http://web.stanford.edu/class/cs107e/memory.pdf

All of Bare Metal!

Processor and memory architecture

Peripherals: GPIO, timers, UART

Assembly language and machine code

From C to assembly language

Function calls and stack frames

Serial communication and strings

Modules and libraries: Building and linking

Memory management: Memory map & heap

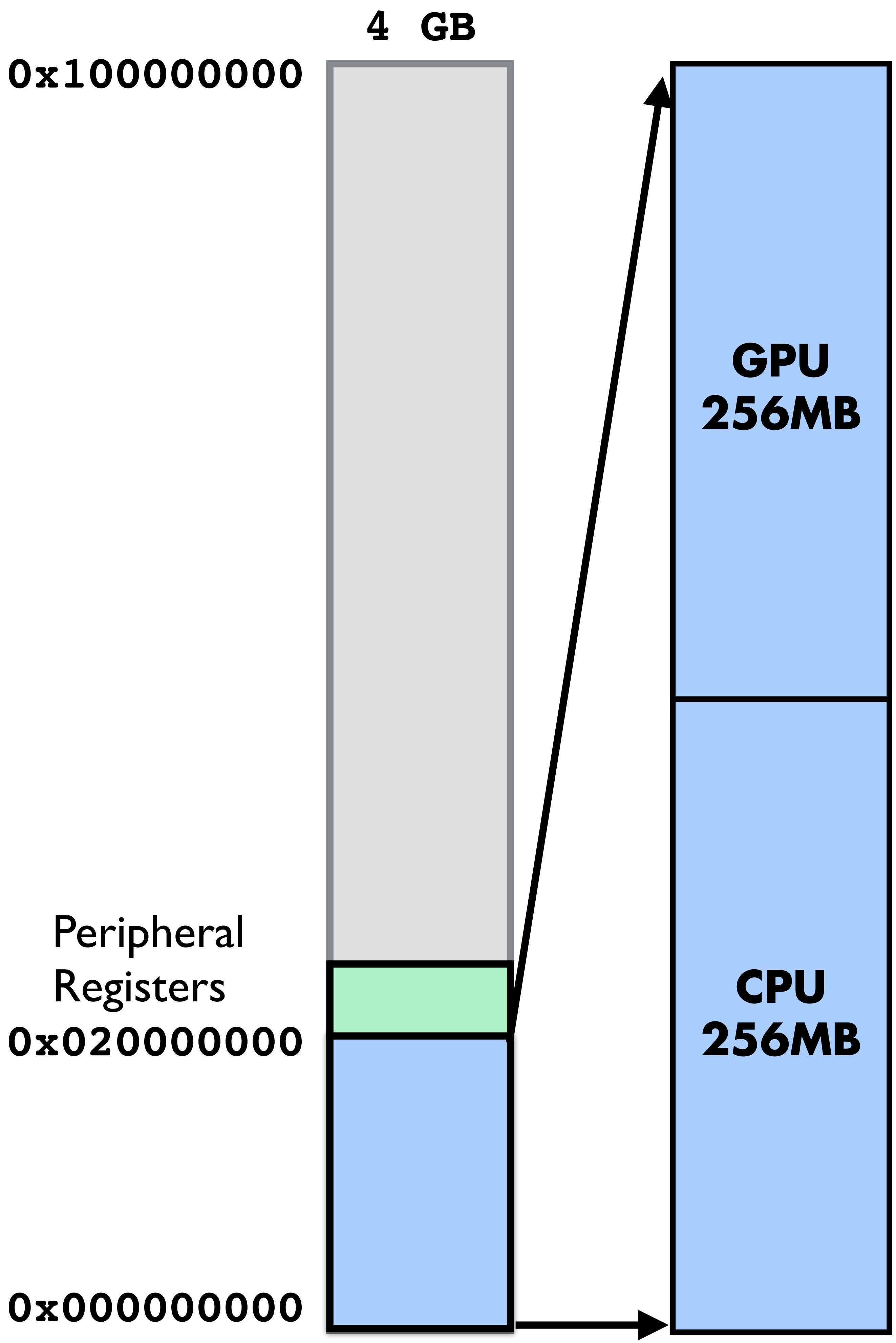
Memory Management

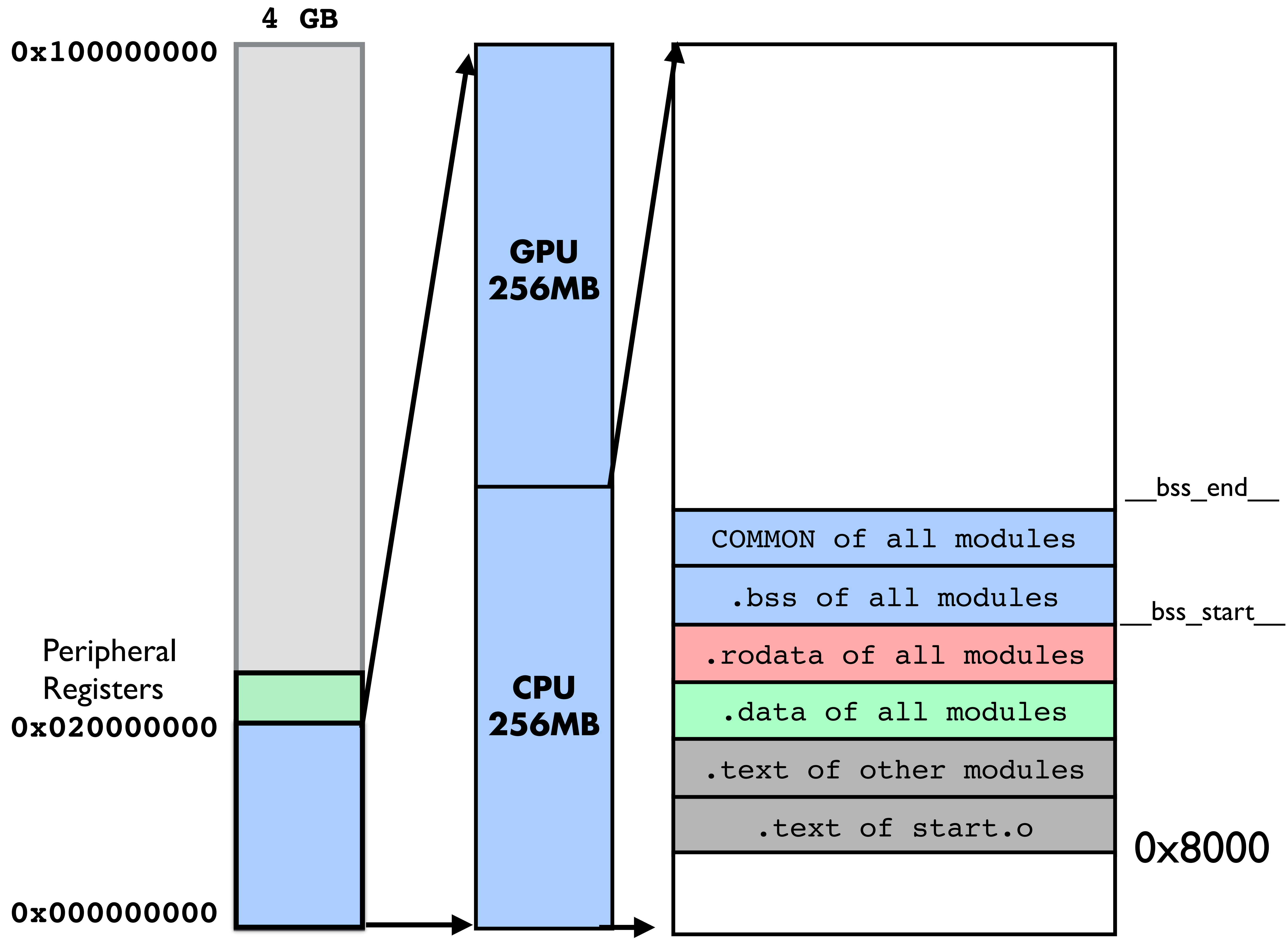
Sections and memory map
Initializing memory
Heap memory allocation

