McGill UNIVERSITY

C LAB 3: POINTERS & MEMORY ALLOCATION I

ECSE 427/COMP 310: WINTER 2022

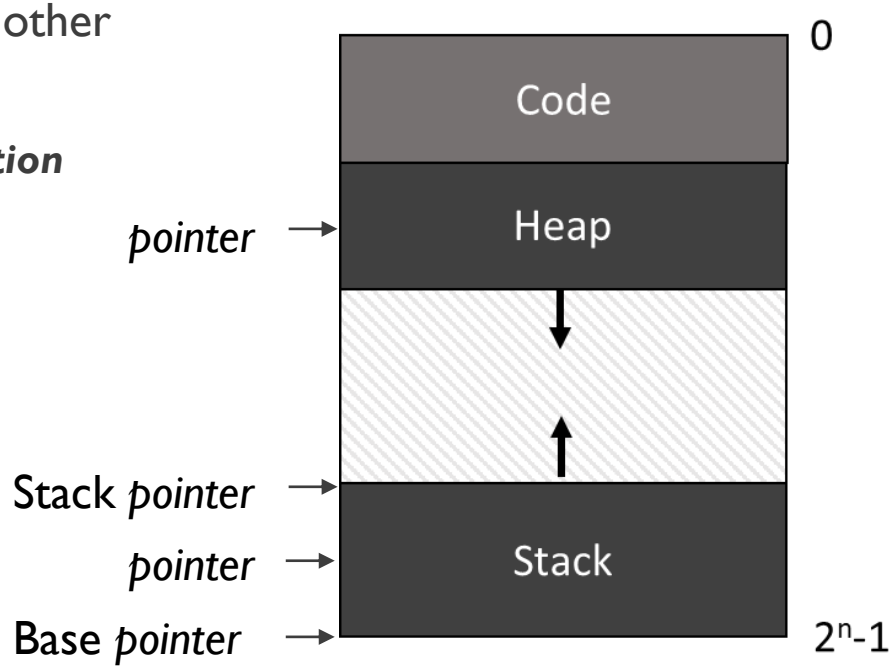
TA: MUSHFIQUE (MOHAMMAD MUSHFIQUR RAHMAN)

CONTENTS

- POINTER INTRODUCTION
- ADDRESS OPERATOR – “&”
- POINTER OPERATIONS
- NULL POINTER
- MEMORY ALLOCATION
- MEMORY LEAK
- POINTER ARITHMETIC
- EXAMPLE USAGE – LINKED LISTS
- Q&A

POINTER INTRODUCTION

- **Pointer** – It is a variable that stores the ***address*** (in memory) of another variable.
  - Simpler way to remember - *A **pointer** is anything that "**points**" to a **location (address) in the memory***.
- A **pointer** provides a way of accessing a variable without referring directly to the variable (will see examples).
- Operations:
  - Declaration
  - Initialization
  - Dereferencing – *New for pointers!*

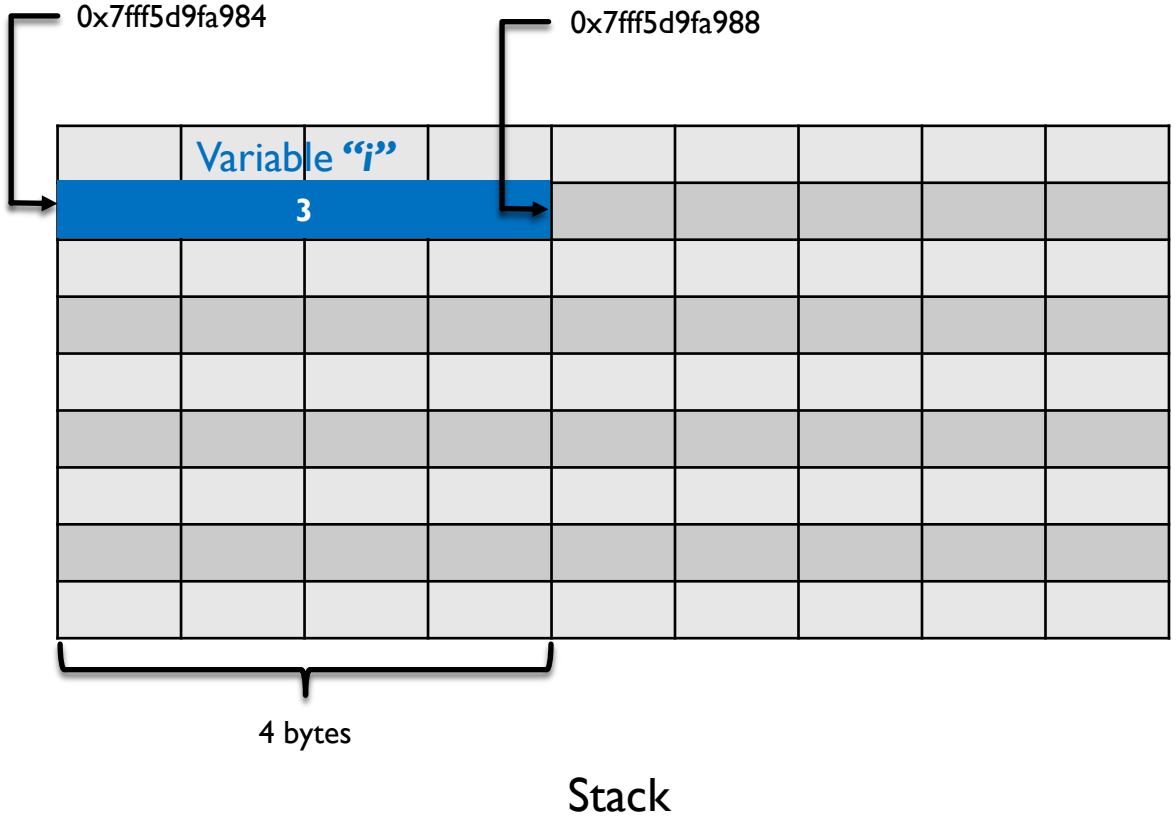


ADDRESS OPERATOR – "&"

- **Variable Identifier –** represents the ***value*** of the variable.
- **Variable Identifier preceded with "&" –** represents the ***address*** of the variable.
- Example:

```
int i;
i = 3;
printf (" The value of i = %d\n", i);
printf (" The address of i = %p\n", &i);
```

```
The value of i = 3
The address of i = 0x7fff5d9fa984
```



POINTER OPERATIONS

- **Declaration**

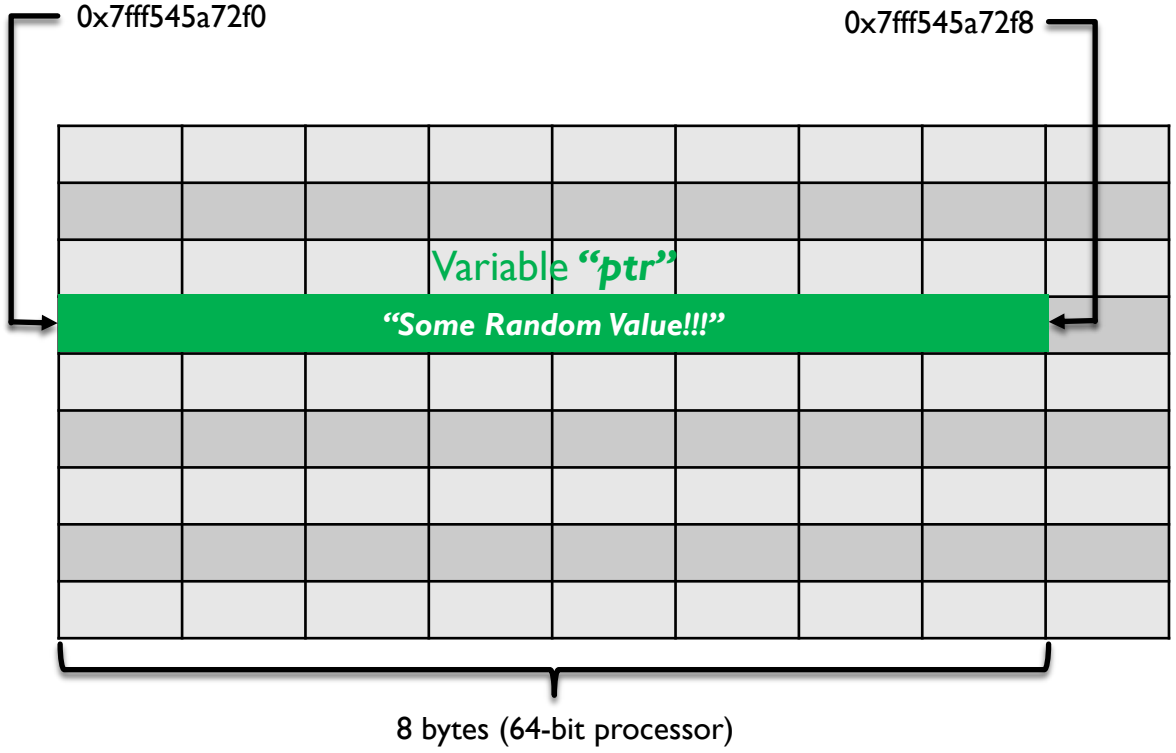
  *int *ptr;*

  - The **asterisk (*)** symbol means that the variable ***"ptr"*** is a ***pointer***.
  - The **type (*int*)** specifies what ***type of variable*** the ***pointer is storing the address of***. Must be a valid C data type/user defined data type (ex – struct).
  - The **actual data type** of the value of all pointers, whether integer, float, character, or otherwise, is the same, ***a long hexadecimal number*** that represents a ***memory address***.

