

# 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**

**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;
```

**% 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__ = . ;
```



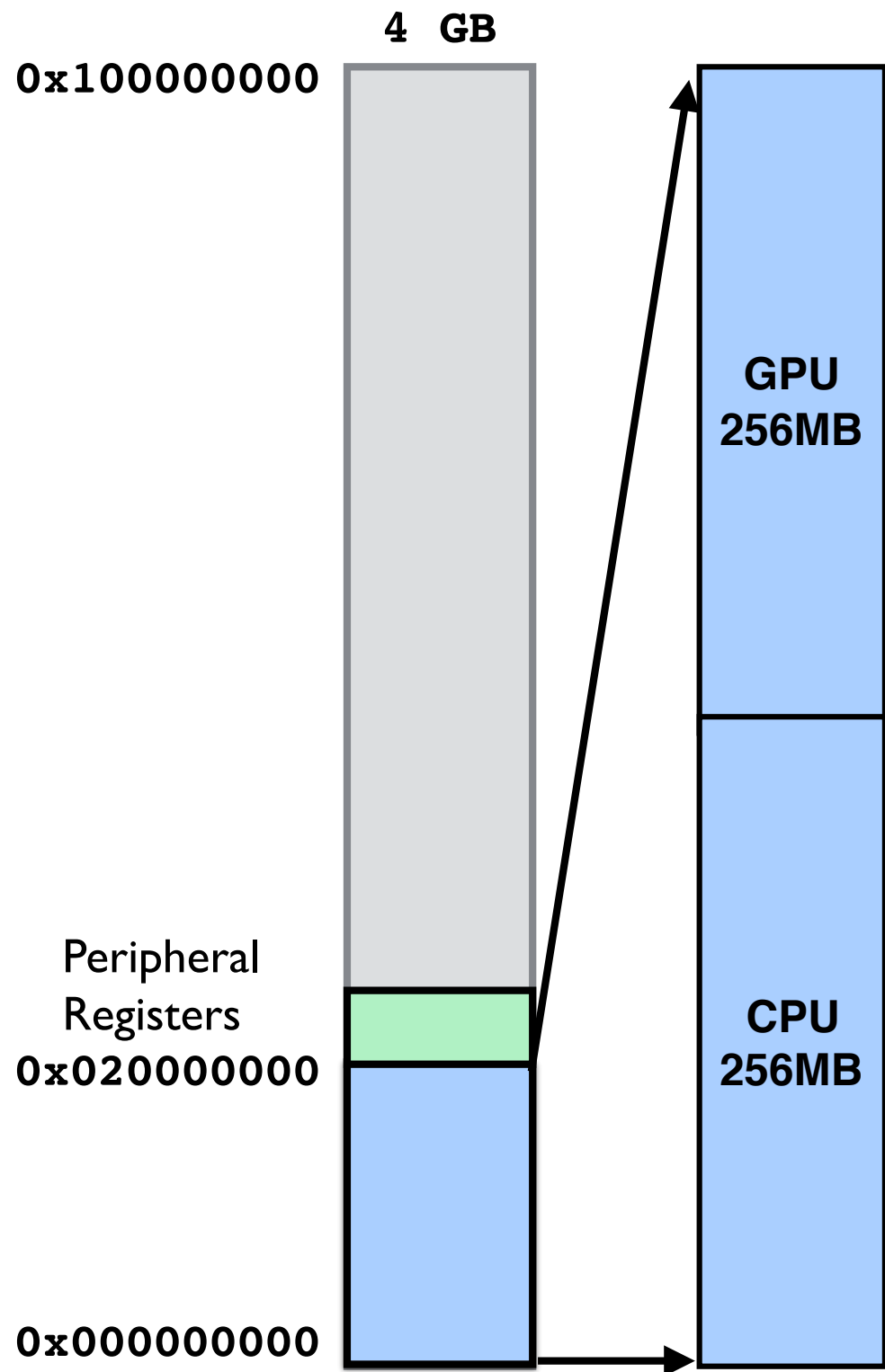
```
% arm-none-eabi-nm -n main.elf
00008000 T _start
00008008 t hang
0000800c T _cstart
0000805c T tricky
000080a8 T main
00008108 D x
0000810c d x_static
00008110 R x_const
00008114 R __bss_start__