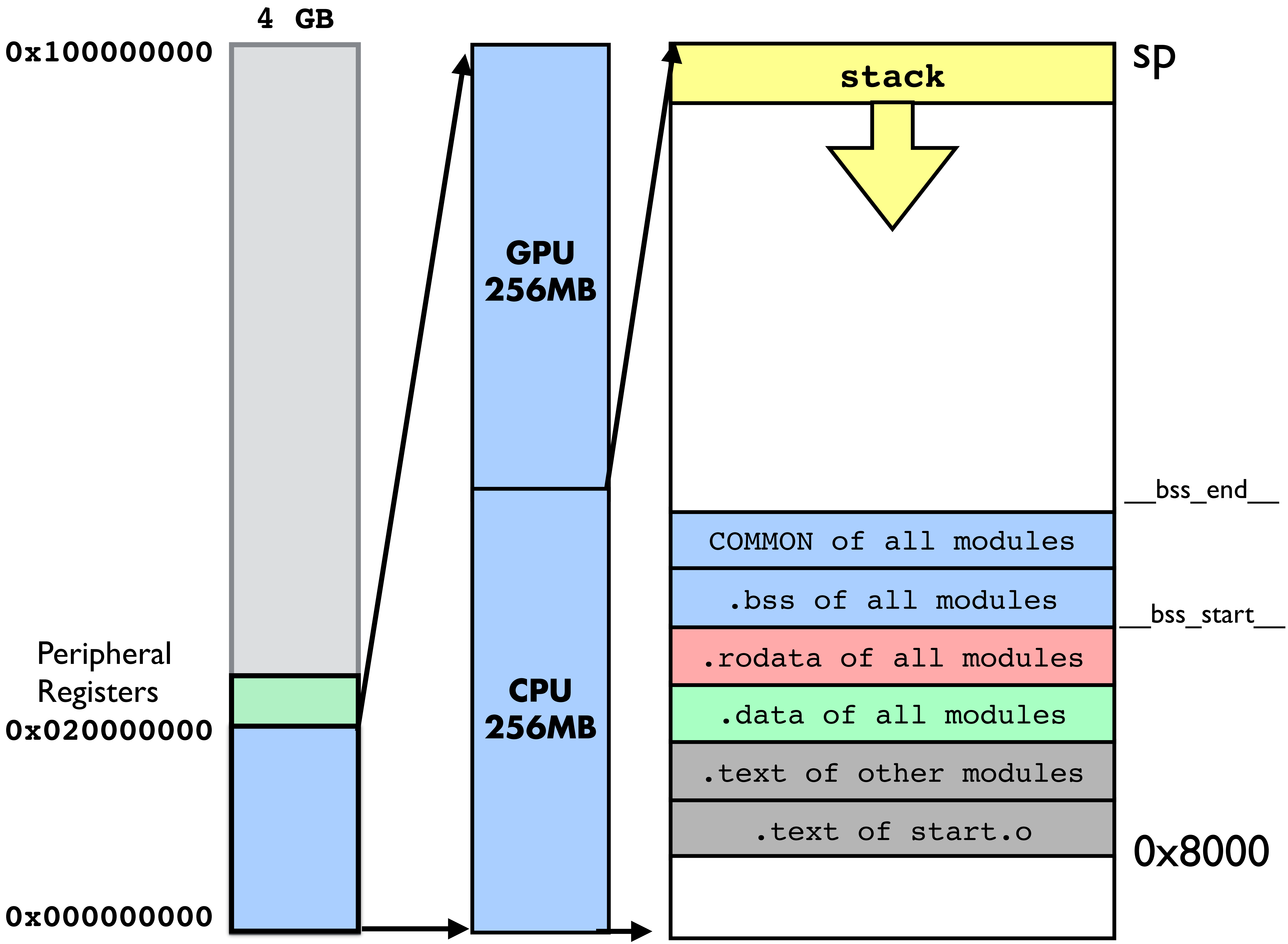
Memory Management

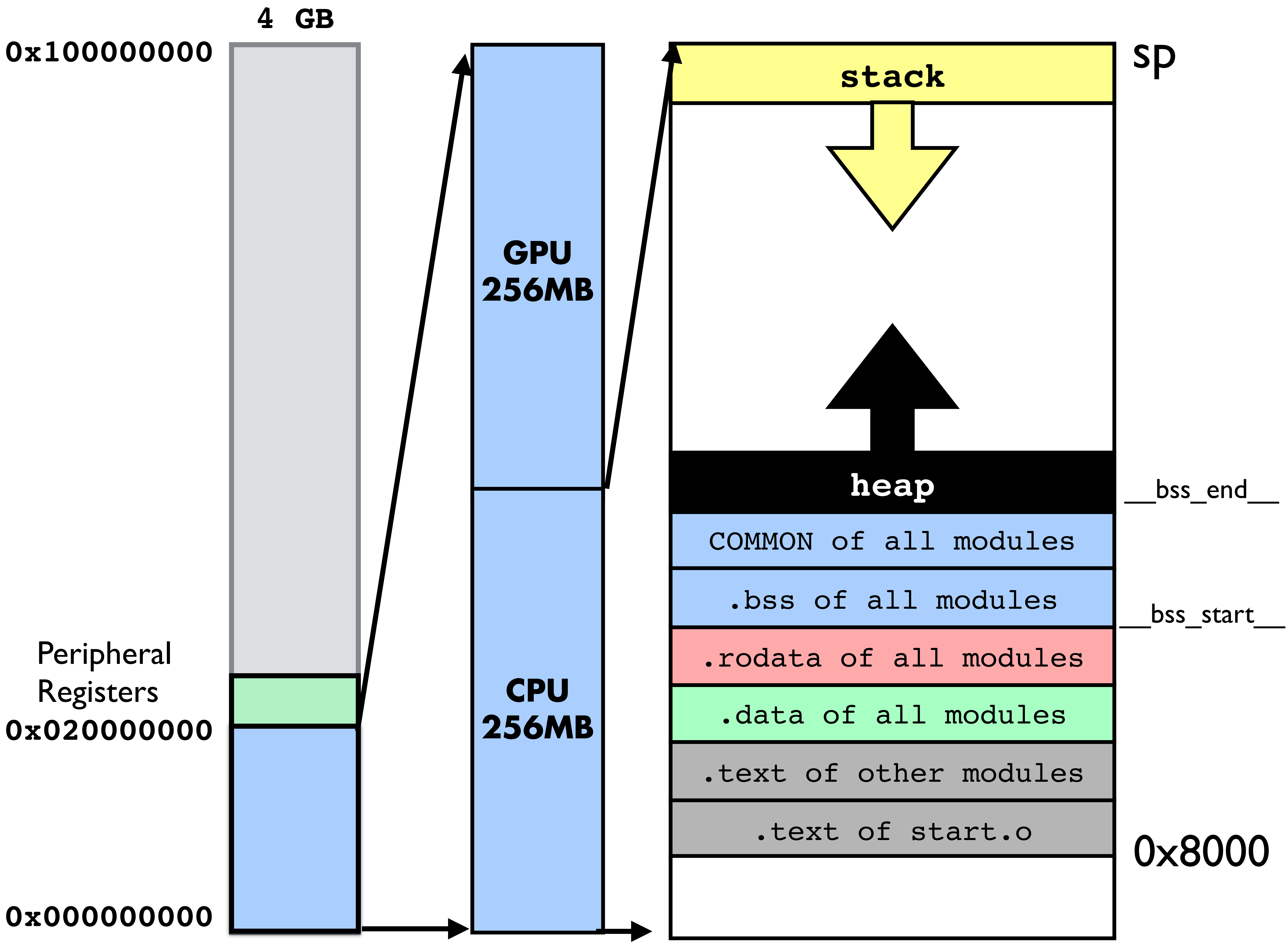
Heap memory allocation


4 GB
0x100000000
Peripheral
Registers
0x020000000
0x000000000


4 GB
0x100000000
Peripheral
Registers
0x020000000
0x000000000
GPU
256MB
CPU
256MB


4 GB
0x100000000
Peripheral
Registers
0x020000000
0x000000000
GPU
256MB
CPU
256MB
__bss_end__
COMMON of all modules
.bss of all modules
__bss_start__
.rodata of all modules
.data of all modules
.text of other modules
.text of start.o
0x8000


