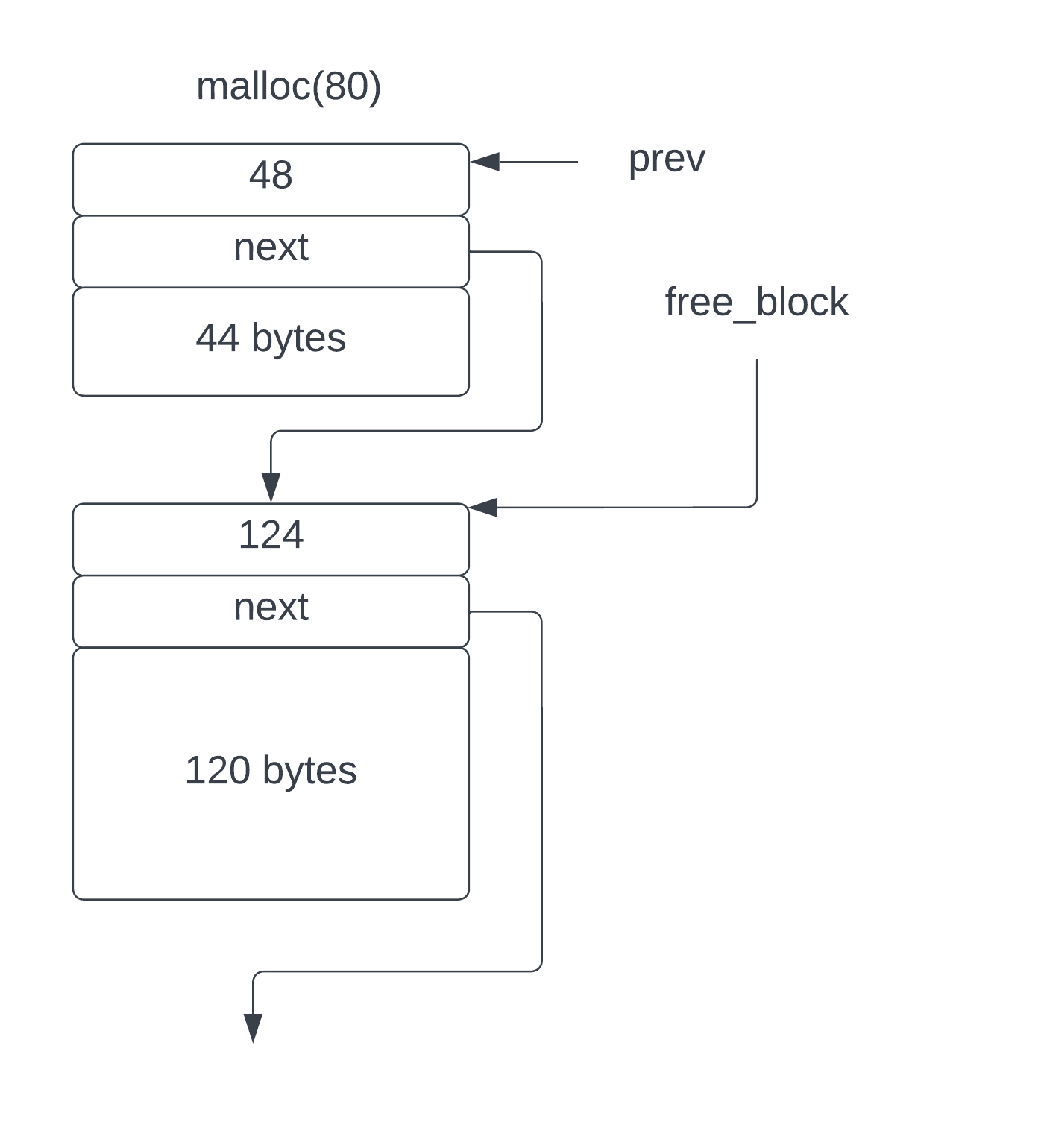
If there are not enough bytes available to make a free block after ***full\_length*** bytes are taken, we just take the whole block.

The function returns the number of bytes that will be written into the length marker located just before the allocated address.

Let’s take an example. The diagram below shows the current state of the free list with 3 free blocks, of sizes 48, 124 and 8192 bytes.

