Systems Architecture

6. Dynamic memory in C

Boni García

boni.garcia@uc3m.es

Telematic Engineering Department
School of Engineering

2024/2025

uc3m | Universidad Carlos III de Madrid

Table of contents

1. Introduction

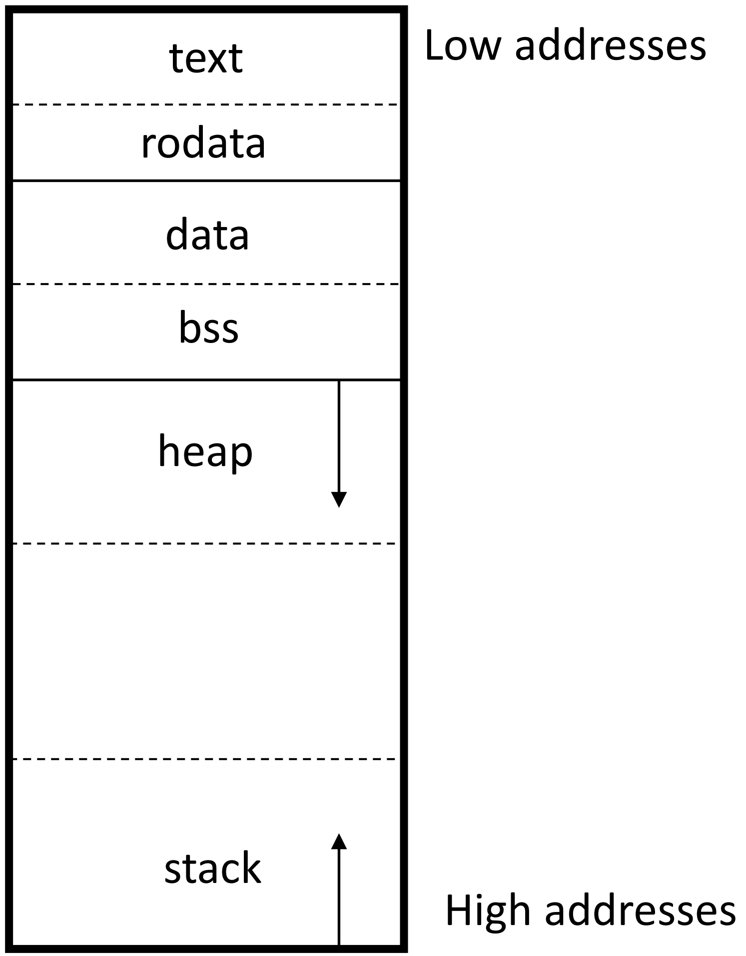
- The data managed by a process (i.e., a program in execution) is stored in the primary memory (RAM) of the computer running it and handled through variables in its source code
- In programming, memory management is the process of allocation (i.e., assign memory) and de-allocation (i.e., release memory)
- In some programming languages, such as Java or Python, memory management is automatic and transparent for the programmer
  - These programming languages use a garbage collector mechanism for releasing memory automaticaly
- In other programming languages, such as C, dynamic memory management is explicit, and it requires specific function to allocate (**`malloc`**, **`calloc`**, **`realloc`**) and de-allocate memory (**`free`**)

Table of contents

2. Memory layout in C

- C programs handle different memory segments, namely:

1. The **code** segment (sometimes called "**text**"), which stores the machine code to be executed
2. The **read-only data** segments (or "**rodata**") which contains constants and string literals (declared with *)
3. The **initialized data** segment (sometimes called simply "**data**"), which contains the global variables and static variables that are initialized by the programmer
4. The **uninitialized data segment** (called **BSS** for historical reasons -comes from *block started by symbol*-), which contains the global variables and static variables that are no initialized by the programmer
   - Data in this segment is initialized by the compiler to arithmetic 0
5. The **stack** segment, which stores local variables during the execution of the functions
6. The **heap** segment, which stores dynamically allocated memory



2. Memory layout in C

- The following program shows an example of different variables stored in different memory segments:

```
#include <stdio.h>

int global1 = 10; // Initialized global variable (data)

int global2; // Uninitialized global variable (BSS)

int main() {
    const int number = 5; // Constant variable (rodata)

    static int min; // Uninitialized static variable (BSS)

    static int max = 20; // Initialized static variable (data)

    char *msg1 = "Hello world"; // Immutable string literal (rodata)

    char msg2[] = "Hello world"; // Mutable string literal (stack)

    msg2[0] = 'h';

    printf("%s\n", msg2);

    return 0;
}
```

What happens if we try to modify `msg1[0]`?

2. Memory layout in C - The stack segment

- The values stored in the stack segment come from local variables (and function arguments)
- The stack is a LIFO (Last-In-First-Out) data structure, which is a linear data structure with two possible operations:
  - Push, which adds an element at the end of the collection
  - Pop, which removes the most recently added

We can think in a stack like a pile of plates: new plates are placed on top, and the last plate is the first one we take out

Source: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

- All data stored on the stack must have a known, fixed size
- Data with an unknown size at compile time or a size that might change must be stored on the heap instead

2. Memory layout in C - The heap segment

- The **heap** is a large pool of memory (not allocated in contiguous order) that can be used dynamically
- Unlike the stack, heap memory is allocated explicitly by programmers and it won't be deallocated until it is explicitly freed
- To manage the heap, we need to use pointers and specific C functions:
  - Allocation: **`malloc`**, **`calloc`**, **`realloc`**
  - De-allocation: **`free`**
- The heap requires **pointers** to access it
- Variables created on the heap are accessible by any function in a C program (heap variables are essentially global in scope)

2. Memory layout in C - The stack vs. the heap

- Some key differences between the stack and the heap are:

	Stack	Heap
Structure	LIFO	Free store (not contiguous order)
Memory allocation	Automatically done (on function start)	Manually done by the programmer (**`malloc`**, **`calloc`**, **`realloc`**)
Memory deallocation	Automatically done (on function exit)	Manually done by the programmer (**`free`**)
Scope	Local (access only in the scope)	Global (access with pointers)
Limit of space size	Dependent on operating system. In Linux, we can check it using a shell command: `ulimit -s`	No restrictions, other than the physical size of the computer memory
Resize	Variables cannot be resized	Variables can be resized (dynamic memory)
Access time	Faster	Slower (compared to stack)
Possible problems	Shortage of memory (*stack overflow*)	Memory leaks (memory allocated but not deallocated)

Table of contents

3. Dynamic memory functions

- Up to now, we have used the stack and data segments in our C programs
  - Variables are allocated automatically as either permanent or temporary variables
- With dynamic memory we will use the heap
  - For that we need pointers and allocation/de-allocation functions
- With dynamic memory we can create data structures that can grow or shrink as needed
  - For instance: linked lists or trees

3. Dynamic memory functions - `malloc`

- The C library function **`malloc`** (short for *memory allocation*) allocates a number of consecutive bytes in the heap and returns a pointer to the first byte
  - This function (and the rest of dynamic memory) is declared in **`stdlib.h`**
  - The prototype of **`malloc`** is as follows:

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

It returns a pointer to the newly allocated memory, or NULL if the reallocation fails

Size for the memory block to be allocated, in bytes

3. Dynamic memory functions - **malloc**

- Since malloc returns a generic pointer (i.e., **void***), although not mandatory, it is a common practice to cast (i.e., explicitly inform the compiler the type of a variable) the resulting pointer:

```
T ptr = (T*) malloc(size);
```

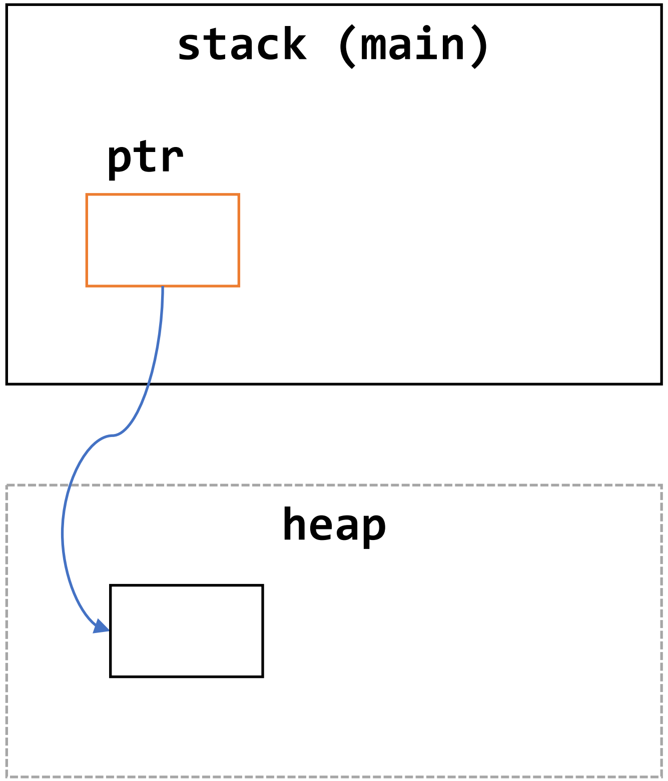
- For instance:

```
int *ptr = (int*) malloc(sizeof(int));
```

See discussion about it in:
https://stackoverflow.com/questions/605845/do-i-cast-the-result-of-malloc

3. Dynamic memory functions - malloc

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int main() {
    int *ptr = (int*) malloc(sizeof(int));

    if (ptr == NULL) {
        fputs("Dynamic memory cannot be allocated\n", stderr);
        exit(1);
    }

    // FIXME: Memory allocated is not released!

    return 0;
}
```

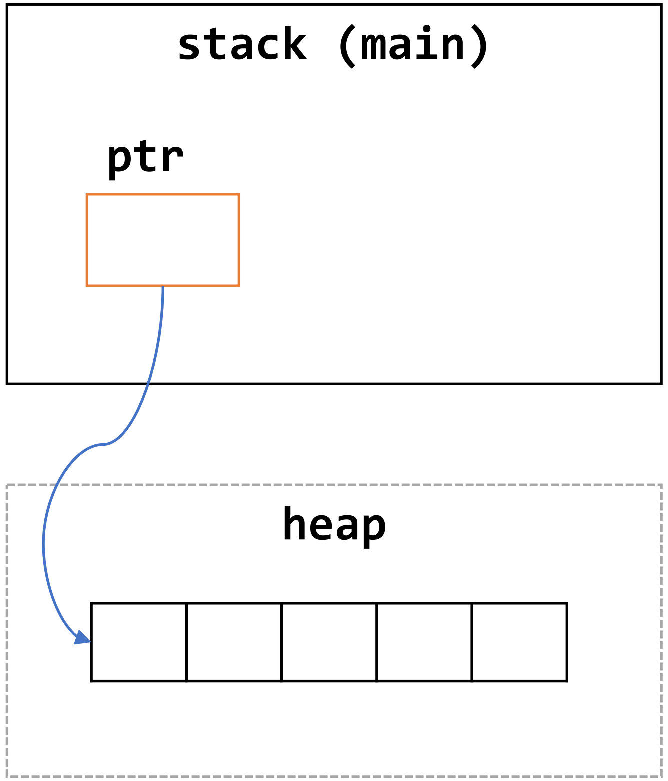


Is there any problem in this program?

3. Dynamic memory functions - malloc

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int main() {
    int *ptr = (int*) malloc(SIZE * sizeof(int));

    if (ptr == NULL) {
        fprintf(stderr, "Dynamic memory cannot be allocated.\n");
        exit(1);
    }

    // FIXME: Memory allocated is not released!

    return 0;
}
```