4 GB
0x100000000
Peripheral Registers
0x020000000
0x000000000
GPU 256MB
CPU 256MB
stack
sp
__bss_end__
COMMON of all modules
.bss of all modules
__bss_start__
.rodata of all modules
.data of all modules
.text of other modules
.text of start.o
0x8000

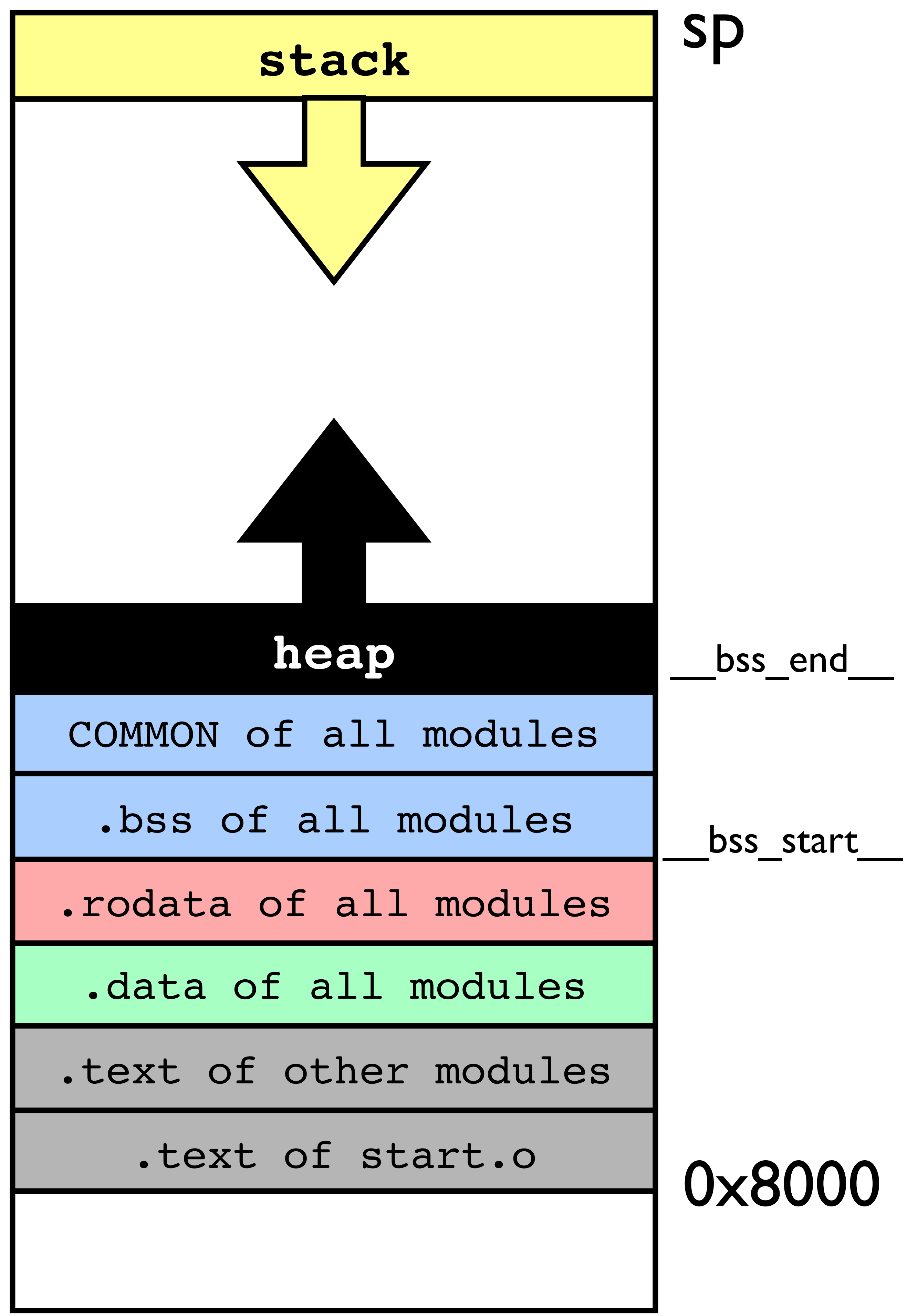

4 GB
0x100000000
Peripheral Registers
0x020000000
0x000000000
GPU 256MB
CPU 256MB
stack
sp
heap
__bss_end__
COMMON of all modules
.bss of all modules
__bss_start__
.rodata of all modules
.data of all modules
.text of other modules
.text of start.o
0x8000

```
void f() {
   int x;
}
```

```
char* ptr = malloc(len);
```

global variables

code



Heap Memory Allocation

Memory Allocation

Compile-time vs. run-time memory allocation

Why run-time memory allocation?

1. Don't know the size of an array when compiling

2. Dynamic data structures such as strings, lists and trees

For example, you cannot return an array from a function, as it is on the stack…we must put it on the heap.

Why do we have both stack and heap allocation?

As we have discussed before, stack memory is limited and serves as a scratch-pad for functions, and it is continually being re-used by your functions. Stack memory isn't persistent, but because it is already allocated to your program, it is fast.

Heap memory takes more time to set up (you have to go through the heap allocator), but it is unlimited (for all intents and purposes), and persistent for the rest of your program.

`malloc`, `free`, and `realloc`

This is what your heap allocator is going to do!

`void *malloc(size_t size)`

Return pointer to memory block >= requested size
(failure returns `NULL` and sets errno)

`void free(void *p)`

Recycle memory block
`p` must be from previous malloc/realloc call

~~`void *realloc(void *p, size_t size)`~~

~~Changes size of block `p`, returns pointer to block (possibly same)~~
~~Contents of new block unchanged up to min of old and new size~~
~~If the new pointer isn't the same as the old pointer, the old block will have been free'd~~

realloc removed from spec (but you might want to know about it)

Bump Memory Allocator

`malloc.c`

Allocator Requirements

The heap allocator must be able to service arbitrary sequence of malloc() and free() requests

- **`malloc`** must return a pointer to contiguous memory that is equal to or greater than the requested size, or **`NULL`** if it can't satisfy the request.
- The *payload* contents (this is the area that the pointer points to) are unspecified — they can be 0s or garbage.
- If the client introduces an error, then the behavior is undefined
  - If the client tries to free non-allocated memory, or tries to use free'd memory.

The heap allocator has some constraints:

It can't control the number, size, or lifetime of the allocated blocks.

It must respond immediately to each malloc request

- I.e., it can't reorder or buffer **`malloc`** requests — the first request must be handled first.
- It *can* defer, ignore, or reorder requests to **`free`**

Allocator Requirements (continued)

Other heap allocator constraints:
The allocator must align blocks so they satisfy all alignment requirements

i.e., 8 byte alignment for `malloc 32`-bit ARM

The allocated payload must be maintained *as-is*

The allocator *cannot* move allocated blocks, such as to compact/coalesce free.

- Why not?

All of the programs with allocated memory would have corrupted pointers!

- The allocator *can* manipulate and modify free memory

Allocator Goals

The allocator should first and foremost attempt to service **`malloc`** and **`free`** requests *quickly.*

Ideally, the requests should be handled in *constant time* and should not degrade to linear behavior (we will see that some implementations can do this, some cannot)

The allocator must try for a *tight space utilization*.

Remember, the allocator has a fixed block of memory to dole out smaller parts — it must try to allocate efficiently

The allocator should try to minimize *fragmentation*.

It should try to group allocated blocks together.

There should be a small overhead relative to the payload (we will see what this mean soon!)

Allocator Goals (continued)

It is desirable for a heap allocator to have the following properties:

Good locality

- Blocks are allocated close in time are located close in space
- "Similar" blocks are allocated close in space

Robust

- Client errors should be recognized
  - What is required to detect and report them?