00008114 b y_static
00008118 B y_const
0000811c B y
00008120 B __bss_end__
```

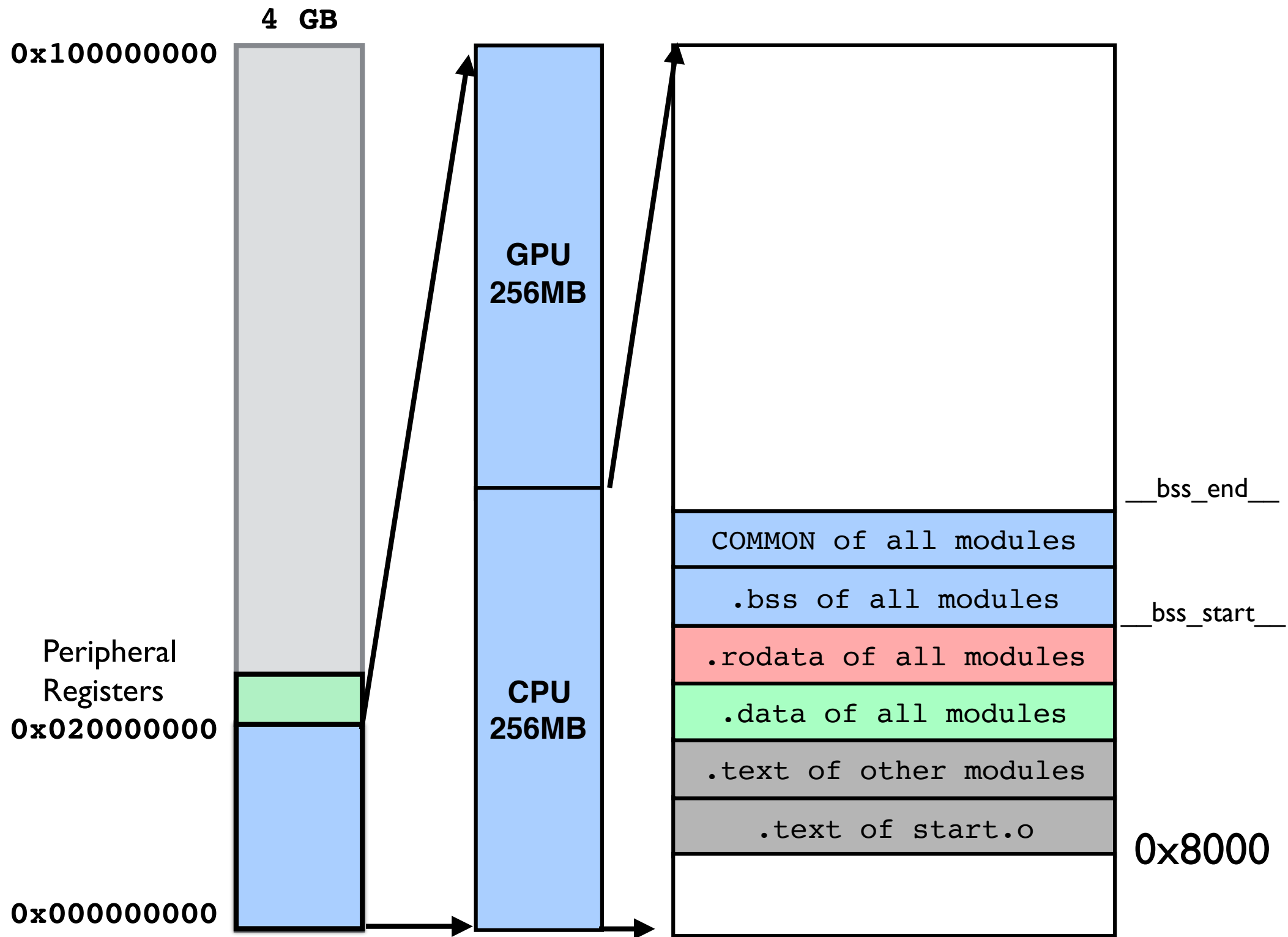
```
// cstart.c - initializes bss to 0
extern int __bss_start__;
extern int __bss_end__;

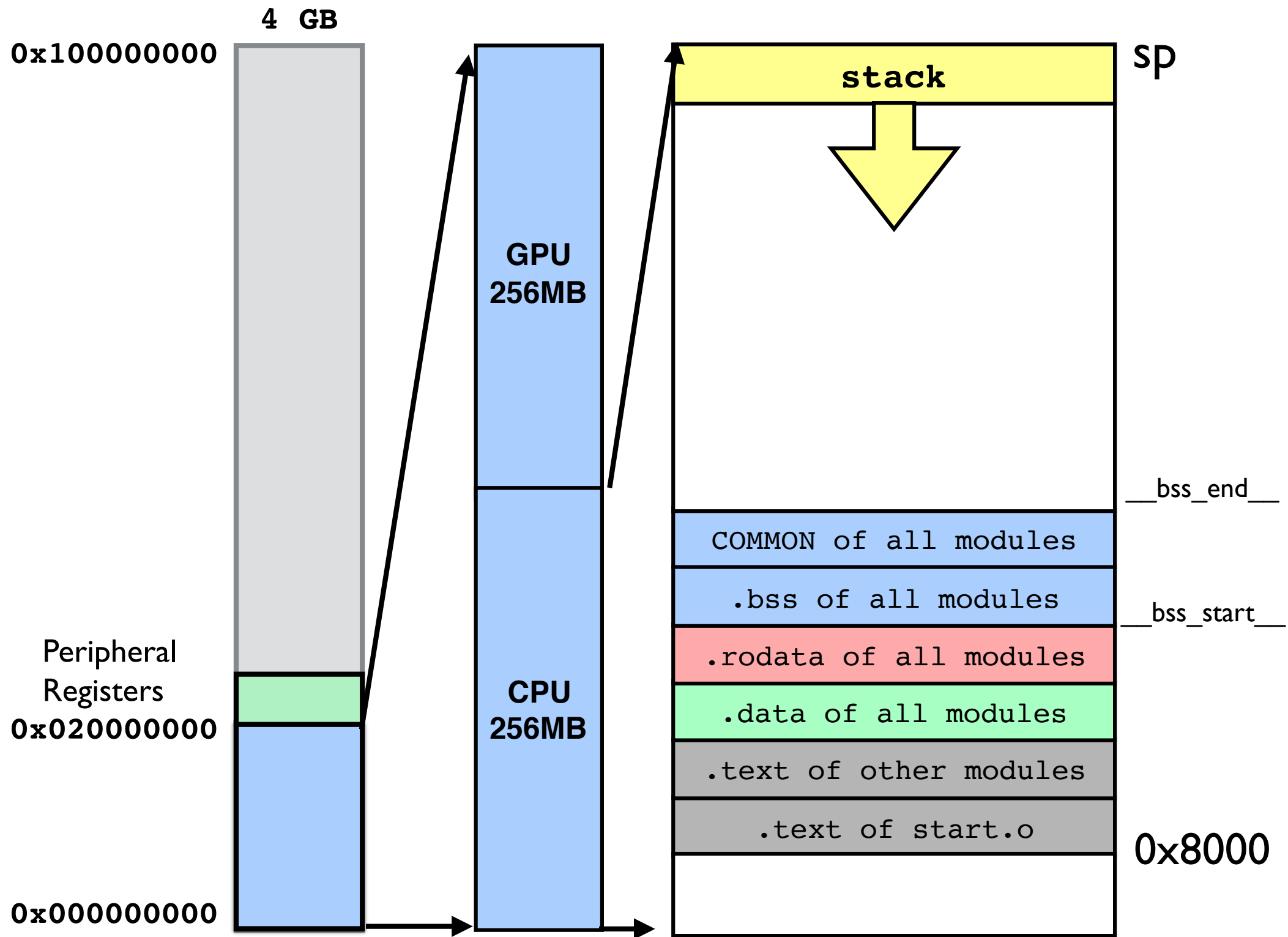
void main();

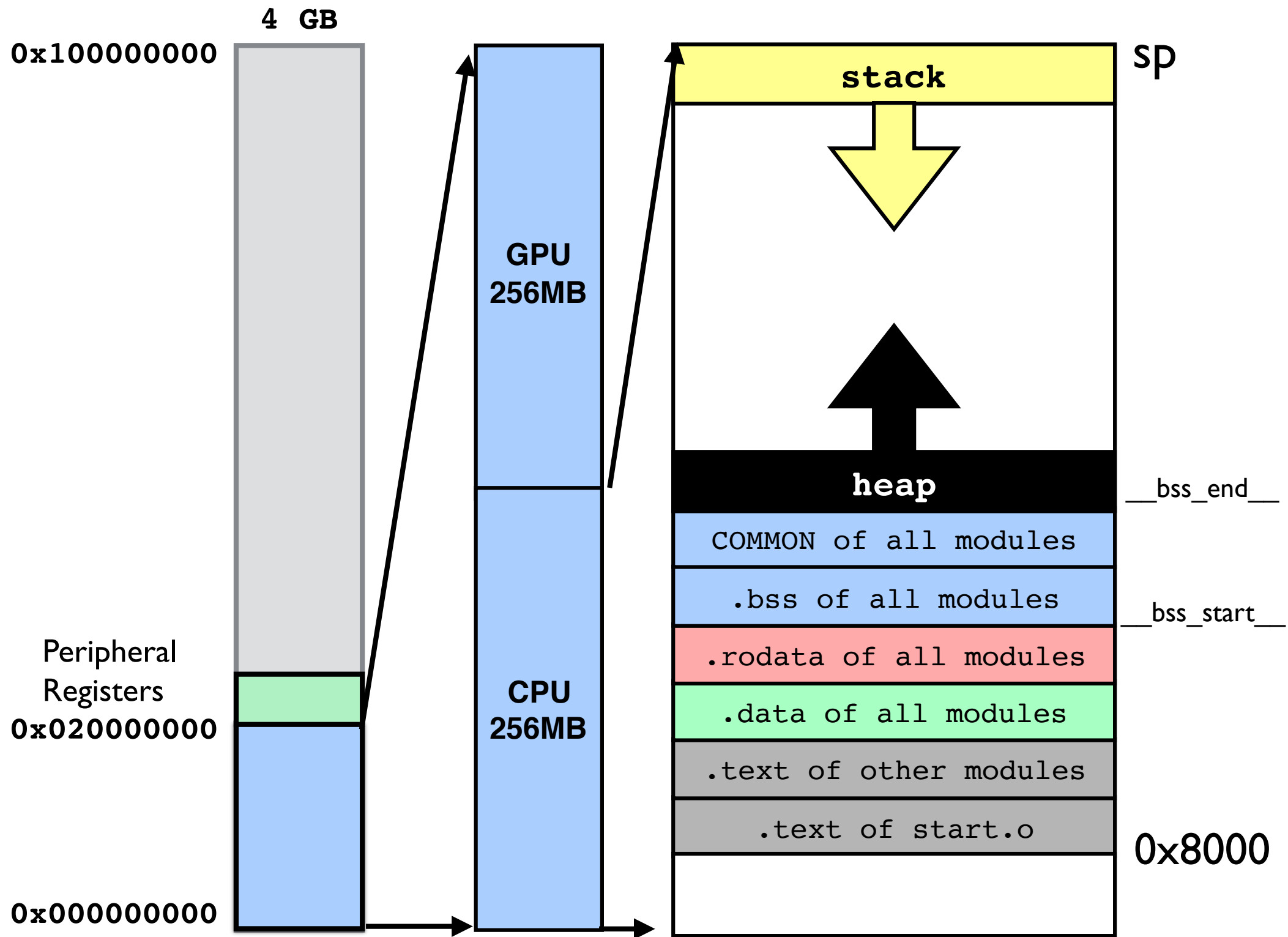
void _cstart() {
    int* bss = &__bss_start__;
    int* bss_end = &__bss_end__;
    while( bss < bss_end )