In this example we allocate memory for a “dynamic array”



3. Dynamic memory functions - `free`

- The C library function **`free`** deallocates the memory previously allocated by a call to **`malloc`**, **`calloc`**, or **`realloc`**
  - Like the other functions related to dynamic memory, **`free`** is declared in **`stdlib.h`**
  - The prototype of **`free`** is as follows:

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

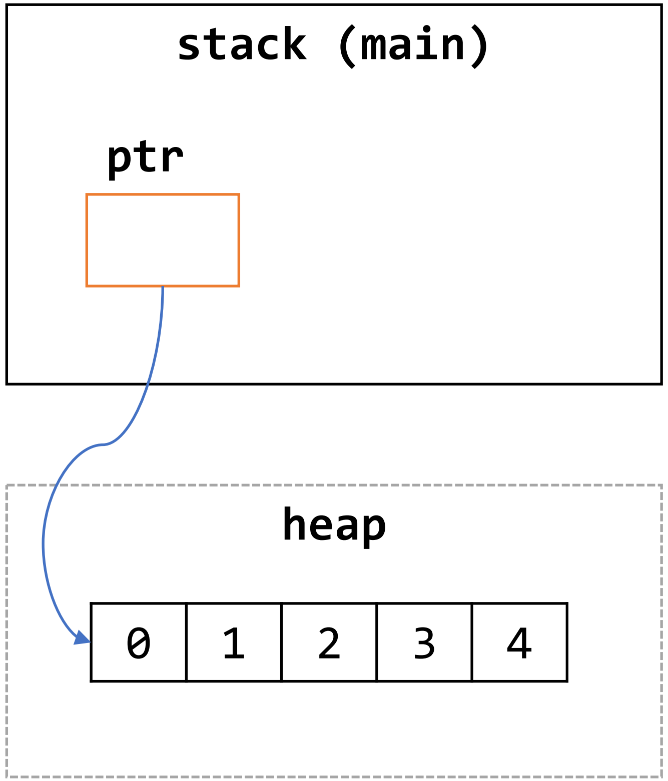
Pointer to a memory block to be deallocated

- We need to invoke **`free`** when we do not need anymore the dynamic memory previously allocated (with **`malloc`**, **`calloc`**, or **`realloc`**)
  - Otherwise, we have a *memory leak* in our program

3. Dynamic memory functions - free

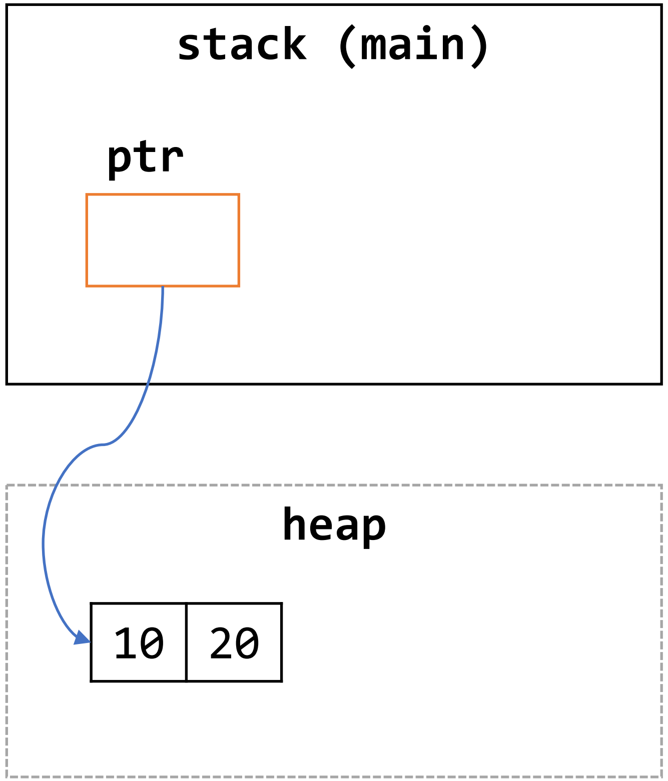
```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int main() {
    int *ptr = (int*) malloc(SIZE * sizeof(int));

    for (int i = 0; i < SIZE; i++) {
        *(ptr + i) = i;  // alternatively: ptr[i] = i;
    }

    for (int i = 0; i < SIZE; i++) {
        printf("The address %p contains %d\n", (ptr + i), *(ptr + i));
    }

    free(ptr);

    return 0;
}
```

```
The address 0x55f6acb0f2a0 contains 0
The address 0x55f6acb0f2a4 contains 1
The address 0x55f6acb0f2a8 contains 2
The address 0x55f6acb0f2ac contains 3
The address 0x55f6acb0f2b0 contains 4
```



3. Dynamic memory functions - free

```
#include <stdio.h>
#include <stdlib.h>

struct cell {
    int a;
    int b;
};

int main() {
    struct cell *ptr = (struct cell*) malloc(sizeof(struct cell));

    ptr->a = 10;
    ptr->b = 20;

    printf("The address %p contains %d and then %d\n",
            ptr, ptr->a, ptr->b);

    free(ptr);
}
```




```
The address 0x5609b97d52a0 contains 10 and then 20
```

3. Dynamic memory functions - **calloc**

- The C library function **`calloc`** (short for *contiguous allocation*) allocates a number of consecutive bytes in the heap and returns a pointer to the first byte
  - Like the other functions related to dynamic memory, **`calloc`** is declared in **`stdlib.h`**
  - It initializes the allocated memory to zero
  - The prototype of **`calloc`** is as follows:

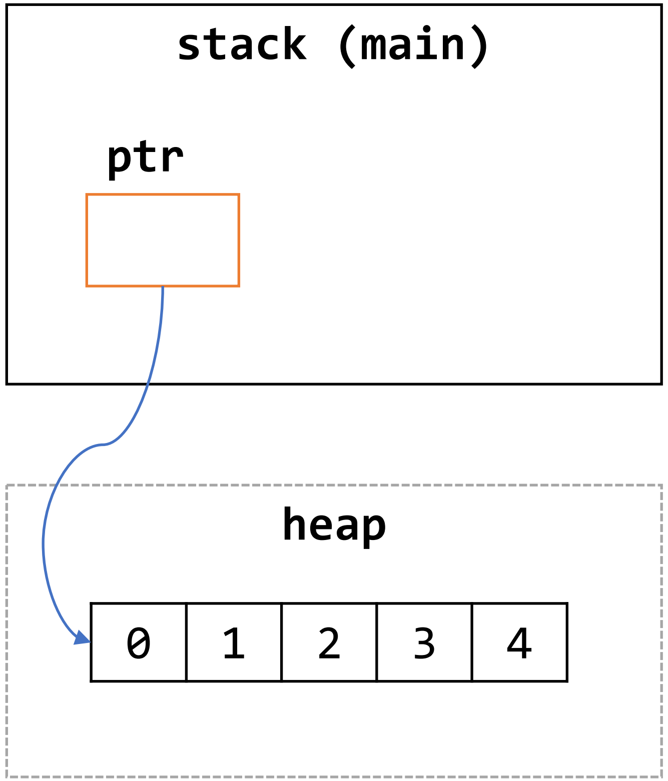
```
void *calloc(size_t nitems, size_t size);
```

- `void *`: It returns a pointer to the allocated memory, or NULL if the allocation fails
- `nitems`: Number of elements to be allocated
- `size`: Size of each element, in bytes

3. Dynamic memory functions - calloc

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 5

int main() {
    int *ptr = (int*) calloc(SIZE, sizeof(int));

    for (int i = 0; i < SIZE; i++) {
        ptr[i] = i;
    }

    for (int i = 0; i < SIZE; i++) {
        printf("The address %p contains %d\n", (ptr + i), ptr[i]);
    }

    free(ptr);

    return 0;
}
```

```
The address 0x5601eb4172a0 contains 0
The address 0x5601eb4172a4 contains 1
The address 0x5601eb4172a8 contains 2
The address 0x5601eb4172ac contains 3
The address 0x5601eb4172b0 contains 4
```



3. Dynamic memory functions - `calloc`

- There key differences between **`malloc`** and **`calloc`** are:

`malloc`	`calloc`
It creates one block of memory of a fixed size	It assigns more than one block of memory to a single variable
It has one argument: • The total size (in bytes) of memory to be allocated	It has two arguments: • The number of items to be allocated • The size (in bytes) of each element
It doesn't initialize the allocated memory	It initializes the allocated memory to zero
It is faster than **`calloc`**	It is slower than **`malloc`**

3. Dynamic memory functions - `realloc`

- The C library function **`realloc`** (short for *reallocation*) resizes the memory block pointed to by a pointer that was previously allocated with **`malloc`** or **`calloc`**
  - Like the other functions related to dynamic memory, **`realloc`** is declared in **`stdlib.h`**
- The prototype of **`realloc`** is as follows:

```
void *realloc (void *ptr, size_t size);
```

It returns a pointer to the newly allocated memory, or NULL if the reallocation fails

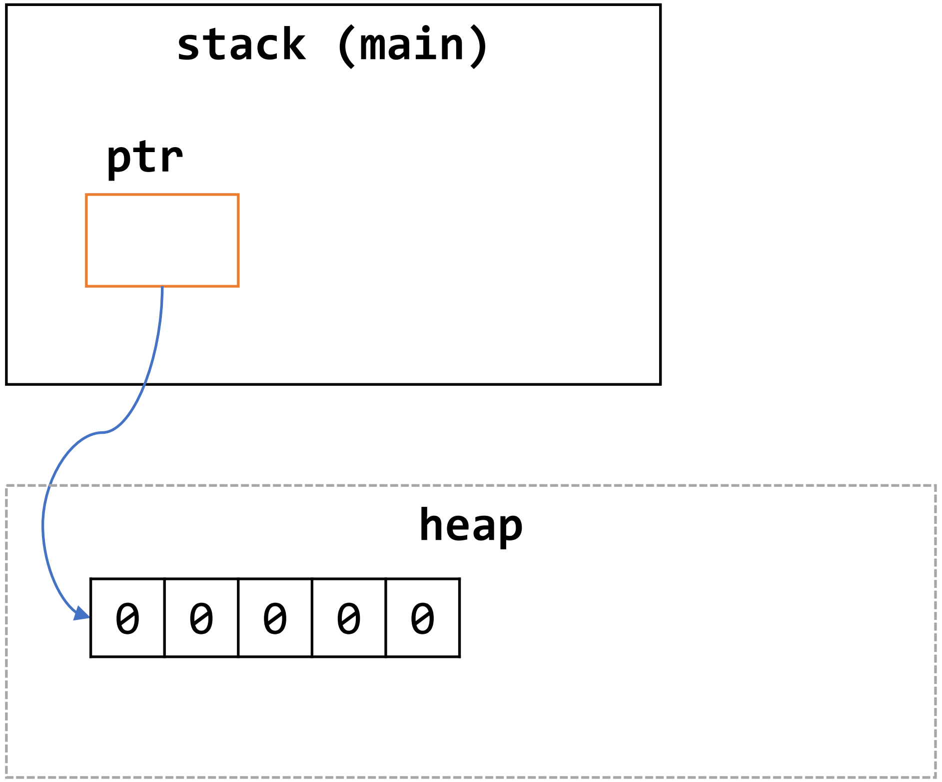
Pointer to a memory block to be reallocated

New size for the memory block, in bytes

Fork me on GitHub

3. Dynamic memory functions - `realloc`

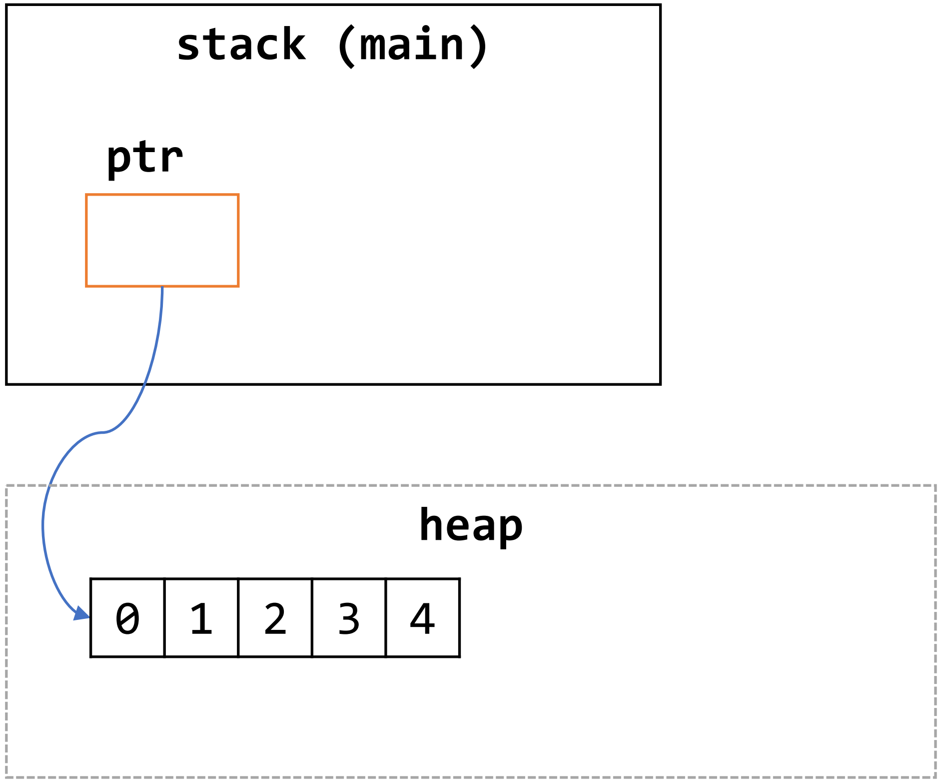
```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE_1 5
#define SIZE_2 10

void fill_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        array[i] = i;
    }
}

void display_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }
    printf("\n");
}

int main() {
    int *ptr = (int*) calloc(SIZE_1, sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, 0, SIZE_1);
    display_array(ptr, 0, SIZE_1);

    ptr = (int*) realloc(ptr, SIZE_2 * sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, SIZE_1, SIZE_2);
    display_array(ptr, 0, SIZE_2);

    free(ptr);
    return 0;
}
```




```
The address of ptr is 0x560b3f37e2a0
```

3. Dynamic memory functions - realloc

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE_1 5
#define SIZE_2 10

void fill_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        array[i] = i;
    }
}

void display_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }
    printf("\n");
}

int main() {
    int *ptr = (int*) calloc(SIZE_1, sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, 0, SIZE_1);
    display_array(ptr, 0, SIZE_1);

    ptr = (int*) realloc(ptr, SIZE_2 * sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, SIZE_1, SIZE_2);
    display_array(ptr, 0, SIZE_2);

    free(ptr);
    return 0;
}
```

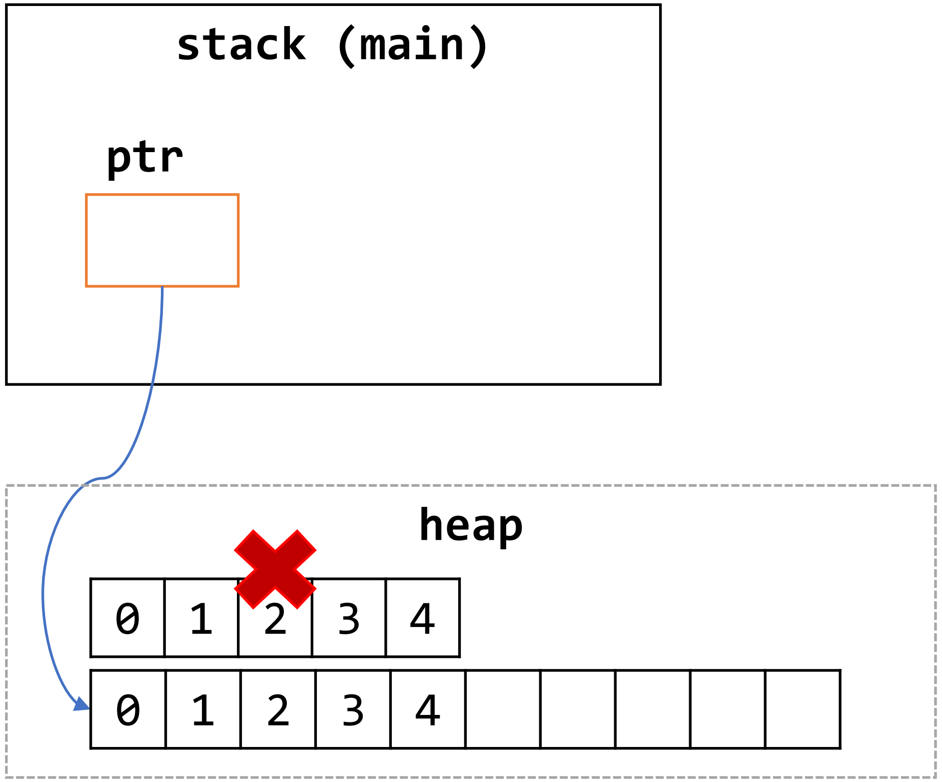



```
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
```

Fork me on GitHub

3. Dynamic memory functions - `realloc`

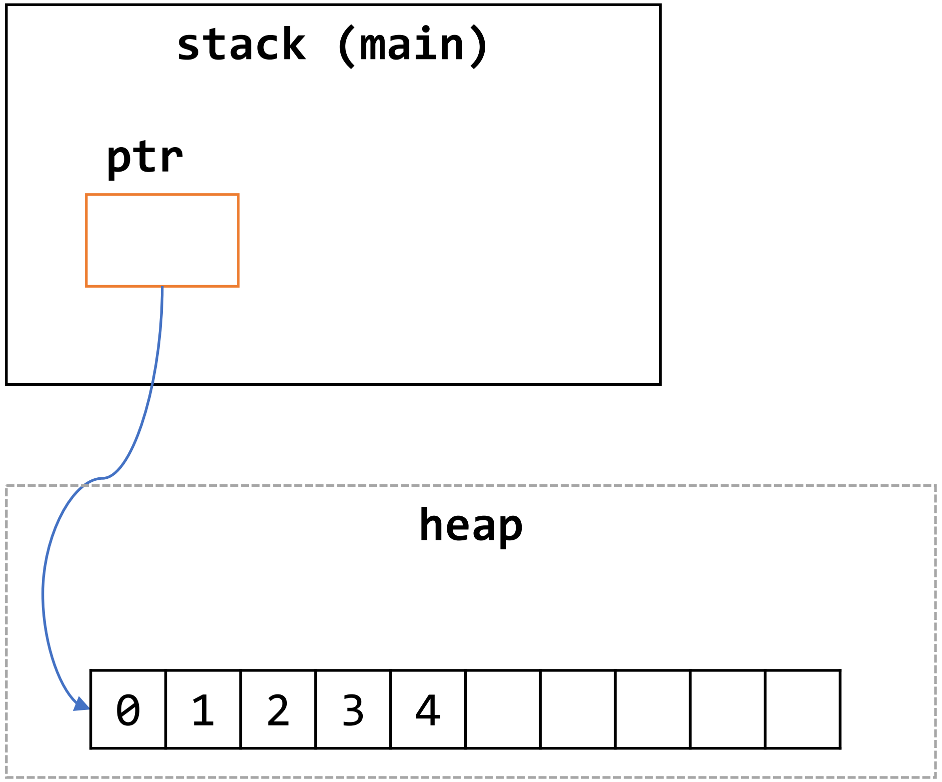
```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE_1 5
#define SIZE_2 10

void fill_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        array[i] = i;
    }
}

void display_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }
    printf("\n");
}

int main() {
    int *ptr = (int*) calloc(SIZE_1, sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, 0, SIZE_1);
    display_array(ptr, 0, SIZE_1);

    ptr = (int*) realloc(ptr, SIZE_2 * sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, SIZE_1, SIZE_2);
    display_array(ptr, 0, SIZE_2);

    free(ptr);
    return 0;
}
```



Internally, realloc allocates memory for the new block, copy the data from the old block over, free the old block and return a pointer to the beginning of the new block