Ease of implementation and maintenance

- Having `*(void **)` all over the place makes for hard-to-maintain code. Instead, use structs, and typedef when appropriate.
- The code is necessarily complex, but the more efforts you put into writing clean code, the more you will be rewarded by easier-to-maintain code.

Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x0**
a	0xffffe800	**0xbeef**

heap

96 bytes

0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x138	0x140	0x148	0x150	0x158

(free)

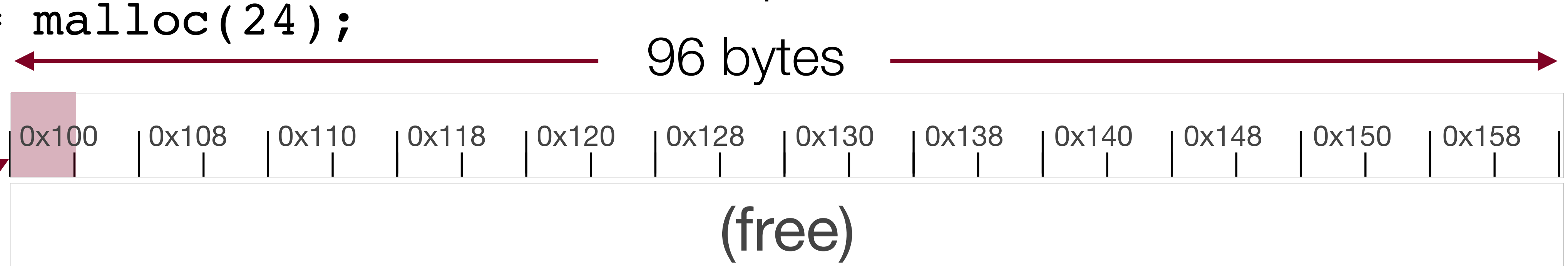
Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x0**
a	0xffffe800	**0xbeef**

heap



Each section represents 4 bytes

Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x0**
a	0xffffe800	**0xbeef**

heap

Each section represents 4 bytes

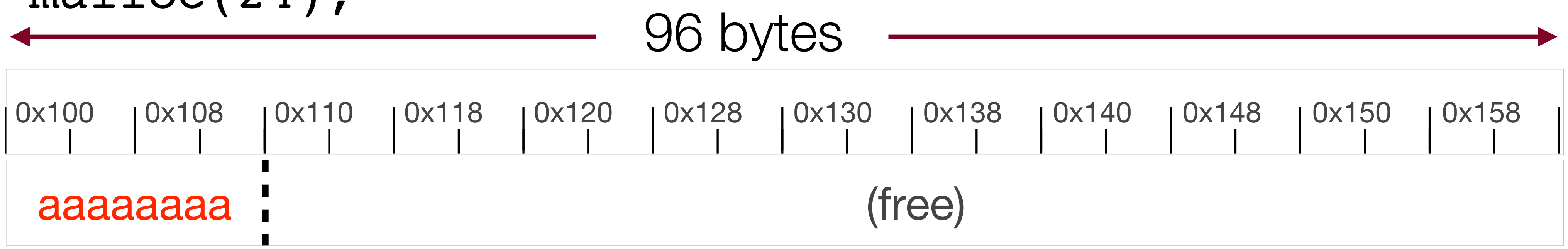
Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x0**
a	0xffffe800	**0x100**

heap



Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**

heap

96 bytes

0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x138	0x140	0x148	0x150	0x158
aaaaaaaa		bbbb	(free)								

Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

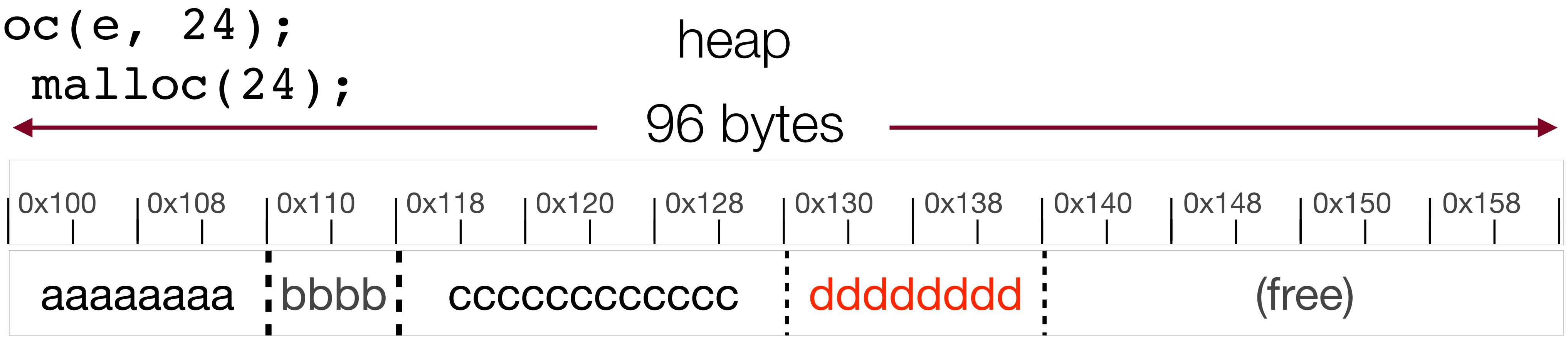
	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**

Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**



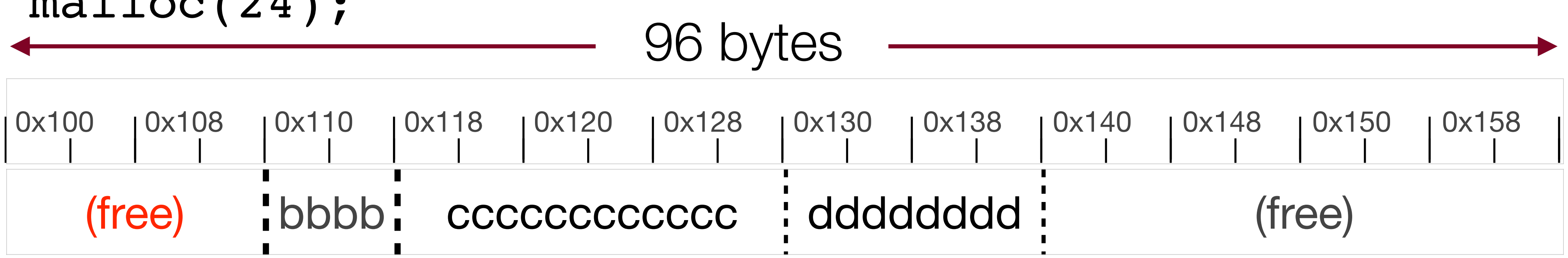
Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**

heap

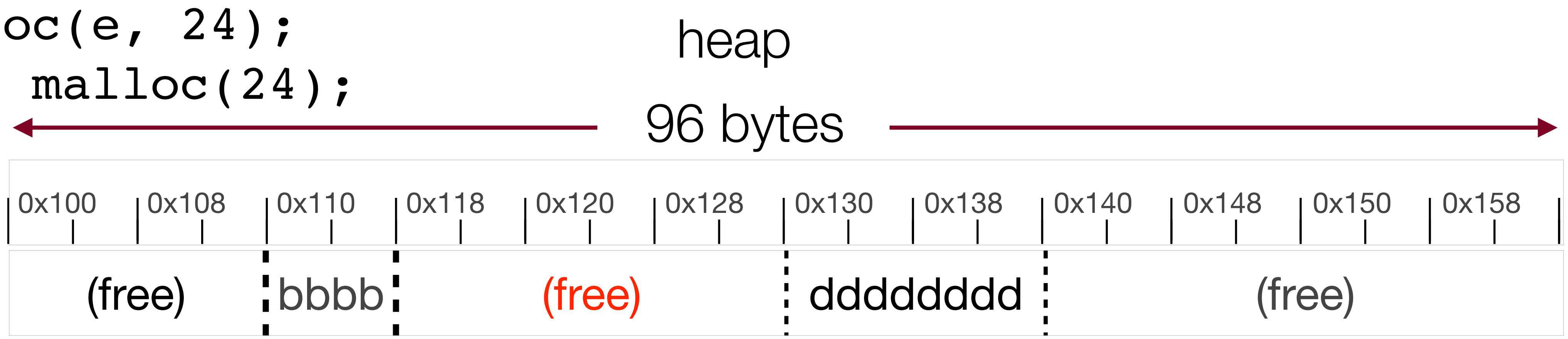


Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**



Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	0x100
d	0xffffe818	0x130
c	0xffffe810	0x118
b	0xffffe808	0x110
a	0xffffe800	0x100

heap

Tracing the Heap (possible implementation)