        *bss++ = 0;
    main();
}
```









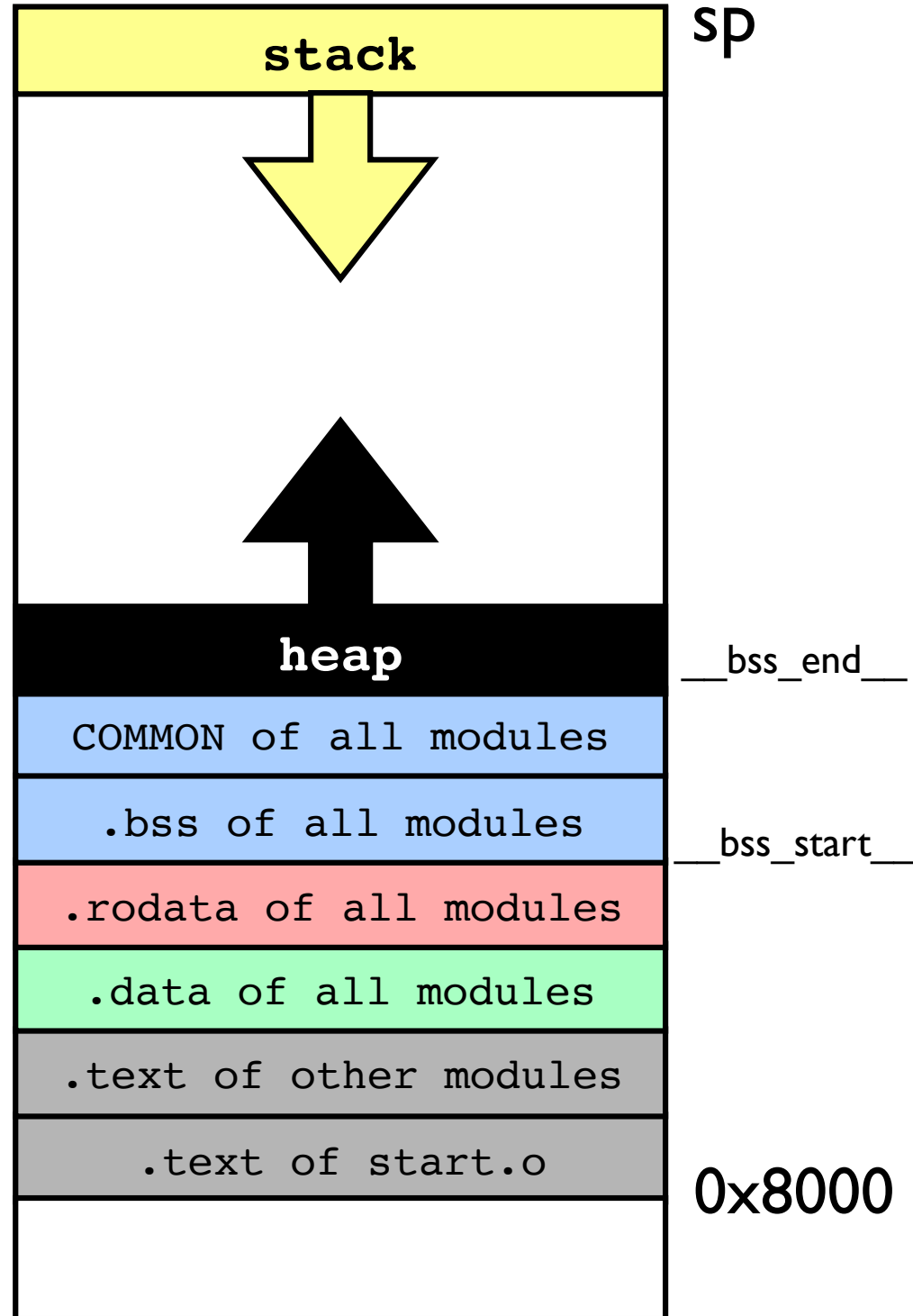


```
void f() {  
    int x;  
}
```

```
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

**Strings**

**strings.c**

# **Bump Memory Allocator**

**malloc.c**

## API

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

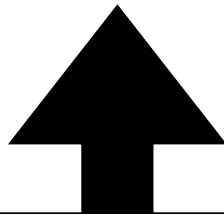
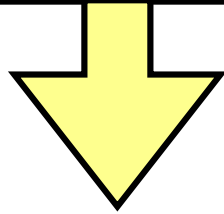
```
void free( void *pointer );
```