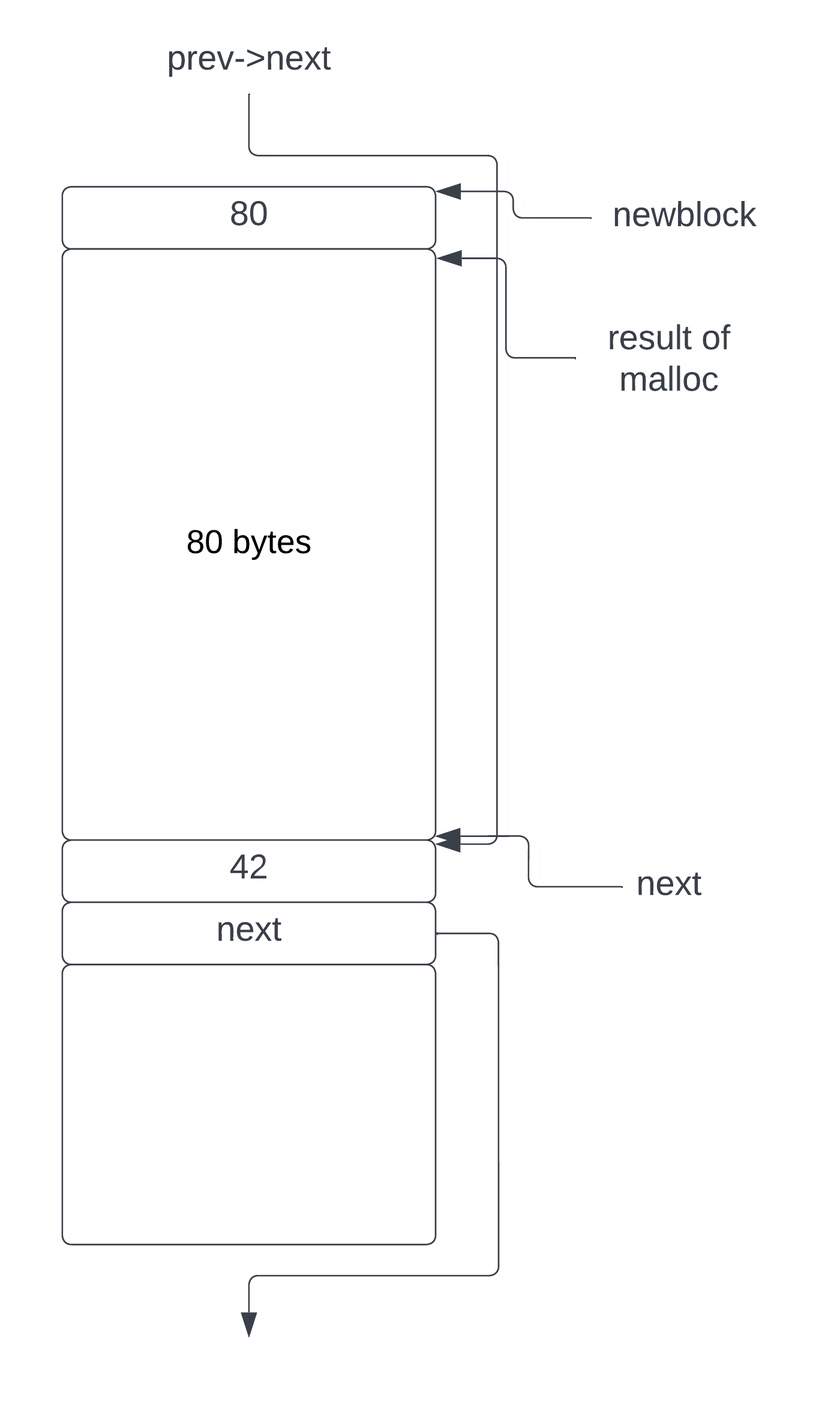


When *malloc(80)* is called we search the free list and find the first block with enough space, in this case the block with 124 bytes available. The previous block has 48 bytes and is too small. The *prev* variable points to the block of 48 bytes and *b* points to the block we want to use.

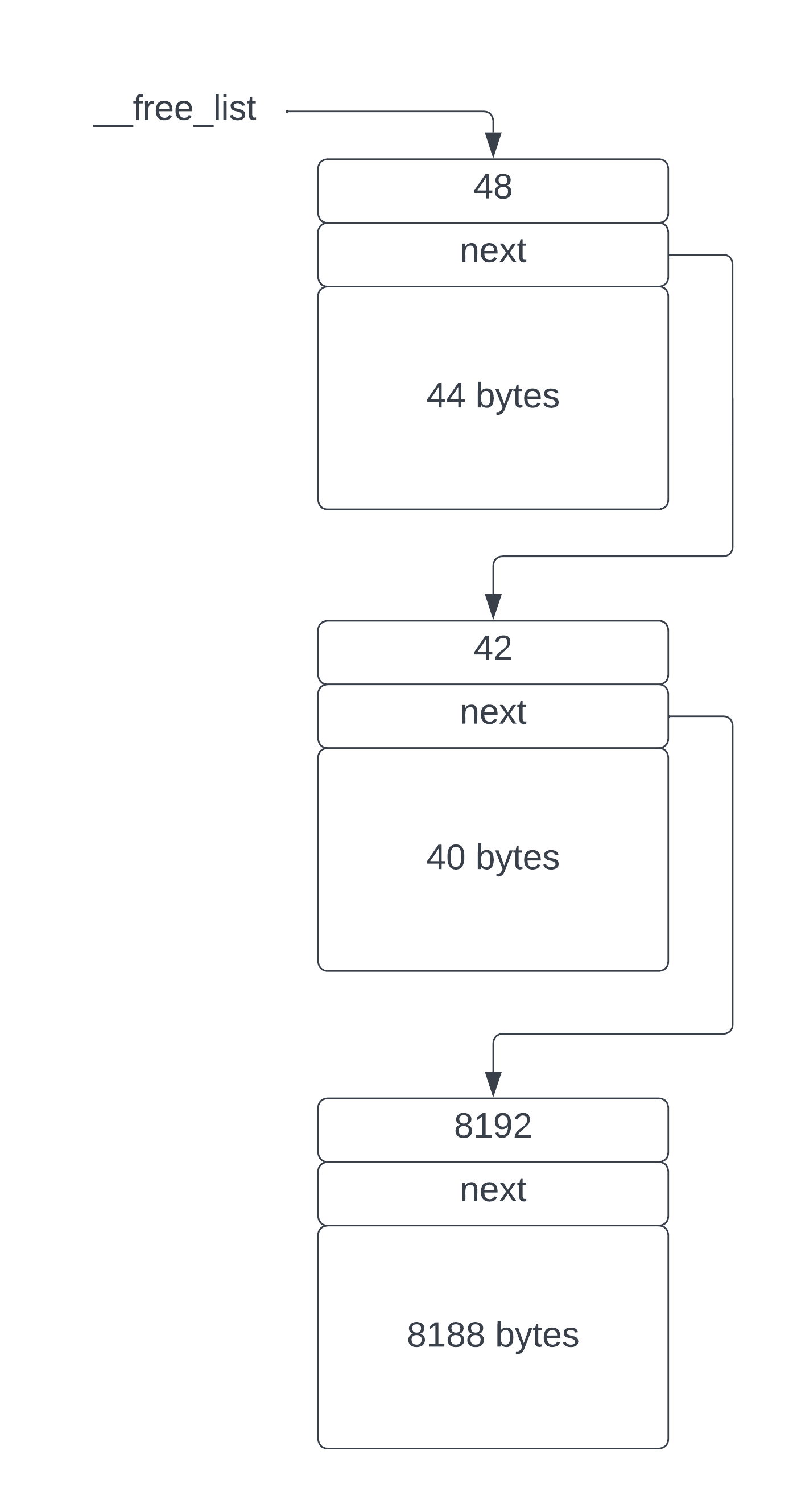


Let’s zoom in on the 124 byte block to see what happens to it as *malloc* does its thing. For this example, let’s assume a 6502 architecture where the size of a pointer, and the size of *size\_t* are both 2 bytes. This means that the size of the *FreeBlockHeader* is 4 bytes and the length indicator for an allocated block is 2 bytes.