3. Dynamic memory functions - realloc

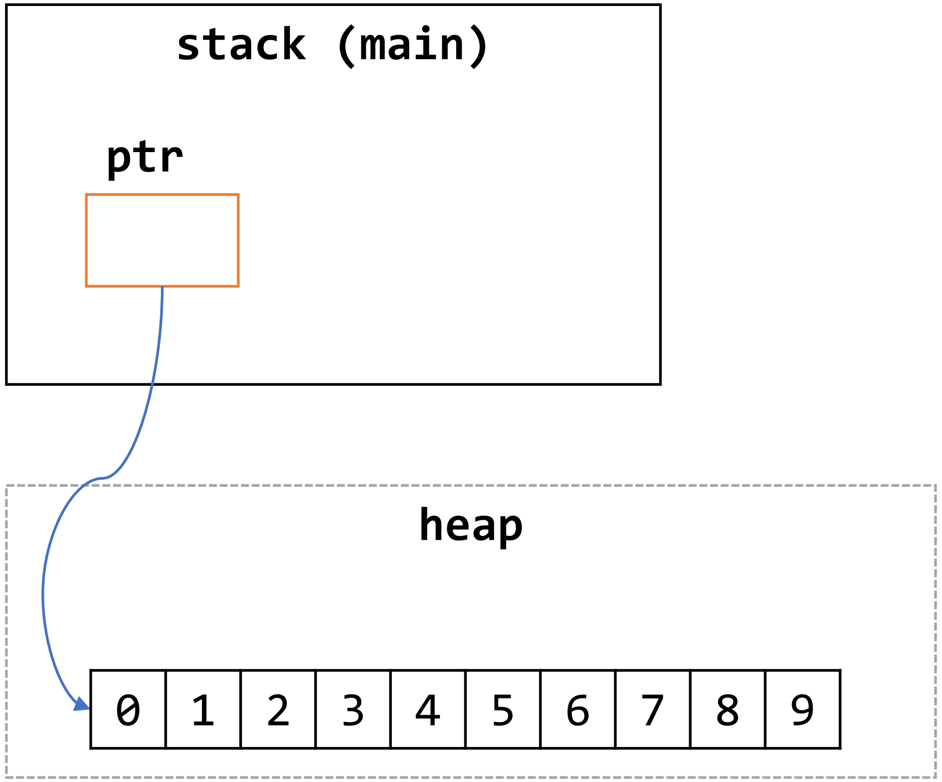
```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE_1 5
#define SIZE_2 10

void fill_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        array[i] = i;
    }
}

void display_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }
    printf("\n");
}

int main() {
    int *ptr = (int*) calloc(SIZE_1, sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, 0, SIZE_1);
    display_array(ptr, 0, SIZE_1);

    ptr = (int*) realloc(ptr, SIZE_2 * sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, SIZE_1, SIZE_2);
    display_array(ptr, 0, SIZE_2);

    free(ptr);
    return 0;
}
```




```
The address of ptr is 0x5628ba8d76d0
```

3. Dynamic memory functions - realloc

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE_1 5
#define SIZE_2 10

void fill_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        array[i] = i;
    }
}

void display_array(int *array, int init, int end) {
    for (int i = init; i < end; i++) {
        printf("array[%d]=%d\n", i, array[i]);
    }
    printf("\n");
}

int main() {
    int *ptr = (int*) calloc(SIZE_1, sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, 0, SIZE_1);
    display_array(ptr, 0, SIZE_1);

    ptr = (int*) realloc(ptr, SIZE_2 * sizeof(int));
    printf("The address of ptr is %p\n", ptr);

    fill_array(ptr, SIZE_1, SIZE_2);
    display_array(ptr, 0, SIZE_2);

    free(ptr);
    return 0;
}
```




```
array[0]=0
array[1]=1
...
array[8]=8
array[9]=9
```

3. Dynamic memory functions - Out of scope allocation

Fork me on GitHub

- Let's consider the following example:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr;

    ptr = (int*) malloc(sizeof(int));

    *ptr = 42;
    printf("*ptr=%d\n", *ptr);

    free(ptr);

    return 0;
}
```

How to do this memory allocation but in a different function?

Fork me on GitHub

3. Dynamic memory functions - Out of scope allocation

```c
#include <stdio.h>
#include <stdlib.h>

void allocate(int *ptr) {
    ptr = (int*) malloc(sizeof(int));
}

int main() {
    int *ptr;
    allocate(ptr);

    *ptr = 42;
    printf("*ptr=%d\n", *ptr);

    free(ptr);

    return 0;
}
```

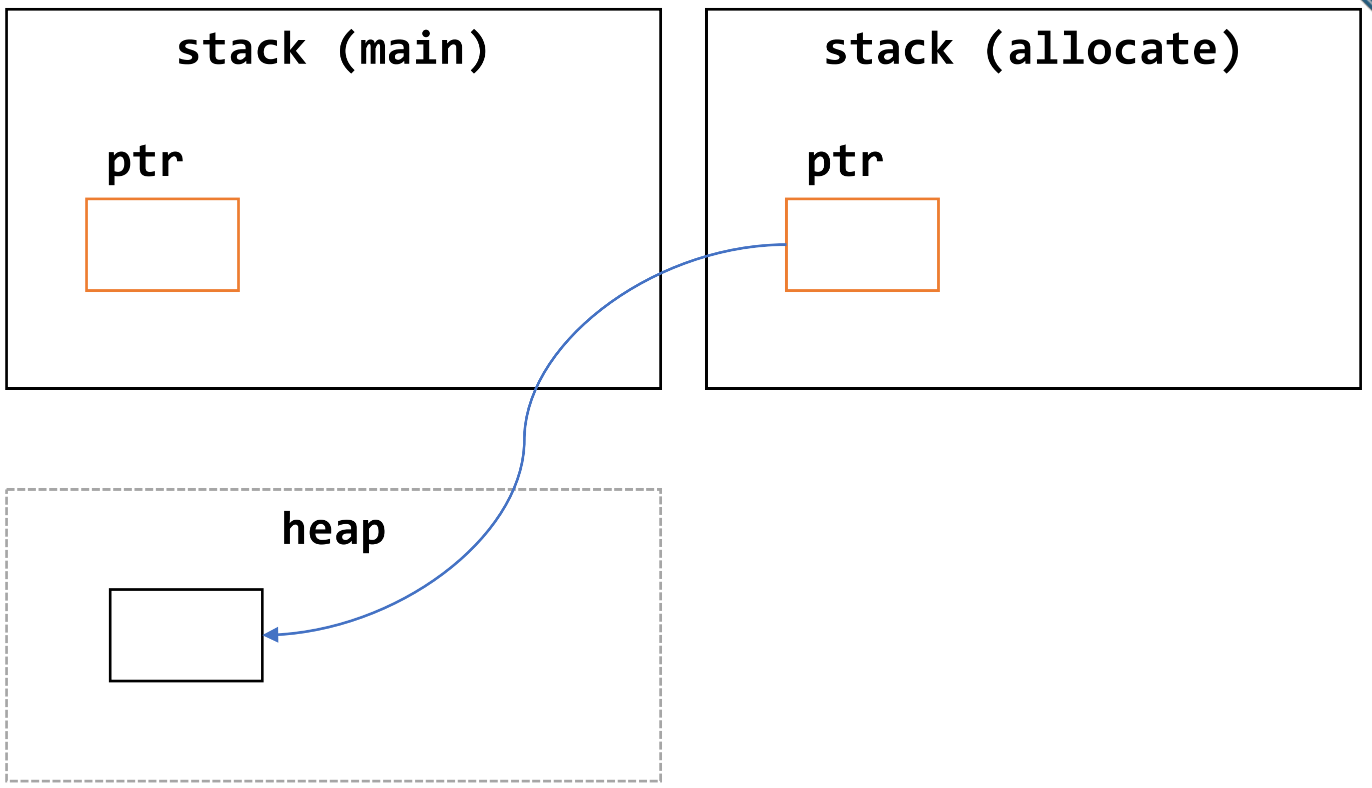
Is this a valid solution?

Fork me on GitHub

3. Dynamic memory functions - Out of scope allocation

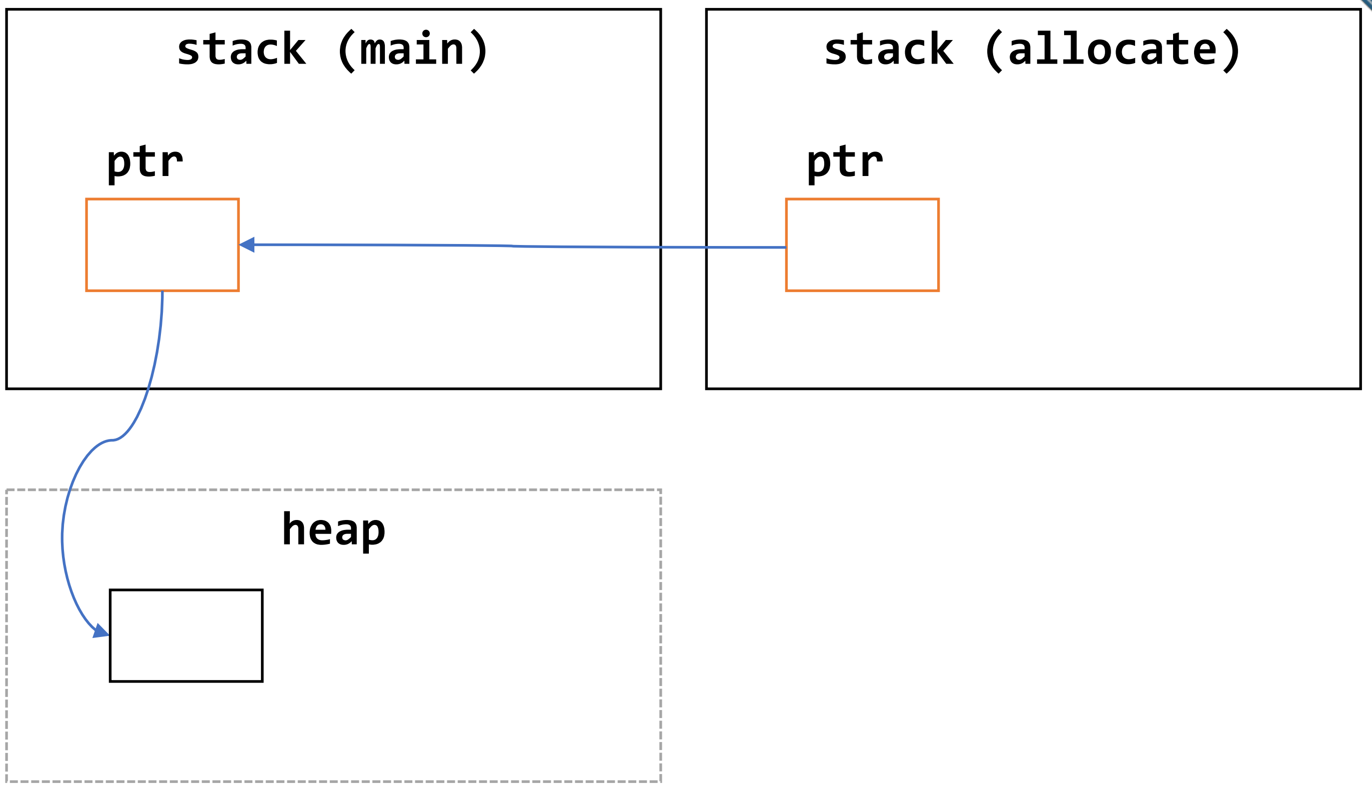
```c
#include <stdio.h>
#include <stdlib.h>

void allocate(int *ptr) {
    ptr = (int*) malloc(sizeof(int));
}

int main() {
    int *ptr;
    allocate(ptr);

    *ptr = 42;
    printf("*ptr=%d\n", *ptr);

    free(ptr);

    return 0;
}
```

Is this a valid solution?