- Example:

```
int *ptr;
printf (" The address of ptr = %p\n", &ptr);
```

```
The address of ptr = 0x7fff545a72f0
```



POINTER OPERATIONS

- **Initialization**

  ptr = &i;

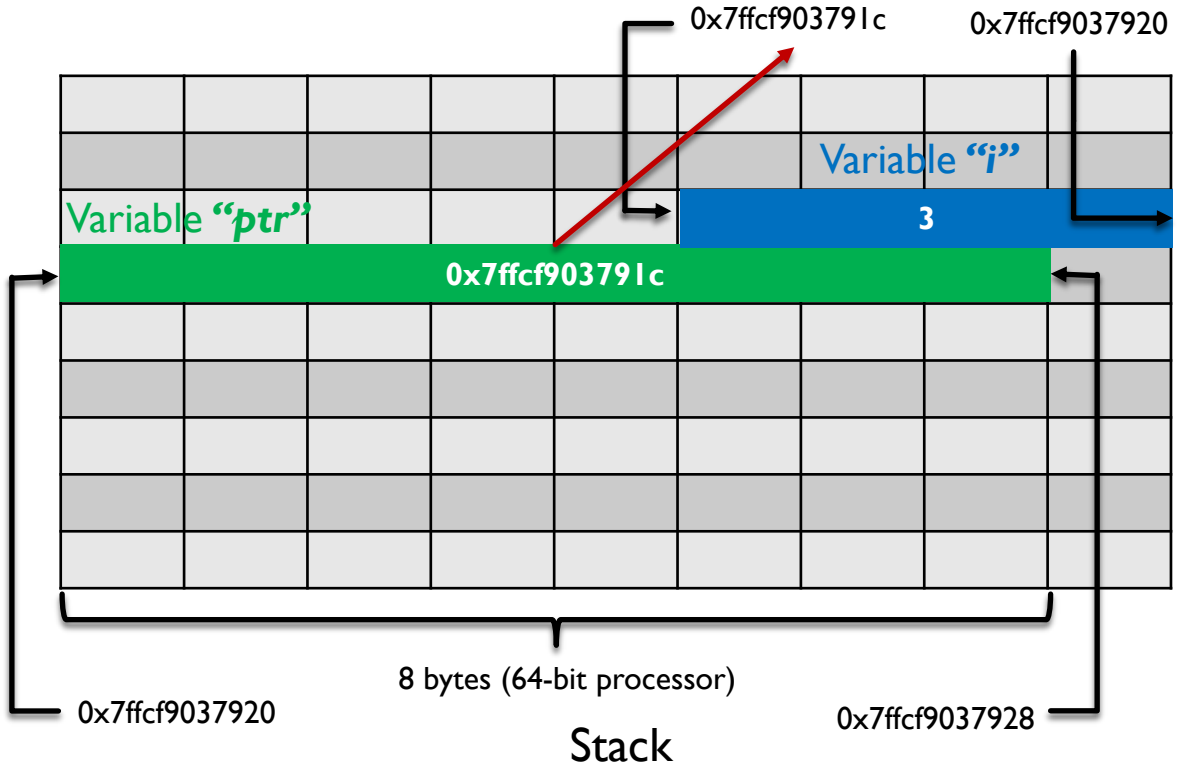
  - Unlike a simple variable that stores a value, a **pointer** must be ***initialized*** with a specified ***address*** prior to its use.

- Example:

```
int *ptr;
ptr = &i;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The address of ptr = %p\n", &ptr);
```

```
The address of i = 0x7ffcf903791c
```

```
The value of ptr / address ptr is pointing to = 0x7ffcf903791c
The address of ptr = 0x7ffcf9037920
```

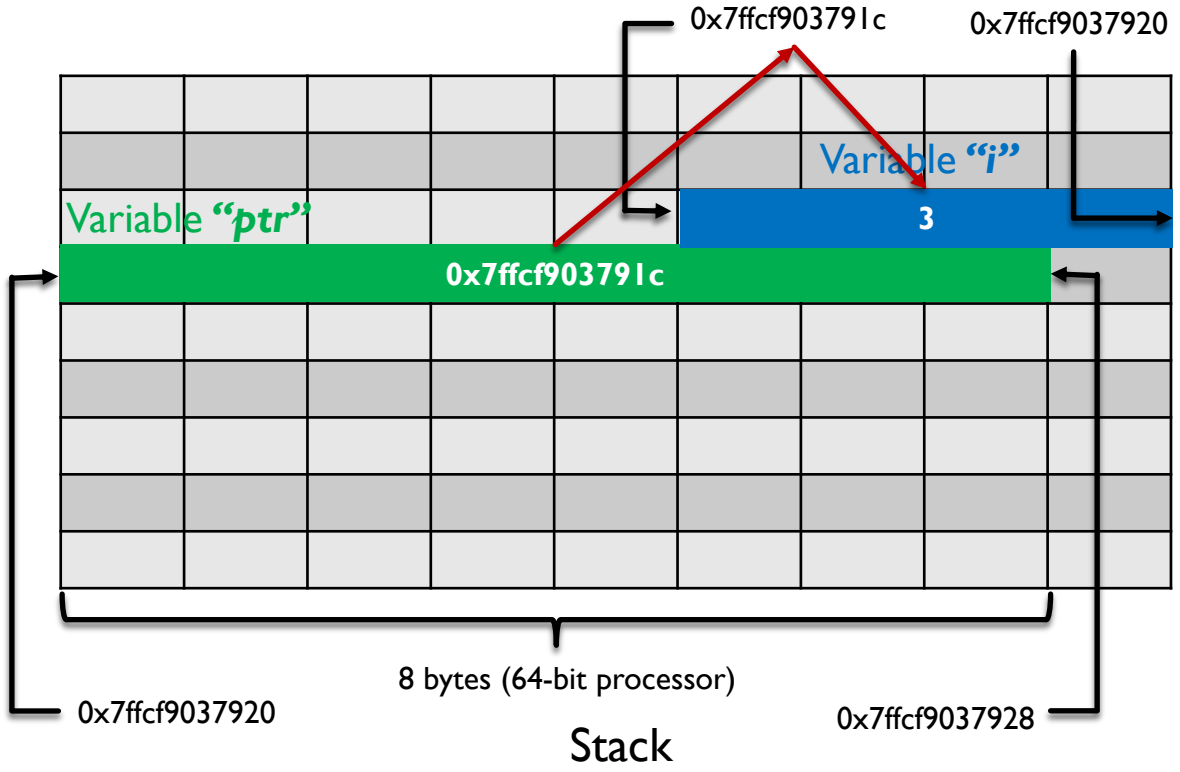


POINTER OPERATIONS

- **Dereferencing - *New for Pointers!***

  **ptr;*

  - The primary use of a **pointer** is to ***access and change*** the ***value of the variable*** that the pointer points to.
  - **Value of the variable** is represented by ***preceding the pointer variable identifier by an asterisk (*)*** sign which literally means ***'value at address'***.
  - The ***'value at address'*** operator is also called **indirection operator** or **dereference operator**.