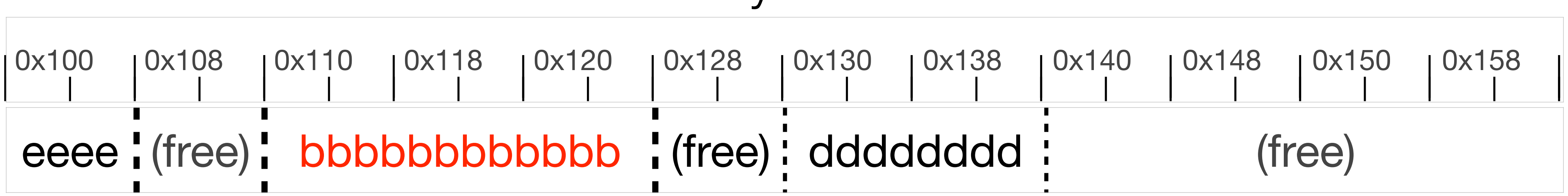
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x100**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**

heap

96 bytes



Tracing the Heap (possible implementation)

```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

	Address	Value
e	0xffffe820	0x140
d	0xffffe818	0x130
c	0xffffe810	0x118
b	0xffffe808	0x110
a	0xffffe800	0x100

heap

96 bytes

0x100	0x108	0x110	0x118	0x120	0x128	0x130	0x138	0x140	0x148	0x150	0x158
(free)		bbbbbbbbbbbb			(free)	dddddddd		eeeeeeeeeeee			(free)

Tracing the Heap (possible implementation)

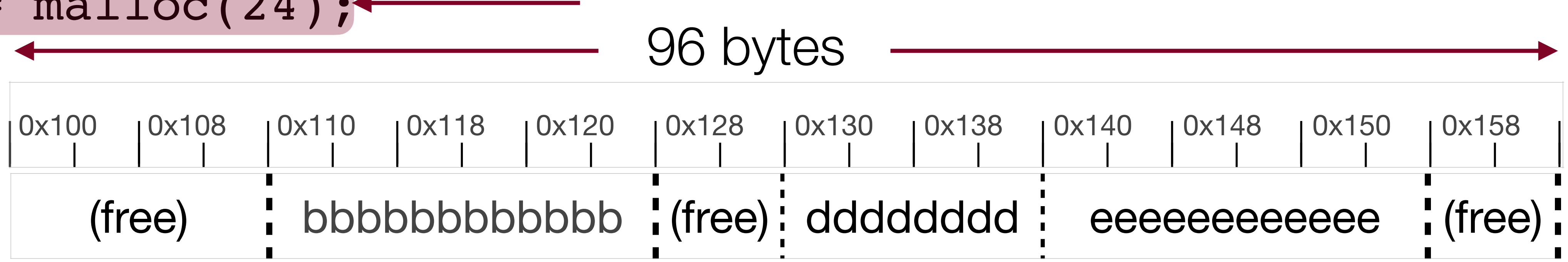
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

Returns **NULL**

All allocated on the stack:

	Address	Value
e	0xffffe820	**0x140**
d	0xffffe818	**0x130**
c	0xffffe810	**0x118**
b	0xffffe808	**0x110**
a	0xffffe800	**0x100**
f	0xffffe7f0	**0x0**

heap



API

```
void *malloc( size_t size );
void free( void *pointer );

// Note that void* is a generic pointer
// Note that size_t is for sizes
```

What is this void * business, anyway?

This is an area that I like to call "the wild west" of pointers. We are going to discuss the **`void *`** pointer, which is a pointer that has an unspecified pointee type. In other words, it is a pointer, but does not have a width associated with the underlying data based on some type.

You can pass **`void *`** pointers to and from functions, and you can assign them values with the **`&`** operator. E.g.,

```
int arr[] = {2,4,6,8,10};
void *arr_p1 = arr;
int *arr_p2 = arr_p;
```

Generic Pointers

You **cannot** dereference a `void *` pointer, nor can you use pointer arithmetic with it. E.g.,

```
int arr[] = {2,4,6,8,10};

void *arr_p1 = arr;

arr_p1++; // gives compiler warning about incrementing void *

printf("%d\n", arr_p1[0]); // warns, but also causes compiler error
                           // because you cannot dereference void *
```

Generic Pointers

Why would we ever want a type where we *lose* information?

Sometimes, a function needs to be *generic* so it can deal with any type. We have seen this with **`realloc`** and **`free`**:

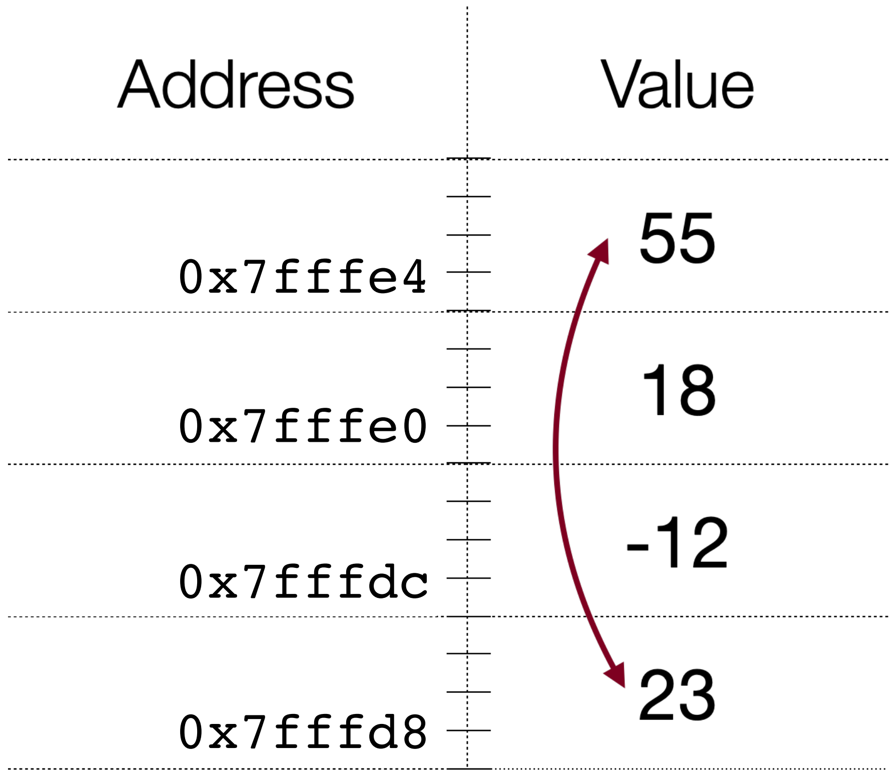
```
void free(void *ptr);
```

It would not be very nice if we had to have a different **`free`** function for every type of pointer!

Generic Pointers

What if you wanted to write a program to swap the first and last element in an **`int`** array? You might write something like this:

```
void swap_ends_int(int *arr, size_t nelems)
{
    int tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) /
                      sizeof(i_array[0]);
    swap_ends_int(i_array,i_nelems);
}
```



Great! But what if you also wanted to swap the first and last element in a **`short`** array?

Generic Pointers

Great! But what if you also wanted to swap the first and last element in a **`short`** array?