We’ve asked for 80 bytes therefore *malloc* needs 82 bytes of memory to accommodate the length indicator. The block we’ve found has 124 bytes available, so we will take 82 of them, leaving 42 bytes (124-82=42) that can remain free. We place a *FreeBlockHeader* at offset 82 into the block and set its *length* to 42 and copy the *next* value from the original *FreeBlockHeader* at the start of the block. We then modify the *next* pointer from the previous block, *prev*, to point to this new *FreeBlockHeader*. Then the length of the allocated block, 80, is written into the first 2 bytes of the newly allocated memory and *malloc* returns the address immediately following that. The effect is that the size of the free block has been reduced by 82 bytes and the start of the block is the allocated memory.



After *malloc* returns, the free list has been modified to be:



For the case where the size of the free block is big enough for the memory needed but isn’t big enough to contain a *FreeBlockHeader*, the whole block is taken and removed from the free list. If this happens, the size of the allocated block is the full size of the free block and is thus more than asked for by the *malloc* call.

After many calls to *malloc*, we may end up with a bunch of small blocks that only have a *FreeBlockHeader* or maybe a few bytes more. If all calls to *malloc* ask for more memory than is held in these blocks they cannot be used and we will have some unusable memory in the heap – blocks that are too small to be useful for the allocation pattern in the program. This is called ***heap fragmentation*** and is always a danger of memory allocation. Advanced algorithms have been developed to reduce heap fragmentation but it’s impossible to eliminate completely without doing some kind of heap compaction and that isn’t usually possible in C[[3]](#footnote-3).

## Freeing blocks using *free*

Now that we have the ability to allocate memory, let’s look at how to free it. The ***free*** function takes a block that has been allocated by *malloc* and inserts it into the free list in the correct place, coalescing adjacent free blocks if possible. Recall that the free list is ordered by increasing address, so in order to free a block we need to search the free list looking for a free block that has an address less than the block being freed. Once we find it, the block being freed is inserted before it in the list.

However, the simplistic approach of just inserting every new free block into the list would mean that our free list grows over time as every call to *free* would result in another entry in the free list. This would slow down any searches of the free list, but can be ameliorated by realizing that if we have two adjacent free blocks, they can be merged into one free block.

Let’s look at the code for ***free***.

void free(void\* p) {

// An allocated block has its length immediately before its address.

size\_t alloc\_length = \*((size\_t\*)p - 1); // Length of allocated block.

// Point to real start of allocated block.

FreeBlockHeader\* alloc\_header =

(FreeBlockHeader\*)((uintptr\_t)p - sizeof(size\_t));

// Insert into free list by searching for the appropriate point in memory

// sorted by address.

FreeBlockHeader\* free\_block = \_\_free\_list;

if (free\_block == NULL) {

// No free list, this block becomes the only block.

alloc\_header->length = alloc\_length + sizeof(size\_t);

alloc\_header->next = NULL;

\_\_free\_list = alloc\_header;

return;

}

FreeBlockHeader\* prev = NULL;

while (free\_block != NULL) {

FreeBlockHeader\*\* next\_ptr;

if (prev == NULL) {

next\_ptr = &\_\_free\_list;

} else {

next\_ptr = &prev->next;

}

// If the current block (b) is at a higher address than t then we know

// that we need to insert t before b.

if (free\_block > alloc\_header) {

// Found a free block after the one being freed.

MergeWithAboveIfPossible(p, alloc\_header, free\_block, next\_ptr,

alloc\_length);

// See if we can merge with prev. If the block just freed is

// immediately contiguous with the previous free block,

// we can merge them,

if (prev != NULL) {

MergeWithBelowIfPossible(alloc\_header, prev);

}

// We're done.

return;

}

// Look at the next free block, keeping track of the previous.

prev = free\_block;

free\_block = free\_block->next;

}

// We reached the end of the free list, insert free block at end.

if (prev != NULL) {

MergeWithBelowIfPossible(alloc\_header, prev);

return;

}

// Can't merge, insert a new free block at end.

InsertNewFreeBlockAtEnd(alloc\_header, prev,

alloc\_header->length + sizeof(FreeBlockHeader\*));

}

The ***free*** function must be passed a block allocated using *malloc*, and therefore there will be length indicator stored immediately before the address it is passed. This code assumes this is the case but advanced implementations add runtime checks to ensure it. The first thing we do is retrieve the length of the allocated block by subtracting *sizeof(size\_t)* bytes from the address passed to free and reading the length from that address. This is placed in the local variable ***alloc\_length***. Then we cast the actual start of the block to a *FreeBlockHeader* and store this in the variable ***alloc\_header***.

It is possible that the program has consumed all the heap in calls to *malloc*, in which case *\_\_free\_list* will be NULL. We check for that and if so, build a new free list out of the block being freed.

We begin a search of the free list starting at *\_\_free\_list*. If we find a free block whose address is greater than ***alloc\_header***, we know that we need to insert the newly freed block just before the found free block in the free list. Just like in *malloc*, we keep track of the previous block in *prev* as we traverse the list.

Once we find the free block we need, in local variable ***free\_block***, we look to see if we can merge it with adjacent blocks.

The ***MergeWithAboveIfPossible*** function checks if the block being freed, in ***alloc\_header***, is adjacent to ***free\_block***, and if so, makes one free block out of them both. Otherwise it just inserts a new free block into the free list before ***free\_block***. Here’s the code to do that.

static void MergeWithAboveIfPossible(FreeBlockHeader\* alloc\_block,

FreeBlockHeader\* alloc\_header,

FreeBlockHeader\* free\_block,

FreeBlockHeader\*\* next\_ptr,

size\_t alloc\_length) {

uintptr\_t alloc\_addr = (uintptr\_t)alloc\_block;

uintptr\_t free\_addr = (uintptr\_t)free\_block;

if (alloc\_addr + alloc\_length == free\_addr) {

// Merge with block above.

alloc\_header->next = free\_block->next;

alloc\_header->length =

alloc\_length + sizeof(FreeBlockHeader\*) + free\_block->length;

\*next\_ptr = alloc\_header;

} else {

// Not adjacent to above; add to free list.