3. Dynamic memory functions - Out of scope allocation

```c
#include <stdio.h>
#include <stdlib.h>

void allocate(int **ptr) {
    *ptr = (int*) malloc(sizeof(int));
}

int main() {
    int *ptr;
    allocate(&ptr);

    *ptr = 42;
    printf("*ptr=%d\n", *ptr);

    free(ptr);

    return 0;
}
```

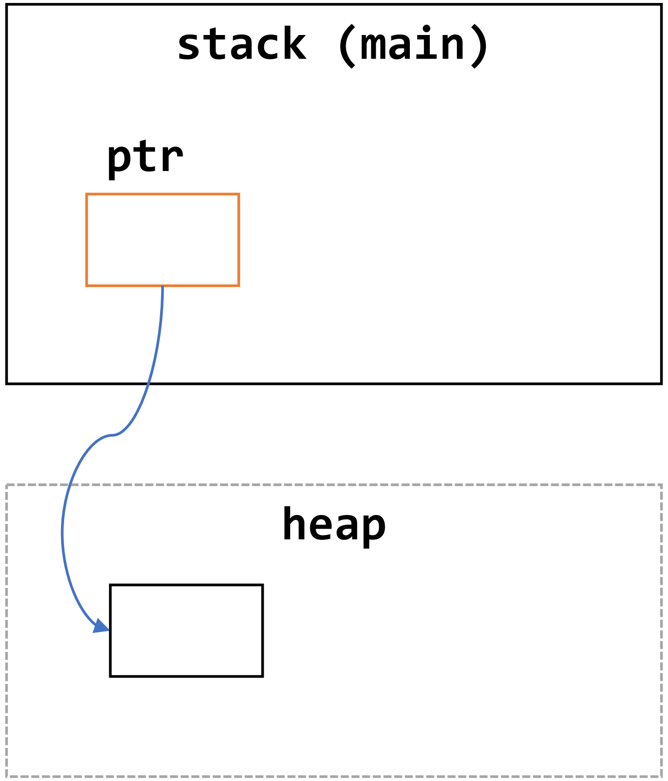


Instead, we need to use a **double pointer**

Fork me on GitHub

3. Dynamic memory functions - Out of scope allocation

```
#include <stdio.h>
#include <stdlib.h>

int* allocate() {
    return (int*) malloc(sizeof(int));
}

int main() {
    int *ptr = allocate();

    *ptr = 42;
    printf("*ptr=%d\n", *ptr);

    free(ptr);

    return 0;
}
```

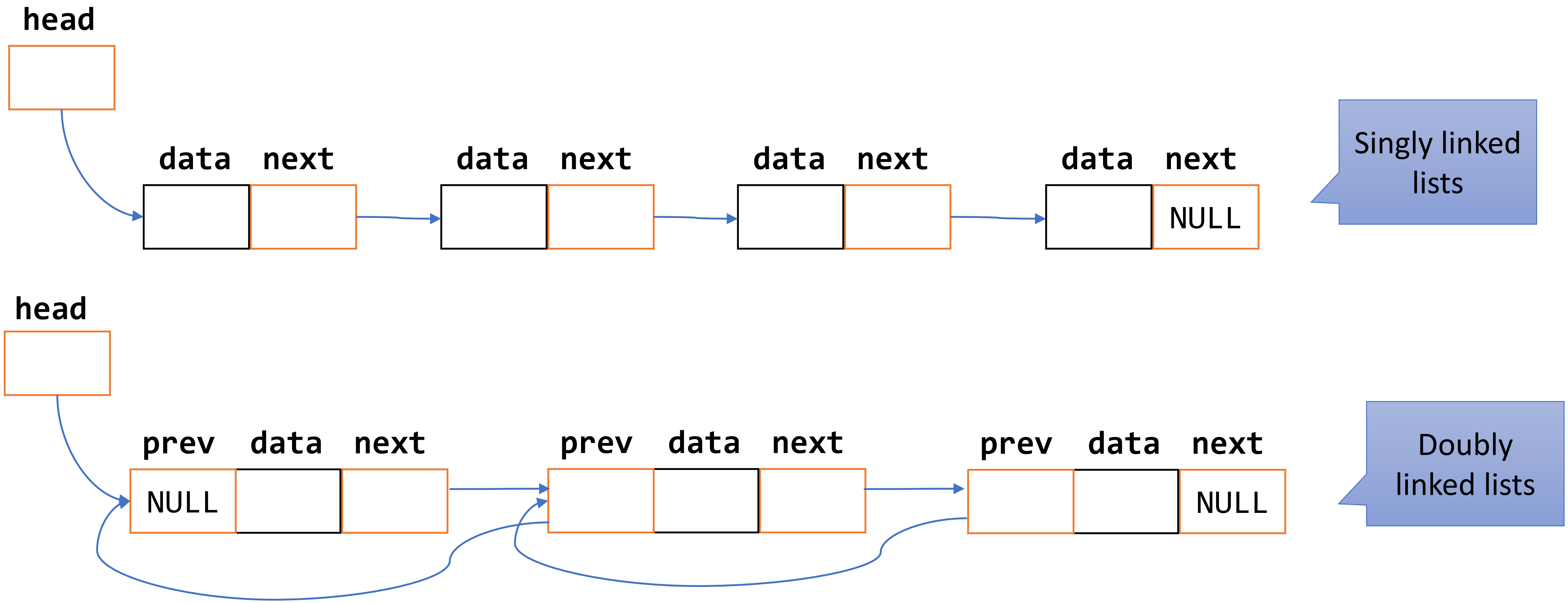


Alternatively, we can make the function return a pointer to the newly allocated memory block

Table of contents

4. Linked lists

- A linked list is a sequence of data structures which are connected together with links (implemented with pointers in C)



Fork me on GitHub

4. Linked lists

- We use an **struct** to define the nodes of the link list
- In singly linked list, in addition to some data (an integer in this example), a pointer to the next element is declared as a member in the structure
- Then, we use a pointer to the structure to declare the first node of the linked list (usually called head)

```
/*
 * Node definition
 */
typedef struct Node {
    int data;
    struct Node *next;
} Node;
```

```
Node *head = NULL;
```

To ensure the linked is empty, we must initialize its head to NULL

4. Linked lists

- Typically, we create nodes using a function, and then, we insert the node in the linked list (head)
  - We need to use malloc to allocate memory for that node:

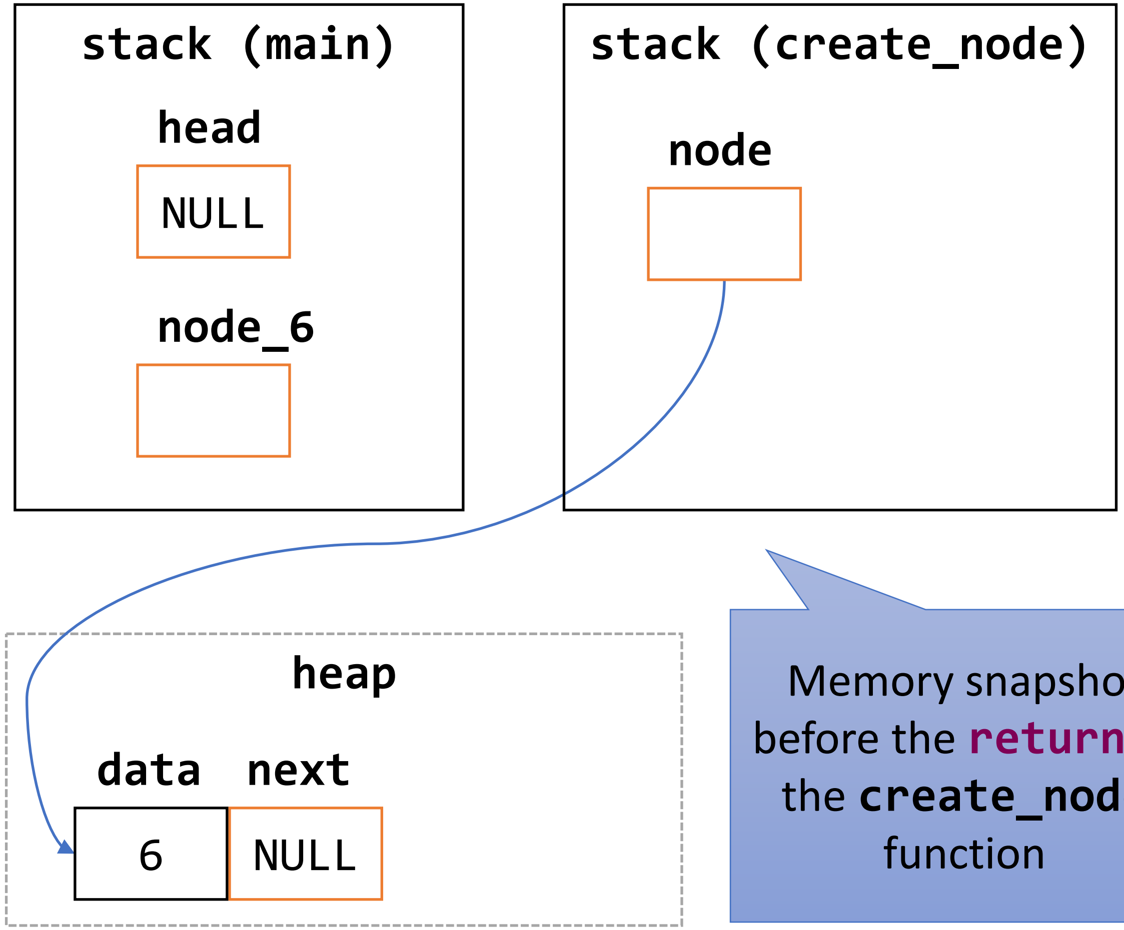
Example of function to create node knowing its content (an integer value in this case)

```c
/*
 * Create new node (using data as input)
 */
Node* create_node(int data) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->data = data;
    node->next = NULL;

    return node;
}
```

