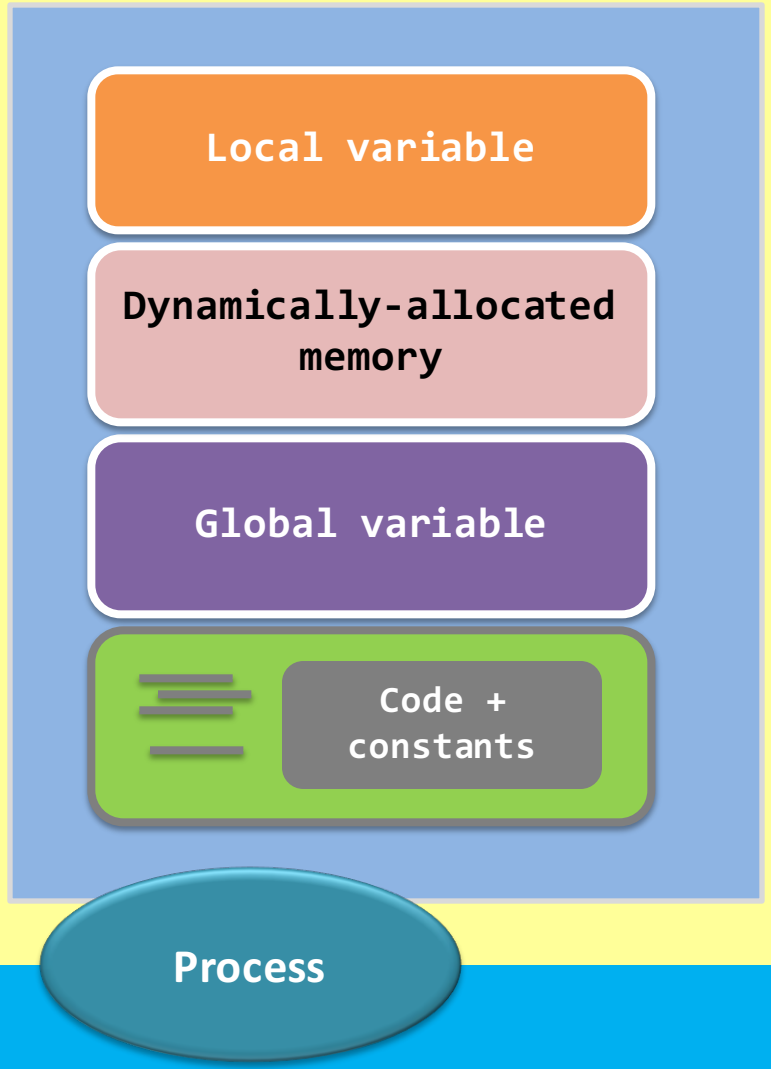
3150 - Operating Systems

Dr. WONG Tsz Yeung

Chapter 4, part 1
User-space Memory of a Process.

- In fact, it is about how modern CPUs treats a process…

Outline



Let's forget about the kernel for a moment. We are going to explore the **user-space memory** first.

If you don't know what is going on within a process, how can you understand the almighty kernel, huh?

User-space memory management

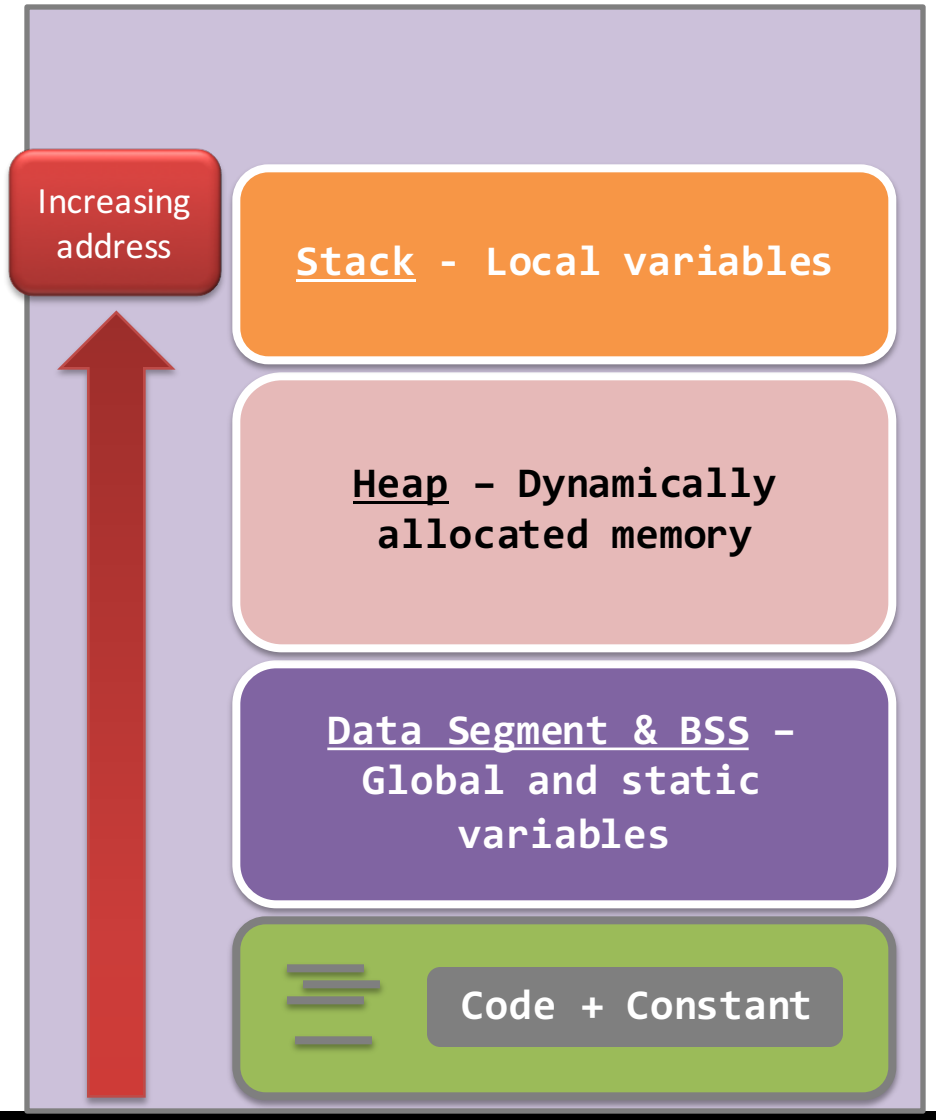
- Addressing;

Address space and pointers

```
$ ./addr
Local variable  = 0xbfa8938c
malloc() space  =  0x915c008
Global variable =  0x804a020
Code & constant =  0x8048550
$ _
```

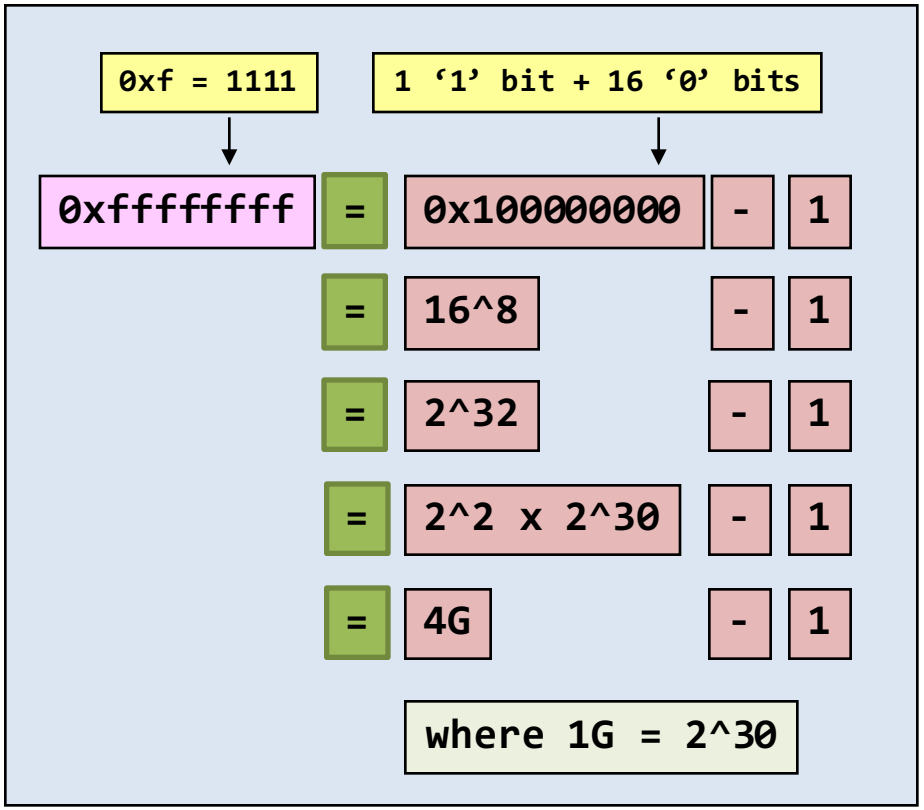
Note

The addresses are not necessarily the same in different processes, e.g., 32-bit VS 64-bit.




```
[examples@3150] cat addr.c
```

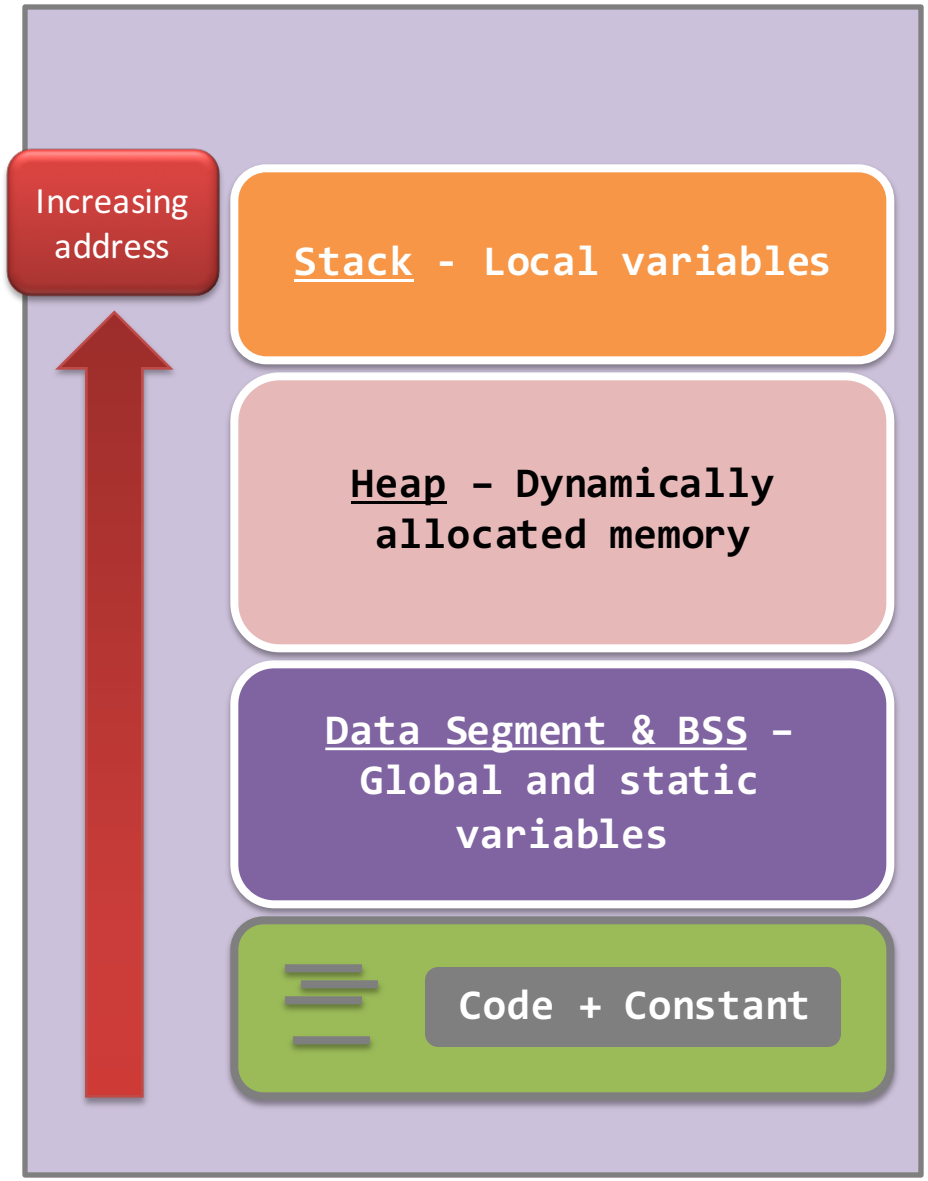
Address space and pointers



In a 32-bit system,

- One address maps to one byte.
- The maximum amount of memory in a process is **4GB**.

Then, how about a 64-bit system?



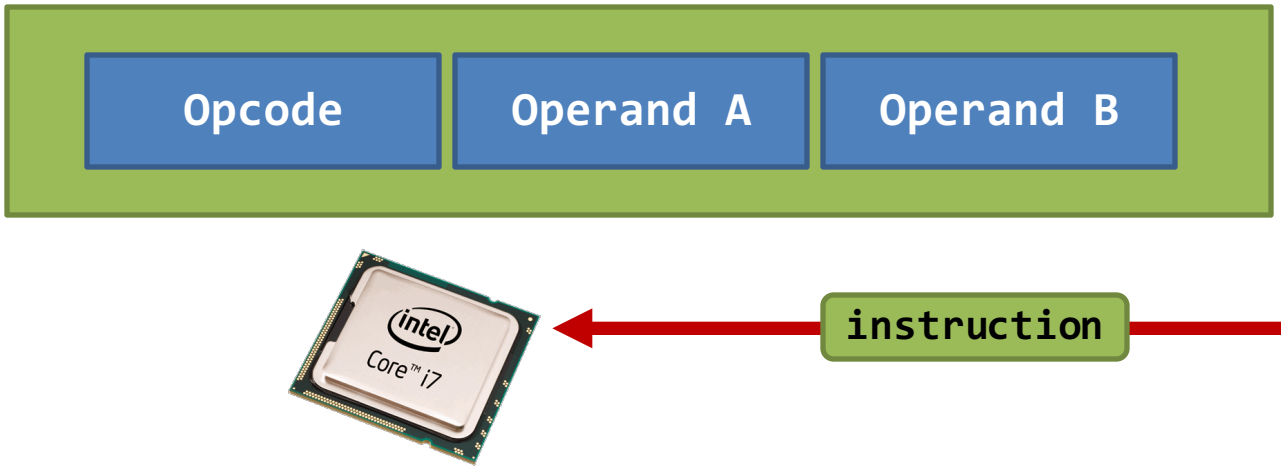
Address space and pointers

Operands are either register names or memory addresses.

So, user-space memory is about:

(1) where does the address point to?
(2) is the memory address valid or allocated?
(3) is the permission granted?

Note that it is about CPU executing an instruction, (*it seems that*) accessing an address does not involve the kernel.



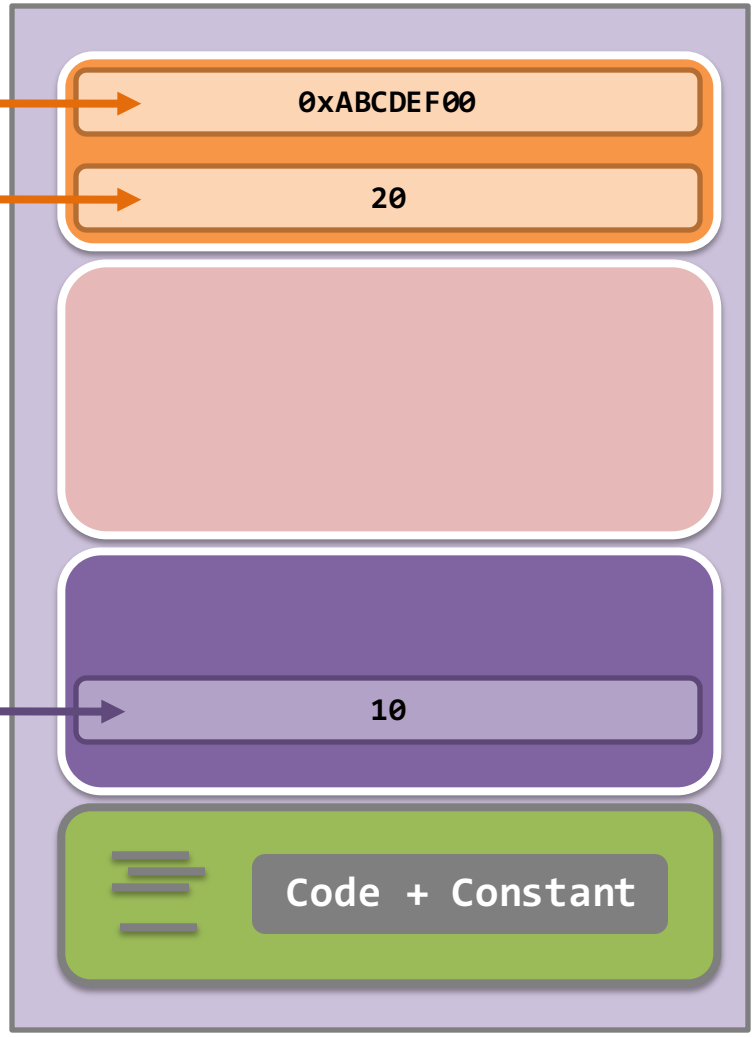
Address space and pointers

So, what is a pointer?

It is just an address or 4 bytes, specifying a memory location.