```
// Note that void* is a generic pointer
```

```
// Note that size_t is for sizes
```

sp

stack



heap

\_\_bss\_end\_\_

COMMON of all modules

\_\_bss\_start\_\_

.bss of all modules

.rodata of all modules

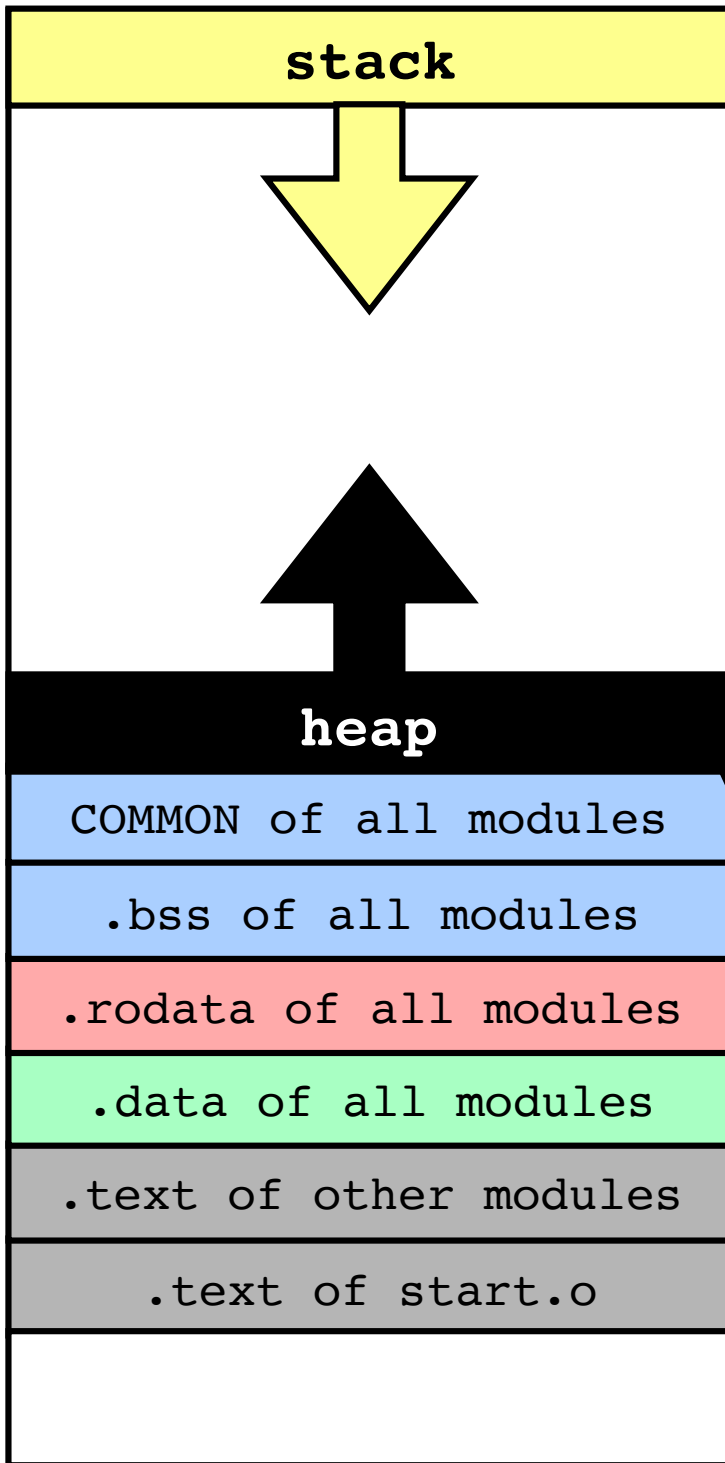
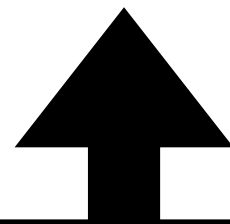
.data of all modules

.text of other modules

.text of start.o

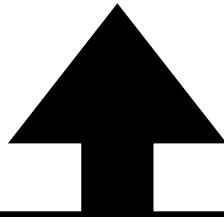
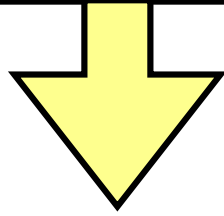