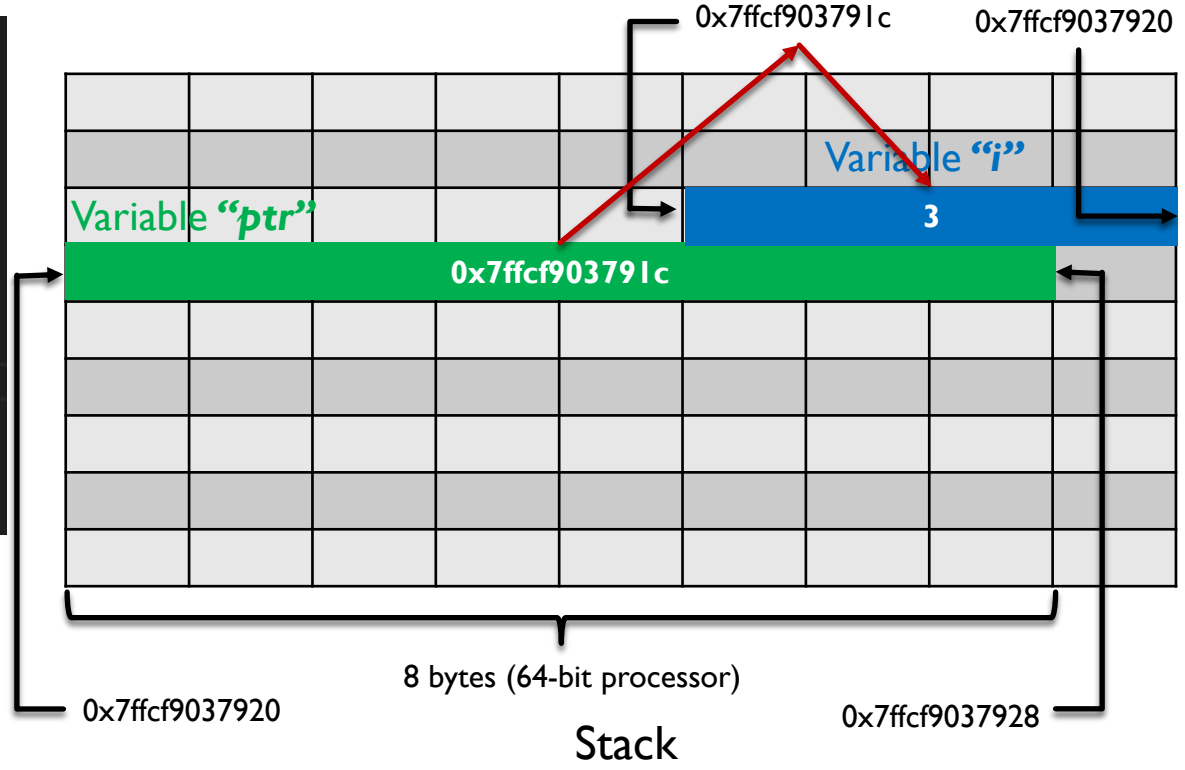
POINTER OPERATIONS

- Example:

```
int i;
i = 3;
printf ("====================================\n");
printf (" The value of i = %d\n", i);
printf (" The address of i = %p\n", &i);
printf ("====================================\n");

int *ptr;
ptr = &i;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The address of ptr = %p\n", &ptr);
printf (" The value stored in the address pointed by ptr = %d\n", *ptr);
printf ("====================================\n");
```

```
====================================
 The value of i = 3
 The address of i = 0x7ffcf903791c
====================================
 The value of ptr / address ptr is pointing to = 0x7ffcf903791c
 The address of ptr = 0x7ffcf9037920
 The value stored in the address pointed by ptr = 3
====================================
```

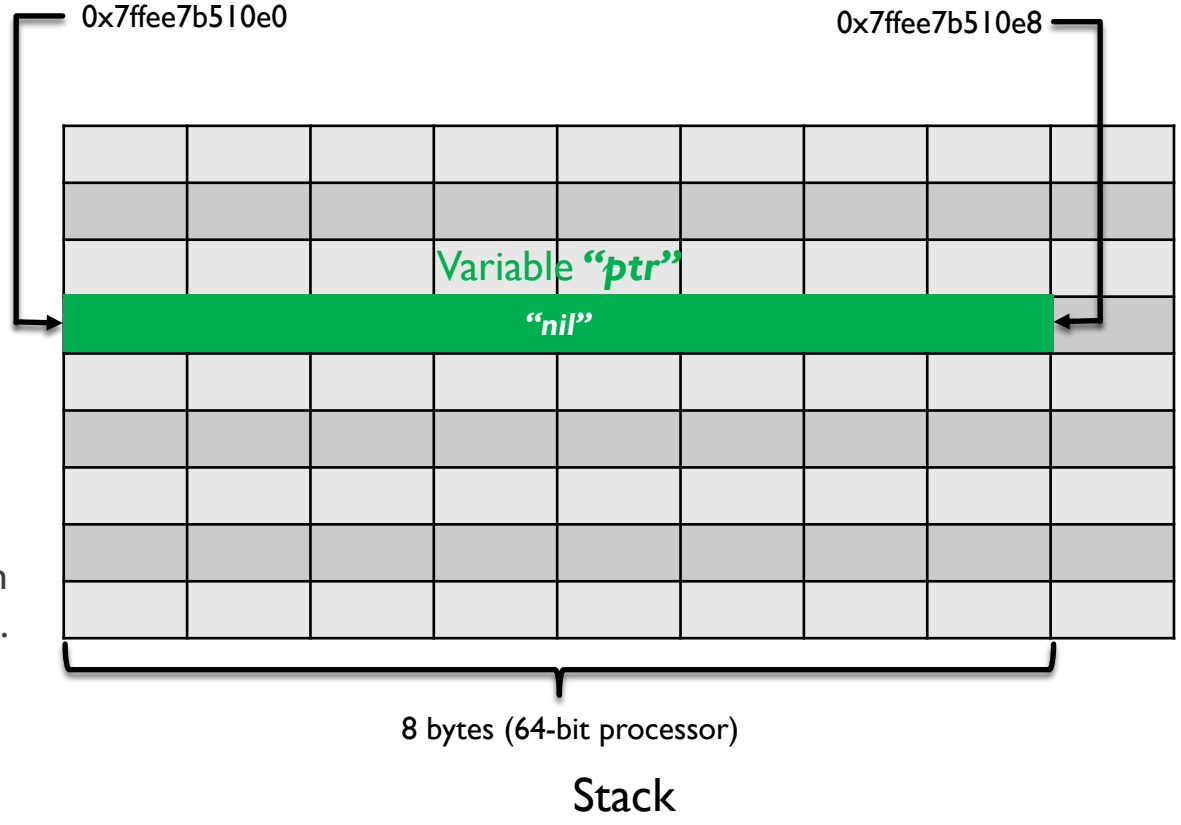


NULL POINTER

- **Null Pointer**

  *int *ptr = NULL;*

  - It is a special pointer value that points to **nowhere**.
  - Value stored is ***"nil"*.**
- It's a good practice to use **NULL pointer** when you don't want to initialize your pointer right away.
  - Makes sure your pointer isn't pointing to some garbage address.
  - Helps to check if a pointer hasn't been initialized yet – we can check for **NULL pointer** before accessing any pointer value. Returns ***"Segmentation Fault"*** if NULL.
- Allows to pass a **Null Pointer** to a function argument when we don't want to pass any valid memory address.

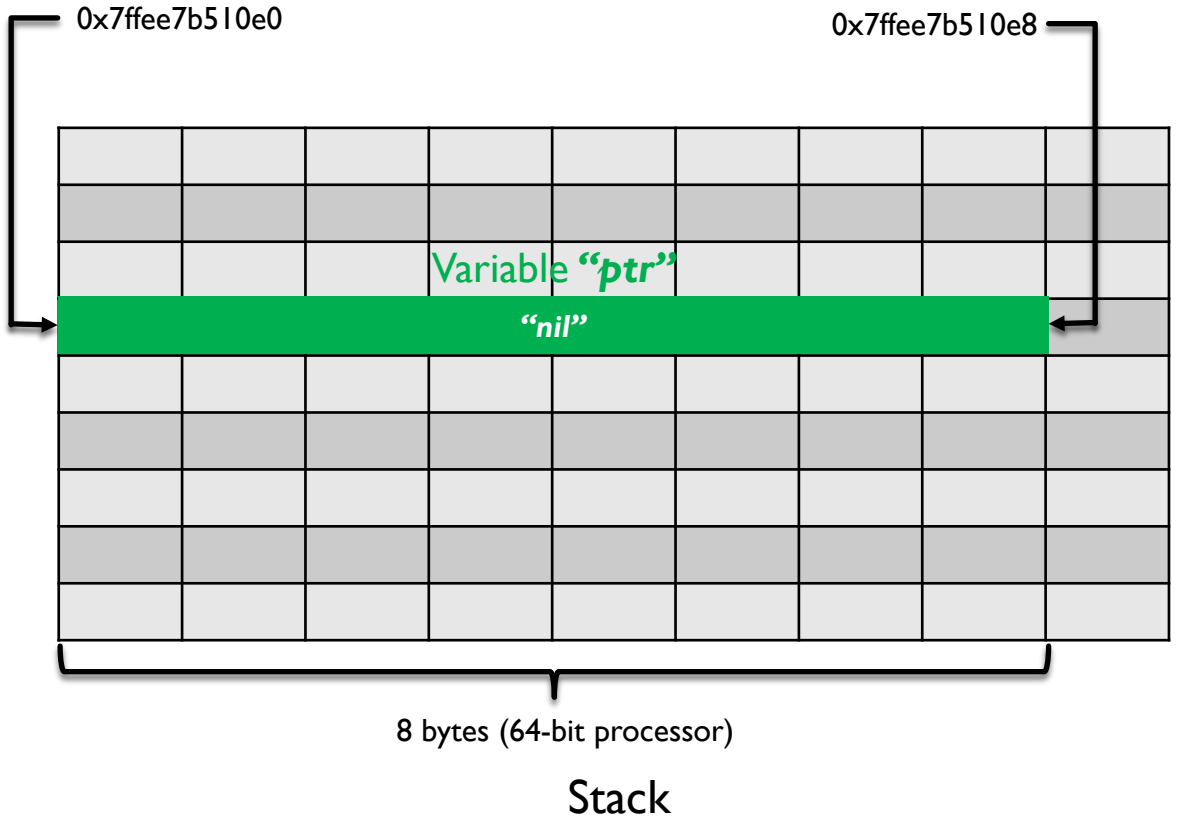


NULL POINTER

- Example:

```
int *ptr;
// ptr = &i;
ptr = NULL;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The address of ptr = %p\n", &ptr);
printf (" The value stored in the address pointed by ptr = %d\n", *ptr);
```

```
 The value of ptr / address ptr is pointing to = (nil)
 The address of ptr = 0x7ffee7b510e0
Segmentation fault
```