But, whether the address is **pointing to something valid** is another story!

```
1   int global_int = 10;
2
3   int main(void) {
4       int local_int = 20, *ptr;
5
6       ptr = &local_int;
7       ptr = &global_int;
8       return 0;
9   }
```



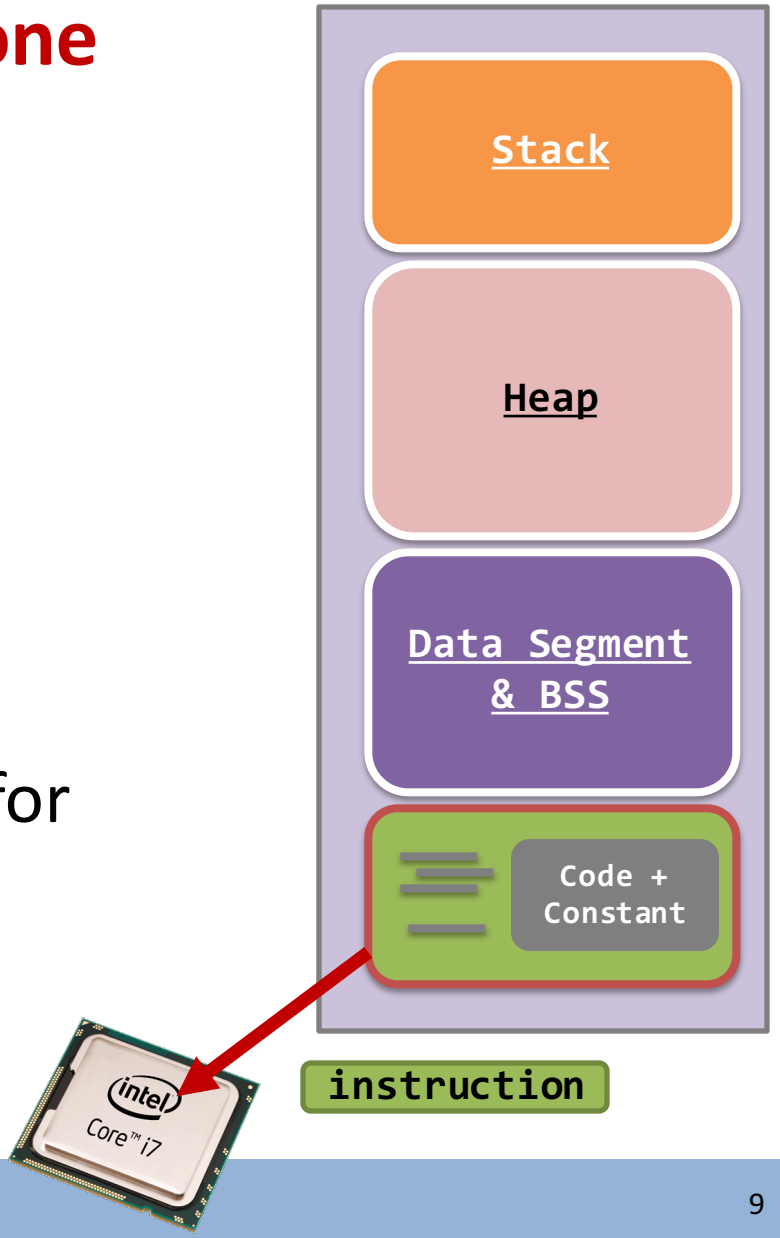
User-space memory management

- Addressing;
- Code & constants;

Program code & constants

- A process is **not bounded to one program code**.
  - Remember **`exec*()`** family?

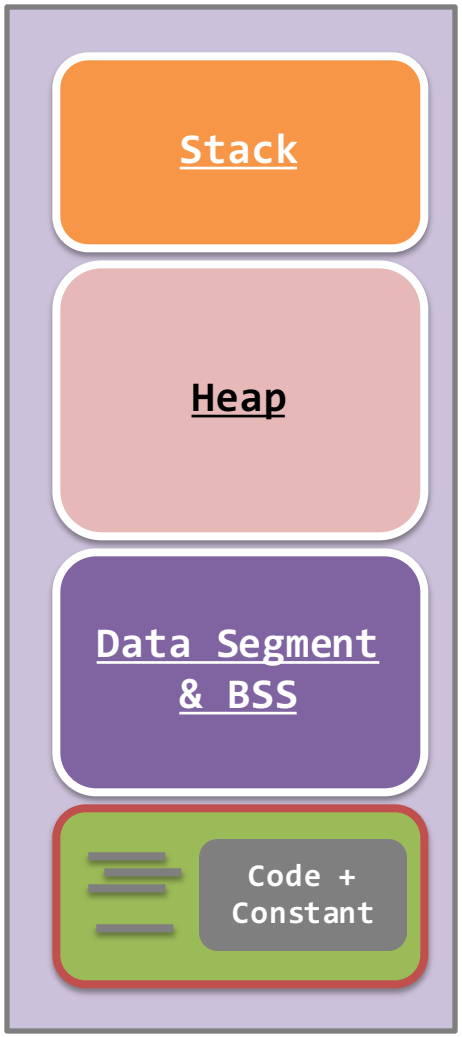
- The program code requires memory space because…
  - The CPU needs to fetch the instructions from the memory for execution.



Program code & constants

```
1  int main(void) {
2      char *string = "hello";
3      printf("\"hello\"       = %p\n", "hello");
4      printf("String pointer = %p\n", string);
5      string[4] = '\0';
6      printf("Go to %s\n", string);
7      return 0;
8  }
```

char string[] = “hello”;



- **Question #1.** What are the printouts from Line 3 & 4?

  both printout is ok

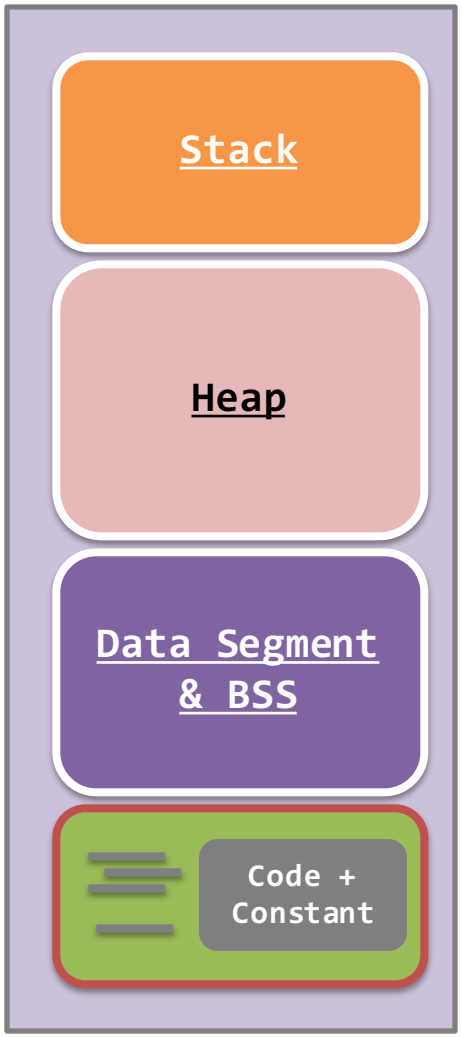
- **Question #2.** What is the printout from Line 6?

  segmentation fault (string copy problem) but the later one can print

```
[examples@3150] cat constant_ptr.c constant_array.c
```

Program code & constants

```
 1   int main(void) {
 2       void *ptr = main;
 3       unsigned char c = *((unsigned char *) ptr);
 4
 5       printf("Read  : 0x%x\n", c);
 6       printf("Write : ");
 7       *((unsigned char *) ptr) = 0xff;
 8       printf("done\n");
 9       return 0;
10   }
```

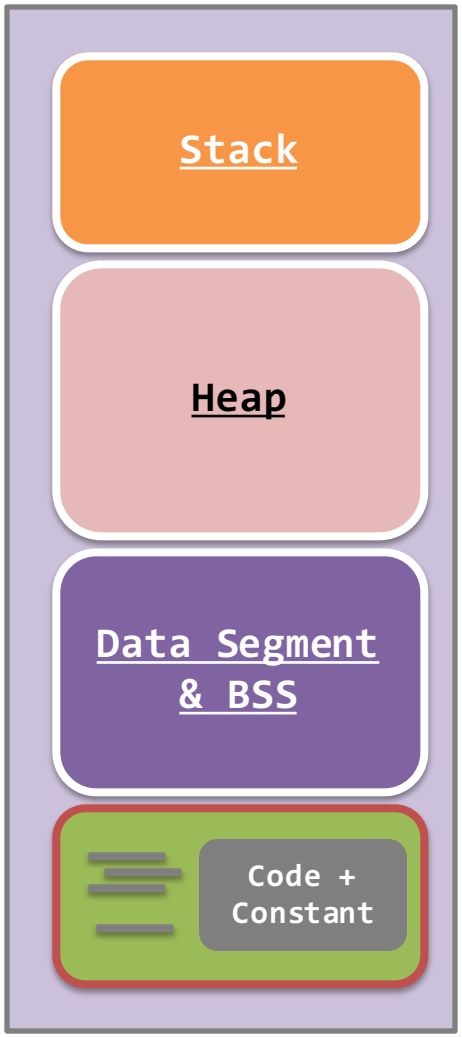


- **Question #1.** What Line 2 is doing?
- **Question #2.** Will Lines 5 or 7 fail?

```
[examples@3150] cat access_code.c
```

Program code & constants

- Summary:
  - Codes and constants are both **read-only**.
    - You cannot change the values of the codes nor the constants during runtime.
  - Constants are stored in code segment.
    - Only store the **unique constants**.
    - Accessing of constants are done using addresses (or pointers).



User-space memory management

- Addressing;
- Code & constants;
- **Data segment;**

Data Segment & BSS – properties

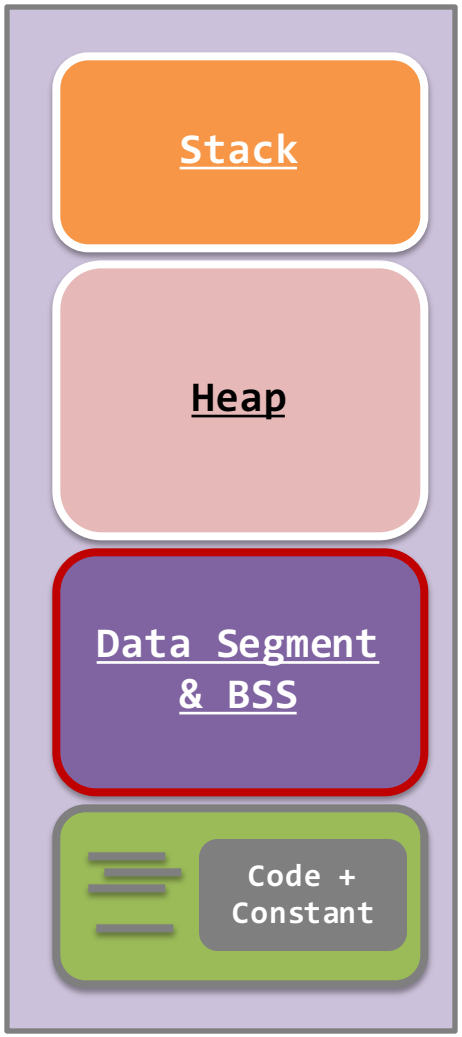
- Data
  - Containing initialized global and static variables.

ulimit -a

- BSS (**B**lock **S**tarted by **S**ymbol)
  - Containing uninitialized global and static variables.

- Both spaces will be allocated at the moment that the program starts.

What?! Are you tell me that a static variable is the same as a global variable?!



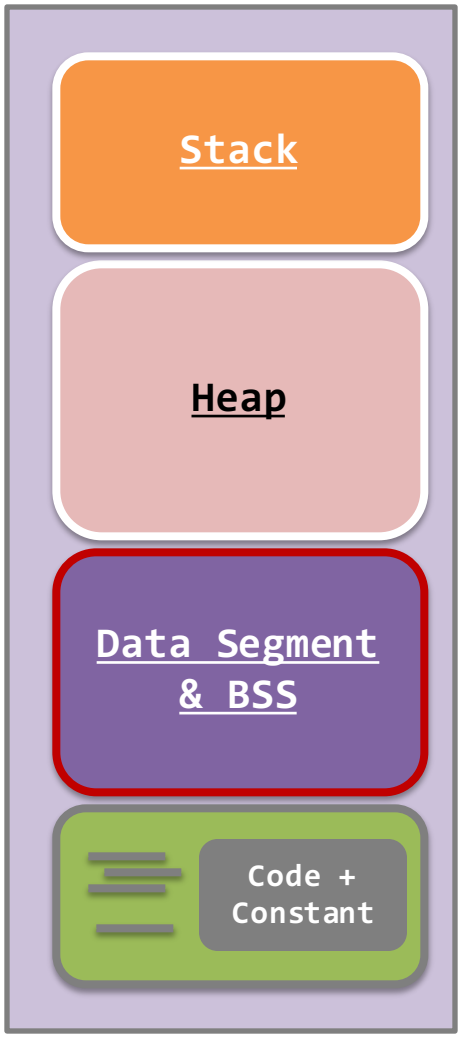
Data Segment & BSS – properties

```
int global_int = 10;
int main(void) {
    int local_int = 10;
    static int static_int = 10;
    printf("local_int  addr = %p\n", &local_int );
    printf("static_int addr = %p\n", &static_int );
    printf("global_int addr = %p\n", &global_int );
    return 0;
}
```

```
$ ./global_vs_static
local_int  addr = 0xbf8bb8ac
static_int addr = 0x804a018
global_int addr = 0x804a014
$_
```

See? They are stored next to each other.

This implies that they are **in the same segment!**

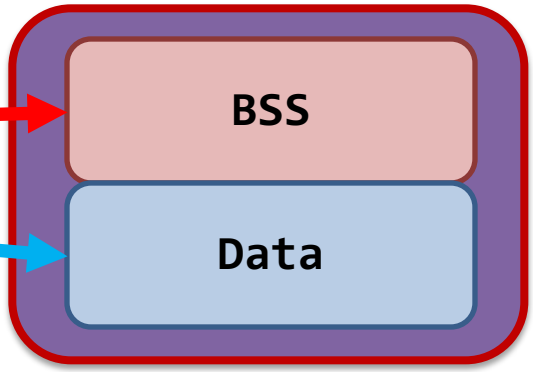


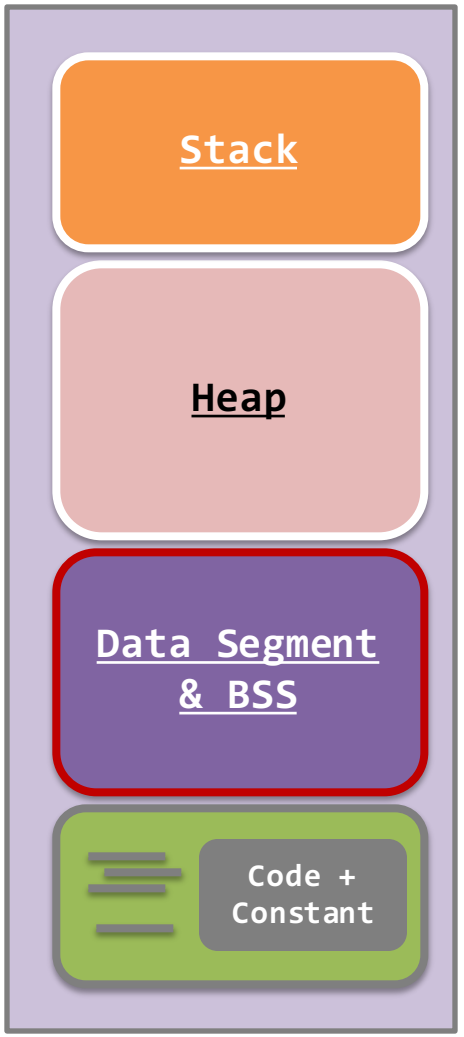
[examples@3150] cat global_vs_static.c

Data Segment & BSS – locations

```
 1  int global_bss;
 2  int global_data = 10;
 3  int main(void) {
 4      static int static_bss;
 5      static int static_data = 10;
 6      printf("global  bss = %p\n", &global_bss  );
 7      printf("static  bss = %p\n", &static_bss  );
 8      printf("global data = %p\n", &global_data );
 9      printf("static data = %p\n", &static_data );
10  }
```

```
$ ./data_vs_bss
global  bss = 0x804a028
static  bss = 0x804a024
global data = 0x804a014
static data = 0x804a018
$_
```






```
[examples@3150] cat data_vs_bss.c
```

Data Segment & BSS – sizes

```
char a[1000000] = {10};

int main(void) {
    return 0;
}
```

Program: data_large.c

```
char a[100] = {10};

int main(void) {
    return 0;
}
```

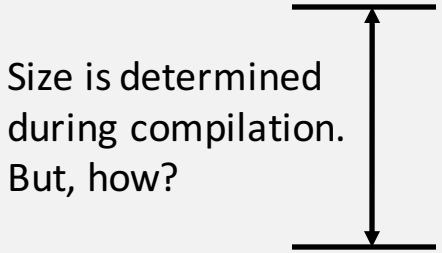
Program: data_small.c

No optimization.

```
$ gcc -O0 -o data_large data_large.c
$ gcc -O0 -o data_small data_small.c

$ ls -l data_small   data_large
```

Guess! Which one is large?

Size is determined during compilation. But, how?




```
[examples@3150] cat data_small.c data_large.c
```

Data Segment & BSS – sizes

```
char a[1000000] = {10};

int main(void) {
    return 0;
}
```

Program: data_large.c

```
char a[100] = {10};

int main(void) {
    return 0;
}
```

Program: data_small.c

```
$ gcc -O0 -o data_large data_large.c
$ gcc -O0 -o data_small data_small.c

$ ls -l data_small   data_large
-rwxr-xr-x ... 1007174 ... data_large
-rwxr-xr-x ...    7240 ... data_small
$_
```

Wow!