0x8000

\_\_bss\_end\_\_



sp

stack



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

\_\_bss\_end\_\_

heap

COMMON of all modules

.bss of all modules

\_\_bss\_start\_\_

.rodata of all modules

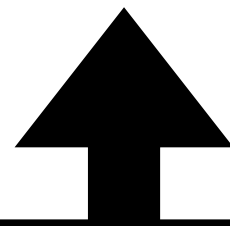
.data of all modules

.text of other modules

.text of start.o

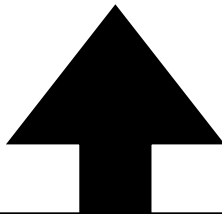
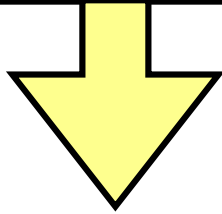
0x8000

\_\_bss\_end\_\_



sp

stack



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

\_\_bss\_end\_\_

heap

COMMON of all modules

.bss of all modules

\_\_bss\_start\_\_

.rodata of all modules

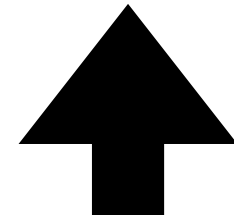
.data of all modules

.text of other modules

.text of start.o

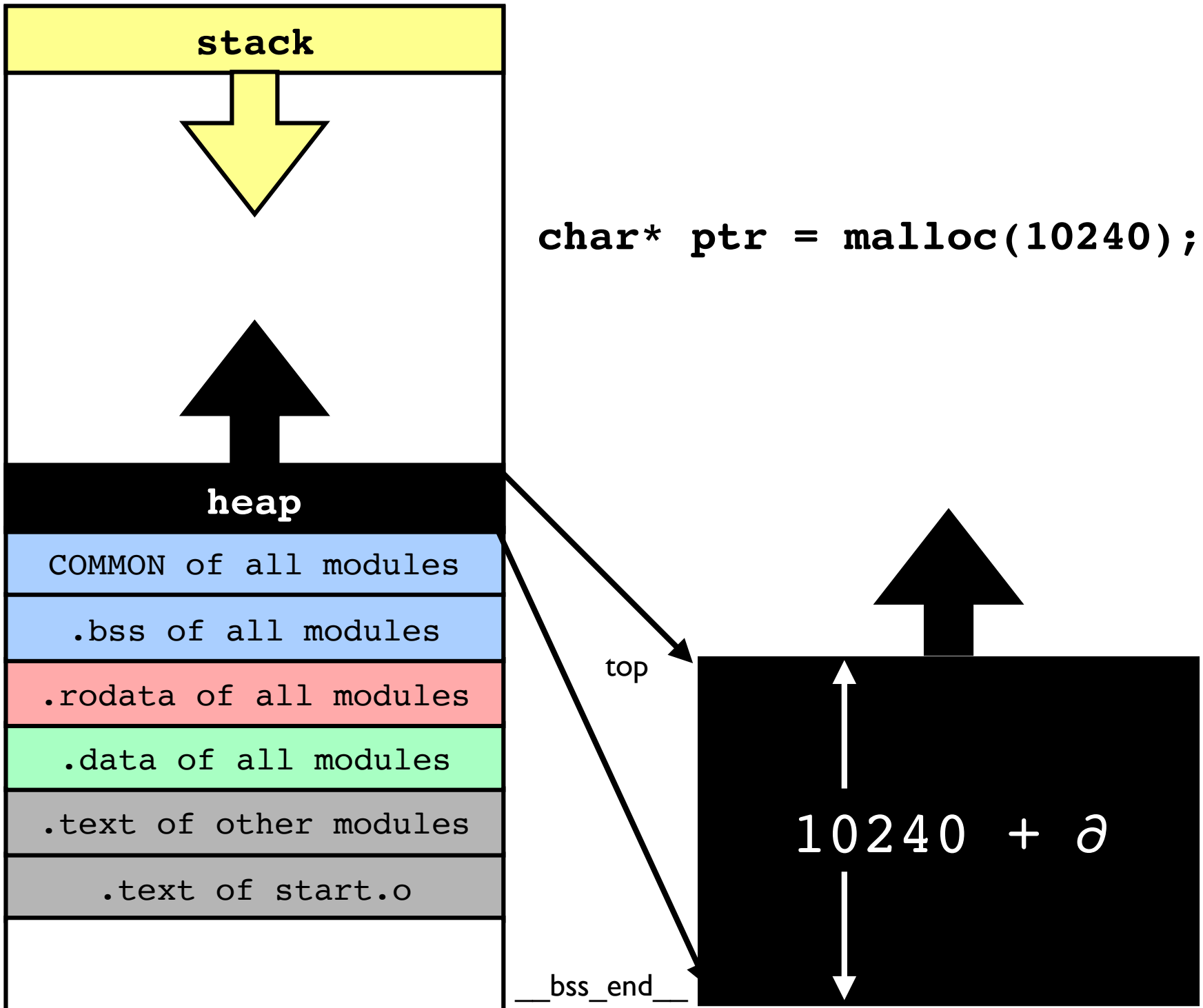
0x8000

top



10240 +  $\partial$

\_\_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?