MEMORY ALLOCATION

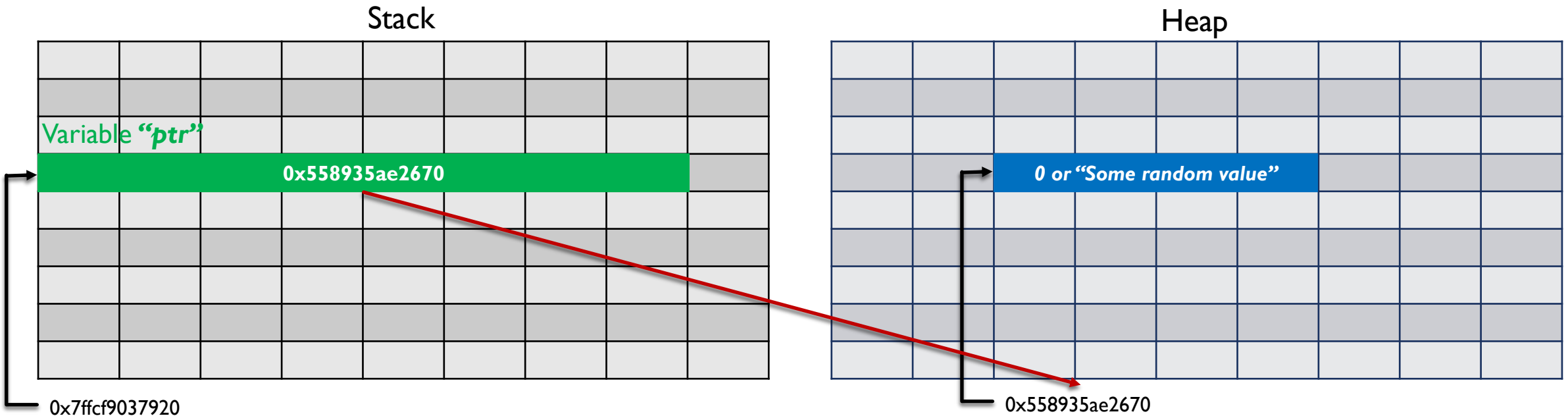
- **Malloc()** – Allocates memory space in the **Heap**.

  *void *malloc(size_t size);*

  - Returns the ***start address*** of a memory block of ***"size"*** bytes.
  - It's important to ***initialize malloc***, otherwise we could get zeroes or some garbage values (from previous use).

```
ptr = malloc(sizeof(int));
*ptr = 5;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The value stored in the address pointed by ptr = %d\n", *ptr);
```

```
The value of ptr / address ptr is pointing to = 0x558935ae2670
The value stored in the address pointed by ptr = 5
```



MEMORY ALLOCATION

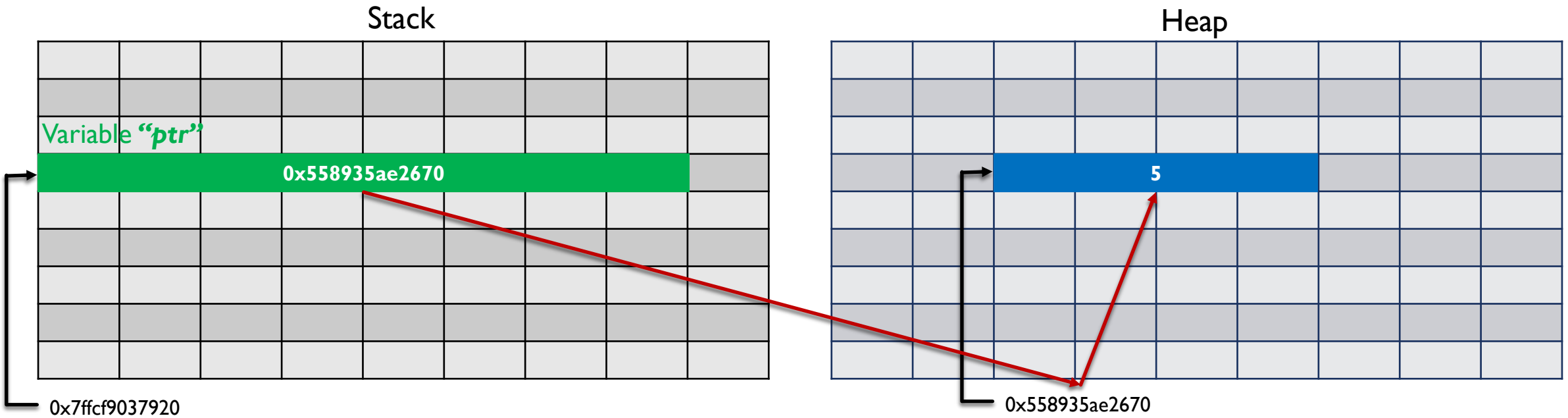
- **Malloc()** – Allocates memory space in the **Heap**.

  *void *malloc(size_t size);*

  - Returns the ***start address*** of a memory block of ***"size"*** bytes.
  - It's important to ***initialize malloc***, otherwise we could get zeroes or some garbage values (from previous use).

```
ptr = malloc(sizeof(int));
*ptr = 5;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The value stored in the address pointed by ptr = %d\n", *ptr);
```

```
The value of ptr / address ptr is pointing to = 0x558935ae2670
The value stored in the address pointed by ptr = 5
```

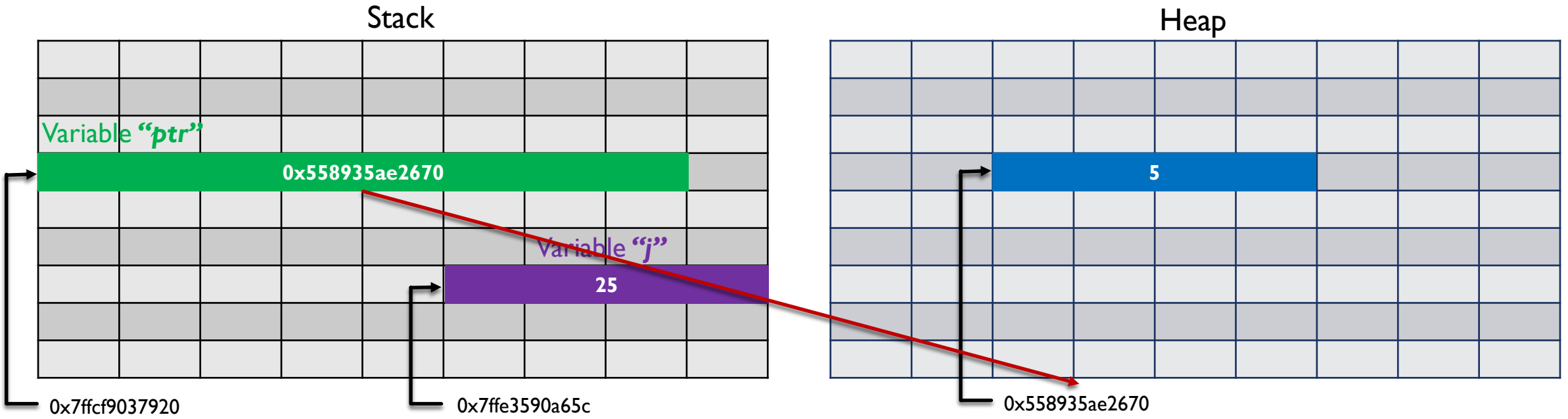


MEMORY LEAK

- Dynamically allocated memory created with malloc() doesn't get ***freed*** on their own.
- For instance, you change your pointer variable to point to a new address, the previous memory block ***can't be reached anymore*** nor ***be reused*** – **Memory Leak!**

```
int j = 25;
ptr = &j;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The value stored in the address pointed by ptr = %d\n", *ptr);
```

```
The value of ptr / address ptr is pointing to = 0x7ffe3590a65c
The value stored in the address pointed by ptr = 25
```