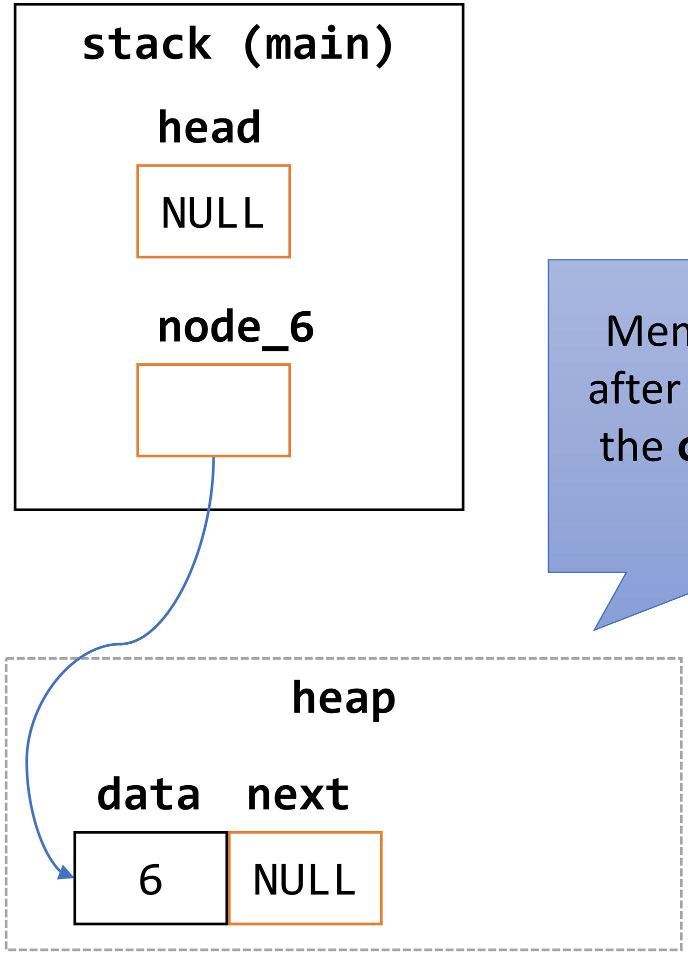
Example of node creation with a given value (6 in this example)

```c
int main() {
    Node *head = NULL;

    Node *node_6 = create_node(6);

    // ...
```

4. Linked lists



Memory snapshot before the **return** of the **create_node** function



Memory snapshot after the **return** of the **create_node** function

4. Linked lists

- Once we have created a node, there are different strategies to insert the node in the linked list
  - It can be inserted at the beginning, at the end, or somewhere in the middle
- The following function shows an example to insert a node at the beginning
  - We need to use **double pointers** (since the push function need to change the original value of head)

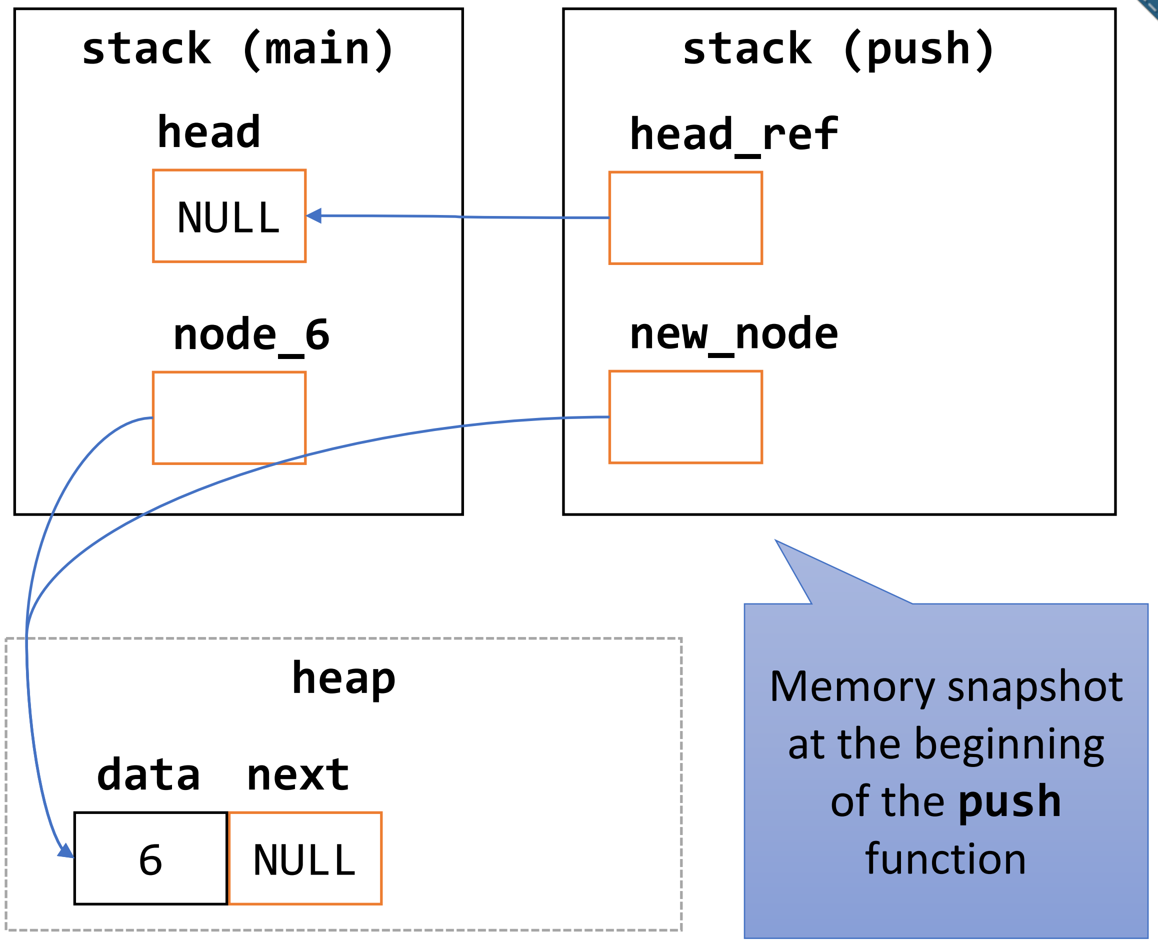
```
/*
 * Insert Node at the beginning
 */
void push(Node **head_ref, Node *new_node) {
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```

4. Linked lists

- When calling the push the first time:

```
int main() {
    Node *head = NULL;

    // Push 6
    Node *node_6 = create_node(6);
    push(&head, node_6);
    printf("Insert 6 at the beginning. Linked list is:");
    print_list(head);

    // ...
```

```
/*
 * Insert Node at the beginning
 */
void push(Node **head_ref, Node *new_node) {
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```