See? The global variables that you're hard-coding will be stored in the code!

Remember, the constant "10" is in the code segment. Yet, the data segment has the required space **already allocated** for the constant.

```
[examples@3150] cat data_small.c data_large.c
```

Data Segment & BSS – sizes

```
char a[1000000];

int main(void) {
    return 0;
}
```

Program: bss_large.c

```
char a[100];

int main(void) {
    return 0;
}
```

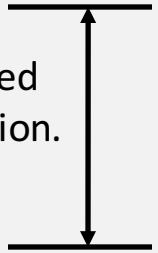
Program: bss_small.c

No optimization.

```
$ gcc -O0 -o bss_large bss_large.c
$ gcc -O0 -o bss_small bss_small.c

$ ls -l bss_small   bss_large
```

Guess! Which one is large?

Size is determined during compilation. But, how?



BSS

Data

```
[examples@3150] cat bss_small.c bss_large.c
```

Data Segment & BSS – sizes

```
char a[1000000];

int main(void) {
    return 0;
}
```

Program: bss_large.c

```
char a[100];

int main(void) {
    return 0;
}
```

Program: bss_small.c

```
$ gcc -O0 -o bss_large bss_large.c
$ gcc -O0 -o bss_small bss_small.c

$ ls -l bss_small   bss_large
-rwxr-xr-x ... 7130 ... bss_large
-rwxr-xr-x ... 7130 ... bss_small
$_
```

Same size!

This is why BSS is called "*Block Started by Symbol*".

To the program, it is just a bunch of symbols. **The space is not yet allocated**.

The space will be allocated to the process once it starts executing.

```
[examples@3150] cat bss_small.c bss_large.c
```

Data Segment & BSS – sizes

The "**size**" program allows you to see the size of the TEXT segment, DATA segment, and the BSS.

```
$ size bss_small bss_large data_small data_large

   text    data     bss     dec     hex filename
    836     260     132    1228     4cc bss_small
    836     260 1000032 1001128    f46a8 bss_large
    836     384       8    1228     4cc data_small
    836 1000284       8 1001128    f46a8 data_large

$ _
```

Just the size in different bases.

Data Segment & BSS – limits

```
$ ulimit -a
  core file size    (blocks, -c) 0
  data seg size     (kbytes, -d) unlimited
  ......
$ _
```

In Linux, "**ulimit**" is a built-in command in "**/bin/bash**".

In Unix, "**ulimit**" is a program; its pathname is "**/bin/ulimit**".

It sets or gets the system limitations in the current shell.

```
#define ONE_MEG (1024 * 1024)

char a[1024 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("1GB OK\n");
}
```

Have you ever tried that? Creating an 1GB array?!

```
[examples@3150] cat global_1gb.c
```

Data Segment & BSS – limits

```
$ gcc -Wall -O0    global_2gb.c    -o global_2gb
global_2gb.c:6: warning: integer overflow in expression
global_2gb.c:6: error: size of array ‘a’ is negative
$ _
```

The size of an array is a 32-bit **signed integer**, no matter 32-bit or 64-bit systems. Therefore…

```
#define ONE_MEG (1024 * 1024)

char a[2048 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf(“2GB OK\n");
}
```

How about 2GB?

```
[examples@3150] cat global_2gb.c
```

Data Segment & BSS – limits

```
#define ONE_MEG (1024 * 1024)

char a[1024 * ONE_MEG];
char b[1024 * ONE_MEG];
char c[1024 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("1GB OK\n");
    memset(b, 0, sizeof(b));
    printf("2GB OK\n");
    memset(c, 0, sizeof(c));
    printf("3GB OK\n");
}
```

Program: global_3gb.c

On 32-bit machine?
How about 64-bit?

```
#define ONE_MEG (1024 * 1024)

char a[1024 * ONE_MEG];
char b[1024 * ONE_MEG];
char c[1024 * ONE_MEG];
char d[1024 * ONE_MEG];

int main(void) {
    memset(a, 0, sizeof(a));
    printf("1GB OK\n");
    memset(b, 0, sizeof(b));
    printf("2GB OK\n");
    memset(c, 0, sizeof(c));
    printf("3GB OK\n");
    memset(d, 0, sizeof(d));
    printf("4GB OK\n");
}
```

Program: global_4gb.c

```
[examples@3150] cat global_3gb.c global_4gb.c
```

Data Segment & BSS – summary

- Remember, "**global variable == static variables**".
  - Only the **compiler cares** about the difference!

- Everything in a computer has a limit!
  - Different systems have different limits: 32-bit VS 64-bit.
  - Your job is to adapt to such limits.
  - On a **32-bit** Linux system, the user-space **addressing space is around 3GB**.
    - The kernel reserves 1GB addressing space.
    - It is just the addressing space, not really allocating 1GB.

User-space memory management

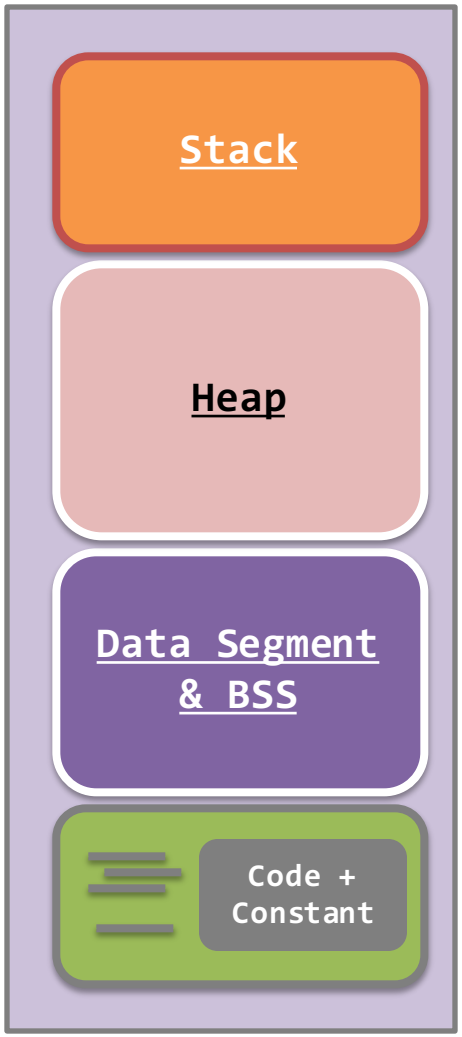
- Addressing;
- Code & constants;
- Data segment;
- **Stack;**

Stack – properties

- The stack contains:
  - all the local variables,
  - all function parameters,
  - program arguments, and
  - environment variables.

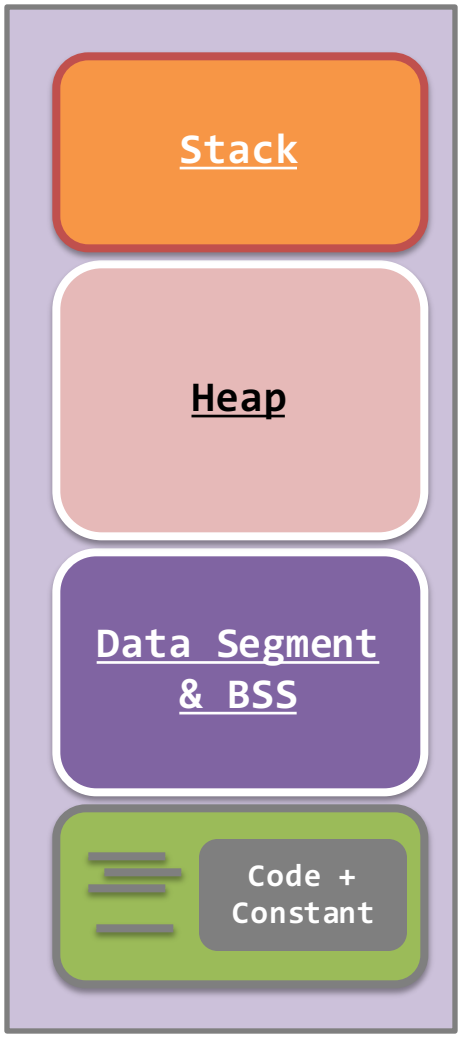
```
int main(int argc, char **argv, char **envp) {
    ......
}
```

Now, you are equipped enough knowledge to decode the above statement!



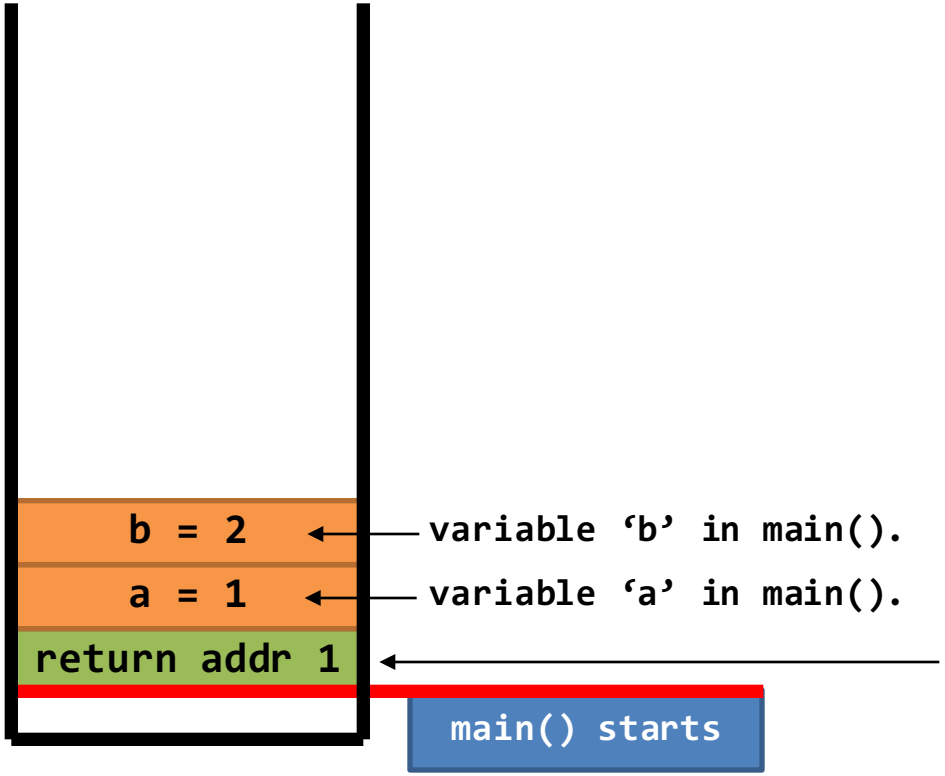
Stack – properties

- When a function is called, the local variables are allocated in the stack.
- When a function returns, the local variables are deallocated from the stack.
- Surprisingly, there is nothing to do with the kernel during function calls.
  - The compiler **hardcodes this mechanism**.
  - For details, look at the assembly codes.



Stack – push & pop mechanisms

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```



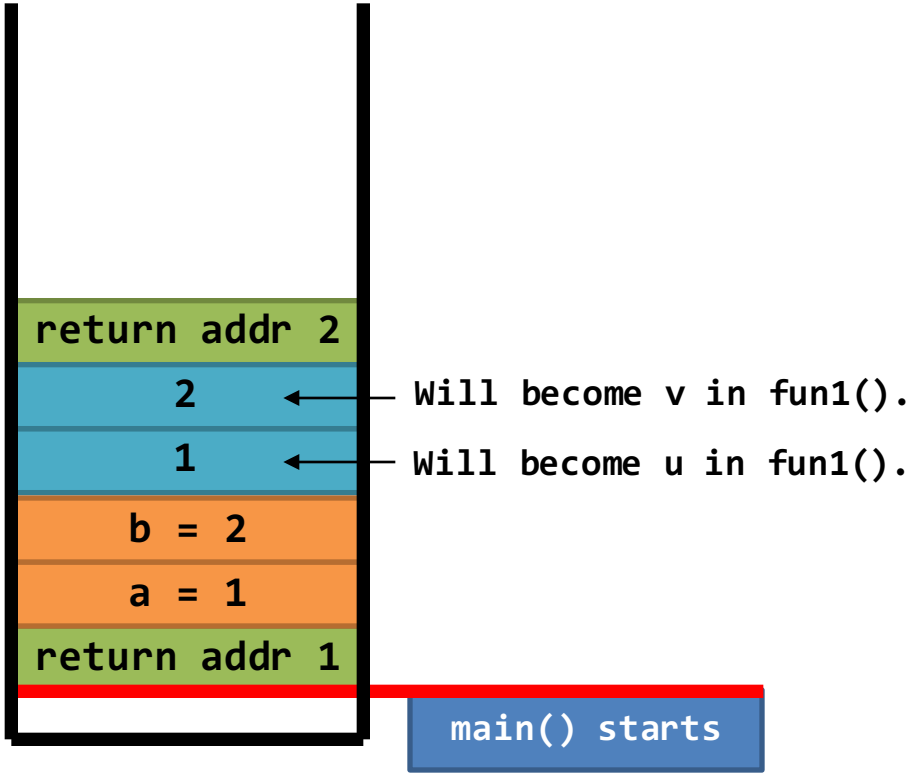
This address tells the program where to return (in the code segment) when fun1() finished running.

For main(), the return location is inside the C library.

[examples@3150] cat stack.c

Stack – push & pop mechanisms

Calling function "**fun1()**" starts.
It is the beginning of the call, and the CPU has not switched to **fun1()** yet.




```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

"return addr 2" is approx. here.

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

Digging deeper into the assembly codes!

For your eyes only!
The real codes are different!

```
push [value of a]
push [value of b]
push [return addr]
call  fun1
movl  eax  [addr of b]
```

return addr

The return address is **an address to the code segment**, pointing the start of an instruction following the "**call**" instruction.

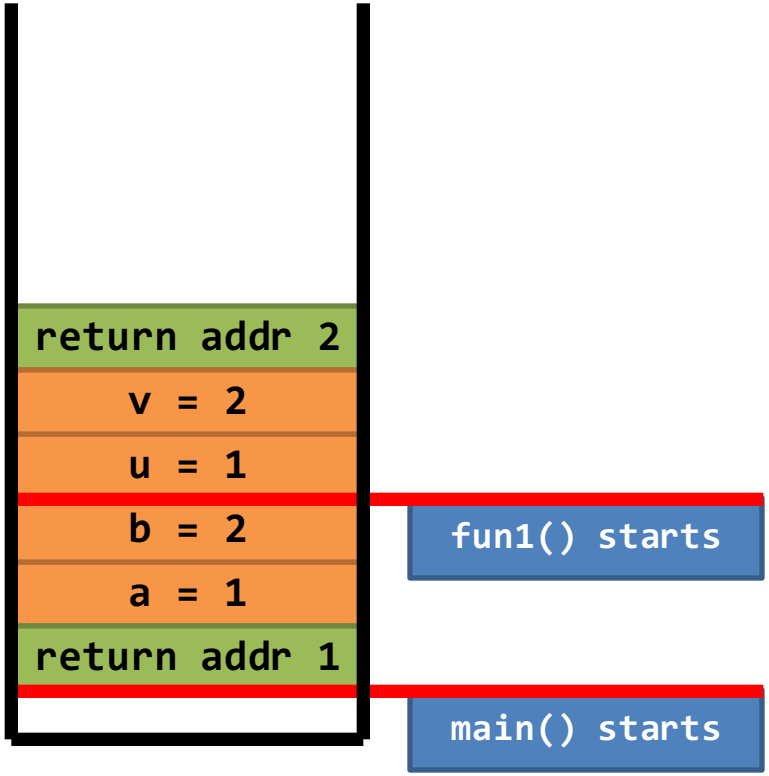
```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

"return addr 2" is approx. here.

[examples@3150] cat stack.c

Stack – push & pop mechanisms

Calling function "**fun1()**" takes place. The CPU has switched to **fun1()** .