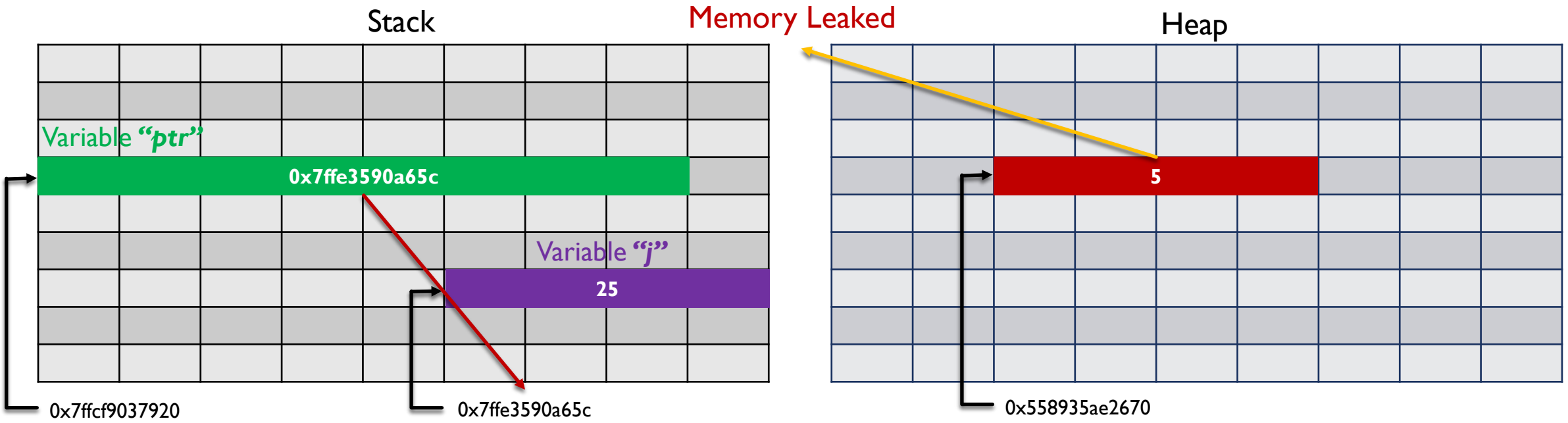


MEMORY LEAK

- Dynamically allocated memory created with malloc() doesn't get ***freed*** on their own.
- For instance, you change your pointer variable to point to a new address, the previous memory block ***can't be reached anymore*** nor ***be reused*** – **Memory Leak!**

```
int j = 25;
ptr = &j;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf (" The value stored in the address pointed by ptr = %d\n", *ptr);
```

```
The value of ptr / address ptr is pointing to = 0x7ffe3590a65c
The value stored in the address pointed by ptr = 25
```



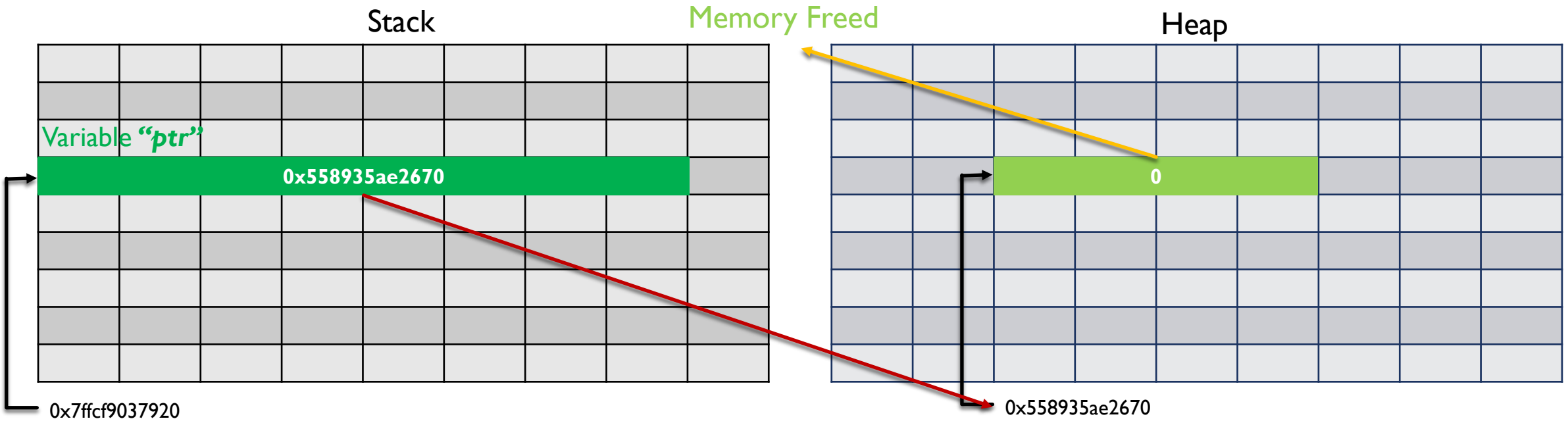
MEMORY LEAK

- To resolve memory leaks, you need to ***manually free*** the memory ***before*** changing the pointer to the new address.
- **Free()** – Releases the memory block specified by address.

  *void free(void *address);*

  - Allows the memory to be ***re-used*** (ex – malloc()).

```
free(ptr);
int j = 25;
ptr = &j;
```

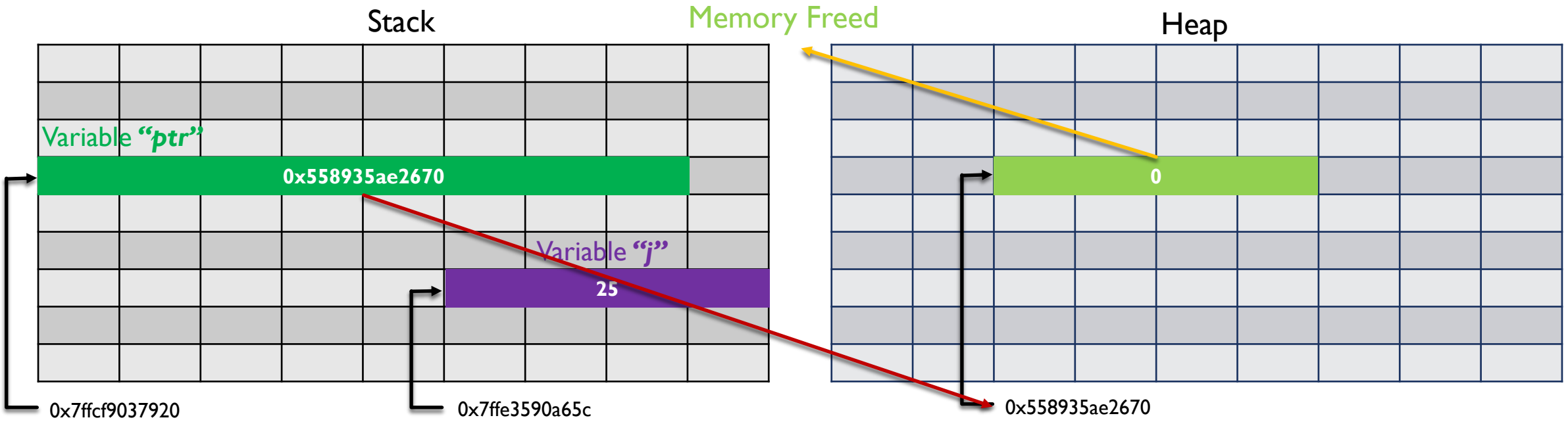


MEMORY LEAK

- To resolve memory leaks, you need to ***manually free*** the memory ***before*** changing the pointer to the new address.
- **Free()** – Releases the memory block specified by address.

  *void free(void *address);*
  - Allows the memory to be ***re-used*** (ex – malloc()).

```
free(ptr);
int j = 25;
ptr = &j;
```

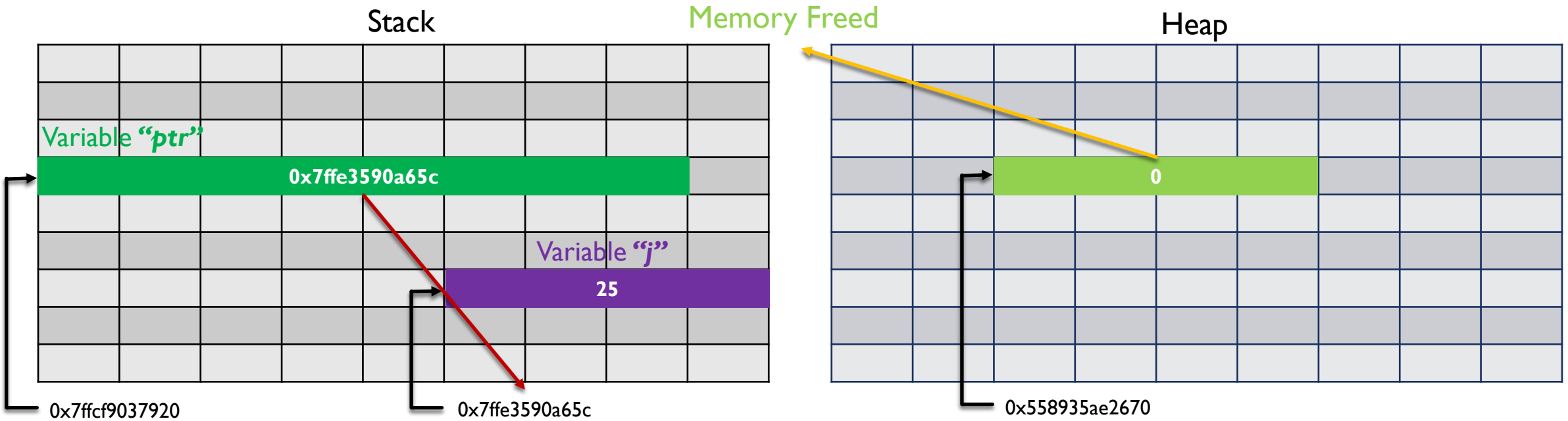


MEMORY LEAK

- To resolve memory leaks, you need to ***manually free*** the memory ***before*** changing the pointer to the new address.
- **Free()** – Releases the memory block specified by address.

