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

Survey Results

The survey is still open: https://goo.gl/forms/a6yjhN3GWHYEhOX02

What is going well:

- Office hours are helpful
- If I start early on assignments, they are doable
- Assignments are difficult but fun

What are you struggling with:

- When I do the assignments on the weekends, there aren't any office hours (*better this past weekend, and going forward*)
- Basic assignments are hard enough! Why require extensions? (*fair point, but part of the course structure*)
- SO MUCH WORK: 100% of time spent on this class, it seems (*This is a reality for CS 107 and CS 107e*)
- Lectures unclear — too much info, slides are not complete (*This is a very valid point. I will try to do better with the slides and in lecture*)
- Labs too long (*Another valid point. We want you to finish in the lab — this means two things: (1) less material in lab (our responsiblity) and (2) more concentration in lab (your responsibility)*)

What can the staff do better?

- More in-class examples! (*will do!*)
- Weekend office hours (*in progress*)
- More time on assignments: later evening due date, longer late days (*very difficult to do given the assignment pace. We have given you one more late day, and we can adjust that for the future if necessary*)
- More opportunities to get an A! (*the course structure is pretty set — in order to keep it fair with prior course offerings, we will probably keep the standard the same*)

Memory Management

Sections and memory map

Initializing memory

Heap memory allocation

A bit more on linking

```
// initialized variables
int x = 1;
const int x_const = 2;
static int x_static = 3;
static const int x_static_const = 4;

// uninitialized variables (equal to 0)
int y;
const int y_const;
static int y_static;
static const int y_static_const;
```

See linking example in today's code

```
% arm-none-eabi-nm main.o
00000000 T main
         U tricky
         U x
         U x_const
         U y
         U y_const
```

```
% arm-none-eabi-nm tricky.o
00000000 T tricky
00000000 D x
00000000 R x_const
00000004 d x_static
00000004 C y
00000004 C y_const
00000000 b y_static
```

Guide to Symbols

T/t - text

D/d - read-write data

R/r - read-only data

B/b - bss (*Block Started by Symbol*)

C - common (instead of B)

lower-case letter means `static`

Data Symbols

Types

- global vs static
- read-only data vs data
- initialized vs uninitialized data
- common (shared data)

```
.text : {
    start.o (.text)
    *(.text*)
} > ram
.data : { *(.data*) } > ram
.rodata : { *(.rodata*) } > ram
__bss_start__ = .;
.bss : {
    *(.bss*)
    *(COMMON)
} > ram
. = ALIGN(8);
__bss_end__ = .;
```

```
% arm-none-eabi-nm -n main.elf
00008000 T _start
00008008 t hang
0000800c T _cstart
0000805c T tricky
000080a8 T main
00008108 D x
0000810c d x_static
00008110 R x_const
00008114 R __bss_start__
00008114 b y_static
00008118 B y_const
0000811c B y
00008120 B __bss_end__
```

```
// cstart.c - initializes bss to 0
extern int __bss_start__;
extern int __bss_end__;

void main();

void _cstart() {
    int* bss = &__bss_start__;
    int* bss_end = &__bss_end__;
    while( bss < bss_end )
        *bss++ = 0;
    main();
}
```