```
void swap_ends_int(int *arr, size_t nelems)
{
    int tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

void swap_ends_short(short *arr, size_t nelems)
{
    short tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

void main(void)
{
    int i_array[]   = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);
    swap_ends_int(i_array,i_nelems);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);
    swap_ends_short(s_array, s_nelems);
}
```

Bummer. We have to write a function that is virtually identical, with the only difference being that we handle the type of the array elements differently.

In other words, the type system is getting in the way! We would like to write a single **`swap_ends`** function that handles *any* array, but the type system foils us.

Generic Pointers

`void *` to the rescue! In this case, the pointer type gives us information about the size of the elements being pointed to (either 4-bytes for **`int`**, or 8-bytes for **`long`**, in the previous example).

By using void * and *explicitly including the width of the type*, we can write a function that can take any type as the elements to swap:

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp,arr,width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}
```

We can copy bytes using **`memcpy!`**

We must pass the width of the elements in the array because the **`void *`** pointer doesn't carry that information.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}
```

Let's look at this function in more detail. First, we have a `void *` pointer passed in as the array.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}
```

Next, we create a **`char`** array to hold the bytes. Remember: **`char`** is the only 1-byte type we have, and using a **`char`** array is how we can create an array that is exactly the number of bytes we want. We will use this almost every time we use **`void *`** pointers, so get used to it!

(we could also use **`malloc`** if we wanted to, but it isn't really necessary here, as the array works just fine. Regardless, we would still use a **`char *`** pointer)

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}
```

We copy the bytes with **`memcpy`**.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}
```

This part takes some time to get used to!

Notice that we need a pointer to the element that we are trying to copy into. We already said that we cannot do pointer arithmetic on a **`void *`** pointer, so we first cast the pointer to **`char *`**, and then manually calculate the pointer arithmetic to get us to the correct location. In this case, because we want the last element in the array, the calculation is:

```
(char *)arr + (nelems - 1) * width
```

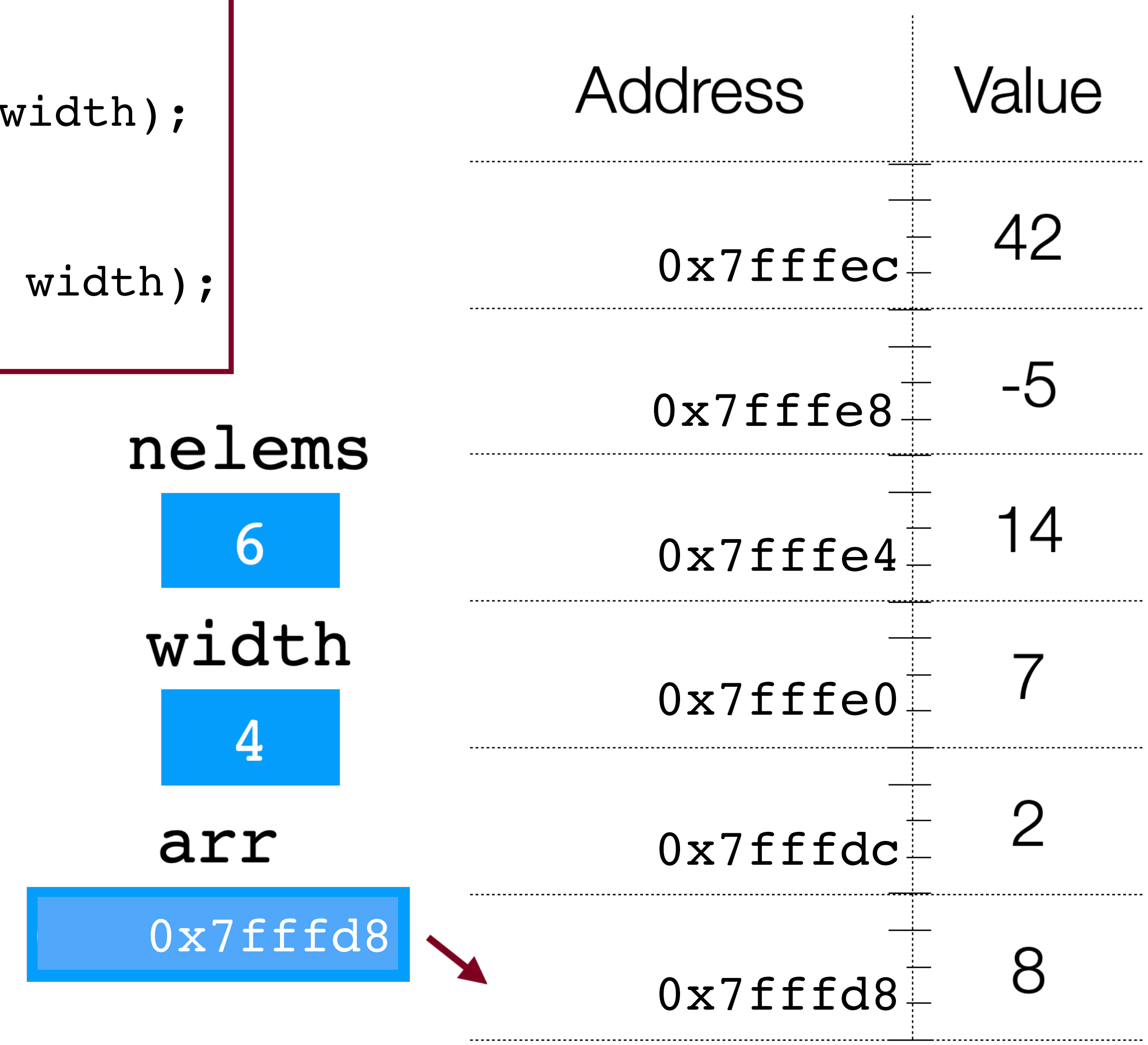
Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}
```

In other words, what is the location of the 42?

`(char *)arr + (nelems - 1) * width`

`0x7fffd8 + (5 * 4) == 0x7fffec`

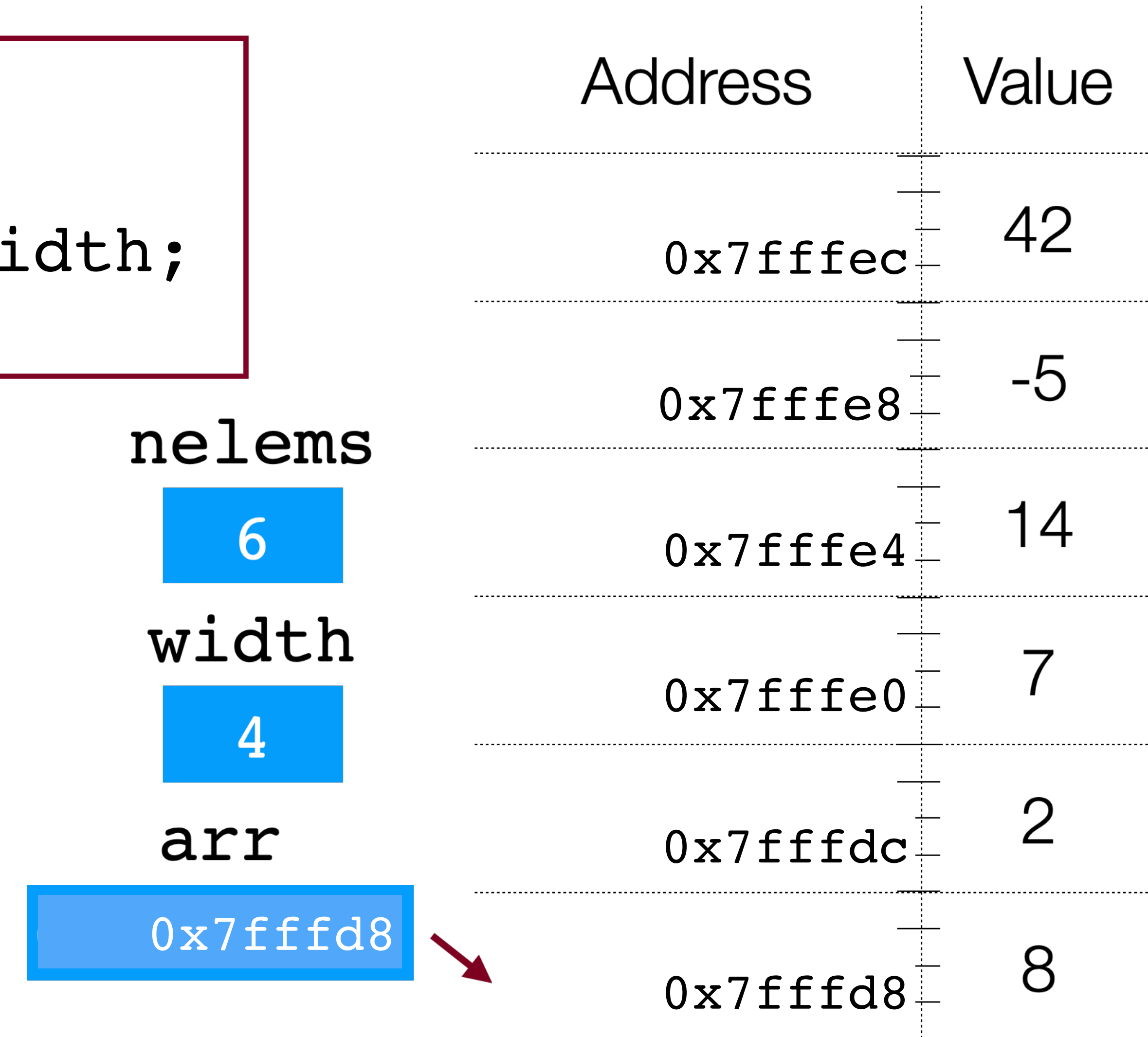


A key point to understand is that the pointer arithmetic increases by exactly 20 because of the **`char *`** cast, which means that +1 equals 1 byte.

Generic Pointers

Very often, we will need to find the ith element in an array. You should become familiar with the following idiom:

```
for (size_t i=0; i < nelems; i++) {
    // get ith element
    void *ith = (char *)arr + i * width;
}
```



Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Let's walk through this example.

First, we create an `int` array, then we find its size.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]    = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Let's walk through this example.

First, we create an `int` array, then we find its size.

Next, we create a short array, then we find its size.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Let's walk through this example.

First, we create an `int` array, then we find its size.

Next, we create a long array, then we find its size.

Then, we call swap_ends on the `int` array.

Note that we pass in the width, which is 4:

`sizeof(i_array[0])`

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Create a char array to hold the width of the element we want to swap.

At this point, all information about the int array is gone, so we just have to rely on the `width` argument.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Move 4 bytes from the first element in the array to **`tmp`**.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Move 4 bytes from the last element in the array to the first element.

Generic Pointers

```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

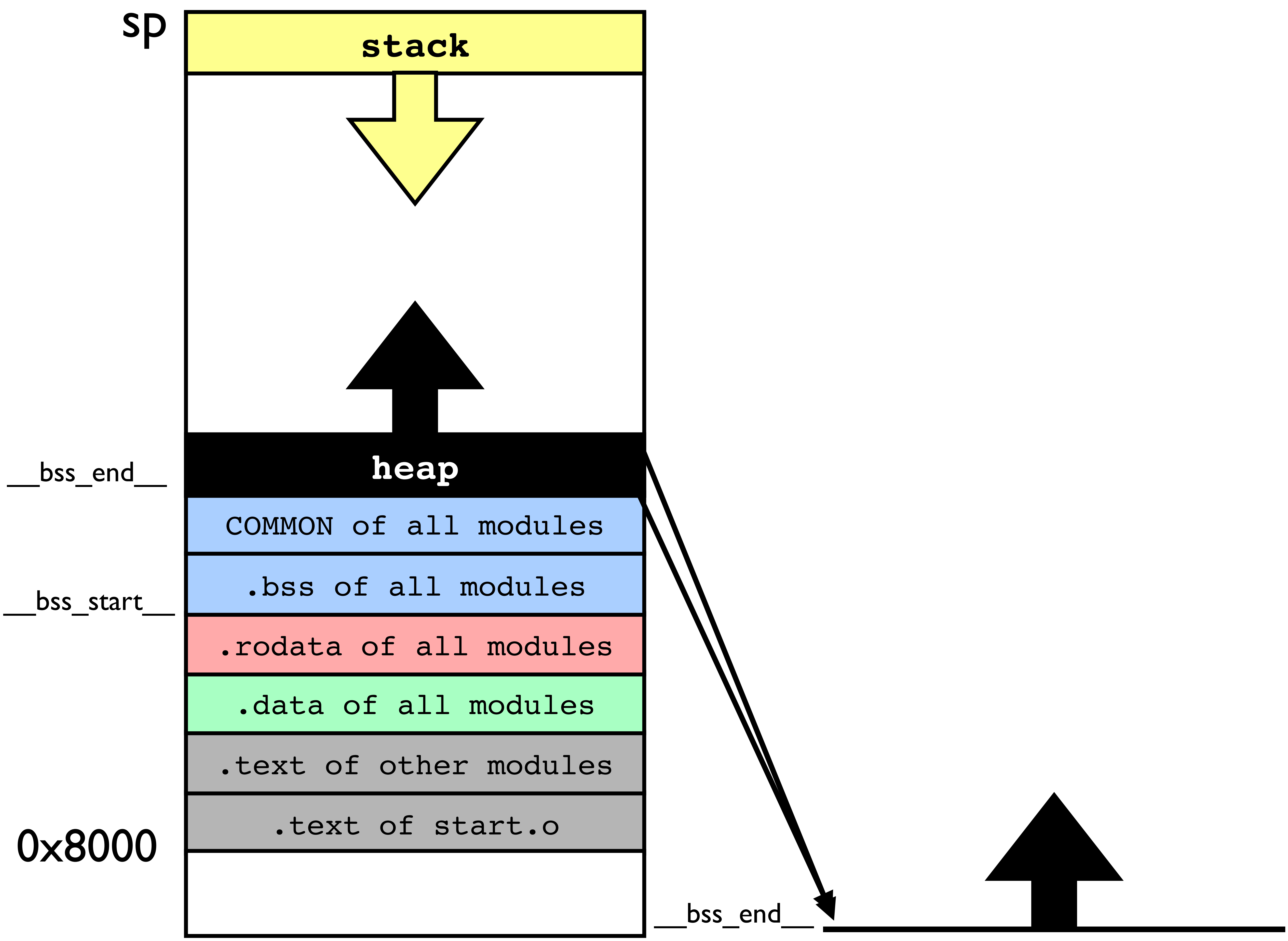
Move 4 bytes from tmp to the last position in the array.