  *void free(void *address);*
  - Allows the memory to be ***re-used*** (ex – malloc()).

```
free(ptr);
int j = 25;
ptr = &j;
```

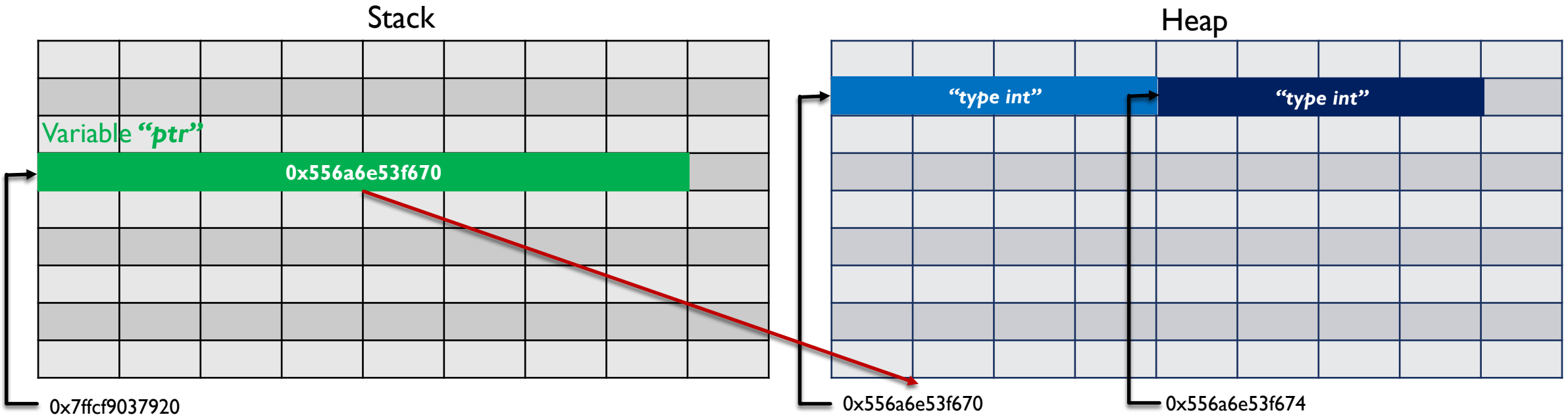


POINTER ARITHMETIC

- **Arithmetic operations** on pointers – **(++, --, +, -)**.
- For instance, **ptr++** will move the ***ptr*** to the next element location.
- **ptr++** is equivalent to ***"ptr + sizeof(ptr type)"***.
- Doesn't affect the value stored in the location.

```
//  Pointer Arithmetic
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
ptr++;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf ("=====================================\n");
```

```
The value of ptr / address ptr is pointing to = 0x556a6e53f670
The value of ptr / address ptr is pointing to = 0x556a6e53f674
```

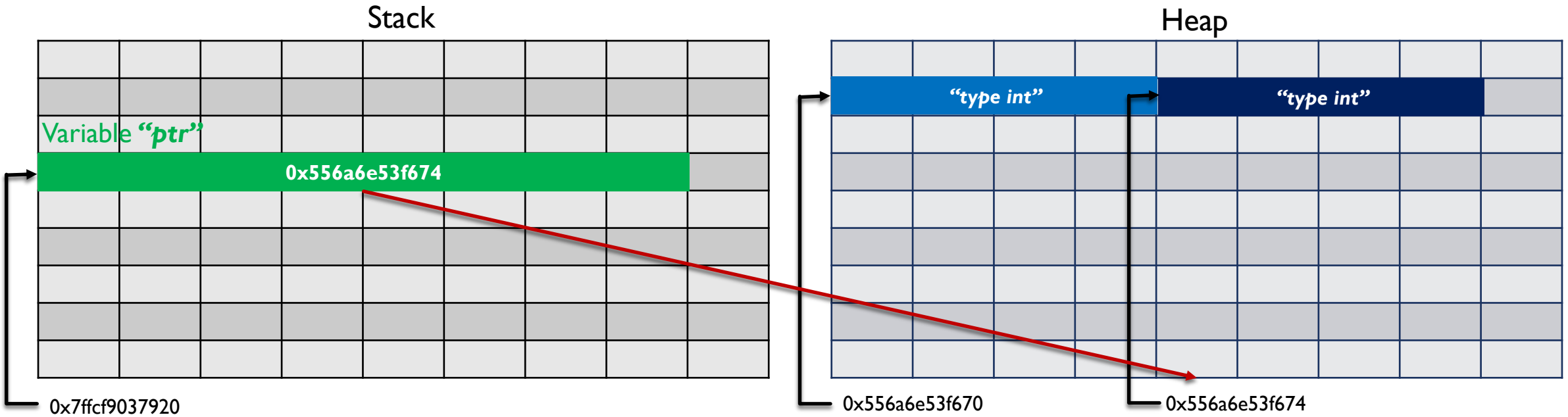


POINTER ARITHMETIC

- **Arithmetic operations** on pointers – **(++, --, +, -)**.
- For instance, **ptr++** will move the ***ptr*** to the next element location.
- **ptr++** is equivalent to ***"ptr + sizeof(ptr type)"***.
- Doesn't affect the value stored in the location.

```
//  Pointer Arithmetic
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
ptr++;
printf (" The value of ptr / address ptr is pointing to = %p\n", ptr);
printf ("=====================================\n");
```

```
The value of ptr / address ptr is pointing to = 0x556a6e53f670
The value of ptr / address ptr is pointing to = 0x556a6e53f674
```



EXAMPLE USAGE – LINKED LISTS

- **Define the "Node" type for linked lists**

```c
typedef struct node
{
    int data;
    struct node *next;
} node_t;
```

Previous data type (***struct node***)

New data type (***node_t***)

Node Structure (***node_t***)

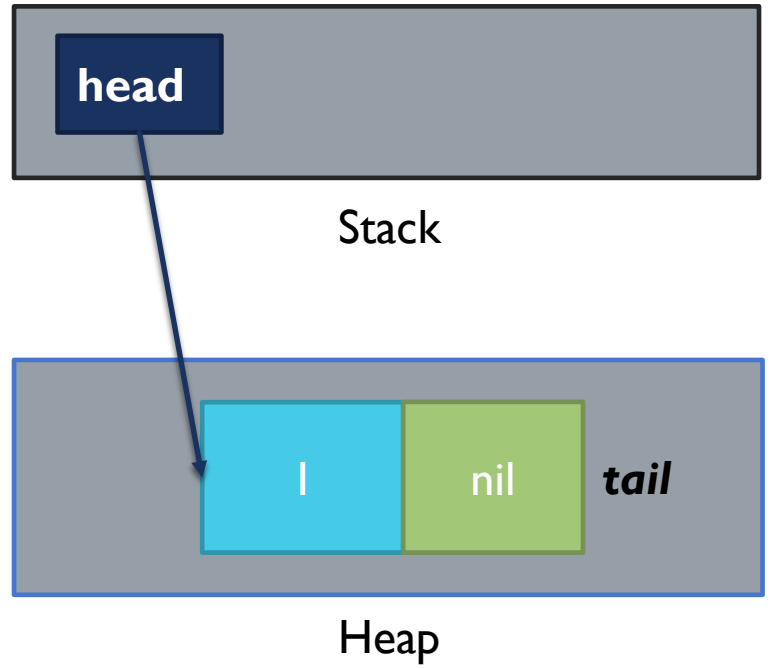
EXAMPLE USAGE – LINKED LISTS

- **Create a node**

```
{
    //  Creating a node named head
    node_t *head = NULL;
    head = malloc(sizeof(node_t));  //   Allocates memory for the node from Heap
    if (head == NULL)   //  Validates memory allocation
    {
        return 1;
    }

    //  Assign values to the "node_t" members
    //  Use "->" instead of "." for pointers (head is a pointer)
    head->data = 1;
    head->next = NULL;
    return 0;
}
```

Put "NULL" in the "next" of the new node since it's also the last node – ***tail***