Memory Management

Heap memory allocation

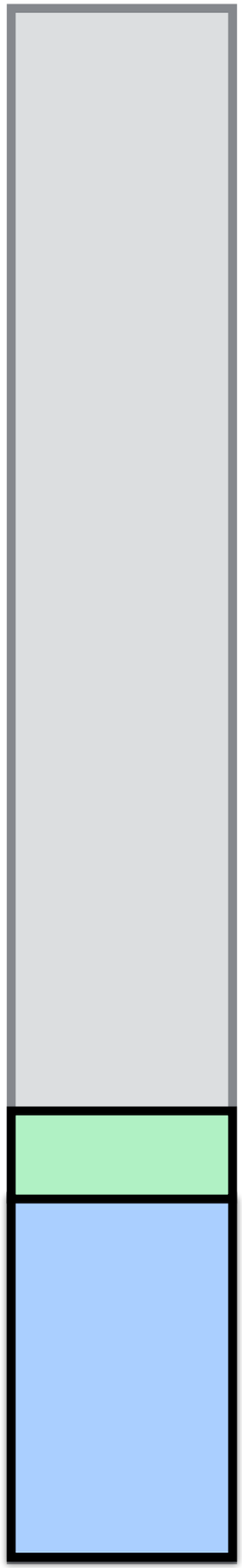
4 GB

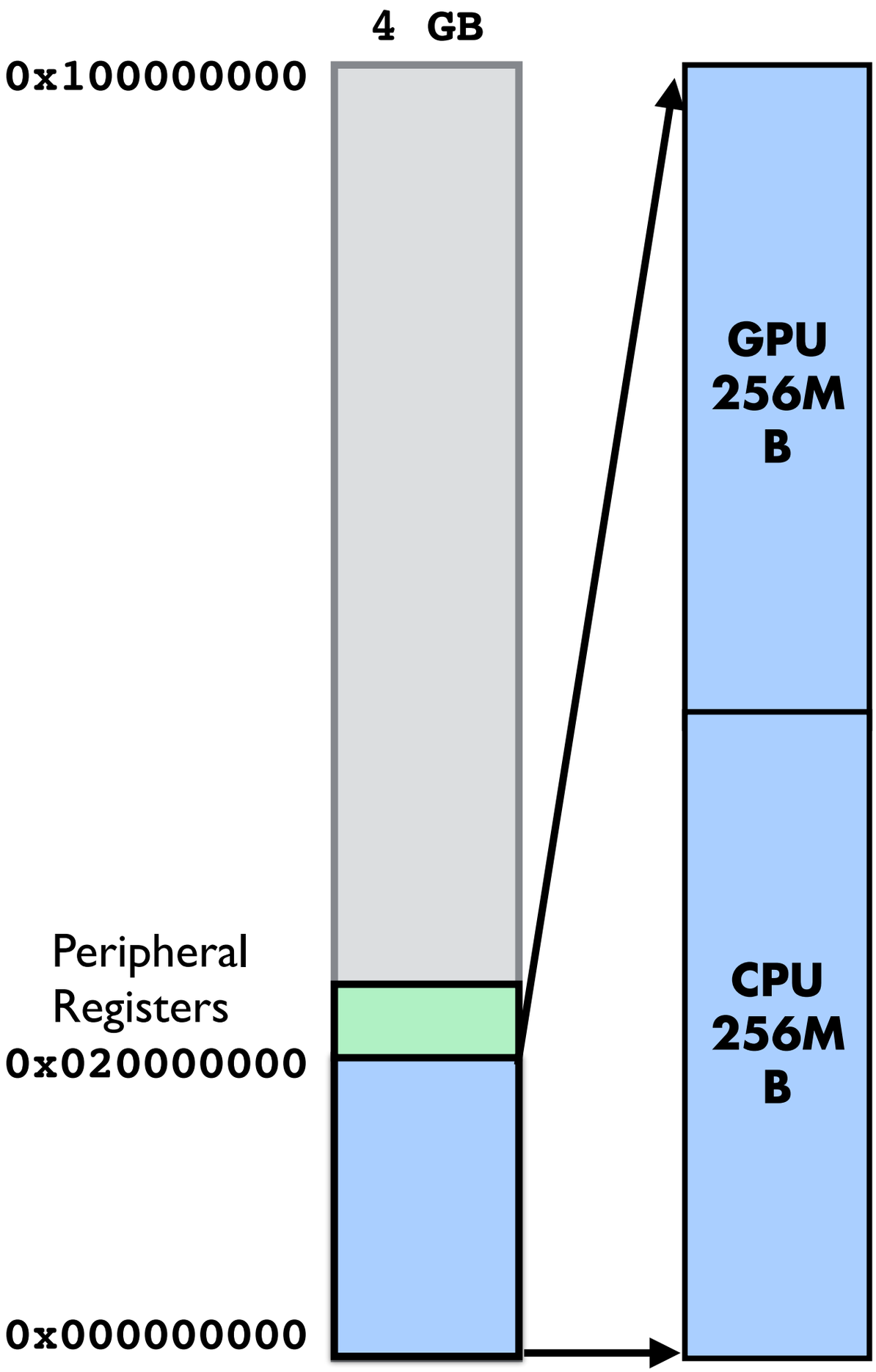
0x100000000

Peripheral
Registers

0x020000000

0x000000000




4 GB
0x100000000
GPU
256M
B
Peripheral
Registers
0x020000000
CPU
256M
B
0x000000000

4 GB

0x100000000

Peripheral Registers

0x020000000

0x000000000

GPU 256MB

CPU 256MB

__bss_end__

COMMON of all modules

.bss of all modules

__bss_start__

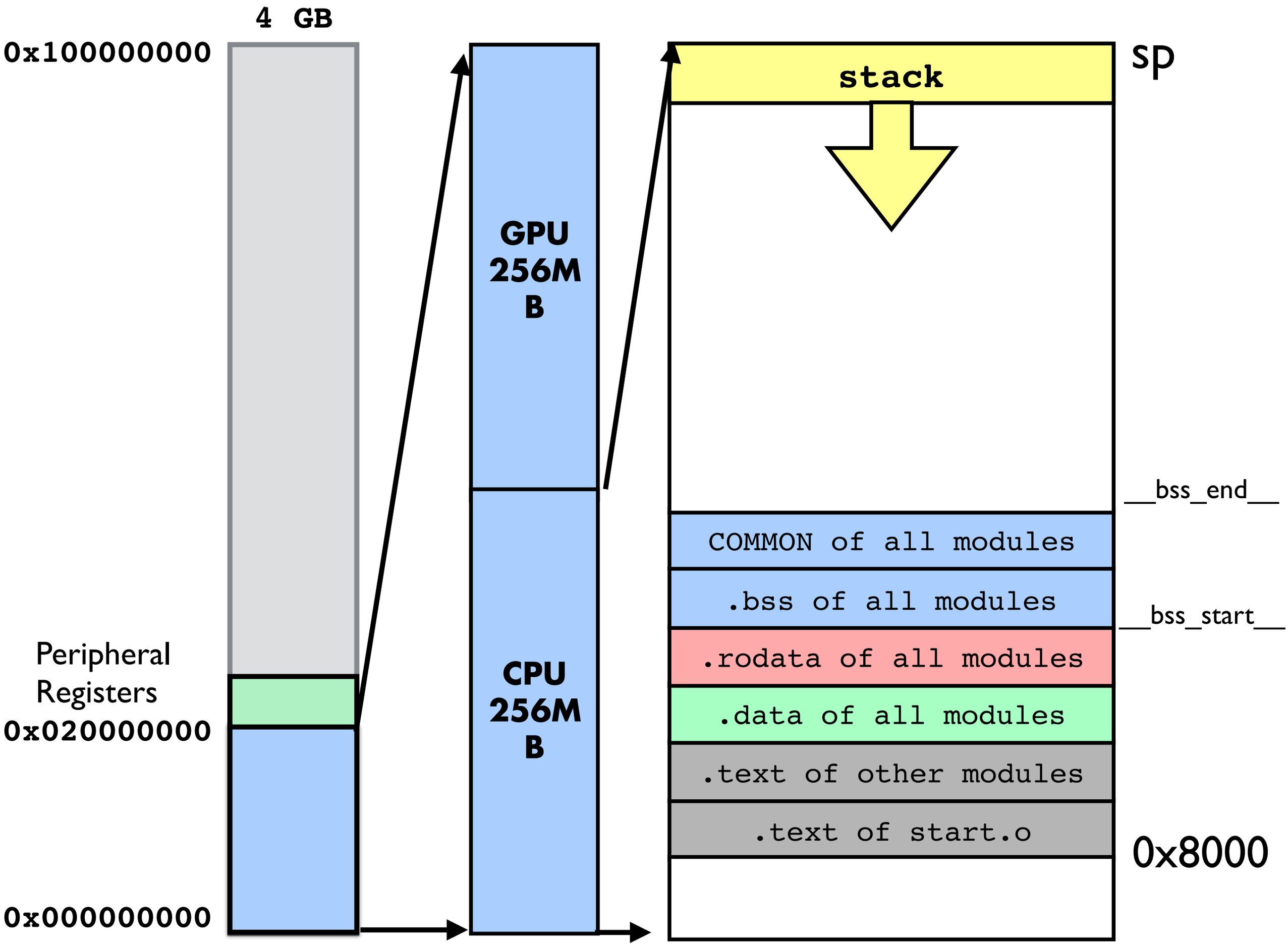
.rodata of all modules

.data of all modules

.text of other modules

.text of start.o

0x8000


4 GB
0x100000000
Peripheral
Registers
0x020000000
0x000000000
GPU
256M
B
CPU
256M
B
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