// t points to the allocated block header which has its length set.

alloc\_header->length += sizeof(FreeBlockHeader\*);

alloc\_header->next = free\_block;

\*next\_ptr = alloc\_header;

}

}

After possibly merging the newly freed block with ***free\_block***, we then check if it can be merged with the free block immediately prior to ***free\_block*** in the free list, in the variable ***prev***. This is done by calling ***MergeWithBelowIfPossible***.

static bool MergeWithBelowIfPossible(FreeBlockHeader\* free\_block,

FreeBlockHeader\* prev) {

uintptr\_t prev\_addr = (uintptr\_t)prev;

if (prev\_addr + prev->length == (uintptr\_t)free\_block) {

// Lower block is adjacent.

prev->next = free\_block->next;

prev->length += free\_block->length;

return true;

}

return false;

}

This is a pretty easy operation: just look at the previous block, add its length and check if the address matches the newly freed block. If so, just increase the length of the previous free block and set its *next* pointer to the same as the newly freed block, thus subsuming the newly freed block into the previous free block. If the merge is possible it returns true.

If we reach the end of the free list, it means that the block being freed has a higher address than all other free blocks and needs to be inserted at the free list. Rather than just blindly inserting a new free block, however, we check if it’s exactly at the end of the free list by calling ***MergeWithBelowIfPossible***. If the merge was done, we’re good and done. If not, we insert the new block at the end of the free list using ***InsertNewFreeBockAtEnd***, simply defined as:

static void InsertNewFreeBlockAtEnd(FreeBlockHeader\* free\_block,

FreeBlockHeader\* prev, size\_t length) {

free\_block->length = length;

free\_block->next = NULL;

if (prev == NULL) {

\_\_free\_list = free\_block;

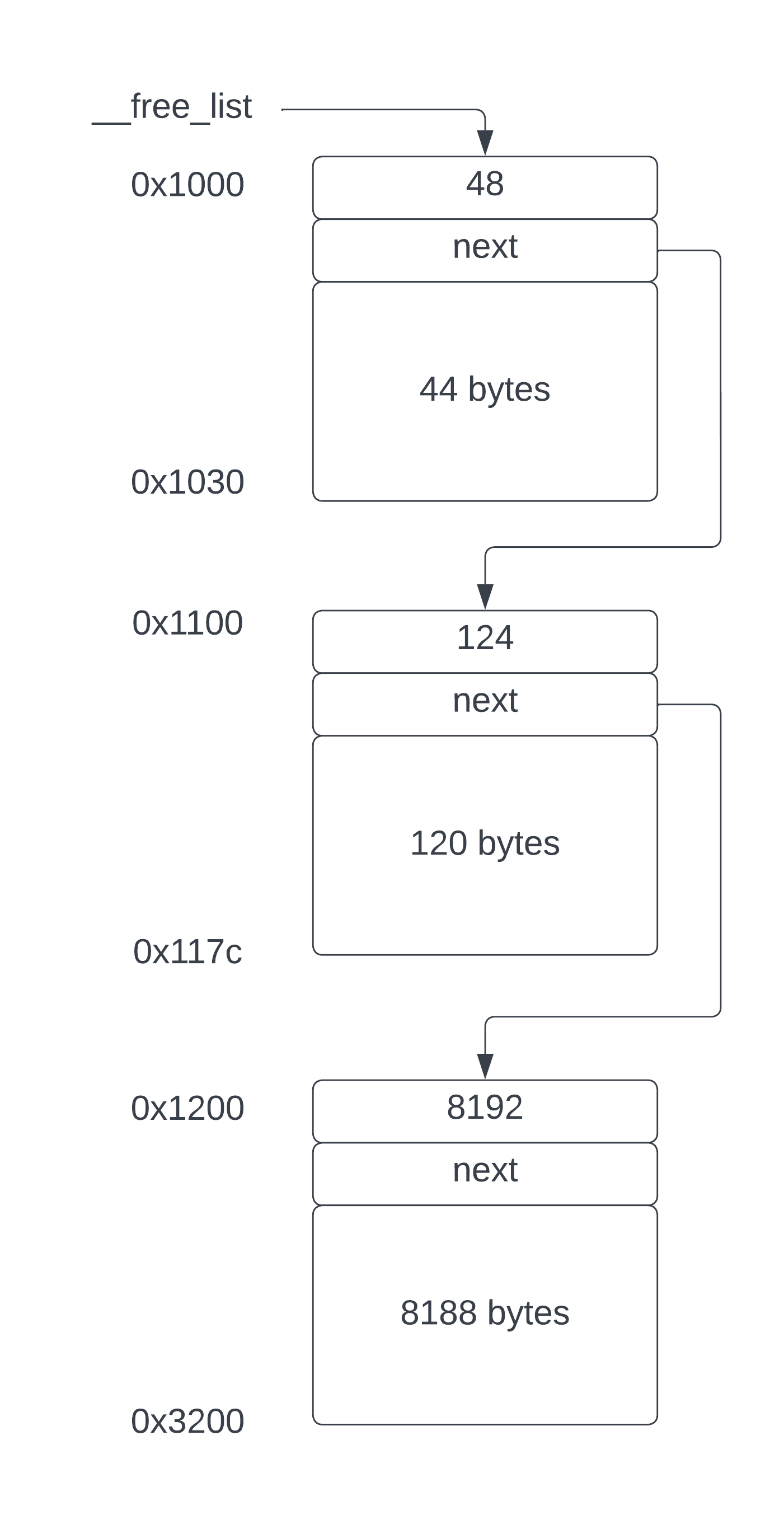
} else {

prev->next = free\_block;