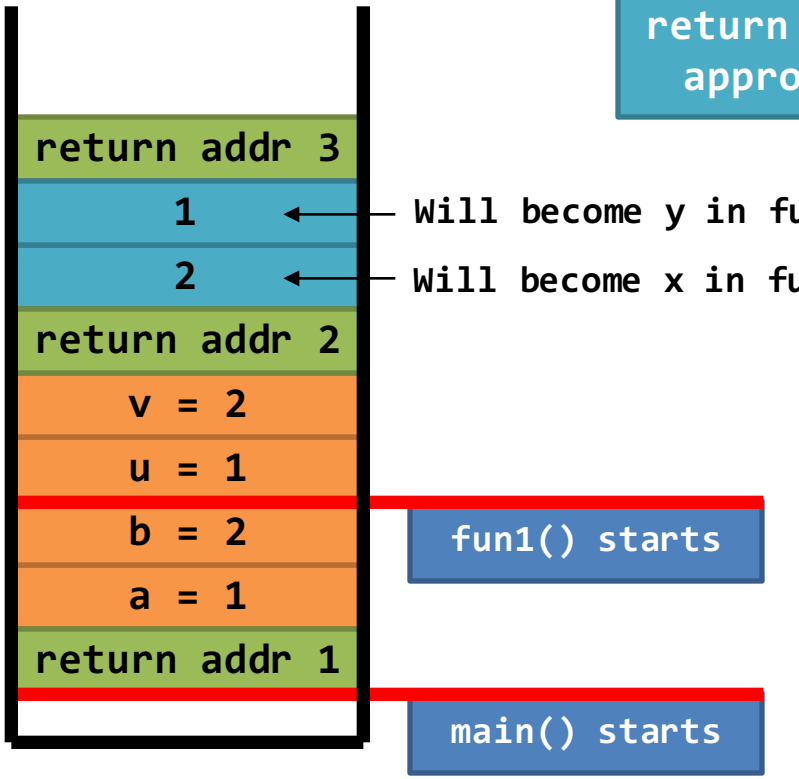



```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

Calling function "**fun2()**" starts.
It is the beginning of the call, and the CPU has not switched to **fun2()** yet.



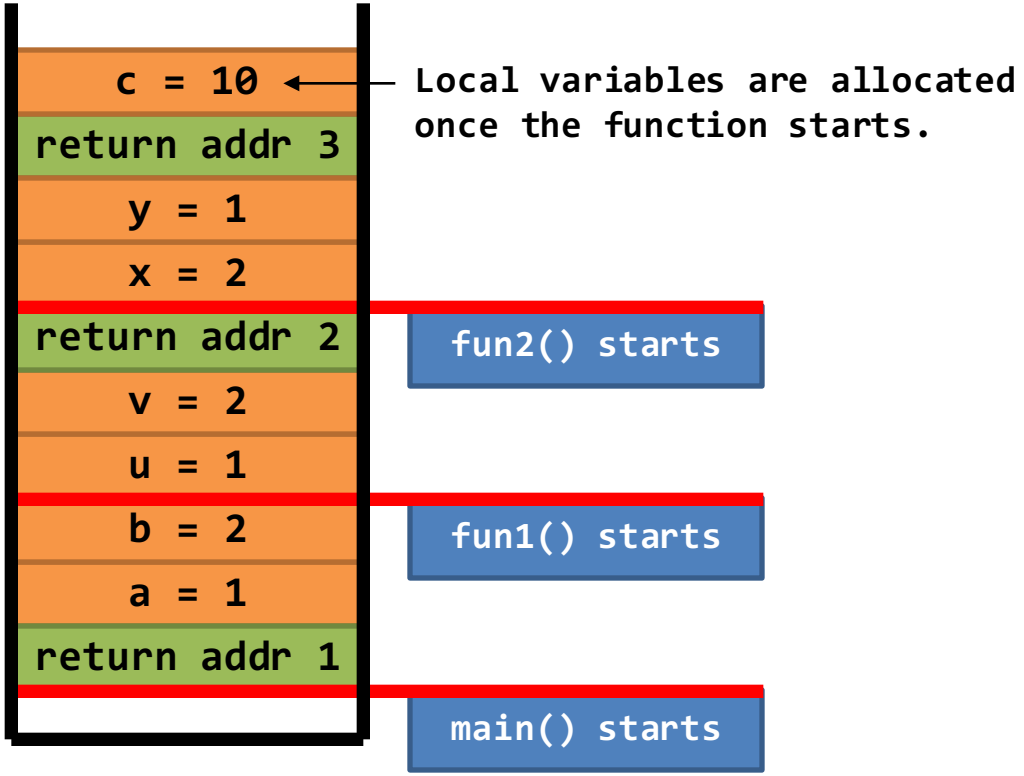
return addr 3 is approx. here.

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

Calling function "**fun2()**" takes place. The CPU has switched to **fun2()** .

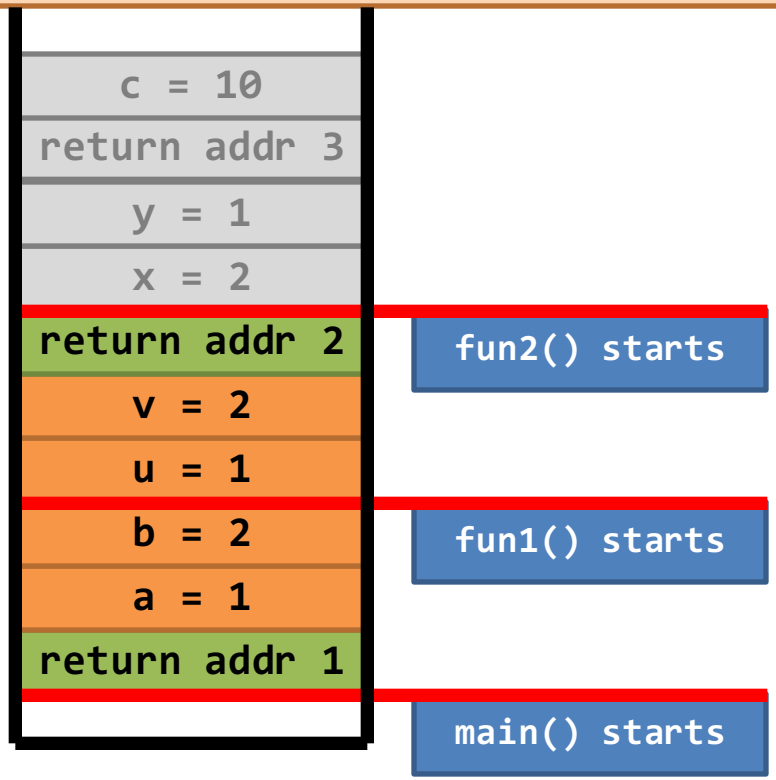



```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

"Return" takes place.
(1) Return value is written to the EAX register.
(2) Stack **shrinks**.
(3) CPU jumps back to **`fun1()`**.



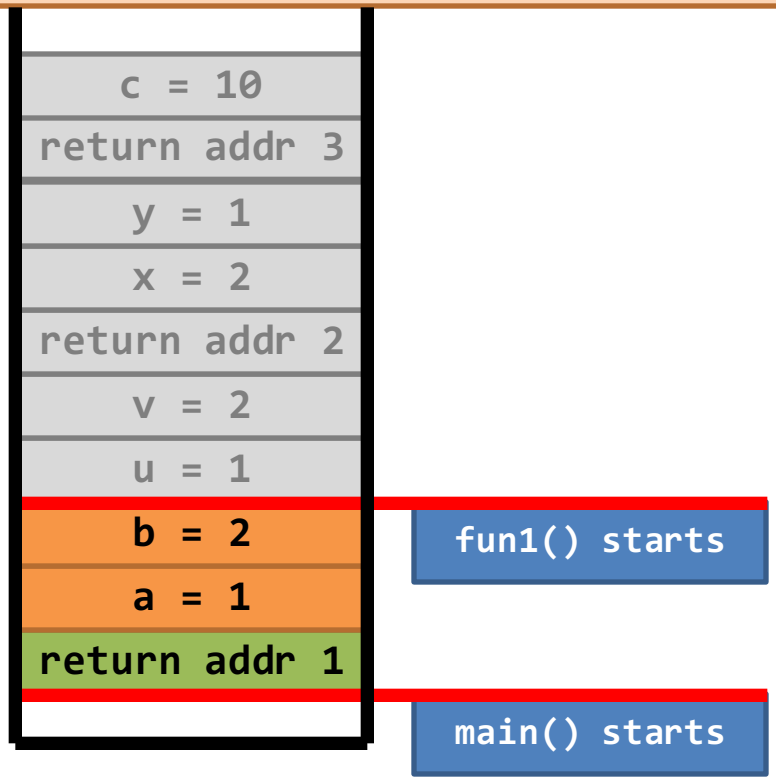

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

EAX: 13


```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms

"Return" takes place.
(1) Return value is written to the EAX register.
(2) Stack **shrinks**.
(3) CPU jumps back to **`main()`**.

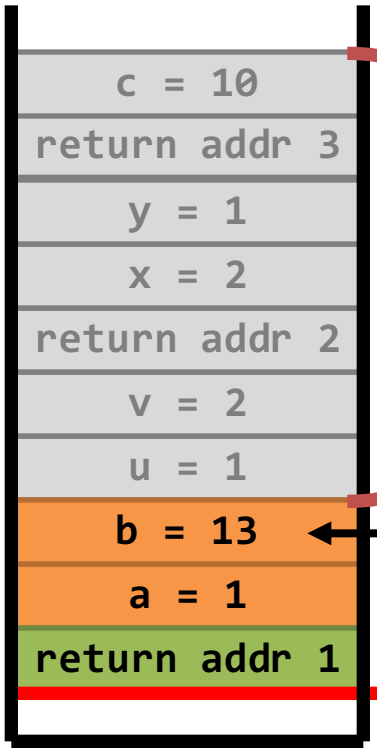



```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    ret 0;
}
```

EAX: 13


```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms



Very Important.

Those memory **is NOT returned to the OS**!!

Those memory will be re-used when you call functions again.

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

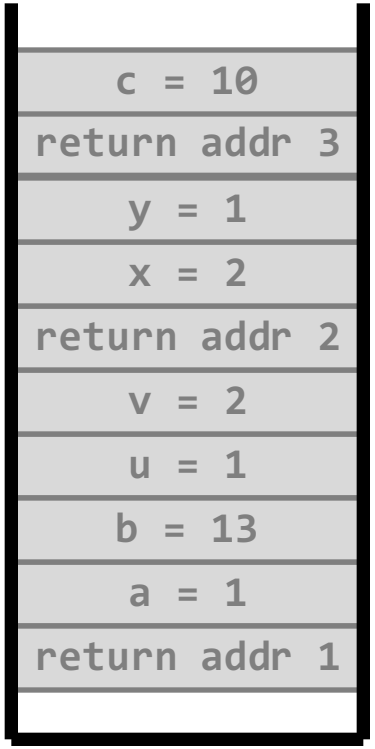
Upon "return", the value of EAX is then copied to "b"

EAX: 13

main() starts

```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms



Eventually, the main function reaches "**`return 0`**".

This takes the CPU pointing to the C library.

Inside the C library, we will eventually reach the system call **`exit()`**.

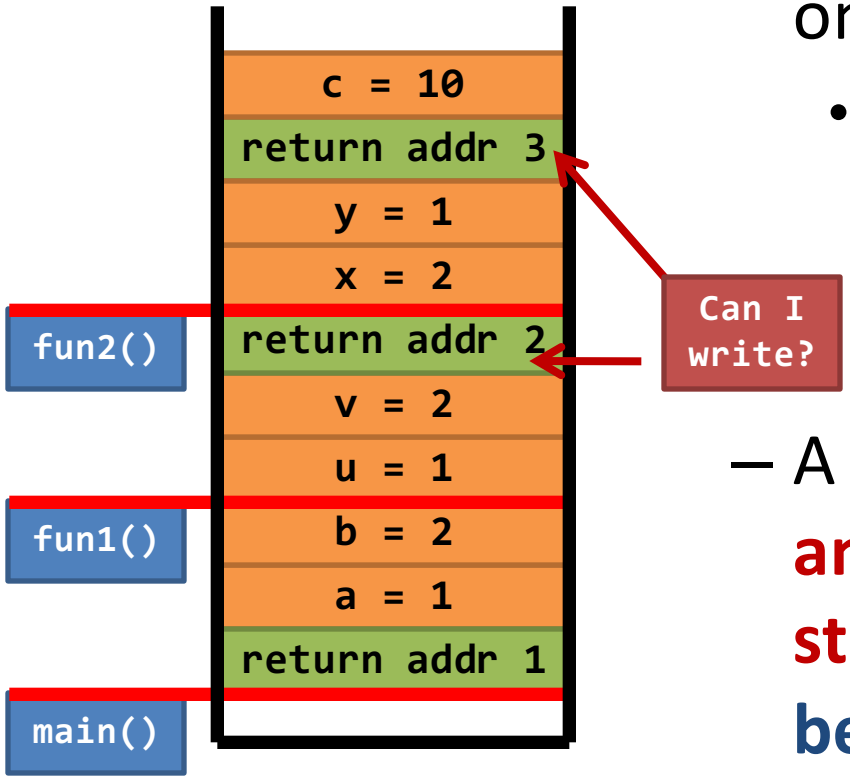
Note. Since the process is gone. The stack will not be gone forever.

```
int fun2(int x, int y) {
    int c = 10;
    return (x + y + c);
}

int fun1(int u, int v) {
    return fun2(v, u);
}

int main(void) {
    int a = 1, b = 2;
    b = fun1(a, b);
    return 0;
}
```

EAX: 0


```
[examples@3150] cat stack.c
```

Stack – push & pop mechanisms



- Important points to note:
  - Stack space is not really allocated on demand;
    - The CPU just moves along the stack space **as if the entire stack space is allocated**.
  - A function can ask the CPU **to read and to write anywhere in the stack**, **not just the "zone" belonging to the running function**!
    - Isn't it horrible (profitable and fun)?

Stack – limits

```
$ ulimit -a
  core file size   (blocks, -c) 0
  data seg size    (kbytes, -d) unlimited
  ......
  stack size       (kbytes, -s) 8192
  ......
$ _
```

So, the limit is:
8192 x 1024 = 8MB.

```
int main(void) {
    char a[8192 * 1024];
    memset(a, 0, sizeof(a));
    printf("OK!\n");
    return 0;
}
```

Can you see "OK"?

[examples@3150] cat max_stack.c

Stack – limits

```
$ ulimit -a
  core file size   (blocks, -c) 0
  data seg size    (kbytes, -d) unlimited
  ......
  stack size       (kbytes, -s) 8192
  ......
$ ulimit -s 81920
```

Now, the limit is:
81920 x 1024 = 80MB.

```
int main(void) {
    char a[8192 * 1024];
    memset(a, 0, sizeof(a));
    printf("OK!\n");
    return 0;
}
```

Can you see "OK"?