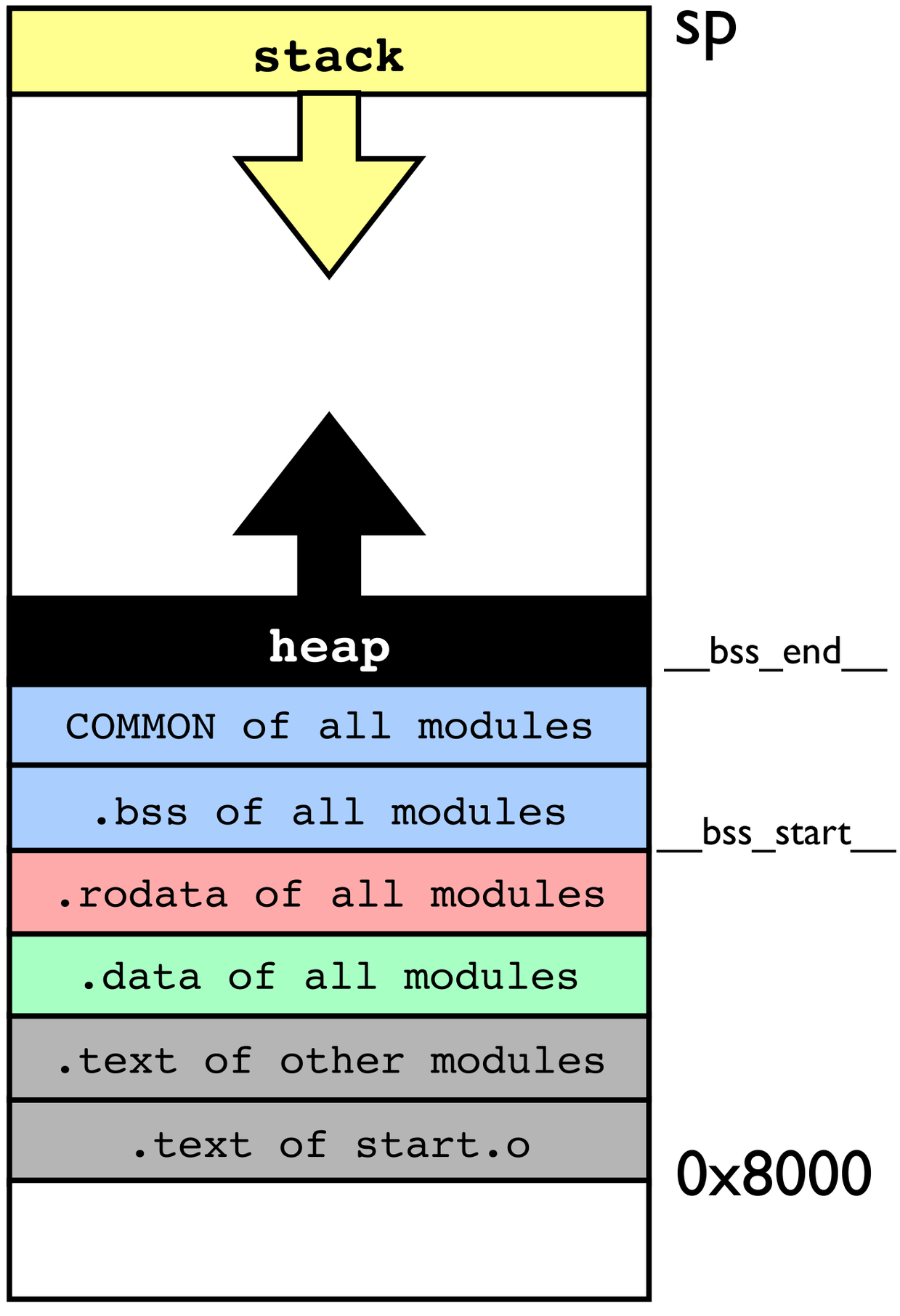
0x8000

```
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`

`void *malloc(size_t size)`

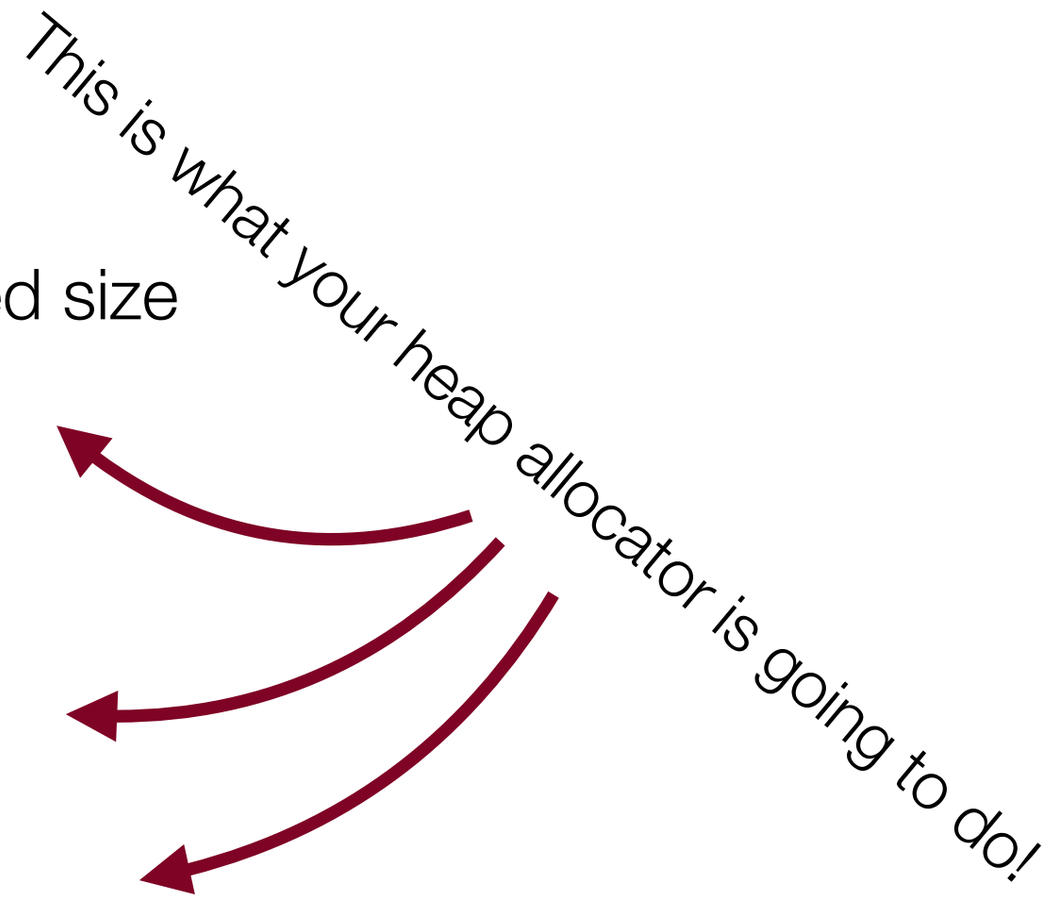
Return pointer to memory block >= requested size
(failure returns **`NULL`** and sets errno)

`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



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

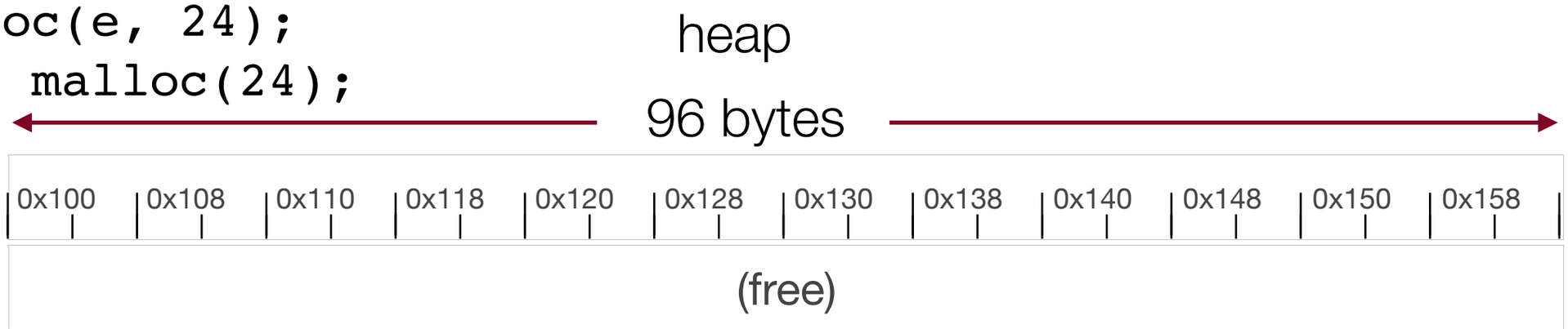
- Having `*(void **)` all over the place makes for hard-to-maintain code. Instead, use structs, and typedef when appropriate.
- The code is necessarily complex, but the more efforts you put into writing clean code, the more you will be rewarded by easier-to-maintain code.

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**



Tracing the Heap (possible implementation)

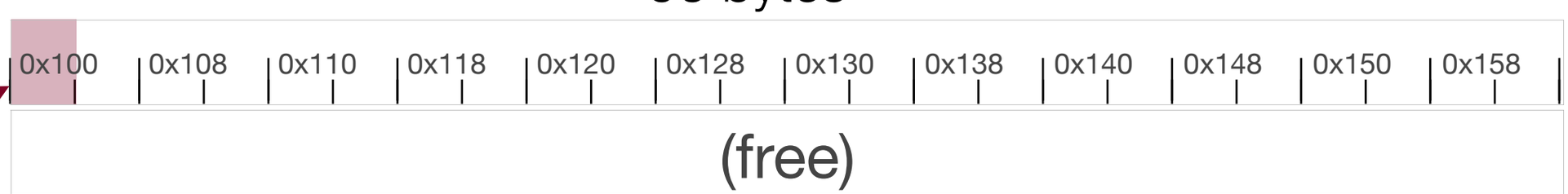