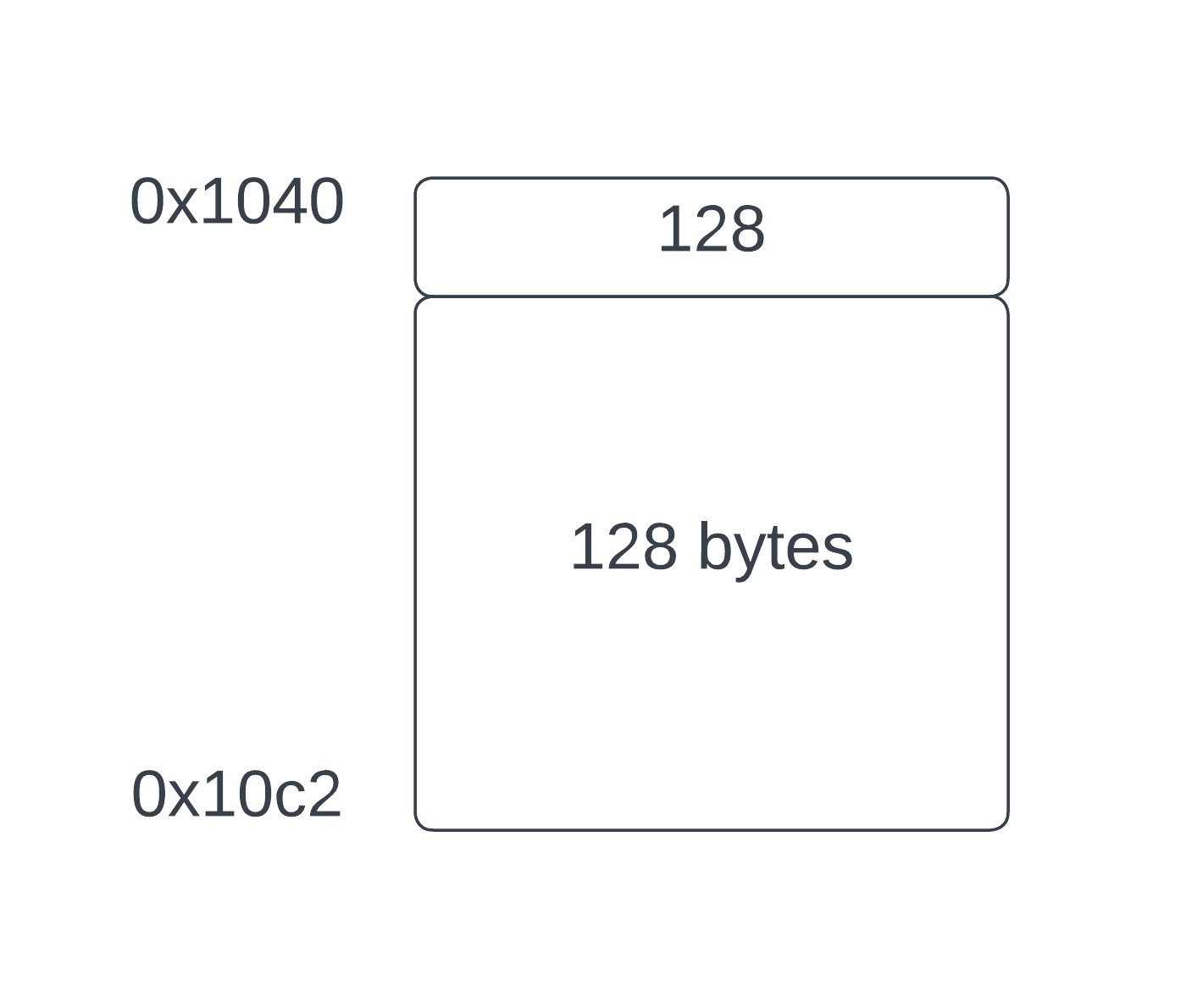
}

}

As an example, consider the same free list we used for the *malloc* example, shown below. Between the free blocks with length 48 and 124 (the first and second blocks) we have allocated blocks. Say we want to free one of those allocated blocks. Let’s also put some addresses onto the blocks. The first block is at address 0x1000, the second at 0x1100 and the third at 0x1200.