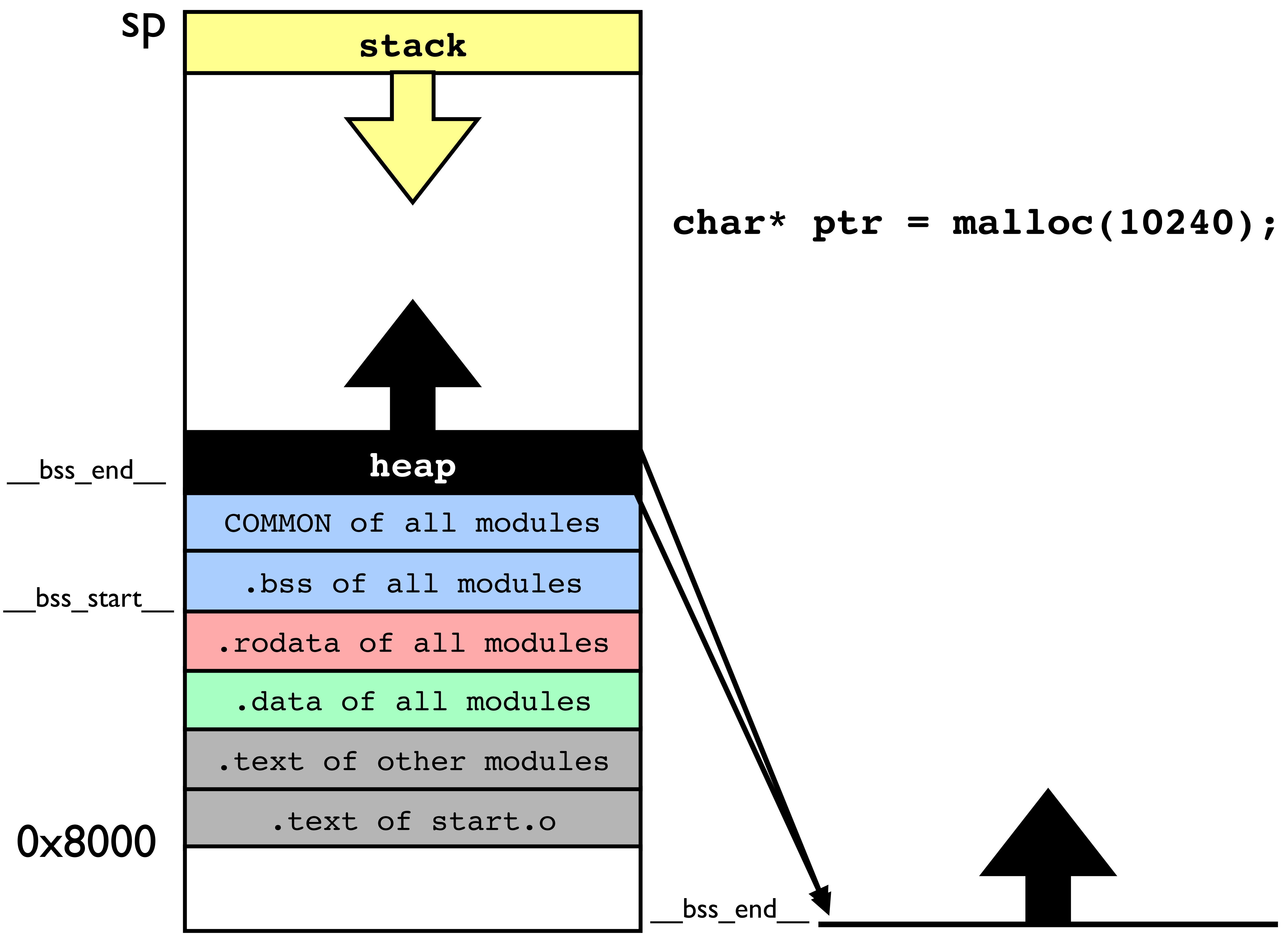
Generic Pointers

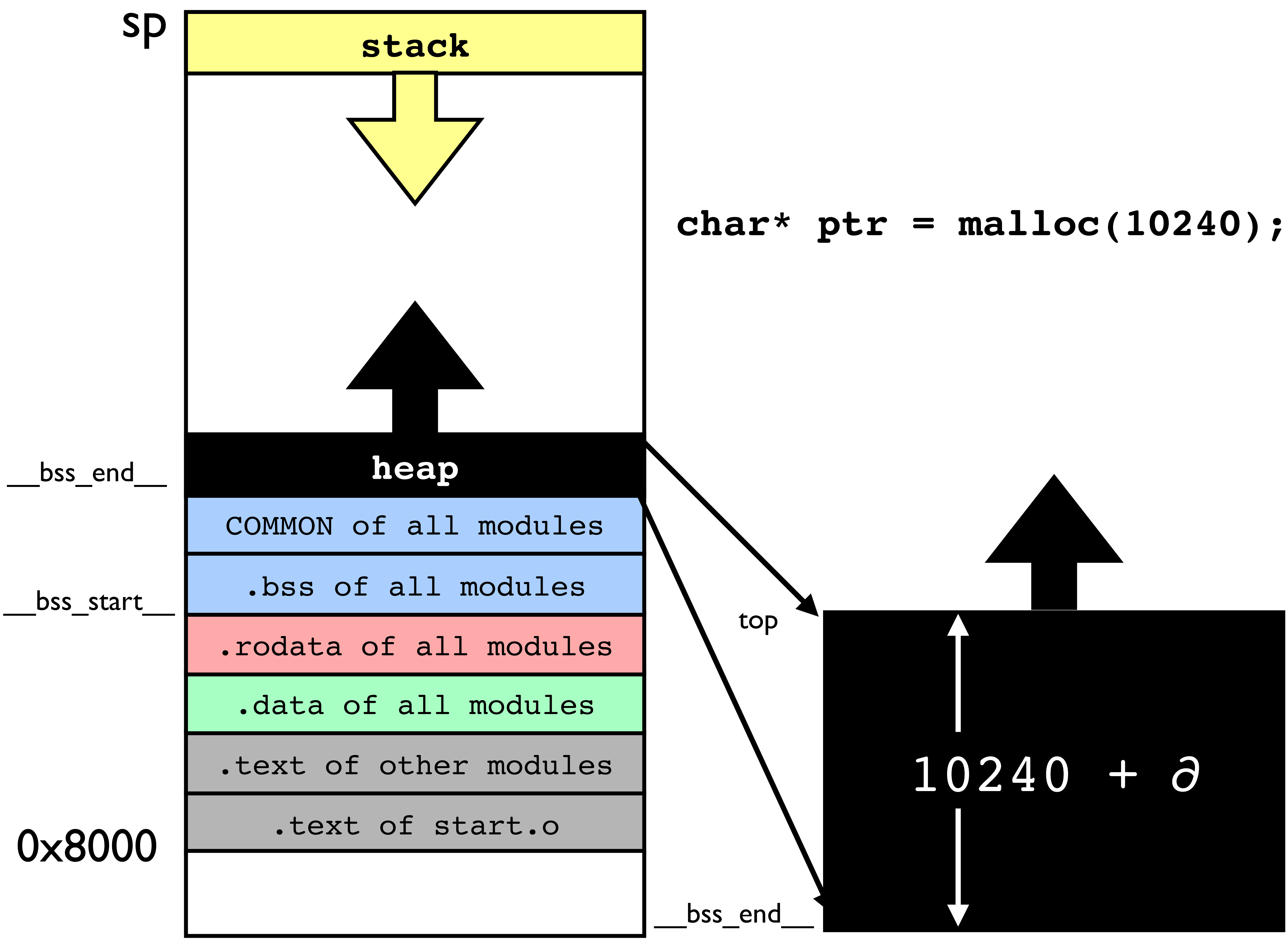
```
void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr,(char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

void main(void)
{
    int i_array[]   = {10, 40, 80, 20, -30, 50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    short s_array[]  = {100, 400, 800, 200, -300, 500};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
  ...
```

Repeat the process for the short array, which will pass in a **`width`** of 2:

```
sizeof(l_array[0]);
```


sp
stack
heap
__bss_end__
COMMON of all modules
.bss of all modules
__bss_start__
.rodata of all modules
.data of all modules
.text of other modules
.text of start.o
0x8000
__bss_end__


sp
stack
char* ptr = malloc(10240);
heap
__bss_end__
COMMON of all modules
.bss of all modules
__bss_start__
.rodata of all modules
.data of all modules
.text of other modules
.text of start.o
0x8000
__bss_end__



sp
stack
char* ptr = malloc(10240);
heap
__bss_end__
COMMON of all modules
.bss of all modules
__bss_start__
.rodata of all modules
.data of all modules
.text of other modules
.text of start.o
0x8000
top
10240 + ∂
__bss_end__

Questions

What happens if you forget to free a pointer after you are done using it?

Can you refer to a pointer after it has been freed?

What is stored in the memory that you malloc?

Calling free with a pointer that you didn't malloc?

Can you free the same pointer twice?

Wouldn't it be nice to not have to worry about freeing memory?

Variable Size `malloc/free`

just `malloc` is easy 🙂

`malloc` with `free` is hard 🙁

- `free` returns blocks that can be re-allocated
- `malloc` should search to see if there is a block of sufficient size. Which block should it choose (best-fit, first-fit, largest)?
- `malloc` may use only some of the block. It splits the block into two sub-blocks of smaller sizes
- splitting blocks causes fragmentation

Buddy allocators, slab allocators, lots of approaches