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

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



Each section represents 4 bytes

Tracing the Heap (possible implementation)

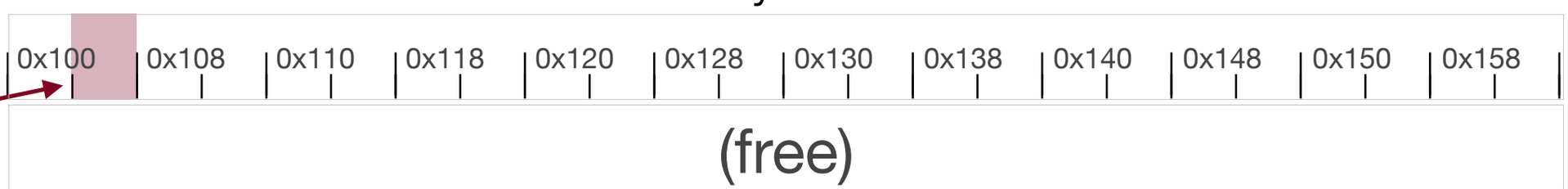
```
void *a, *b, *c, *d, *e;
a = malloc(16);
b = malloc(8);
c = malloc(24);
d = malloc(16);
free(a);
free(c);
e = malloc(8);
b = realloc(b, 24);
e = realloc(e, 24);
void *f = malloc(24);
```

All allocated on the stack:

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

heap

96 bytes



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:

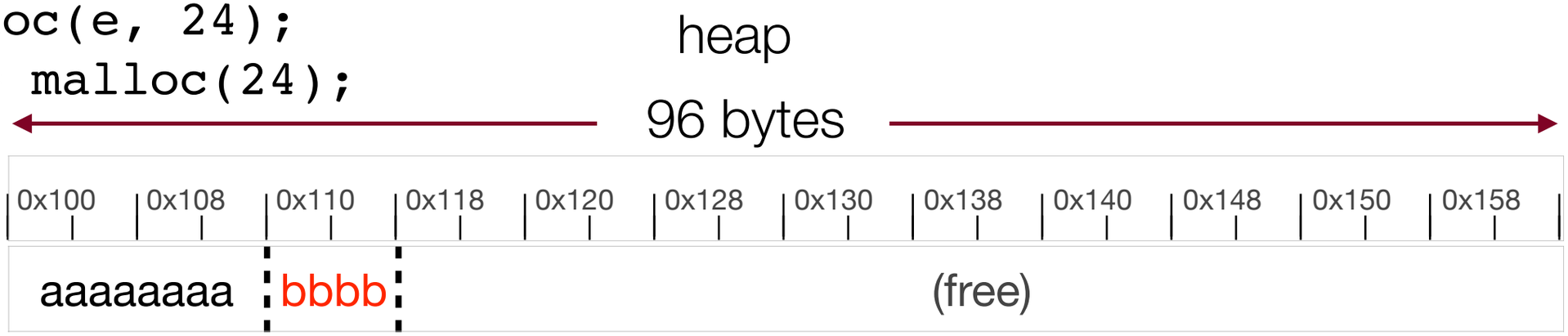
	Address	Value
e	0xffffe820	**0x0**
d	0xffffe818	**0xabcde**
c	0xffffe810	**0xf0123**
b	0xffffe808	**0x0**
a	0xffffe800	**0x100**

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**

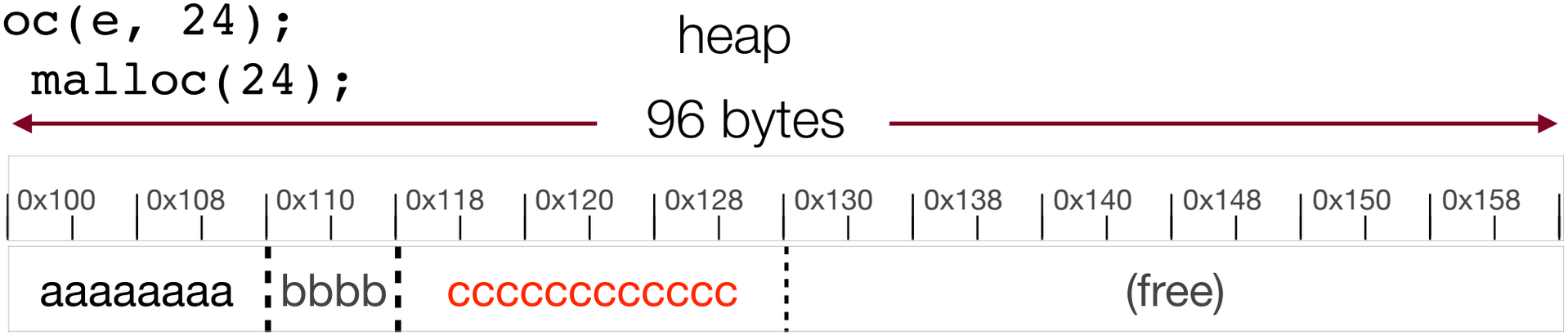


Tracing the Heap (possible implementation)