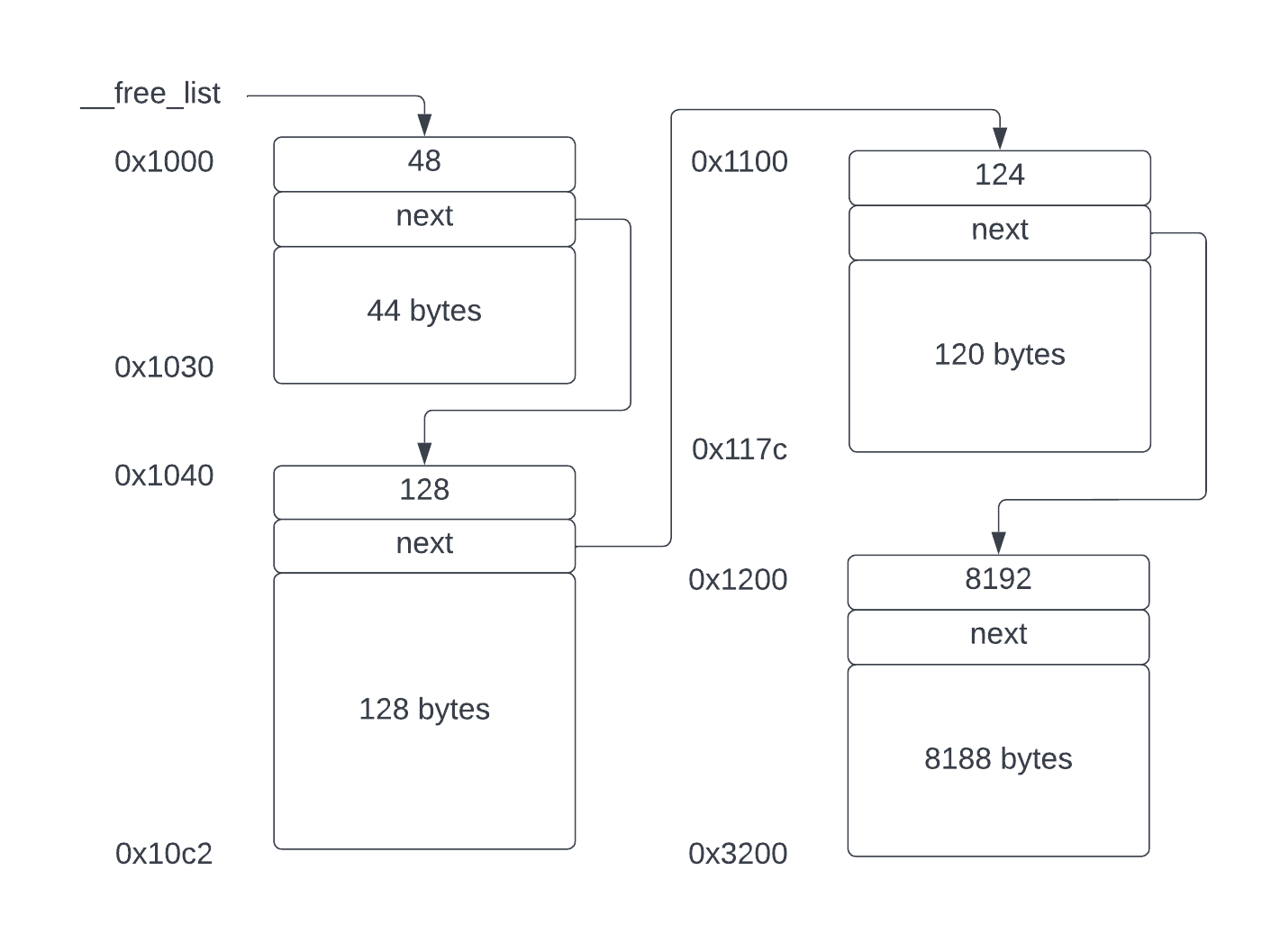


Let’s consider freeing a block of length 128 bytes at address 0x1042. Since *malloc* returns the address after the length indicator, the actual address of the block is 0x1040. The end address of the block is 0x1042 + 128 = 0x10c2.



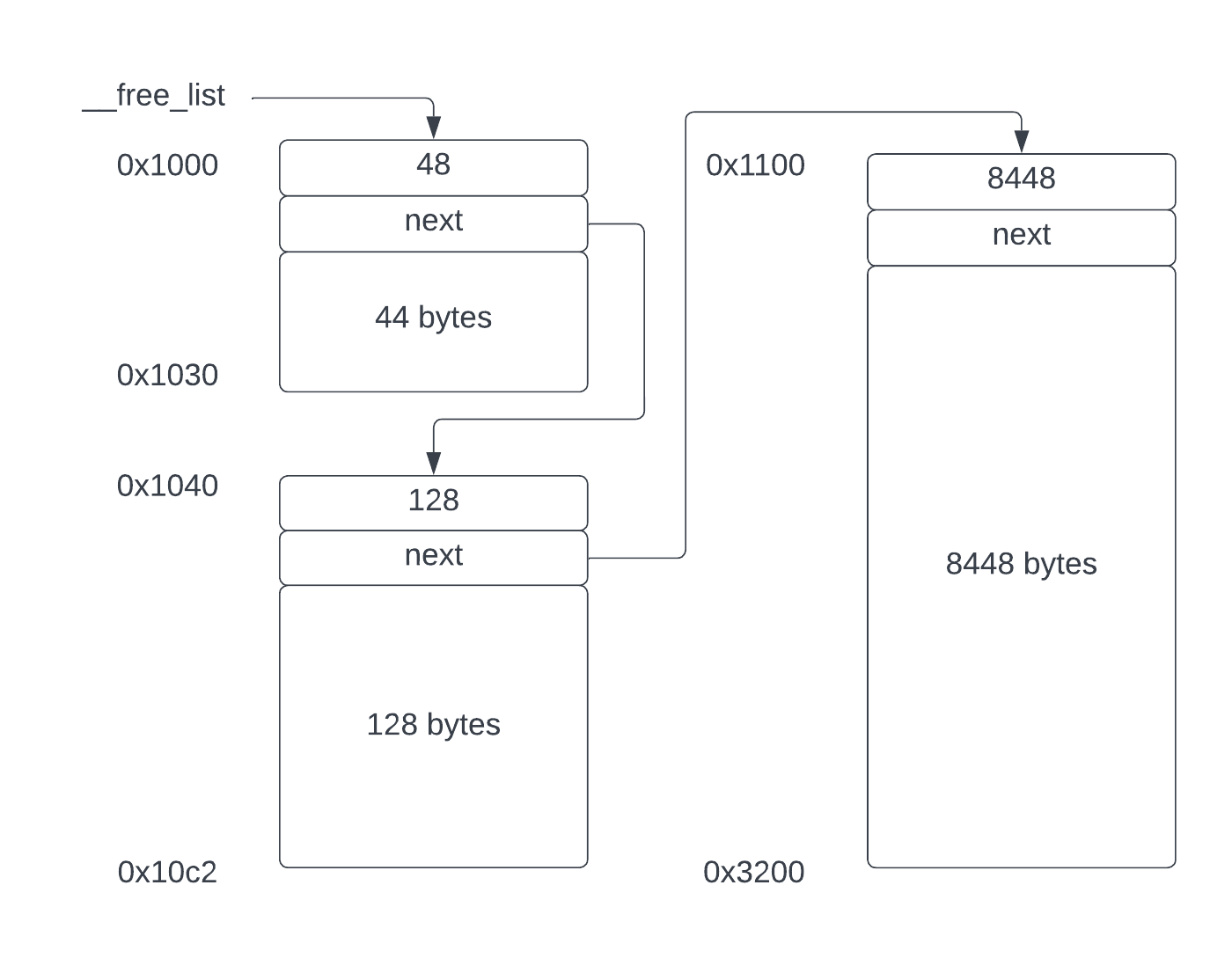
When we search the free list, this block is located between the blocks at 0x1000 and 0x1100. However, the start address of the block, 0x1040 is not adjacent to the block at 0x1000, which ends at 0x1030, and it is also not adjacent to the block at 0x1100, so it needs to be inserted as a free block between the first and second blocks.