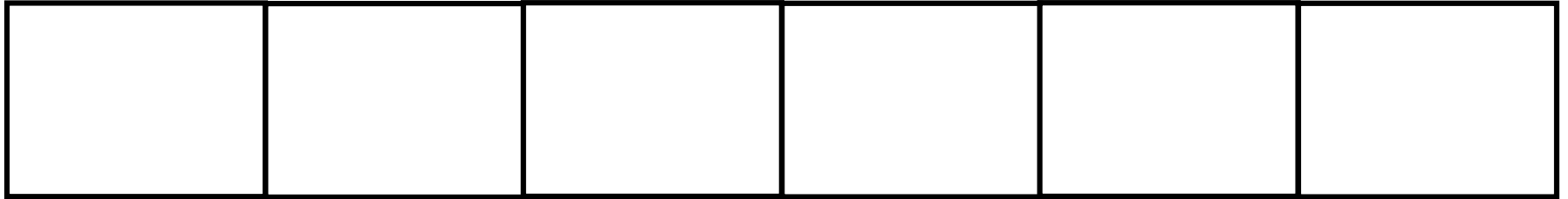
Wouldn't it be nice to not have to worry about freeing memory?

**Lists**

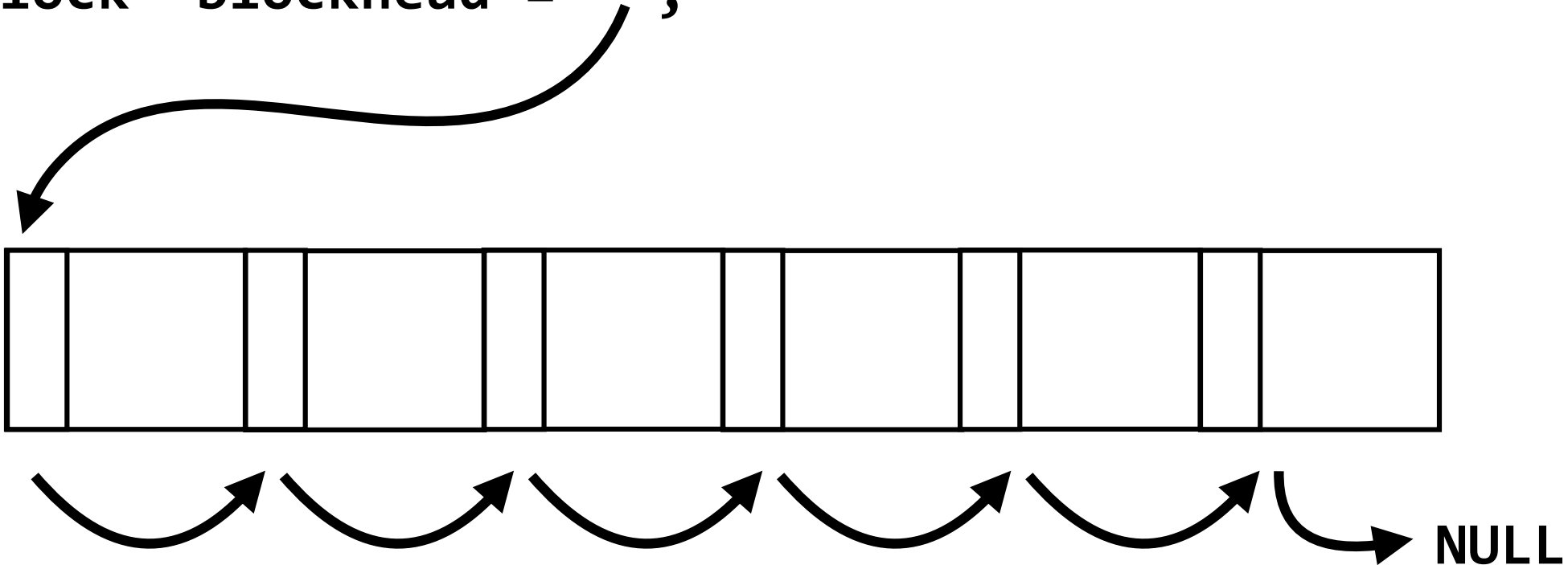
**list.c**

**block.c**

```
newblock( nelements=6, nsize=16 );
```

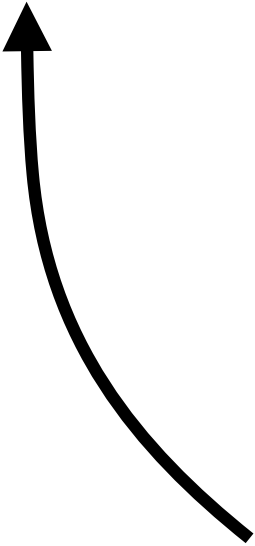
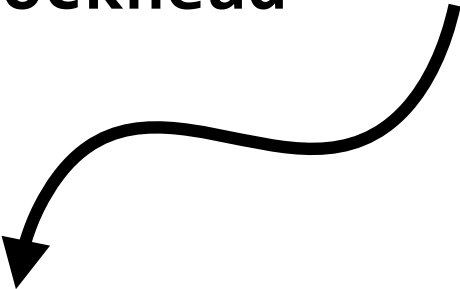
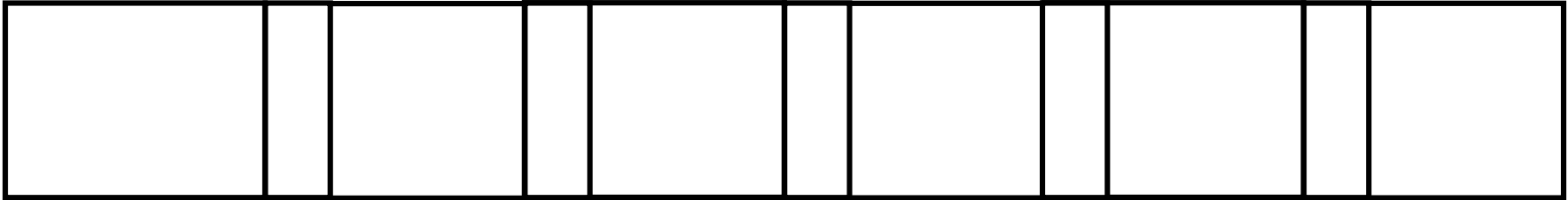