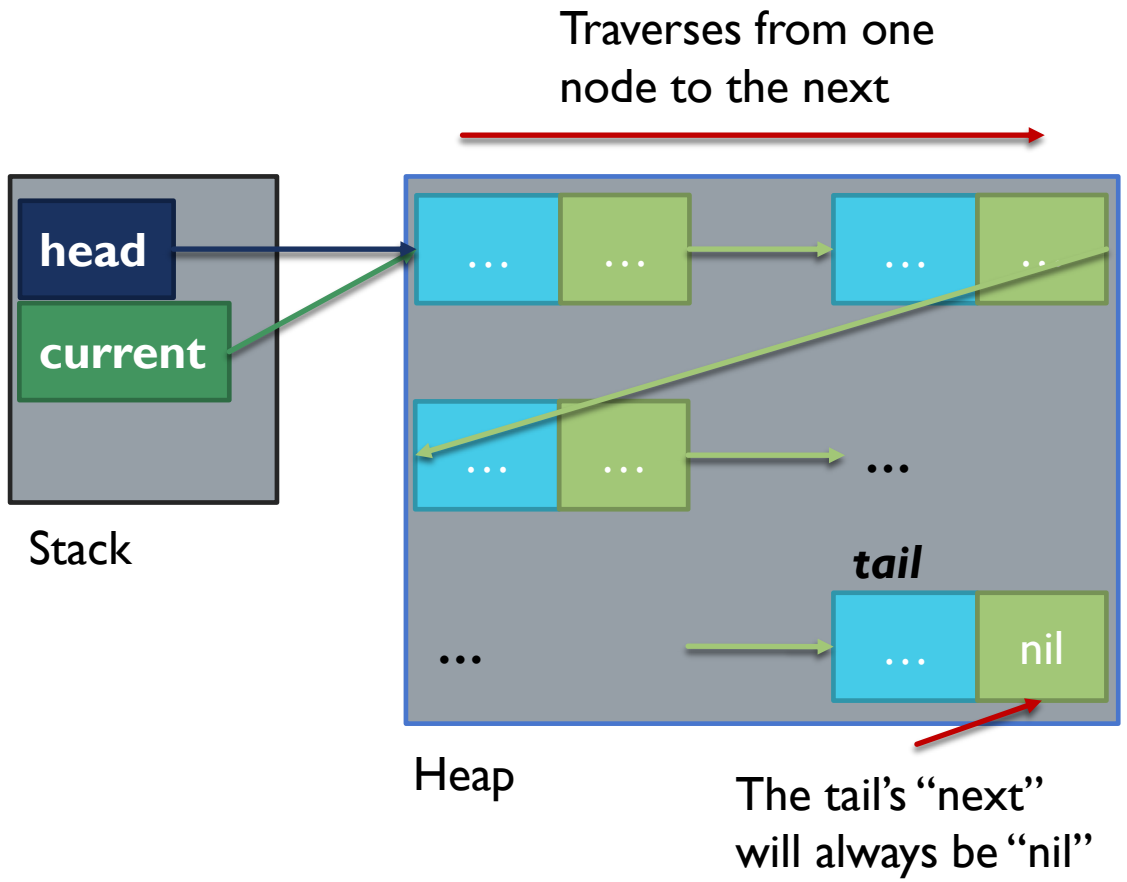


EXAMPLE USAGE – LINKED LISTS

- **Traverse the list and print its contents**

```c
/*
Interates through the list and prints the values of each node
*/
void print_list(node_t *head)
{
    node_t *current = head;

    while (current != NULL)
    {
        printf("%d\n", current->data);
        current = current->next;
    }
}
```

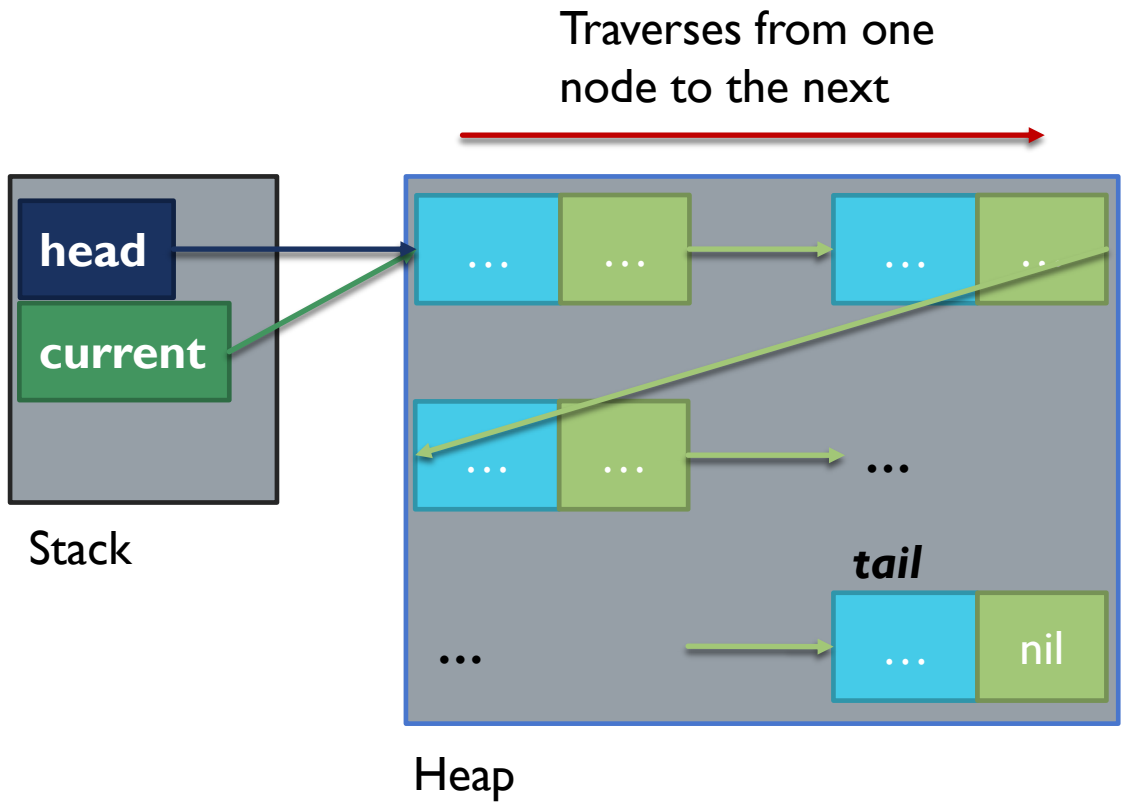


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the end of the list**

```c
/*
Append an item at the end of the Linked List
*/
void append(node_t *head, int val)
{
    node_t *current = head;
    //  Finds the end of the list
    while (current->next != NULL)
    {
        current = current->next;
    }

    // Now we can add a new variable
    current->next = (node_t *)malloc(sizeof(node_t));
    current->next->data = val;
    current->next->next = NULL;
}
```

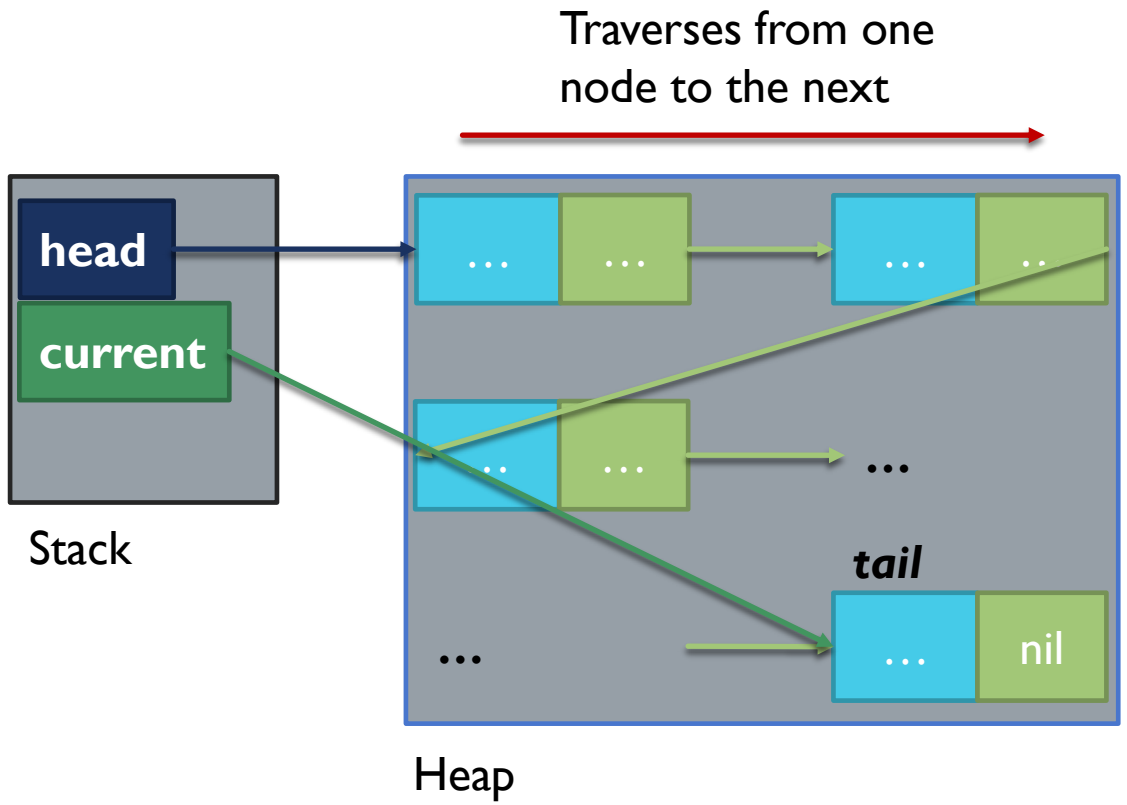


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the end of the list**

```c
/*
Append an item at the end of the Linked List
*/
void append(node_t *head, int val)
{
    node_t *current = head;
    //  Finds the end of the list
    while (current->next != NULL)
    {
        current = current->next;
    }

    // Now we can add a new variable
    current->next = (node_t *)malloc(sizeof(node_t));
    current->next->data = val;
    current->next->next = NULL;
}
```