```
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**

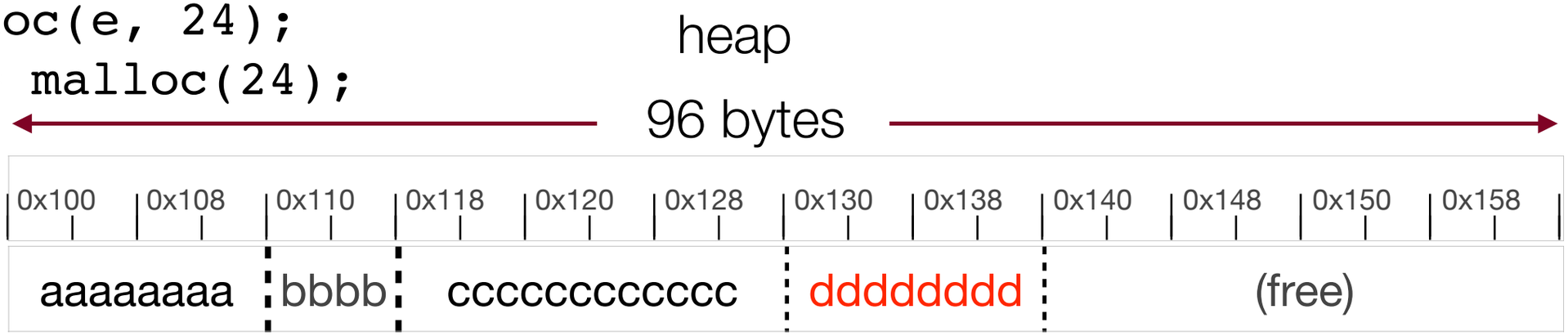


Tracing the Heap (possible implementation)

```
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**

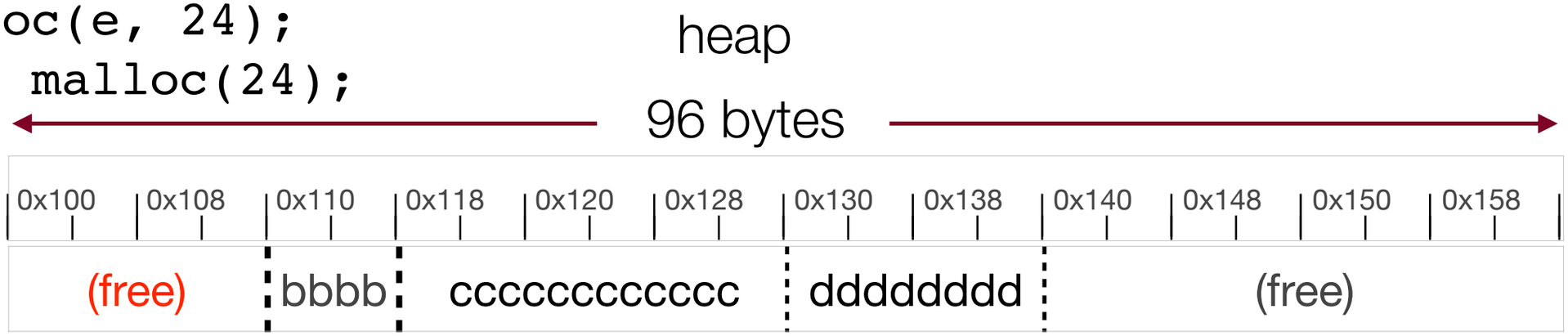


Tracing the Heap (possible implementation)