```
Block *blockhead = ;
```



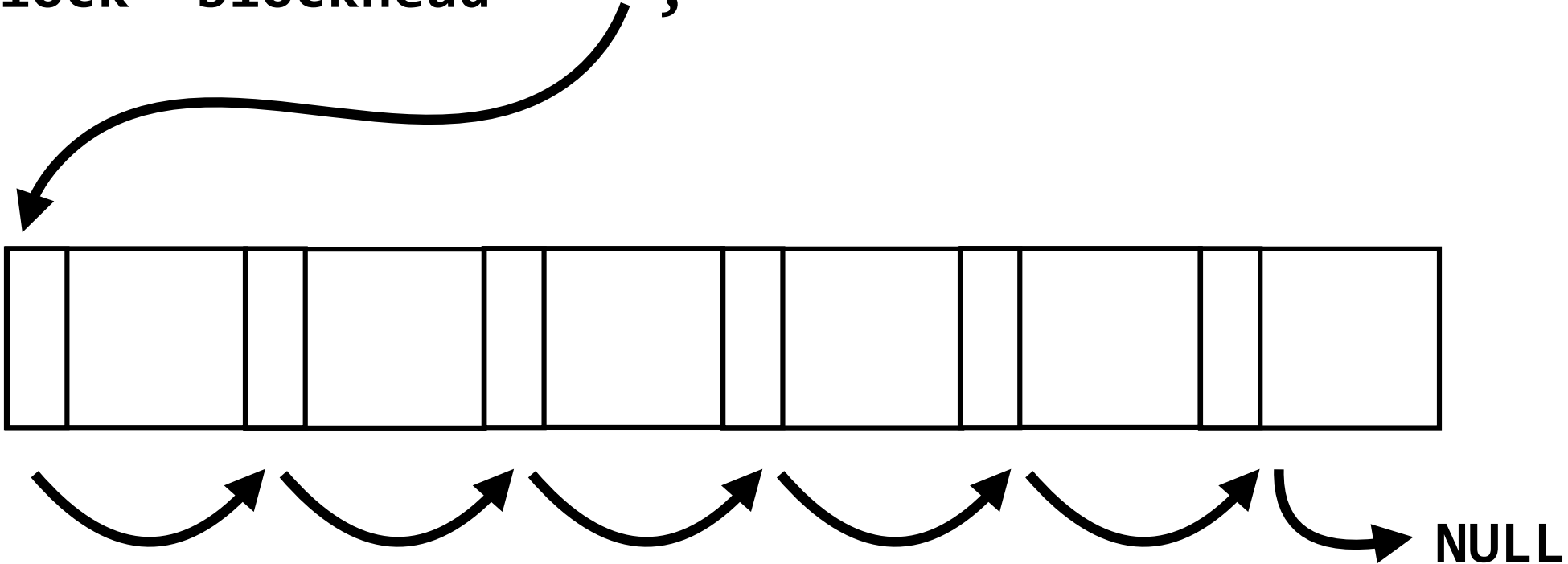
```
typedef struct b {  
    struct b *next;  
} Block;
```

```
Block *blockhead = ;
```



```
block = getblock( &blockhead, 16 );
```

```
Block *blockhead = ;
```



```
getblock( &blockhead, block );
```

# 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