```
[examples@3150] cat max_stack.c
```

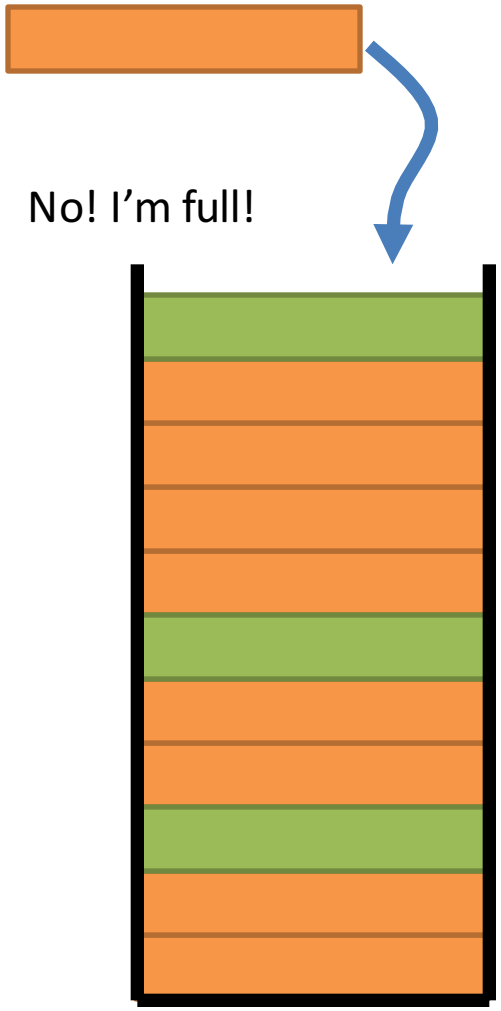
Stack – summary

- The total size of the local variables changes dynamically.

- The consumption of the stack depends on:
  - number of parameters;
  - number of local variables;
  - number of nested function calls;

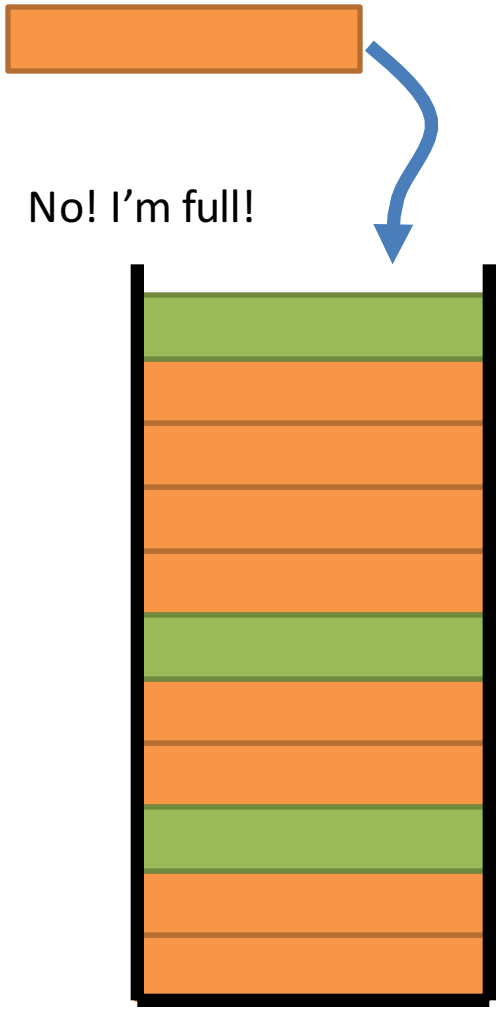
- But, the compiler cannot estimate the stack's size in compile time.
  - Because the number of function calls depends on the program state, the user inputs, etc.
  - The kernel can only reserve a large enough space for the stack.

Stack – summary



- What if there is no more space left in the stack?
  - Enlarge the stack?
  - What if it is a chain of endless recursive function calls?
- What will happen?
  - **Exception caught by the CPU!**
  - **Stack overflow exception!**
  - **Program terminated!**

Stack – summary



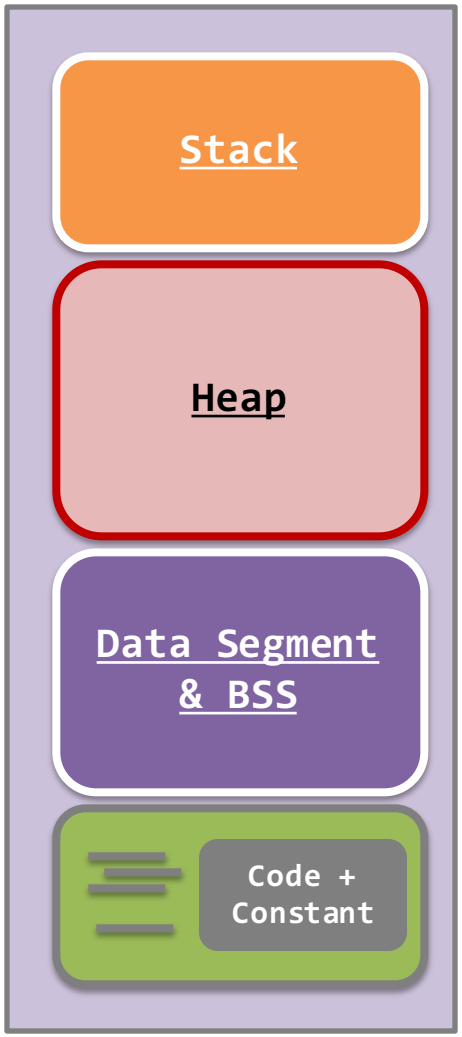
- *"I really need to play with recursions."* Any workaround?
  - Minimizing the number of arguments.
  - Minimizing the number of local variables.
  - Minimizing the number of calls (if you can).
  - Use global variables!
  - **More information: tail recursion**

User-space memory management

- Addressing;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**

Dynamically allocated memory – properties

- Its name tells you its nature:
  - **Dynamic**: not defined at compile time.
  - **Allocation**: only when you *ask for memory, you would be allocated the memory.*
  - The dynamically allocated memory is called the **heap**.
    - Don't mix it up with the binary heap;
    - it has nothing to do with the binary heap.



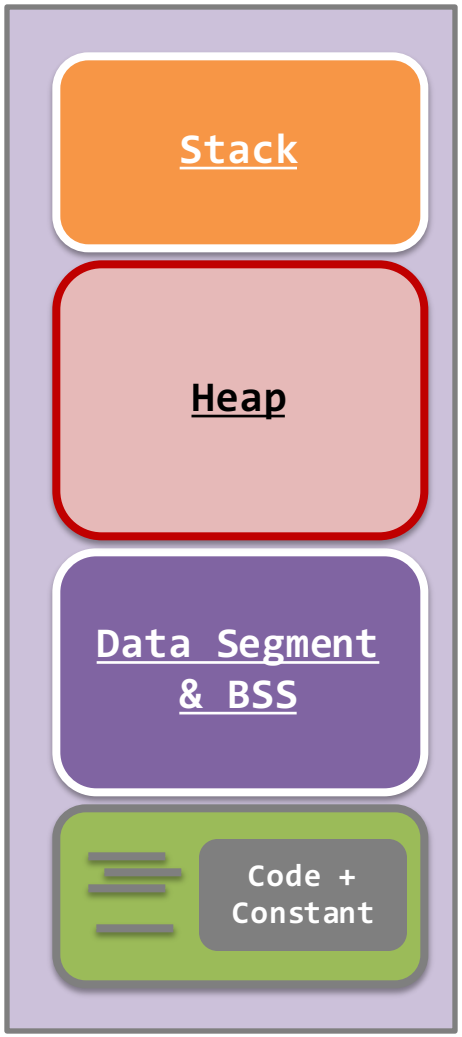
Dynamically allocated memory – properties

- Lecturers of the intro. programming course would tell you the following:

  – *"**malloc()**" is a function that allocates memory for you.*

  – *"**free()**" is a function that gives up a piece of memory that is produced by previous "**malloc()**" call.*

- The lecturer of the OS course is **to define and to defy** what you know about the **`malloc()`** and the **`free()`** library functions.



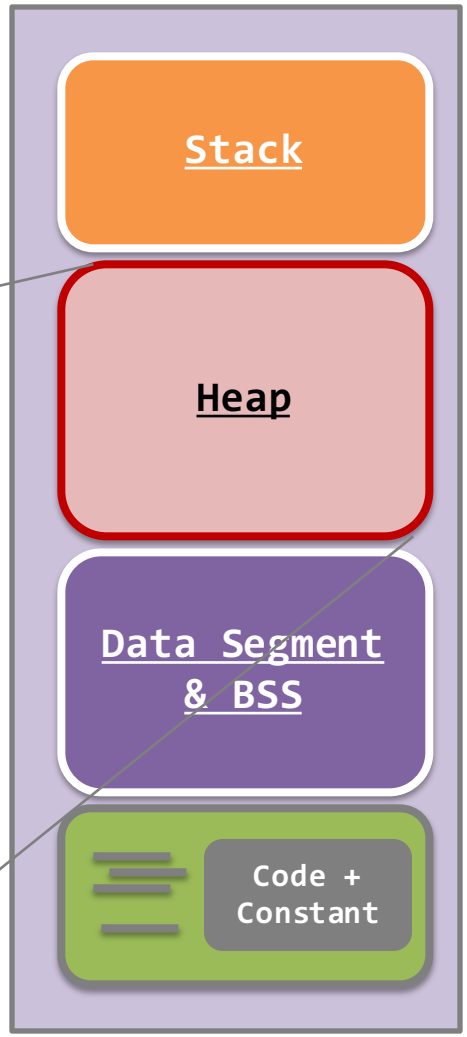
User-space memory management

- Addressing;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**
  - **malloc(), free();**

De-mystifying `malloc()`

When a program just starts running, the entire heap space is unallocated, or empty.

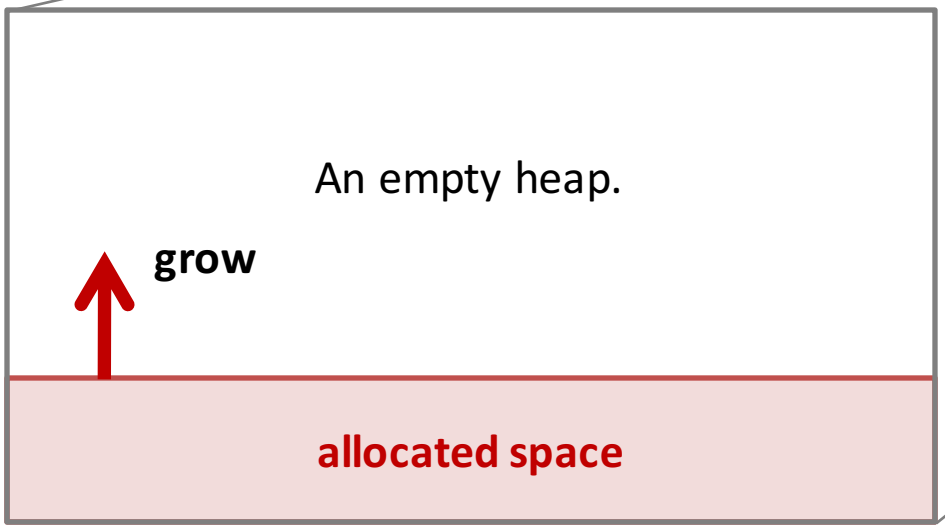
An empty heap.

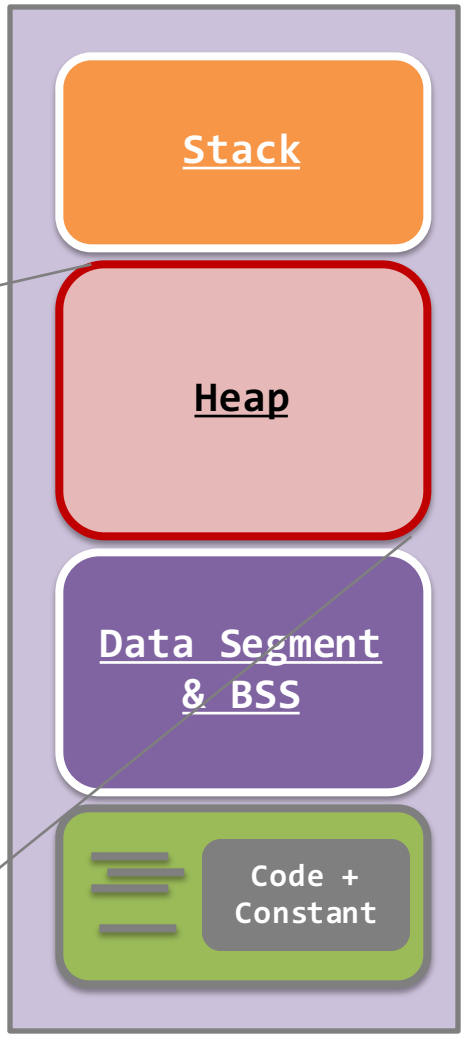


De-mystifying **`malloc()`**

When "**`malloc()`**" is called, the "**`brk()`**" system call is invoked accordingly.

"**`brk()`**" allocates the space required by "**`malloc()`**". But, it doesn't care how "**`malloc()`**" uses the space.



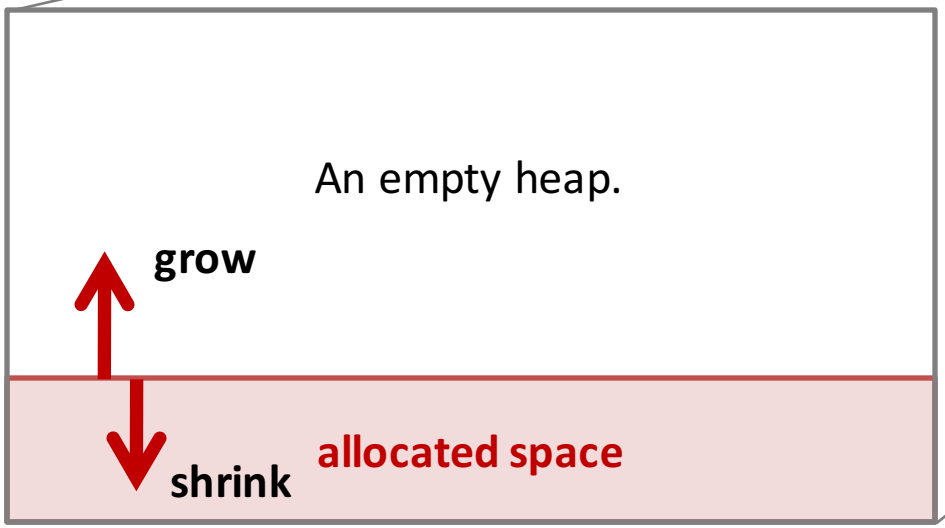


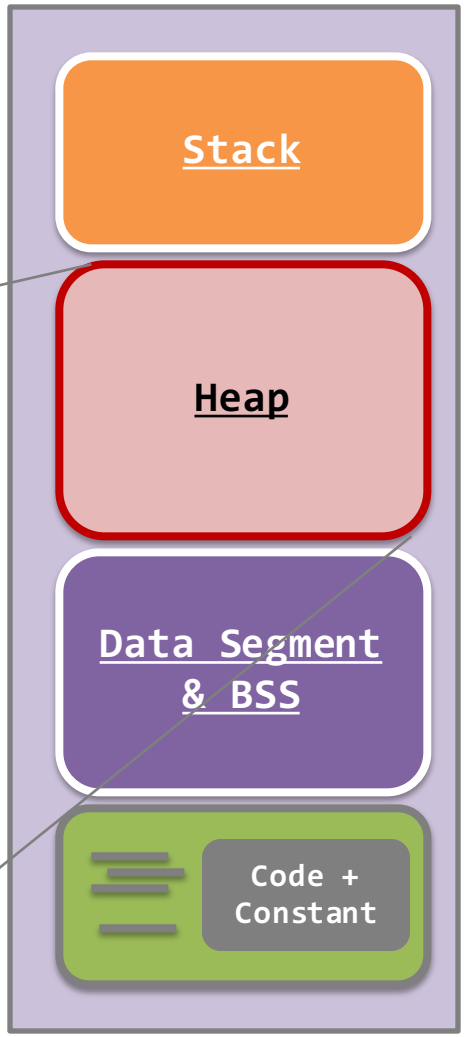
De-mystifying **malloc()**

The allocated space growing or shrinking depends on the further actions of the process. That means the "**brk()**" system call can grow or shrink the allocated area.

In **malloc()**, the library call just invoke **brk()** for growing the heap space.

The **free()** call may shrink the heap space.





De-mystifying **malloc()**

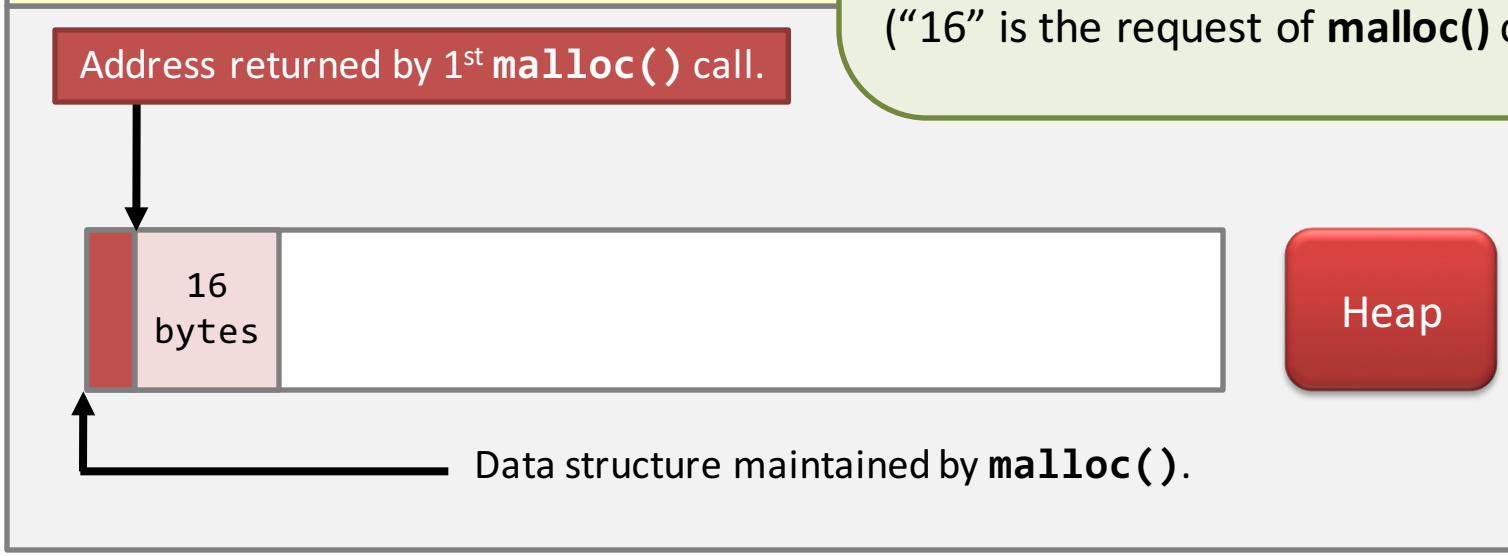
```
int main(void) {
  char *ptr1, *ptr2;
  ptr1 = malloc(16);
  ptr2 = malloc(16);

  printf("Distance between ptr1 and ptr2: %d bytes\n",
         ptr2 – ptr1);
  return 0;
}
```

De-mystifying **malloc()**

```
int main(void) {
  char *ptr1, *ptr2;
  ptr1 = malloc(16);
  ptr2 = malloc(16);

  printf("Distance between ptr1 an
         ptr2 - ptr1);
  return 0;
}
```

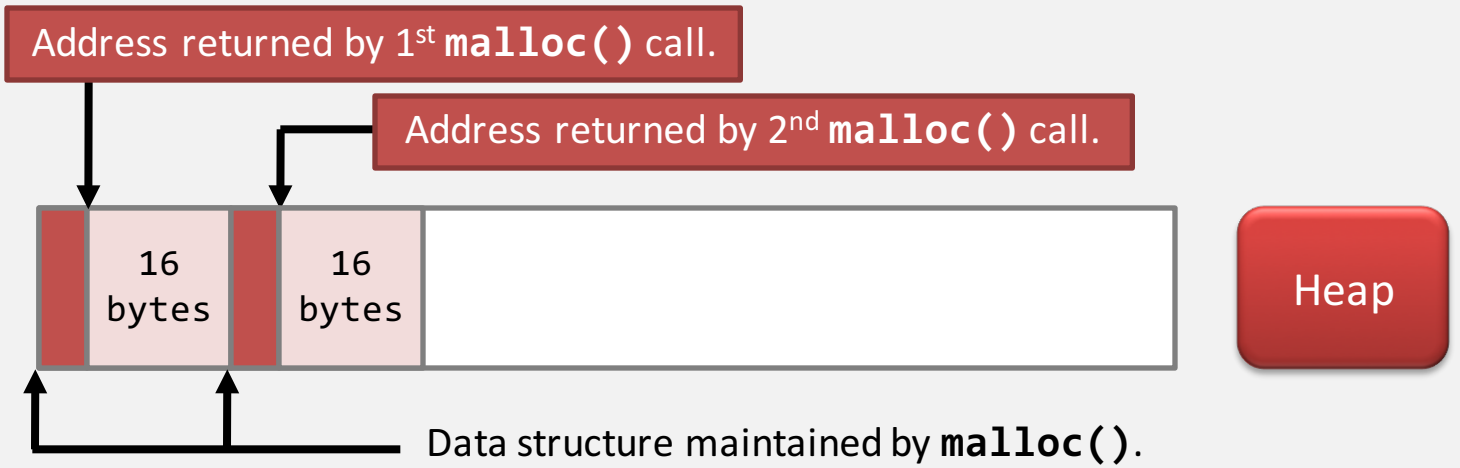
The return value of **malloc()** is of type "**void ***", which means it is just a memory address only, and can be of any data types.

Such a memory address is the starting address of a piece of memory of 16 bytes ("16" is the request of **malloc()** call).



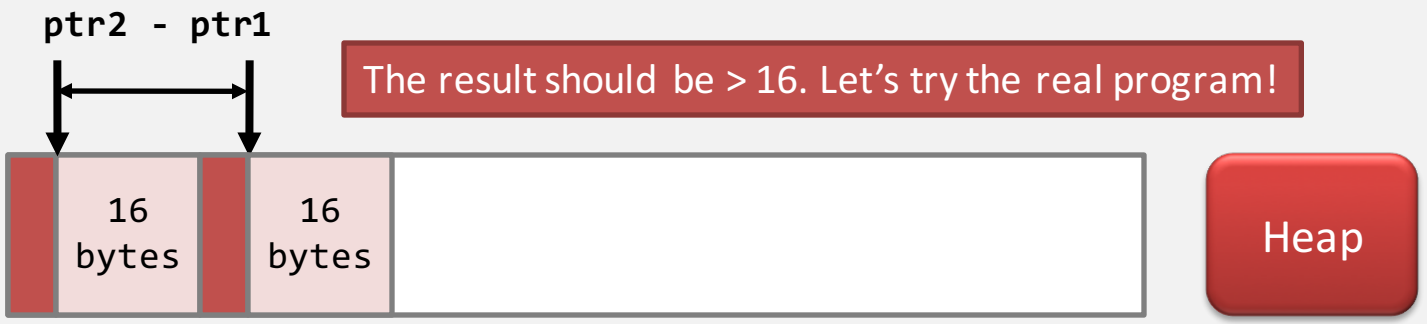
De-mystifying `malloc()`