Here’s what the free list looks like after inserting the block.



Now let’s consider inserting an allocated block at address 0x117e with length 130 bytes. You can see where I’m going with this. The actual address of the allocated block is 0x117e – 2 = 0x117c, which is immediately after the block at 0x1100. And the length of the allocated block puts its end at 0x1200 (0x117c + 130 + 2 = 0x1200). Therefore, this block fits exactly between the free blocks at 0x1100 and 0x1200.

The *free* function will find block 0x1200 because it’s the first block whose address is greater than 0x117c. The calculation alloc\_addr + alloc\_length == free\_addr will be true (0x117e + 130 == 0x1200) so it will merge the newly freed block with the free block at 0x1200. Then it will check if it can merge with the previous block at 0x1100. The condition prev\_addr + prev->length == (uintptr\_t)free\_block also yields the value true (0x1100 + 124 == 0x117c), so the block at 0x1100 will consume the newly freed block at 0x117c, resulting in a big block of length 8448 at address 0x1100.



As the program runs, calling *malloc* and *free*, if all the allocated blocks are freed, the free list will be returned to one big block that has the size of the full heap.

# Reallocating a block

Although the use of *malloc* and *free* is very common in C programs, their lesser used cousin, ***realloc***, allows an allocated block to be changed in size. The naïve way to change the size of an allocated block is to allocate a new block of the new size, copy the contents of the old block to the new one and free up the old block. However, this is inefficient and can result in heap fragmentation, leading to a waste of memory and increased CPU time since the free lists get longer.

A better approach is to manipulate the free list itself, expanding or contracting the block as necessary, and only copying the block if there is no other option.

The change to a block’s size can either increase it or decrease it. If the block is shrinking, the part of the block no longer needed is returned to the free list. If the block is growing, we might be able to shrink an adjacent free block and take some of its memory for the growing block.

The *realloc* function takes two arguments:

1. A pointer to a block allocated by malloc (or NULL)
2. The new size of the block.

If the first argument is NULL, the *realloc* function reduces to *malloc*, otherwise it changes the size of the block referred to.

Here’s the code for the *realloc* function. It’s a little longer than *malloc* and *free* but still not difficult.

void\* realloc(void\* p, size\_t n) {

if (p == NULL) {

// No block to realloc, just call malloc.

return malloc(n);

}

// The allocated block has its length immediately prior to its address.

size\_t\* len\_ptr = (size\_t\*)p - 1;

size\_t orig\_length = \*len\_ptr;

FreeBlockHeader\* alloc\_block =

(FreeBlockHeader\*)((uintptr\_t)p - sizeof(size\_t));

uintptr\_t alloc\_addr = (uintptr\_t)p;

n = AlignSize(n); // Aligned.

if (n == orig\_length) {

// Same size as current block, nothing to do.

return p;

}

if (n < orig\_length) {

// Decreasing in size. Free the remaining part.

ShrinkBlock(alloc\_block, orig\_length, n, len\_ptr);

return p;

}

// Increasing in size.

// See if there's a free block immediately following allocated block.

FreeBlockHeader\* free\_block = \_\_free\_list;

FreeBlockHeader\* prev = NULL;

FreeBlockHeader\* prev\_prev = NULL;

while (free\_block != NULL) {

FreeBlockHeader\*\* next\_ptr;

if (prev == NULL) {

next\_ptr = &\_\_free\_list;

} else {

next\_ptr = &prev->next;

}

if (free\_block > alloc\_block) {

uintptr\_t free\_addr = (uintptr\_t)free\_block;

size\_t diff = n - orig\_length;

if (alloc\_addr + orig\_length == free\_addr) {

// There is a free block above. See if has enough space.

if (free\_block->length > diff) {

ssize\_t freelen = free\_block->length - diff;

if (freelen > sizeof(FreeBlockHeader)) {

ExpandIntoFreeBlockAbove(free\_block, n, diff, freelen, len\_ptr,

next\_ptr);

return p;

}

}

}

// Check for free block adjacent below.

if (prev != NULL) {

uintptr\_t prev\_addr = (uintptr\_t)prev;

if (prev\_addr + prev->length == (uintptr\_t)alloc\_block &&

prev->length >= diff) {

// Previous free block is adjacent and has enough space in it.

// Use start of new block as new address and place FreeBlockHeader

// at newly free part.

return MergeWithFreeBlockBelow(p, prev\_prev, prev, n, orig\_length);

}

// Block doesn't have enough space.

break;

}

}

prev\_prev = prev;

prev = free\_block;

free\_block = free\_block->next;

}

// If we get here we can't reuse the existing block. We allocate a new

// one, copy the memory and free the old block. We are guaranteed that

// the new block is larger than the original one since if it was smaller

// we can always reuse the block.

void\* newp = malloc(n);

if (newp == NULL) {

return NULL;

}

memcpy(newp, p, orig\_length);

free(p);

return newp;

}

If the value of ***p***, the block to be reallocated is NULL, we just call *malloc*, easy. Otherwise, we know that the block has been allocated by *malloc*, so it will have a length indicator immediately preceding it in memory. The function loads this length and if the length is the same as the value of the second argument, ***n***, no reallocation is necessary, and we just return ***p***.

The block will either be growing or shrinking. If it is shrinking we call ***ShrinkBlock*** and return the original pointer. Here’s ***ShrinkBlock***:

static void ShrinkBlock(FreeBlockHeader\* alloc\_block, size\_t orig\_length,

size\_t new\_length, size\_t\* len\_ptr) {

assert(new\_length < orig\_length);

size\_t rem = orig\_length - new\_length;

if (rem >= sizeof(FreeBlockHeader)) {

// If we are freeing enough to make a free block, free it, otherwise

// there's nothing we can do and we just keep the block the same size.