```
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**

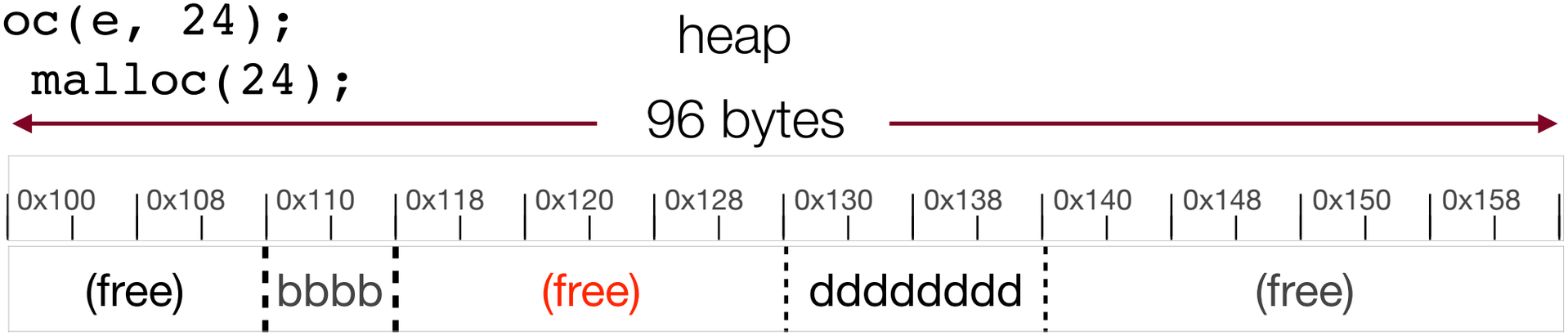


Tracing the Heap (possible implementation)

```
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**

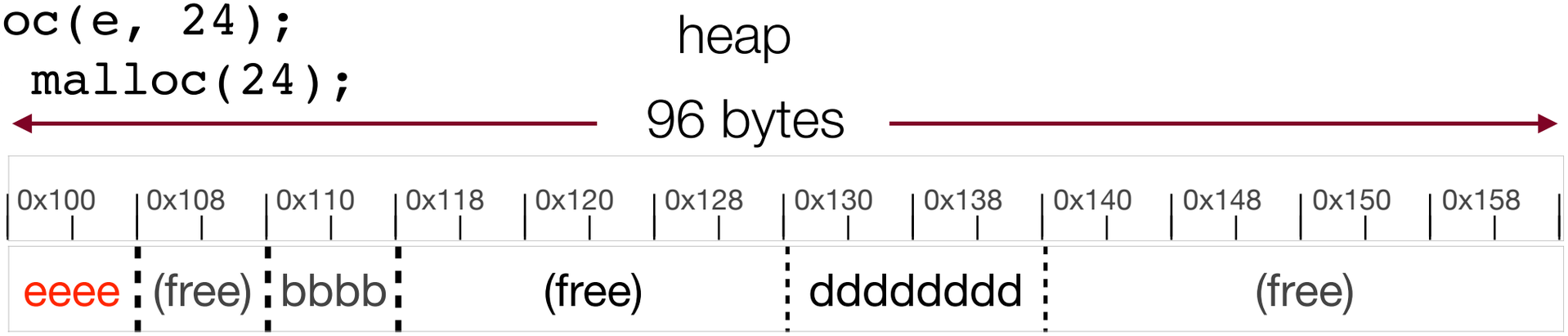


Tracing the Heap (possible implementation)

```
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**

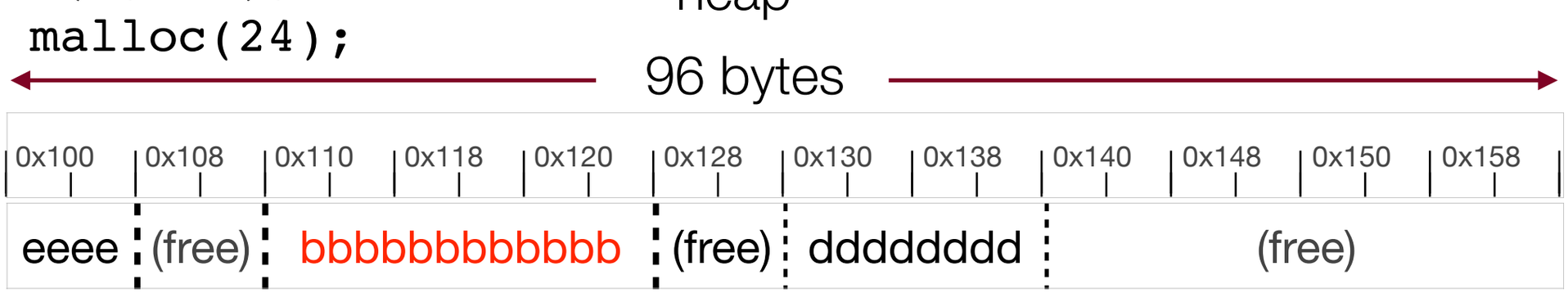


Tracing the Heap (possible implementation)

```
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**

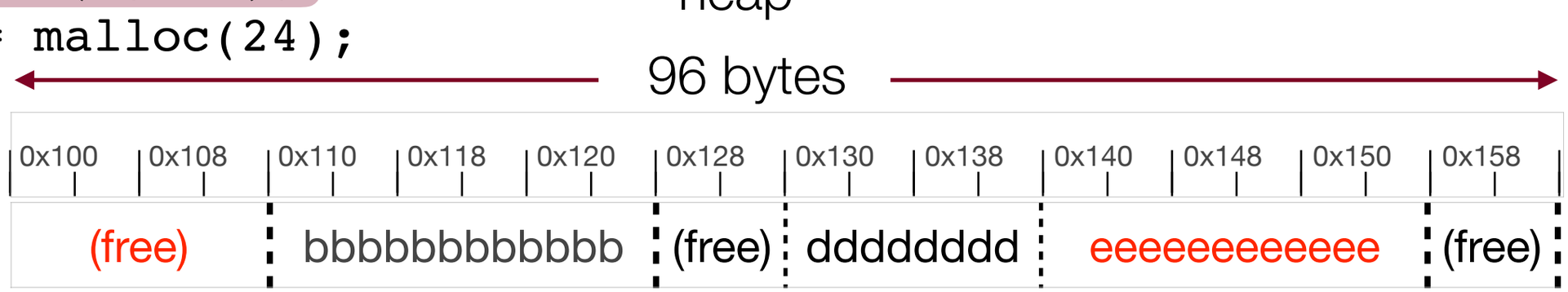


Tracing the Heap (possible implementation)