```
int main(void) {
  char *ptr1, *ptr2;
  ptr1 = malloc(16);
  ptr2 = malloc(16);

  printf("Distance between ptr1 and ptr2: %d bytes\n",
         ptr2 – ptr1);
  return 0;
}
```



De-mystifying `malloc()`

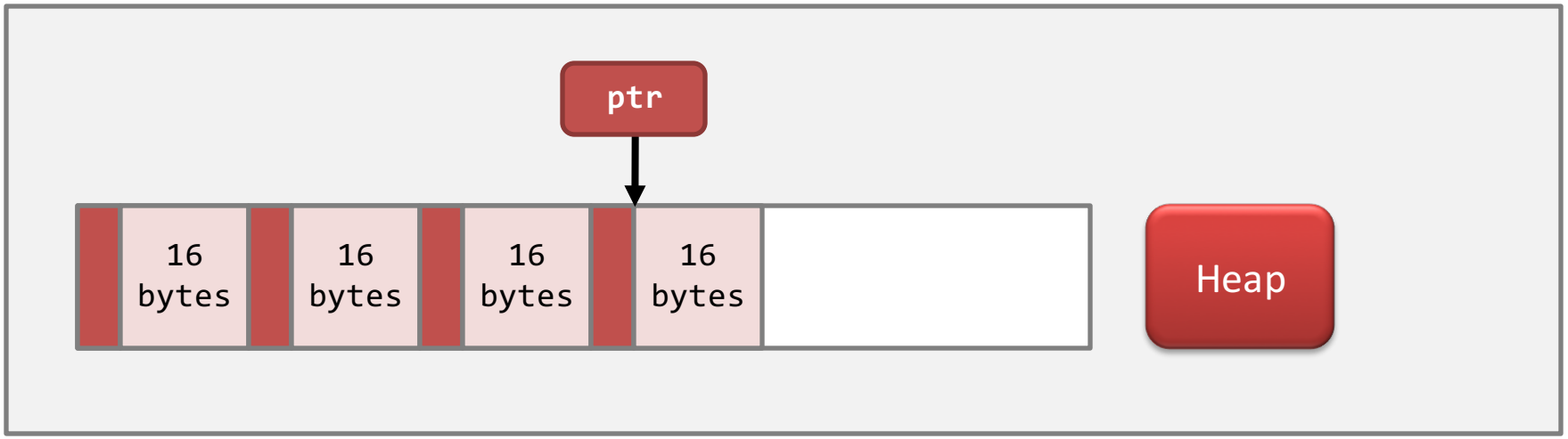
```
int main(void) {
  char *ptr1, *ptr2;
  ptr1 = malloc(16);
  ptr2 = malloc(16);

  printf("Distance between ptr1 and ptr2: %d bytes\n",
         ptr2 – ptr1);
  return 0;
}
```