\*len\_ptr = new\_length; // Change size of block.

size\_t\* newp = (size\_t\*)((char\*)alloc\_block + sizeof(size\_t) +

new\_length);

\*newp = rem - sizeof(size\_t); // Add header for free.

free(newp + 1);

}

}

After asserting that the new length is actually less than the original length, we calculate the difference. If there is sufficient space remaining to fit a free block, we pretend that the unused part of the block was allocated by *malloc* by writing its length marker and calling free on the now unused part of the block. If there isn’t enough space for a free block, we just ignore the change request and return the original block unchanged.

If the block is growing things are more complex. We need to search the free list to see if we can find a free block either above or below the block being reallocated. The preference is to locate a free block immediately adjacent to the block at its upper address because then we can simply grow into that block without copying any memory around. If we find such a free block and it has enough space to expand into, we call ***ExpandIntoFreeBlockAbove*** to perform the expansion. This is defined as:

static void ExpandIntoFreeBlockAbove(FreeBlockHeader\* free\_block,

size\_t new\_length, size\_t len\_diff,

size\_t free\_remaining, size\_t\* len\_ptr,

FreeBlockHeader\*\* next\_ptr) {

assert(free\_remaining > sizeof(FreeBlockHeader));

FreeBlockHeader\* next = free\_block->next;

// The free block has enough space.

\*len\_ptr = new\_length;

FreeBlockHeader\* new\_block =

(FreeBlockHeader\*)((uintptr\_t)free\_block + len\_diff);

new\_block->length = free\_remaining;

new\_block->next = next;

\*next\_ptr = new\_block;

}

It’s fairly easy. We just write the new length into the length marker of the block being reallocated and then place a new *FreeBlockHeader* at the remaining free space in the free block. The *next* pointer of the previous block is then updated to refer to the relocated *FreeBlockHeader*.

If we can’t expand into an adjacent free block at the upper address, we look for a free block that ends at the lower address. This is not as good an option because it involves copying the memory and changing the address returned by realloc. If we are able to do it, a call to ***MergeWithFreeBlockBelow*** will do it.

static size\_t\* MergeWithFreeBlockBelow(void\* alloc\_block,

FreeBlockHeader\* prev,

FreeBlockHeader\* free\_block,

size\_t new\_length, size\_t orig\_length) {

uintptr\_t free\_addr = (uintptr\_t)free\_block;

FreeBlockHeader\*\* next\_ptr;

if (prev == NULL) {

next\_ptr = &\_\_free\_list;

} else {

next\_ptr = &prev->next;

}

// Move FreeBlockHeader to end of allocated block. This is inside

// the combined free block and block being reallocated.

FreeBlockHeader\* next = free\_block->next;

FreeBlockHeader\* newb =

(FreeBlockHeader\*)(free\_addr + new\_length + sizeof(size\_t));

newb->length = free\_block->length + orig\_length - new\_length;

newb->next = next;

\*next\_ptr = newb;

size\_t\* len\_ptr = (size\_t\*)free\_block;

\*len\_ptr = new\_length;

memmove(len\_ptr + 1, alloc\_block, orig\_length);

return len\_ptr + 1;

}

The procedure involves merging the allocated block into a free block and making a new free block out of the unused space at the end of the combined block. We have to copy the contents of the block being reallocated to the start of the new memory. Since this may overlap with the existing contents, we use the *memmove* function, which handles overlapping memory copies (despite its name, it doesn’t move the memory, it just copies it). Since the block being reallocated has changed addresses, we write the new length marker into the new address and return the address after the marker.

If all else fails and we can’t change the size of the block in place, either by shrinking or growing it, we have to copy the block, we return the address of the newly allocated block.

# The Rarely Used *calloc* Function

Most of the time the obscure function *calloc* isn’t really needed. The function declaration is:

void\* calloc(size\_t n, size\_t m);

It does two things:

1. Allocates ***n\*m*** bytes of memory using *malloc*.
2. Clears the allocated memory

That’s it. Here’s the function with no additional explanation required.

void\* calloc(size\_t n, size\_t m) {

void\* p = malloc(n\*m);

if (p == NULL) {

return NULL;

}

memset(p, 0, n\*m);

return p;

}

# Optimizations

The presented algorithm works well but it is not really suitable for a real-world application in an industrial strength application. It is too slow and will fragment the heap too much over time. What can be done to improve it?

The algorithm is an intrusive linked list with a linear search, which results in a *first-fit* allocation, meaning that the first free block it finds that is big enough will be used. This tends to fragment the heap, creating lots of little blocks that are not big enough for most allocations. The resulting increase in the length of the free free list decreases the speed of each allocation and the little free blocks waste memory.

Rather than first-fit, it would be better to perform a *best-fit* algorithm by using a data structure that takes into account the size of the blocks in its ordering. This would mean that the free block closest to the required size would be chosen first. One way to do this, and one I used many years ago, was to keep the free list sorted by size rather than address, but this complicates the free function as it is harder to know when blocks can be coalesced. If you are interested, I suggest you check out US Patents US 6363468 B1 and US 6430665 B1.

A real-world program will tend to allocate lots of blocks of the same size, most of the them being small blocks. Many industrial allocators keep a bucket of blocks of various sizes outside of the main free list for these allocators. For the various block sizes found to be common in programs (structs tend to be small and fixed size), the allocator keeps separate data structures. The allocation of these known size blocks can be done quickly without any fragmentation because their size is known. Probably the best allocator today is Google’s *TCMalloc* (<https://github.com/google/tcmalloc>). It is worth studying the documentation if you are interested in how a real allocator works in modern computers.

1. I’ve written my own C compiler and linker and this is what my linker does. [↑](#footnote-ref-1)
2. Most programs don’t behave well if the stack overlaps the top of the heap or the read/write data. [↑](#footnote-ref-2)
3. The reason is that, in C, pointers are just addresses and moving the thing they are pointing to would require finding all the used pointers in memory and registers and changing them. Managed languages like Java and Go have garbage collectors that actually do that so heap compaction is possible for them. [↑](#footnote-ref-3)