```
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**



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**
f	0xffffe7f0	**0x0**

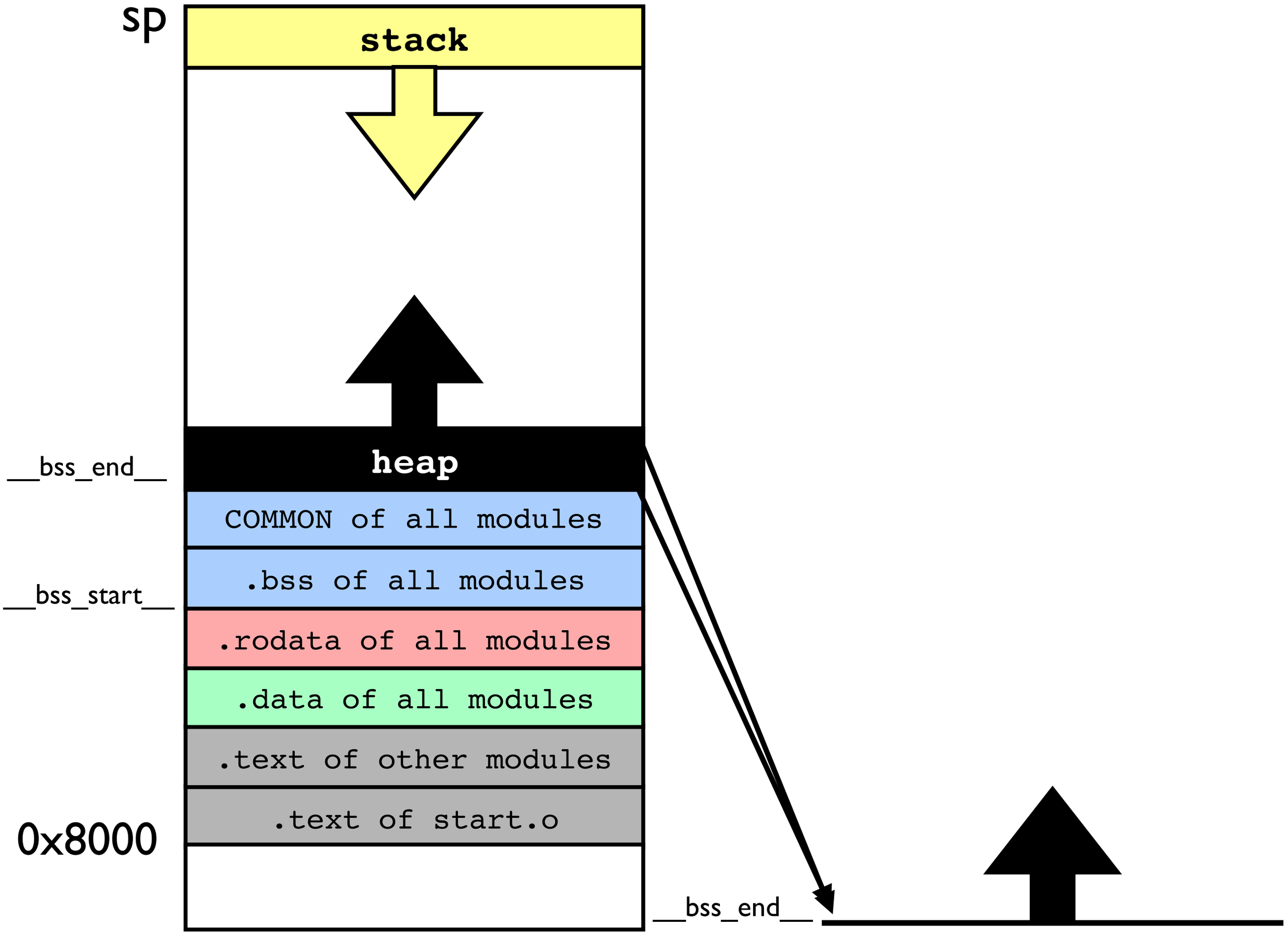
Returns `NULL`

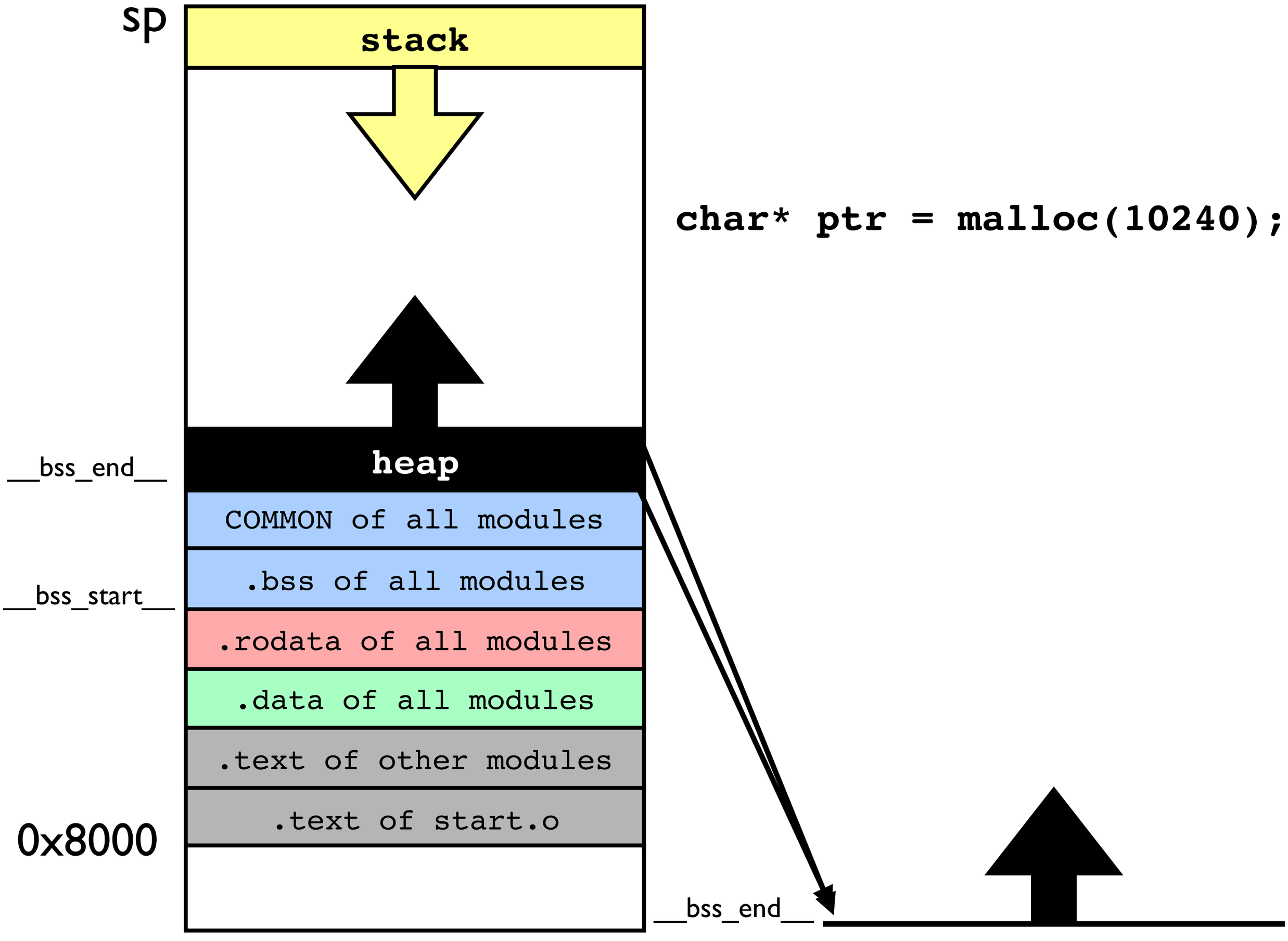
heap

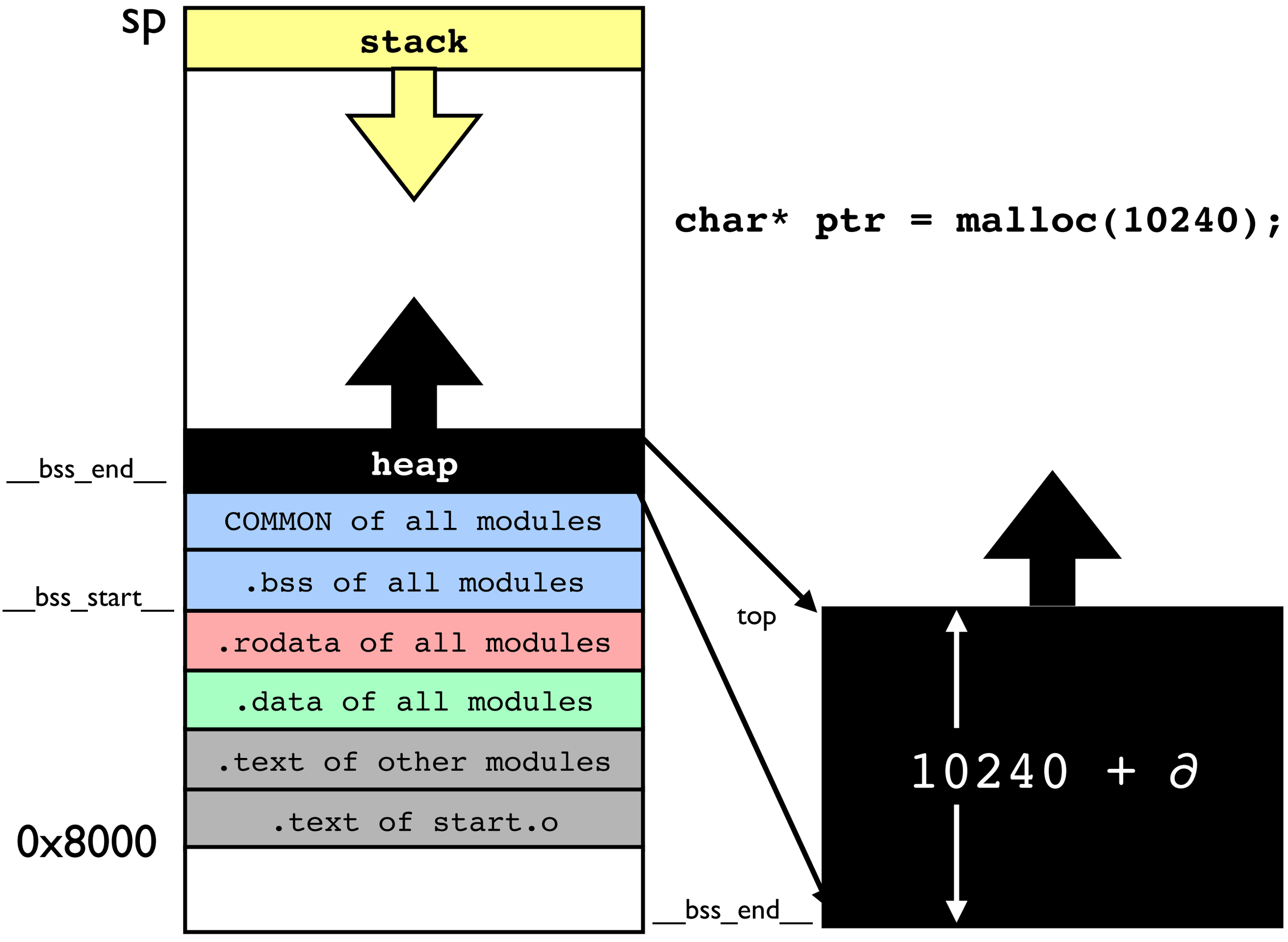
API

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

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


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
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__
char* ptr = malloc(10240);




```
char* ptr = malloc(10240);
```

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