```
[examples@3150] cat malloc_distance.c # 32-bit & 64-bit machines yield different results
```

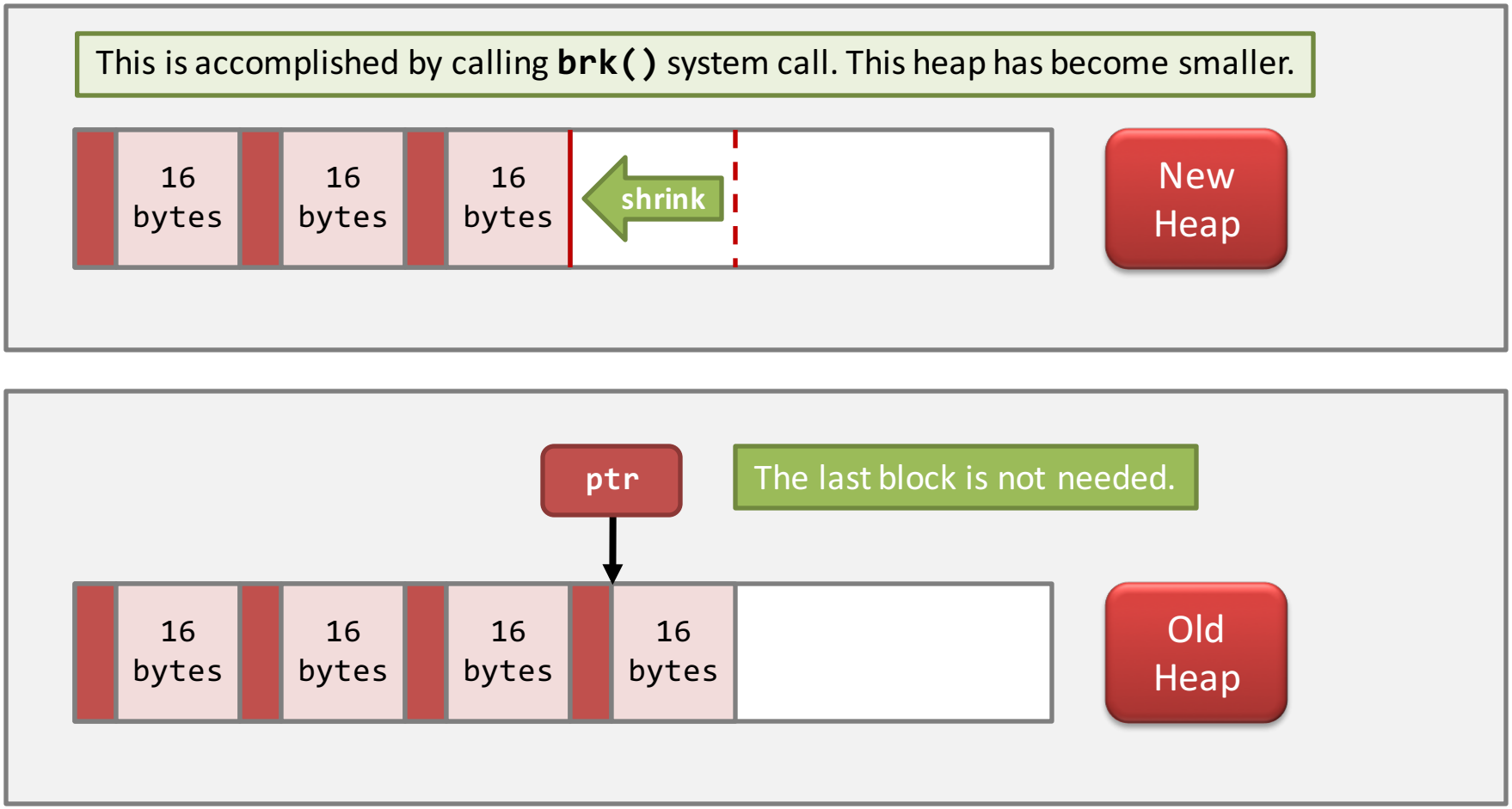
De-mystifying **`free()`**

- "**`free()`**" *seems to* be the opposite to "**`malloc()`**":
  - It de-allocates any allocated memory.
  - When a program calls "**`free(ptr)`**", then the address "**`ptr`**" must be the start of a piece of memory obtained by a previous "**malloc()**" call.



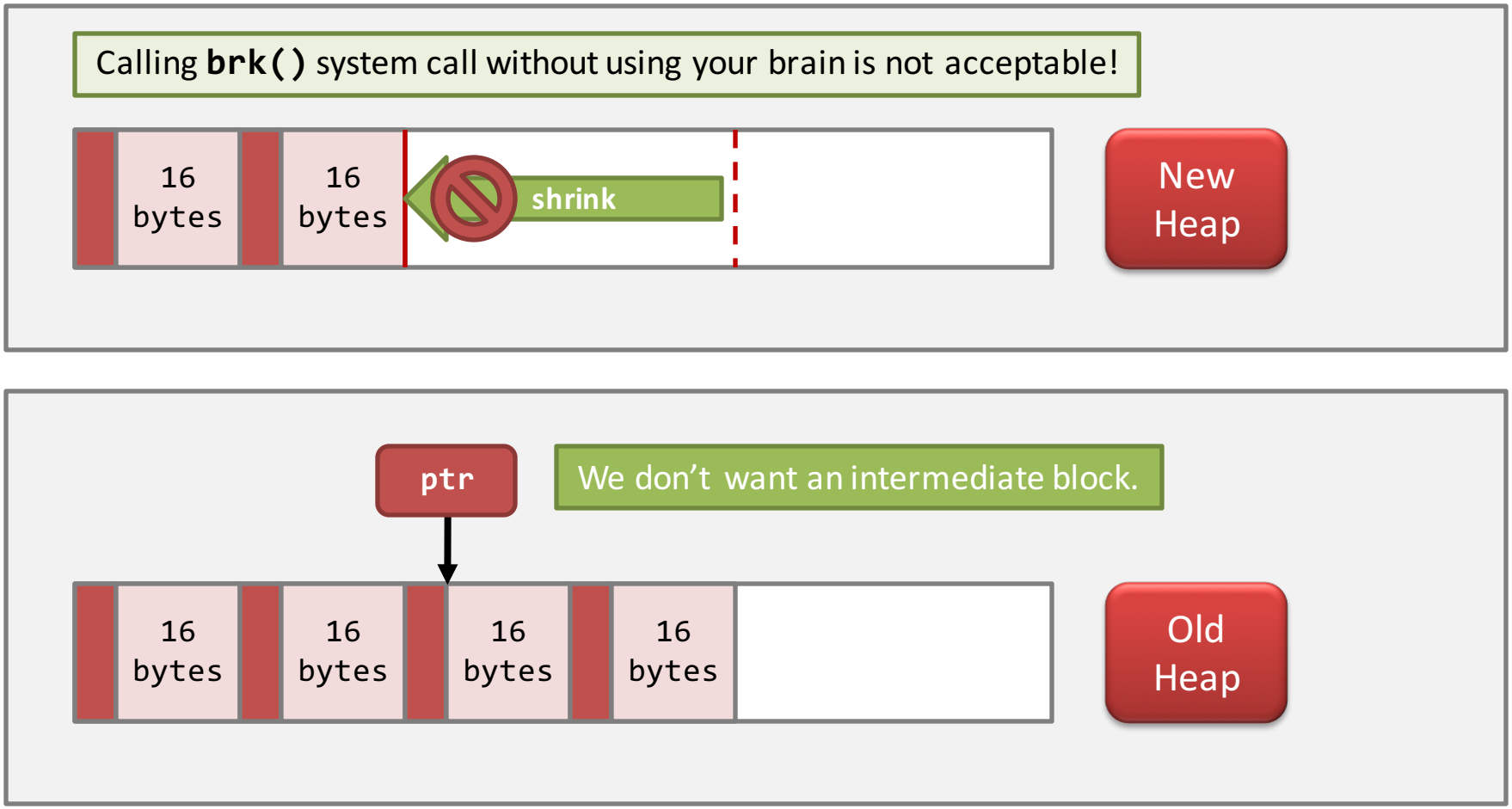
De-mystifying `free()` – case #1

- Case #1: de-allocating the last block.



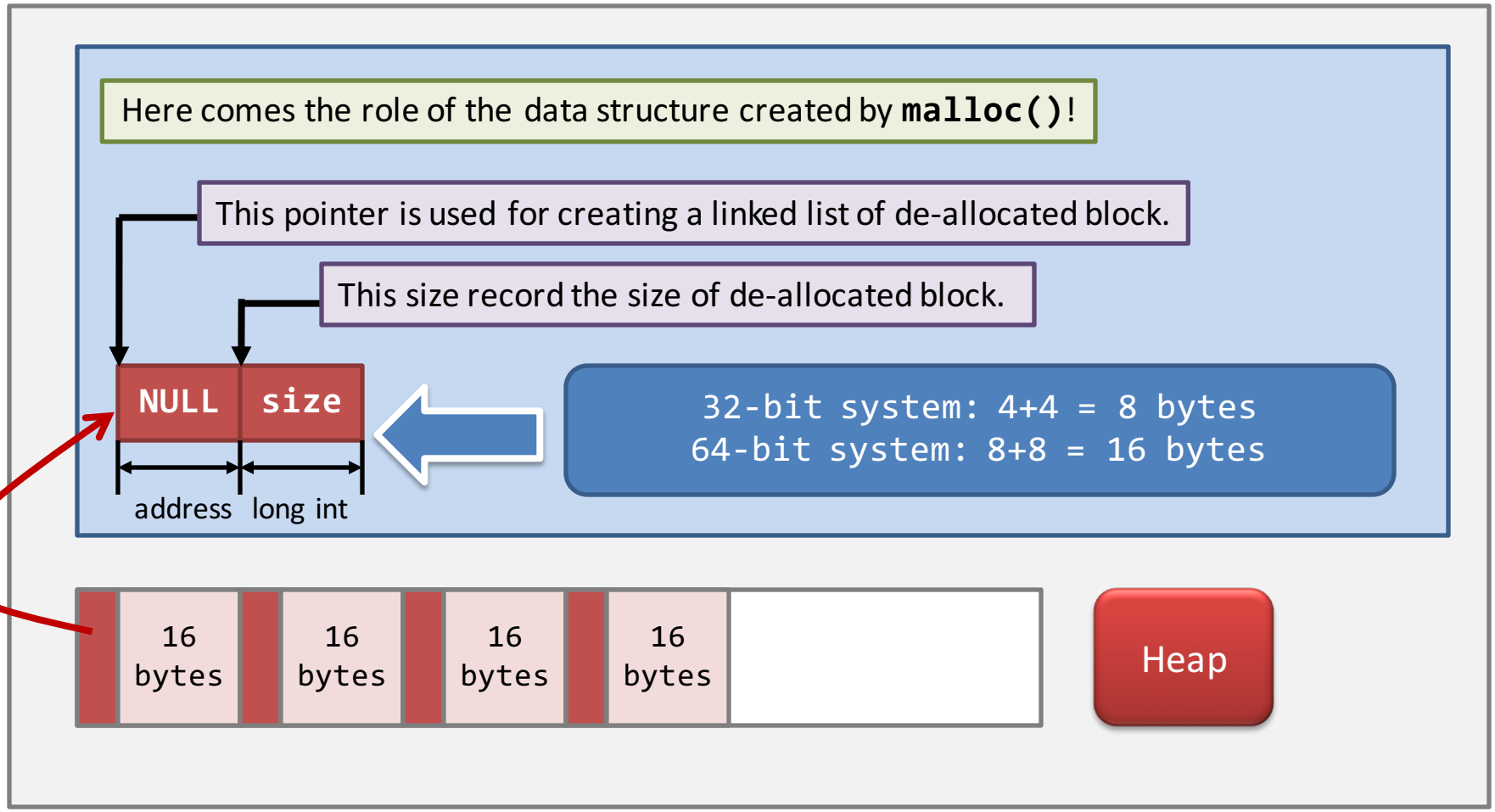
De-mystifying `free()` – case #2

- Case #2: de-allocating an intermediate block.



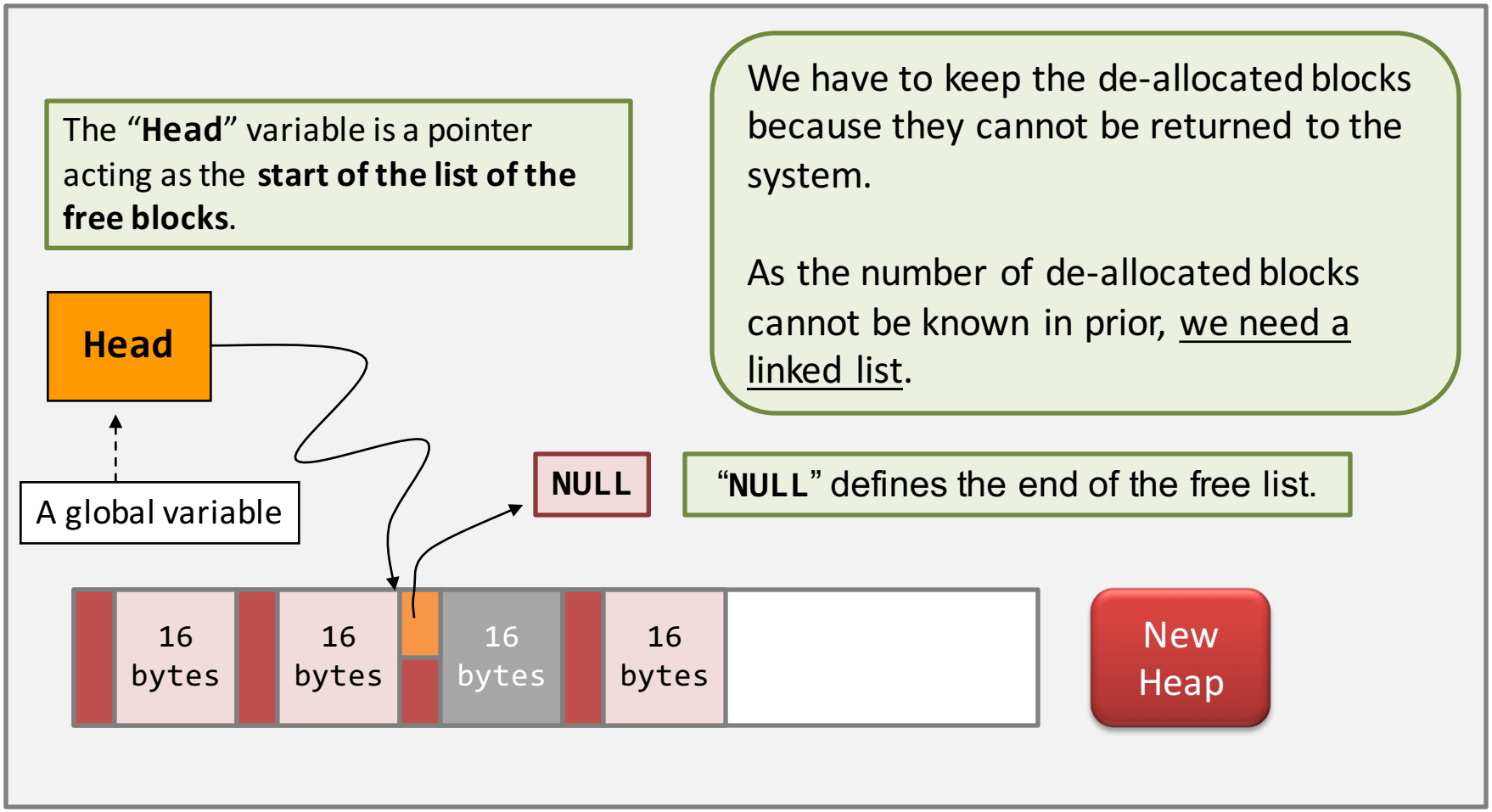
De-mystifying **free()** – case #2

- Case #2: de-allocating an intermediate block.



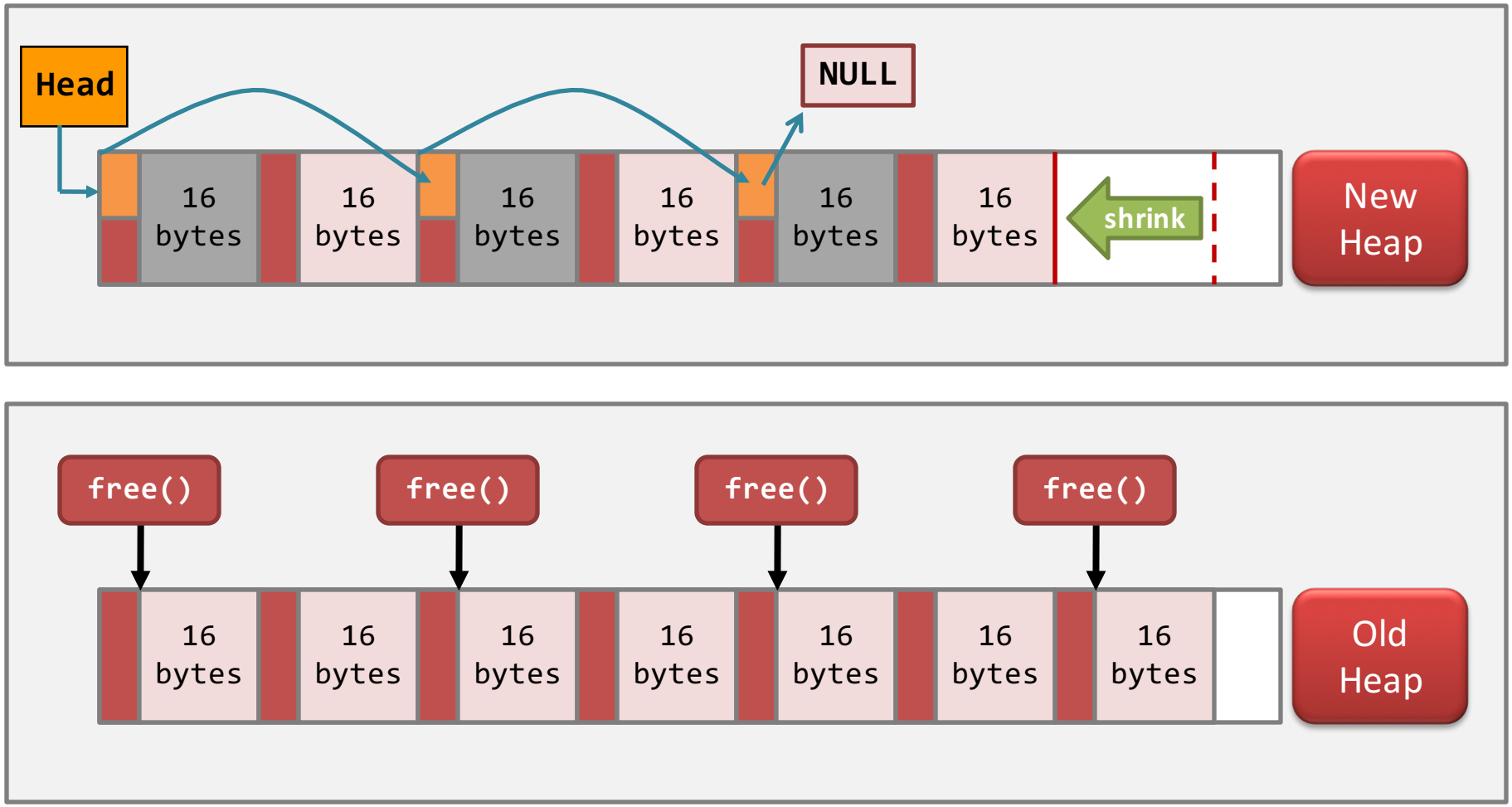
De-mystifying `free()` – case #2

- Case #2: de-allocating an intermediate block.



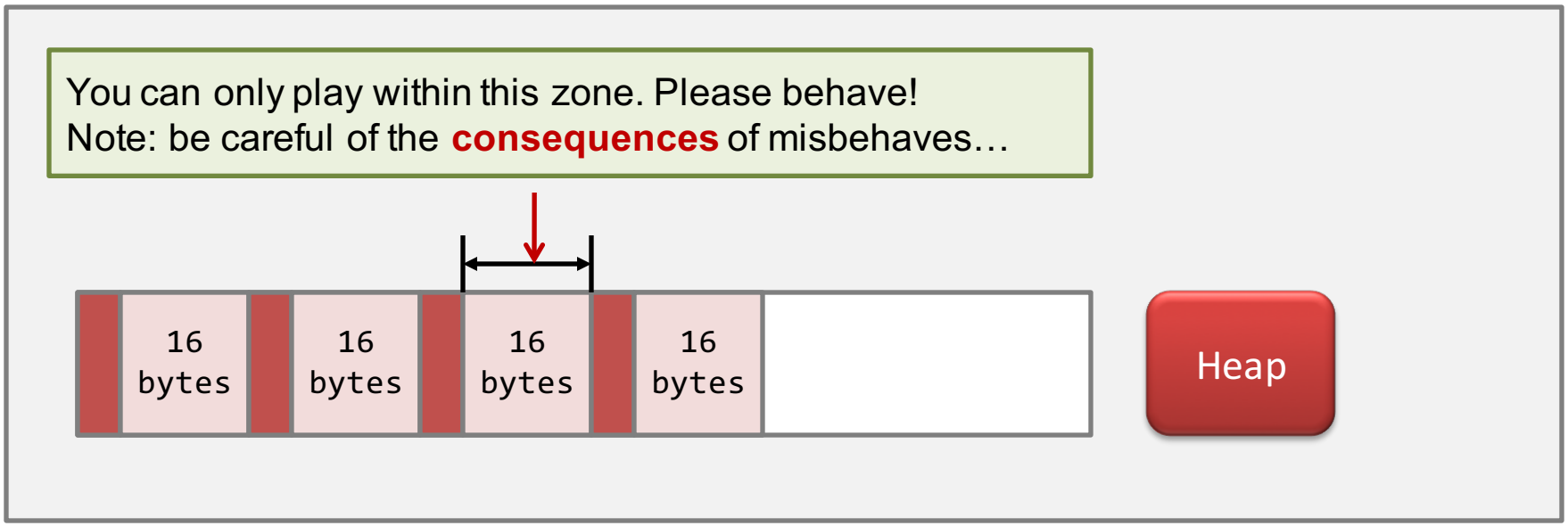
De-mystifying `free()` – case #2

- Case #2: another example.



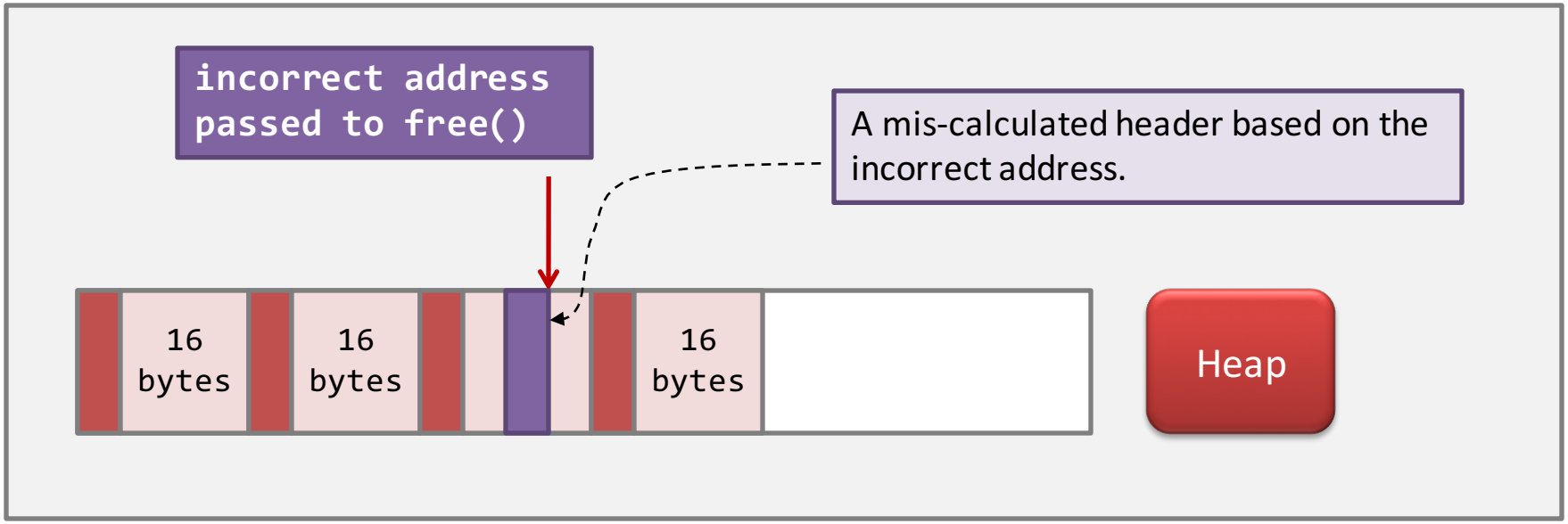
De-mystifying `free()` – cautions

- The calling program is **assumed** to be carefully written.
  - After `malloc()` has been invoked, the program should read and write inside the requested area only.
  - Now, you know why you'd **have troubles** when you write data outside the allocated space.



De-mystifying `free()` – cautions

- The calling program is **assumed** to be carefully written.
  - When **`free()`** is called, the program should provide **`free()`** with the correct address…
    - i.e., the address previously returned by a **`malloc()`** call.
    - What if the address is wrong?

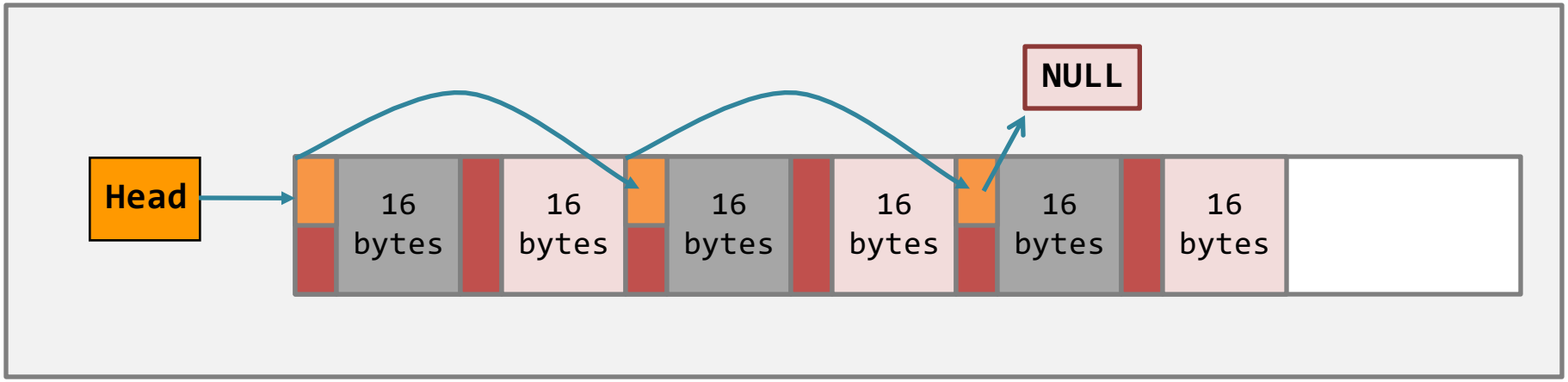


User-space memory management

- Addressing;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**
  - malloc(),free();
  - **malloc() + free();**

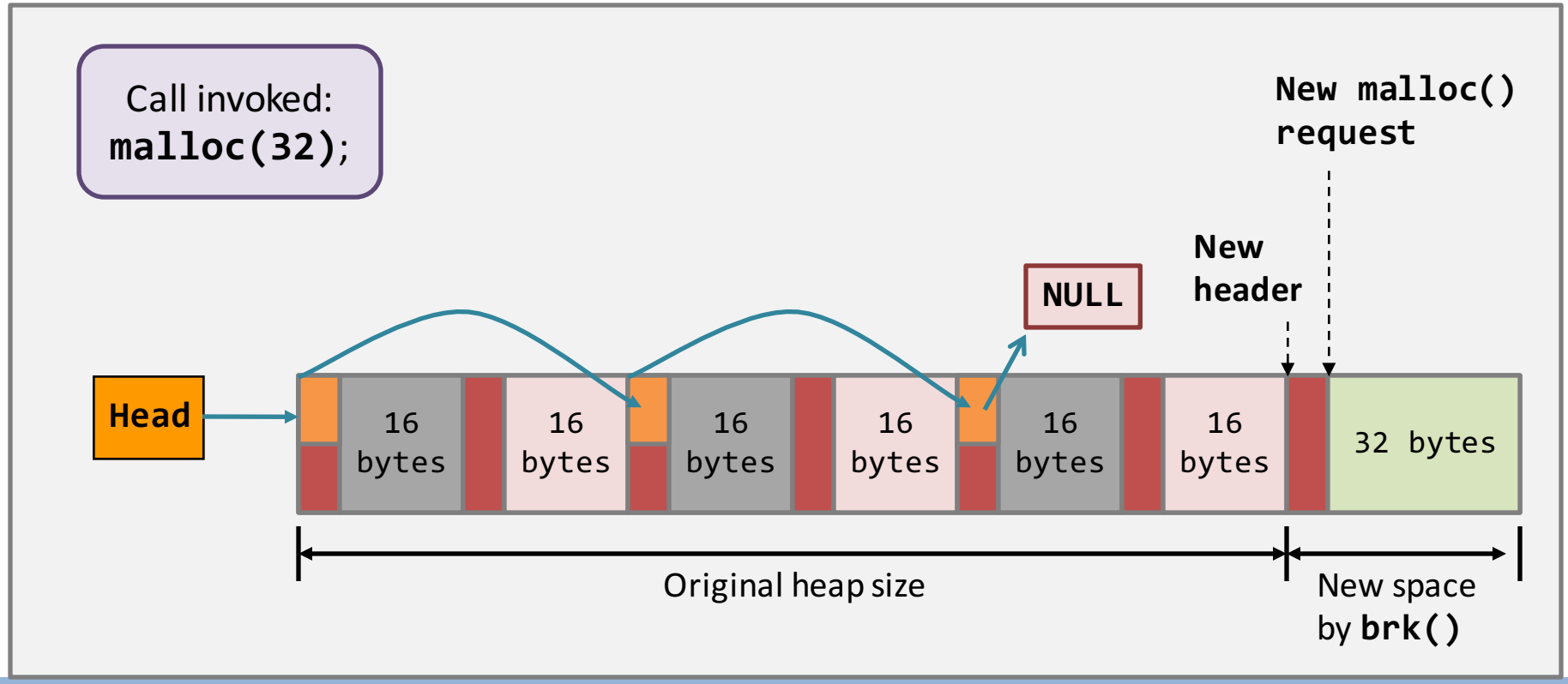
When `malloc()` meets free blocks…

- Problem: whether to use the free blocks or not?
  - *Is there any free block that is **large enough** to satisfy the need of the **`malloc()`** call?*



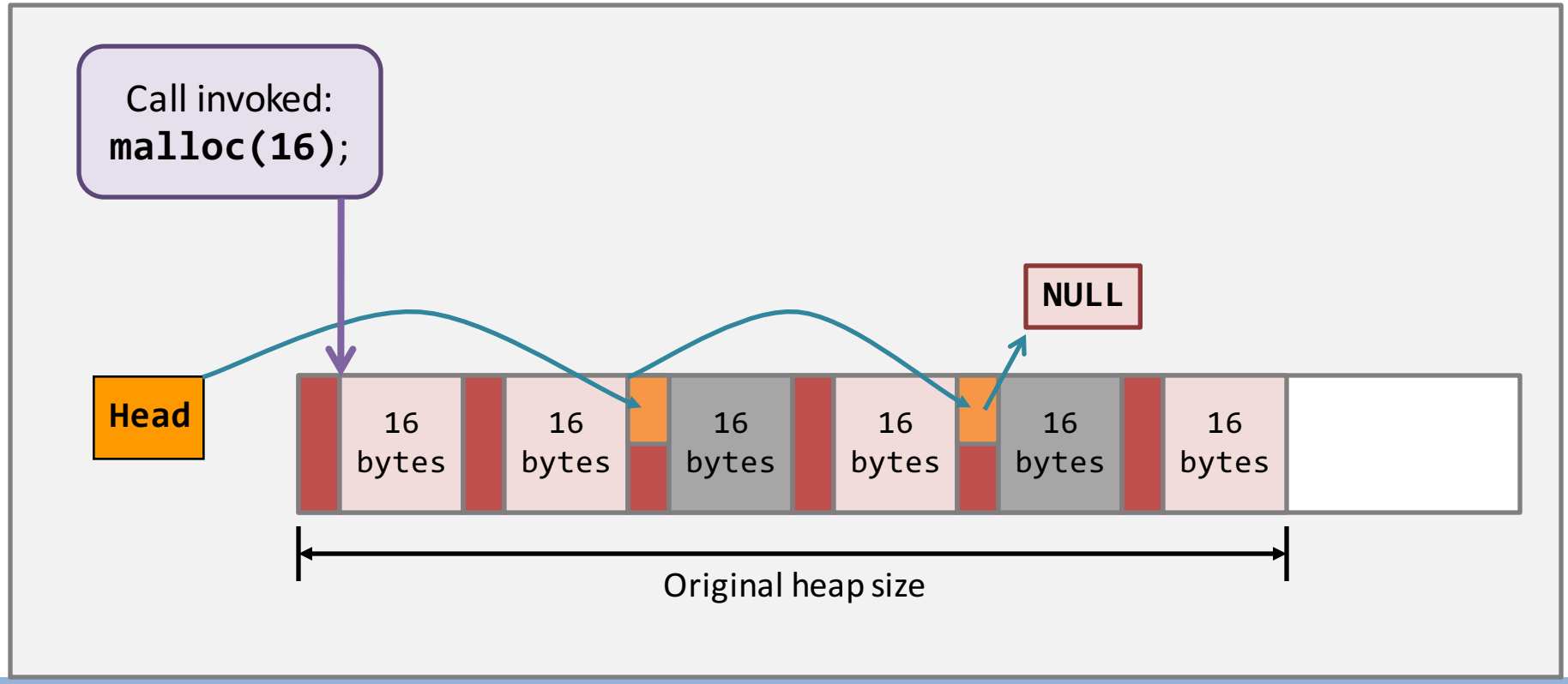
When `malloc()` meets free blocks…case 1

- Case #1: if there is **no suitable free block**…
  - then, the **`malloc()`** function should call **`brk()`** system call…in order to claim more heap space.



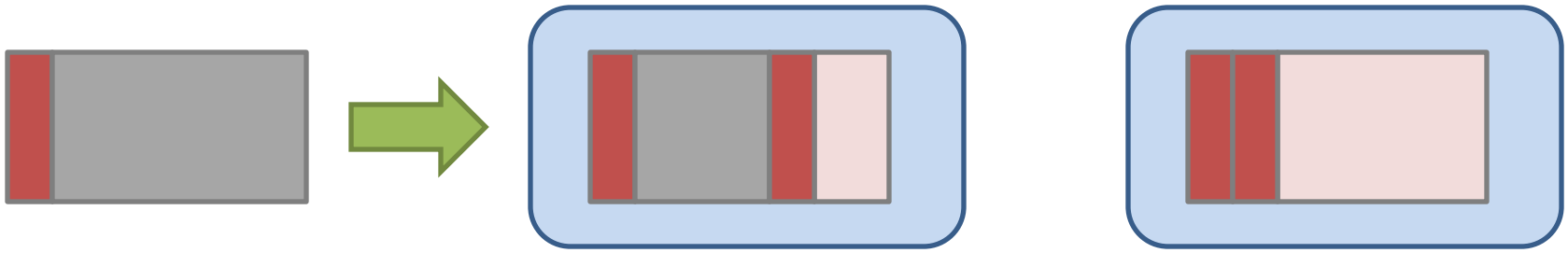
When `malloc()` meets free blocks…case 2

- Case #2: if there is a suitable free block
  - then, the **`malloc()`** function should reuse that free block.



When `malloc()` meets free blocks…

- There can be other cases:
  - A **`malloc()`** request that takes a partial block;
  - A **`malloc()`** request that takes a partial block, but leaving no space in the previously free block.



- We will skip those subtle cases…
  - It boils to implementation only…
  - You already have the **big picture** about **`malloc()`** and **`free()`**.

User-space memory management

- Addressing;
- Code & constants;
- Data segment;
- Stack;
- **Heap;**
  - malloc(),free();
  - malloc() + free();
  - **Out of memory?!**

Out of memory?

- The kernel knows how much memory should be given to the heap.
  - When you call **`brk()`**, the kernel **tries** to find the memory for you.

- Then…one natural question…
  - Is it possible to cause the system to **run out of memory (OOM)**?

Out of memory?

- Try this! An OOM Generator!