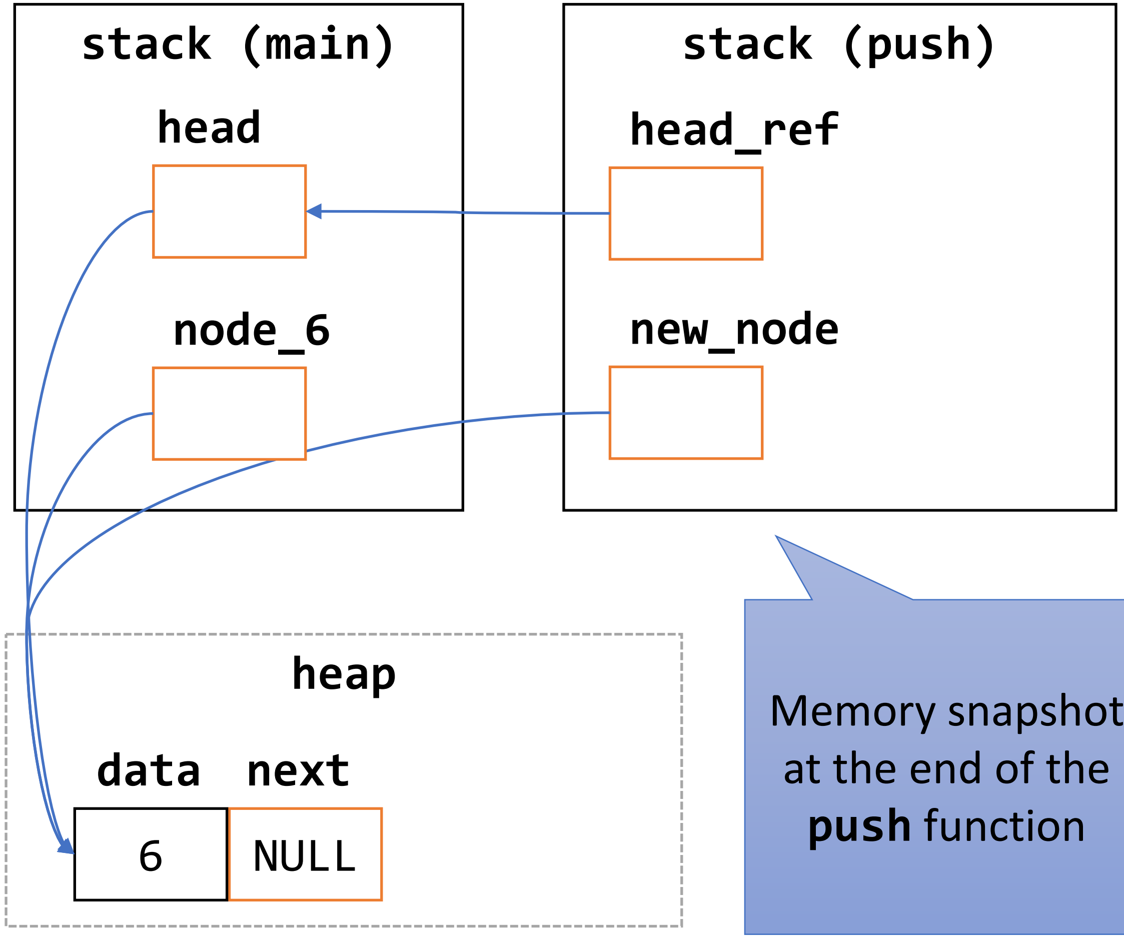


4. Linked lists

- When calling the push the first time:

```c
int main() {
    Node *head = NULL;

    // Push 6
    Node *node_6 = create_node(6);
    push(&head, node_6);
    printf("Insert 6 at the beginning. Linked list is:");
    print_list(head);

    // ...
```

```c
/*
 * Insert Node at the beginning
 */
void push(Node **head_ref, Node *new_node) {
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```



Memory snapshot at the end of the **push** function

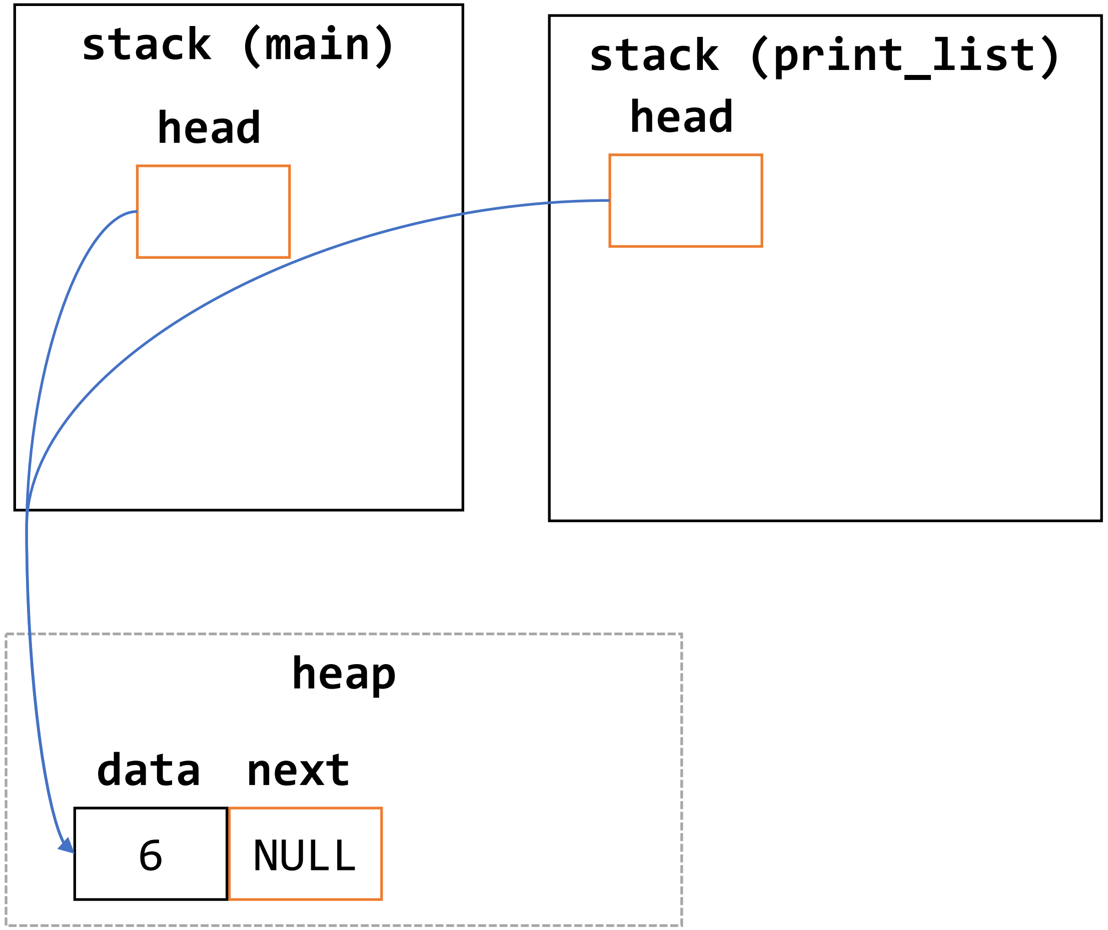
4. Linked lists

- We use another function to display the content of our linked list:

```
int main() {
    // ...

    printf("Insert 6 at the beginning. Linked list is:");
    print_list(head);

    // ...
```

```
/*
 * Display list content in the standard output
 */
void print_list(Node *head) {
    while (head != NULL) {
        printf(" %d", head->data);
        head = head->next;
    }
    printf("\n");
}
```

```
Insert 6 at the beginning. Linked list is: 6
```



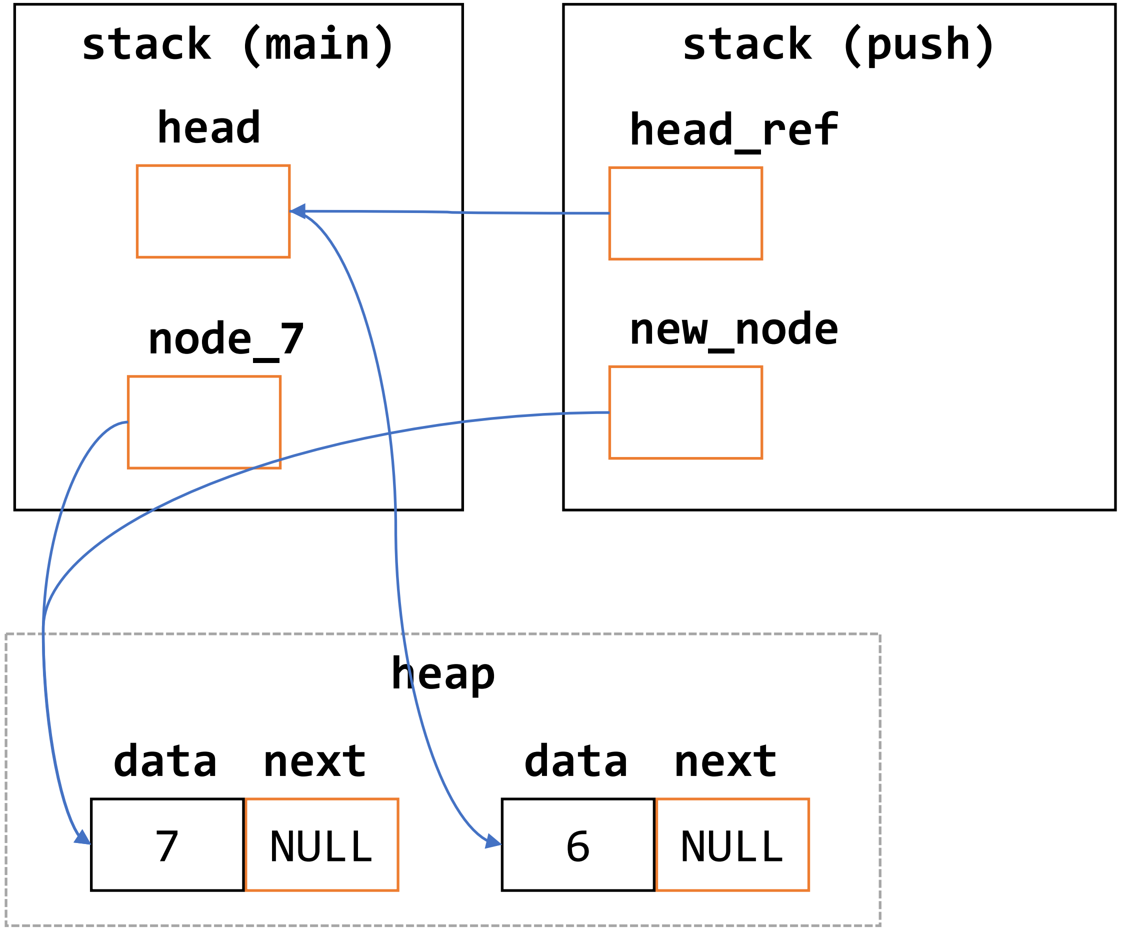
4. Linked lists

- When calling the push the second time:

```
    // Push 7
    Node *node_7 = create_node(7);
    push(&head, node_7);
    printf("Insert 7 at the beginning. Linked list is:");
    print_list(head);

    // ...
```

```
/*
 * Insert Node at the beginning
 */
void push(Node **head_ref, Node *new_node) {
    new_node->next = *head_ref;
    *head_ref = new_node;
}
```

Memory snapshot at the beginning of the **push** function the second time



Fork me on GitHub

4. Linked lists

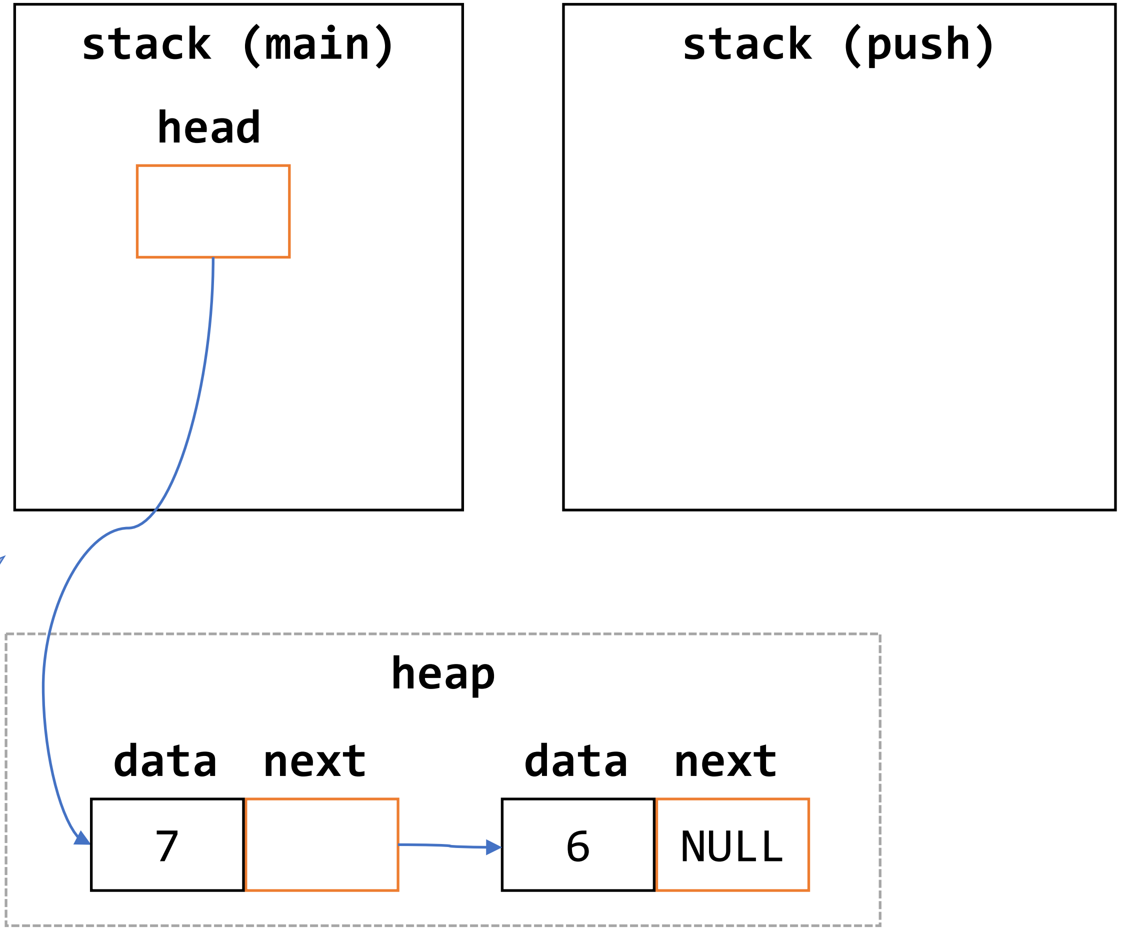
- When calling the push the second time:

```
    // Push 7
    Node *node_7 = create_node(7);
    push(&head, node_7);
    printf("Insert 7 at the beginning. Linked list is:");
    print_list(head);

    // ...
```

```
/*
 * Display list content in the standard output
 */
void print_list(Node *head) {
    while (head != NULL) {
        printf(" %d", head->data);
        head = head->next;
    }
    printf("\n");
}
```

```
Insert 7 at the beginning. Linked list is: 7 6
```

Memory snapshot after calling the **push** function the second time



4. Linked lists

- To delete a node, several cases need to be considered
  - If the node to be deleted is at beginning of the list
  - If the node to be deleted is after the first node
  - If the node to be deleted is not in the list

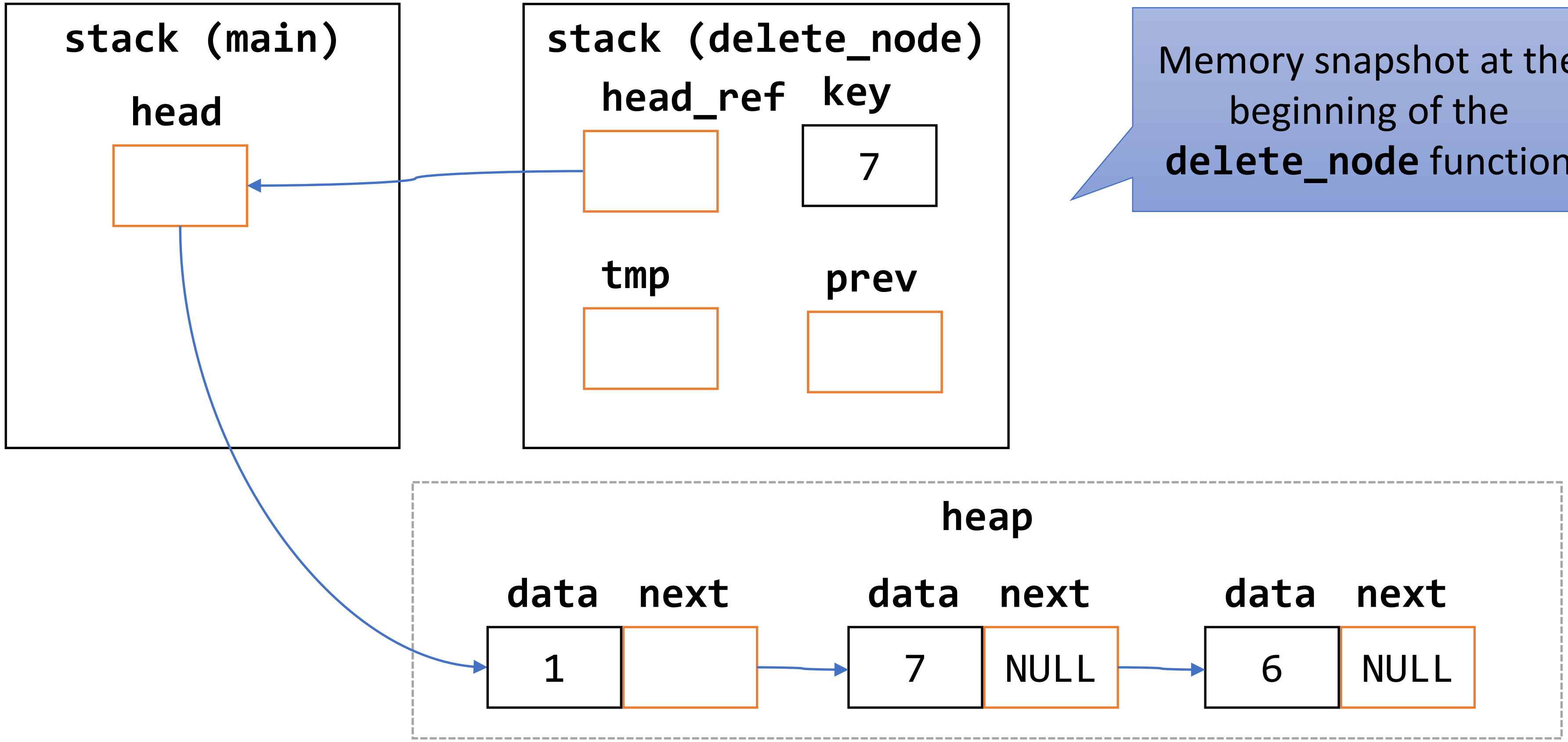
We need to use to auxiliar pointers (called `tmp` and `prev` in this example) to keep references to the node to be found (`tmp`) and the previous one (`prev`)

```
/*
 * Delete node by value
 */
void delete_node(Node **head_ref, int key) {
    Node *tmp = *head_ref, *prev;

    // The node to be deleted is the first position
    if (tmp != NULL && tmp->data == key) {
        *head_ref = tmp->next;
        free(tmp);
        return;
    }

    // If not, we find the matching node (if any)
    while (tmp != NULL && tmp->data != key) {
        prev = tmp;
        tmp = tmp->next;
    }

    // If not found, nothing is done
    if (tmp == NULL) {
        return;
    }

    // If found, the previous node is connected to the next
    // and then, the memory of the matching node is released
    prev->next = tmp->next;
    free(tmp);
}
```

Fork me on GitHub

4. Linked lists

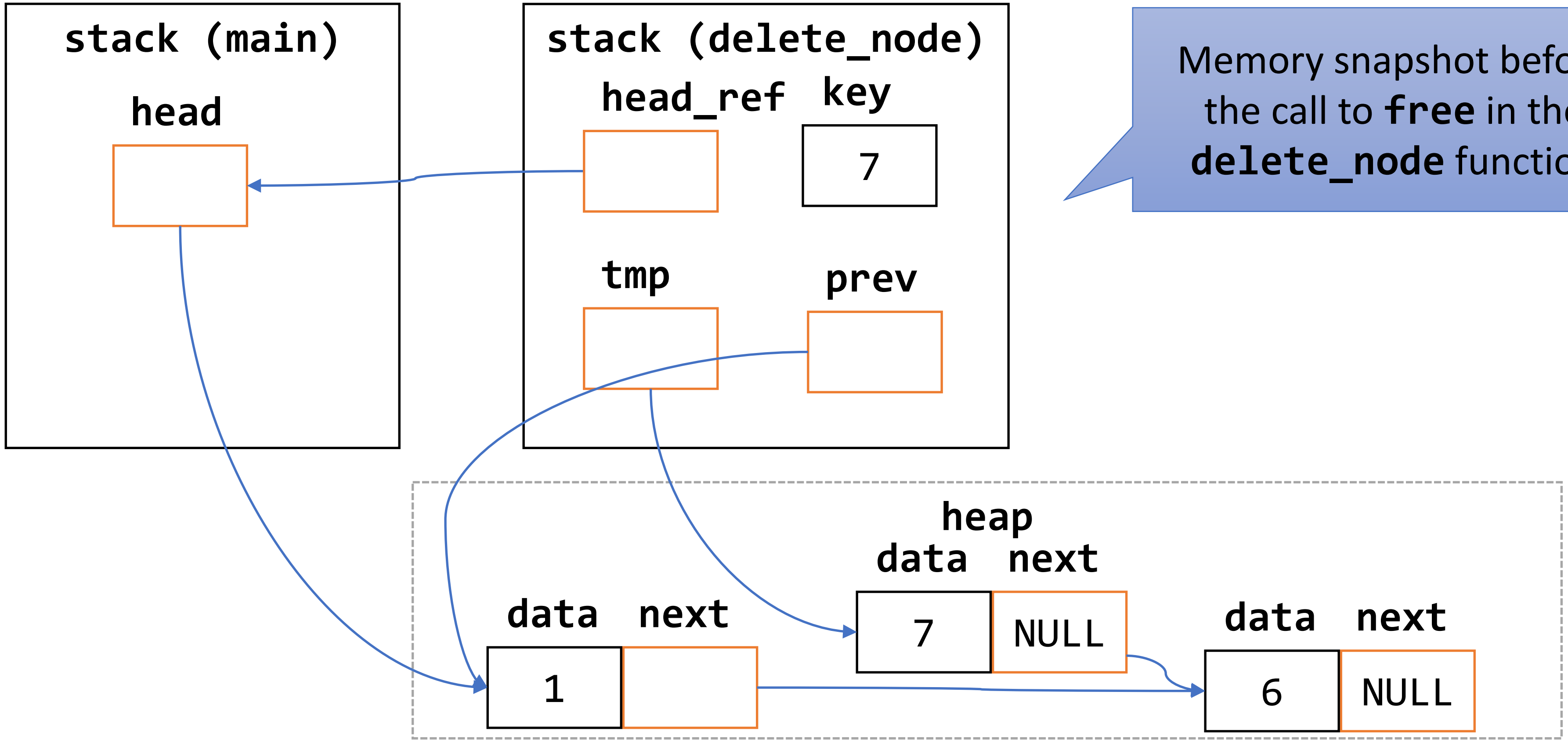
```
// Delete 7
delete_node(&head, 7);
printf("Delete node with value 7. Linked list is:");
print_list(head);
```



Memory snapshot at the beginning of the **delete_node** function

4. Linked lists

```c
// Delete 7
delete_node(&head, 7);
printf("Delete node with value 7. Linked list is:");
print_list(head);
```



Memory snapshot before the call to **free** in the **delete_node** function

Fork me on GitHub

4. Linked lists

- The following functions show how to insert a node at end and after a giving position

```
void append(Node **head_ref, Node *new_node) {
    // If list is empty, the node is inserted at the beginning
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    // If list is not empty, we look for the last node
    Node *last = *head_ref;
    while (last->next != NULL) {
        last = last->next;
    }

    last->next = new_node;
}
```

```
/*
 * Insert Node after a giving position
 */
void insert_after(Node *prev_node, Node *new_node) {
    if (prev_node == NULL) {
        printf("The previous node cannot be NULL\n");
        return;
    }

    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
```

4. Linked lists

- We need to clear all the allocated memory for the linked list before the program exit

```
/*
 * Delete list (free memory)
 */
void clear_list(Node **head_ref) {
    Node *current = *head_ref;
    Node *next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *head_ref = NULL;
}
```

Table of contents

5. Takeaways

- There are four memory segments for a C program: **code** (which stores the machine code to be executed), **data** (which stores global variables, static variables, constants, and string literals), **stack** (which stores local variables during the execution of the functions) and **heap** (which stores dynamically allocated memory)
- Dynamic memory management is explicit in C, and it requires the use of pointers and specific function to allocate (**`malloc`**, **`calloc`**, **`realloc`**) and de-allocate (**`free`**) memory
- A **linked list** is a sequence of data structures, which are connected together with links (implemented with pointers in C)