```
#define ONE_MEG  1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Are you "*man*" enough to execute this program on a 32-bit machine?

What is the output?

```
[examples@3150] cat oom_v1.c
```

Out of memory?

- On 32-bit Linux, why does the OOM generator always stop at around 3055MB?

- Still remember what we said when we are talking about data segment?
  - On a 32-bit Linux system, the user-space **addressing space is around 3GB**.
    - The kernel reserves 1GB addressing space.
    - It is just the addressing space, not really allocating 1GB.

Out of memory?

- Try this! Yet another OOM Generator!

```
#define ONE_MEG  1024 * 1024
char global[1024 * ONE_MEG];
int main(void) {
    void *ptr;
    int counter = 0;
    char local[8000 * 1024];
    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

I guess you are now man enough to run this program on a 32-bit machine

Yet, what is the output?

```
[examples@3150] cat oom_v2.c
```

Real OOM!

Explanation is in Part 2.

```
#define ONE_MEG  1024 * 1024

int main(void) {
    void *ptr;
    int counter = 0;

    while(1) {
        ptr = malloc(ONE_MEG);
        if(!ptr)
            break;
        memset(ptr, 0, ONE_MEG);
        counter++;
        printf("Allocated %d MB\n", counter);
    }

    return 0;
}
```

Warning #1. Don't run this program on any department's machines.

Warning #2. Don't run this program when you have important tasks running at the same time.

User-space memory management

- Addressing;
- Code & constants;
- Data segment;
- Stack;
- Heap;
- **Segmentation fault;**

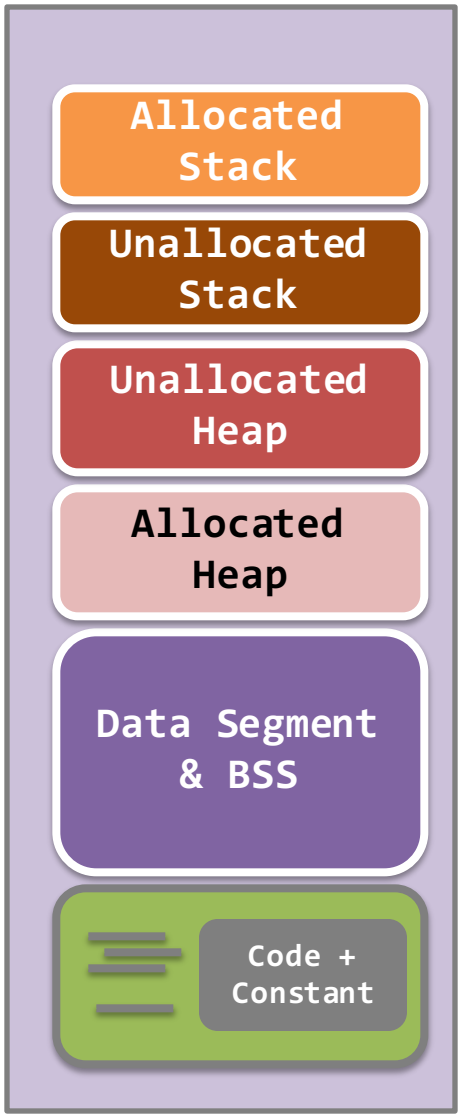
What is segmentation fault?

- Someone must have told you:

  – When you are accessing a piece of memory that is not allowed to be accessed, then the OS returns you an error called – segmentation fault.

- As a matter of fact, **how many ways** are there to generate a segmentation fault?

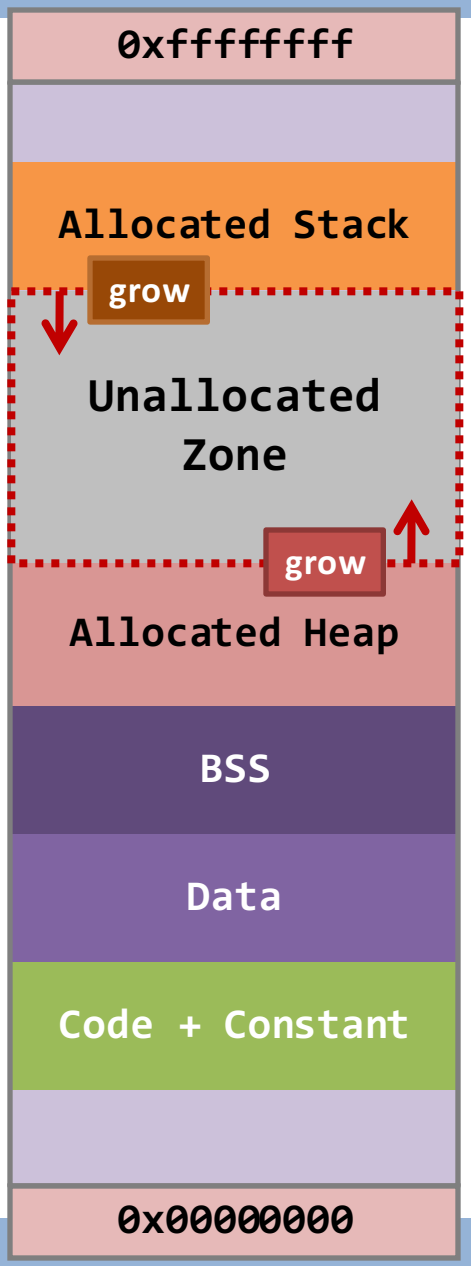
What is segmentation fault?



From illustration to reality…

Forget about the illustration, the memory in a process is separated into **segments**.

So, when you visit a segment in an illegal way, then…**segmentation fault**.



How to “segmentation fault”?

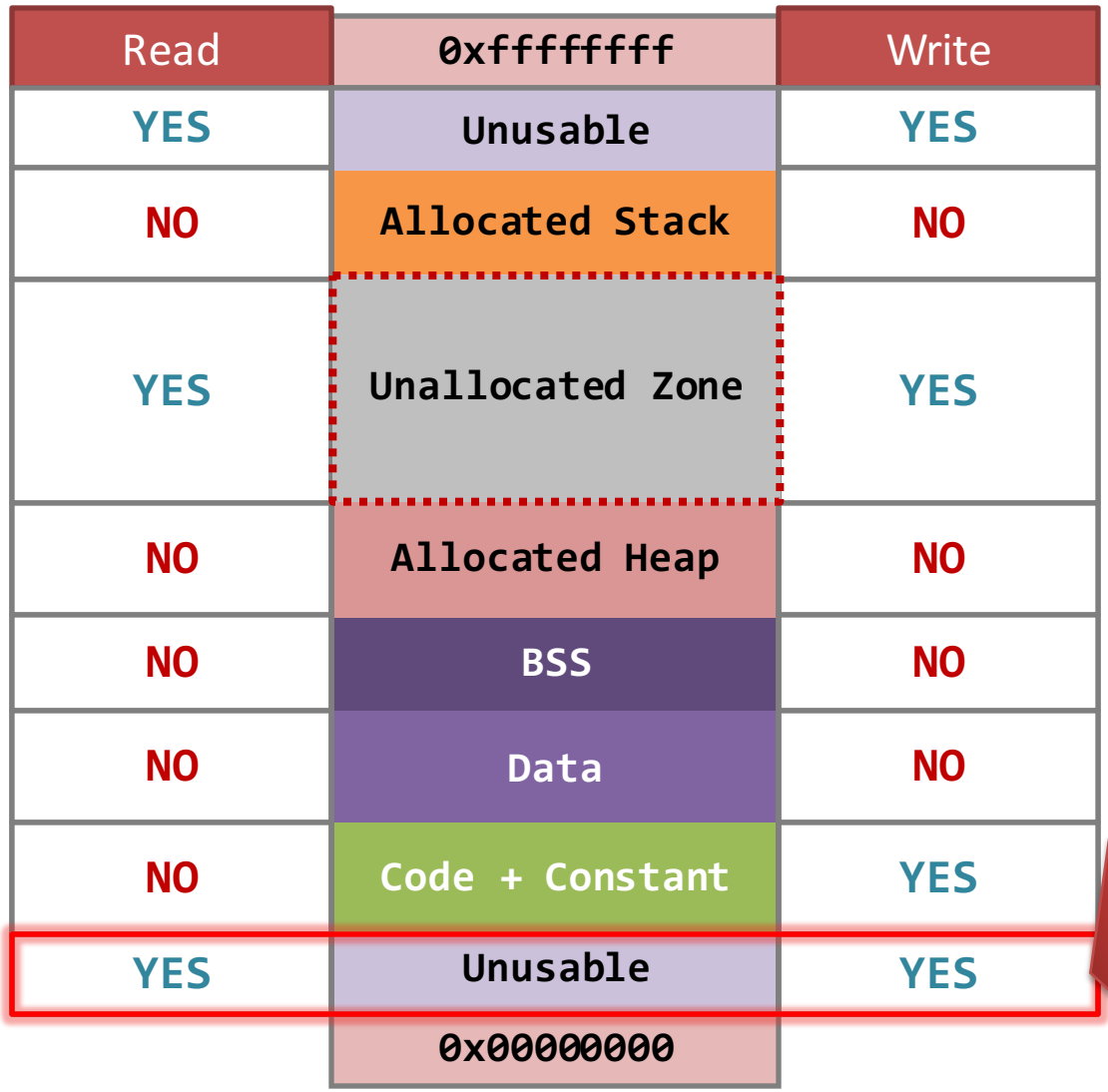
Read	0xffffffff	Write
YES	Unusable	YES
NO	Allocated Stack	NO
YES	Unallocated Zone	YES
NO	Allocated Heap	NO
NO	BSS	NO
NO	Data	NO
NO	Code + Constant	YES
YES	Unusable	YES
	0x00000000	

If you think that when you read from or write to a particular region will generate segmentation fault,

then put a “YES” in the box.

Else put a “NO” there.

How to “segmentation fault”?



Now, we can explain why:

```
char *ptr = NULL;
char c = *ptr;
```

generates segmentation fault.

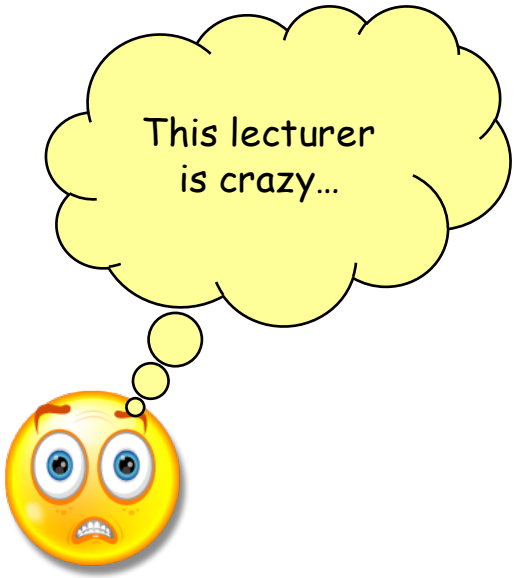
`NULL = 0x00000000`

`*ptr is reading`

Having fun with “segmentation fault”!

- What a code!

```
char *ptr = NULL;

void handler(int sig) {
    printf("TEXT segment starts at %p\n", ptr);
    exit(0);
}

int main(void) {
    char c;
    signal(SIGSEGV, handler);
    ptr = (char *) main;
    while(1) {
        ptr--;
        c = *ptr;    /* wanna generate SIGSEGV */
    }
}
```




```
[examples@3150] cat code_start.c
```

Summary

- Now, you know what is a segmentation fault, and the cause is always **carelessness**!
  - Now, you know why "`free()`" sometimes give you segmentation fault…
    - because **you corrupt the list of free blocks**!
  - Now, you know why "`malloc()`"-ing a space that is smaller than required is ok…
    - because **you are overwriting the neighboring blocks**!
  - Now, you know why sometimes writing or reading **out of bound** is ok…
    - because **you are lucky**!

Summary

- When you have a **so-called address** (maybe it is just a random sequence of 4 bytes), you de-reference it…then, one of the following cases happens:

See if you have luck…

	Read-only segments	Allocated segments	Unused or unallocated segments
Reading	No problem	No problem	**Segmentation fault**
Writing	**Segmentation fault**	No problem	**Segmentation fault**