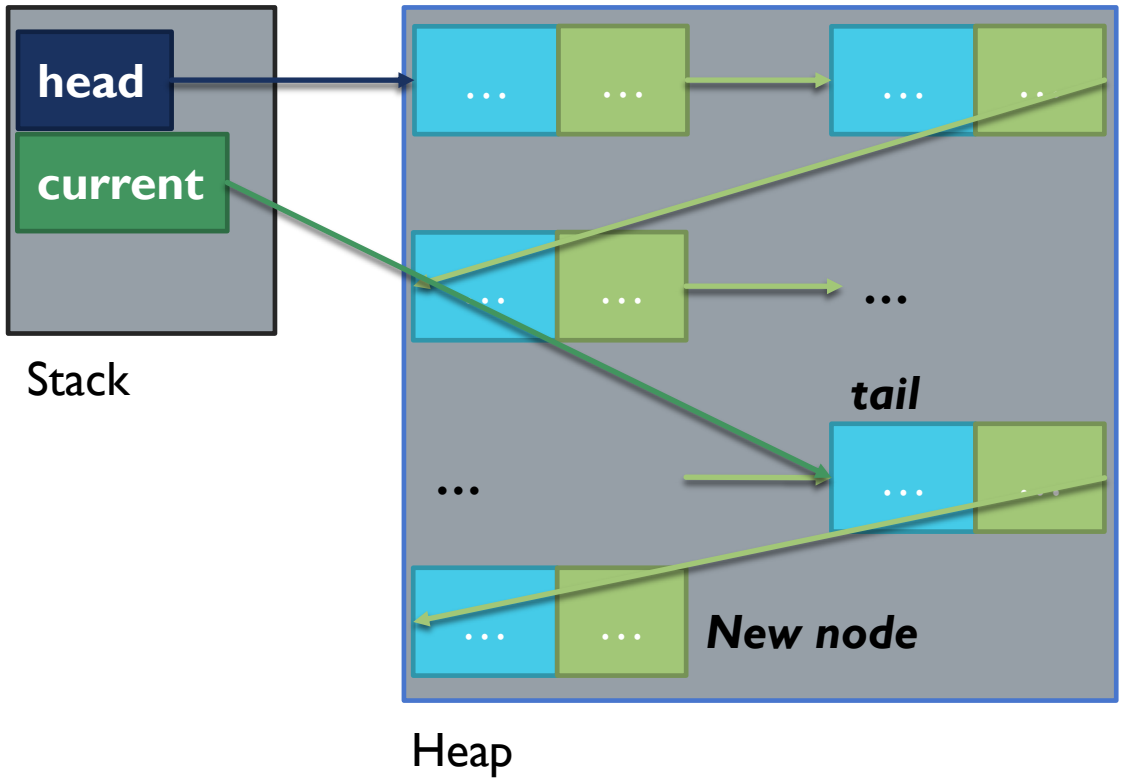


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the end of the list**

```c
/*
Append an item at the end of the Linked List
*/
void append(node_t *head, int val)
{
    node_t *current = head;
    //  Finds the end of the list
    while (current->next != NULL)
    {
        current = current->next;
    }

    // Now we can add a new variable
    current->next = (node_t *)malloc(sizeof(node_t));
    current->next->data = val;
    current->next->next = NULL;
}
```

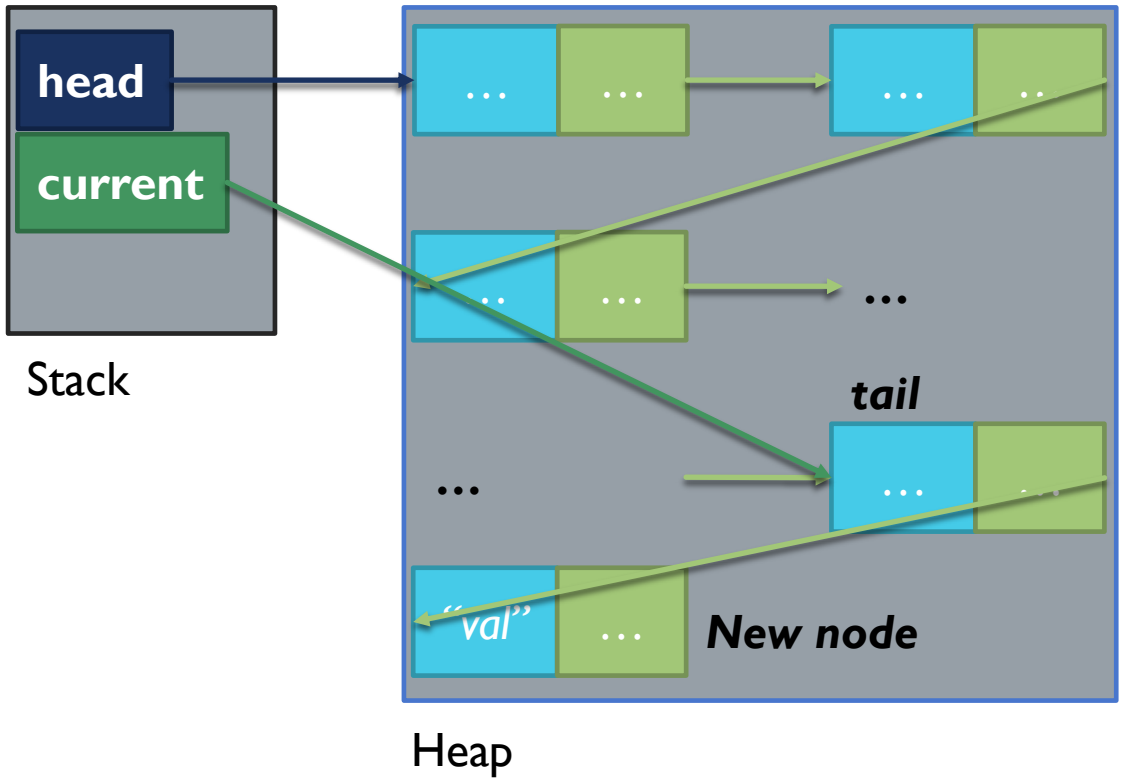


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the end of the list**

```c
/*
Append an item at the end of the Linked List
*/
void append(node_t *head, int val)
{
    node_t *current = head;
    //  Finds the end of the list
    while (current->next != NULL)
    {
        current = current->next;
    }

    // Now we can add a new variable
    current->next = (node_t *)malloc(sizeof(node_t));
    current->next->data = val;
    current->next->next = NULL;
}
```

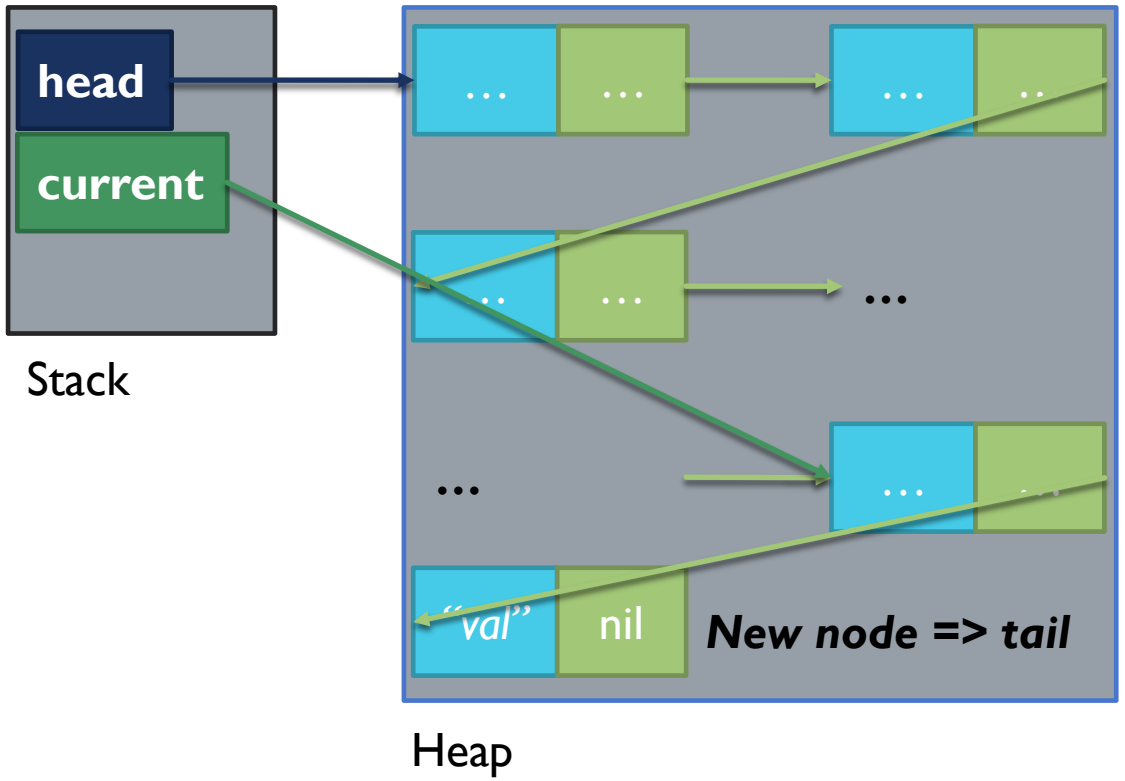


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the end of the list**

```c
/*
Append an item at the end of the Linked List
*/
void append(node_t *head, int val)
{
    node_t *current = head;
    //  Finds the end of the list
    while (current->next != NULL)
    {
        current = current->next;
    }

    // Now we can add a new variable
    current->next = (node_t *)malloc(sizeof(node_t));
    current->next->data = val;
    current->next->next = NULL;
}
```

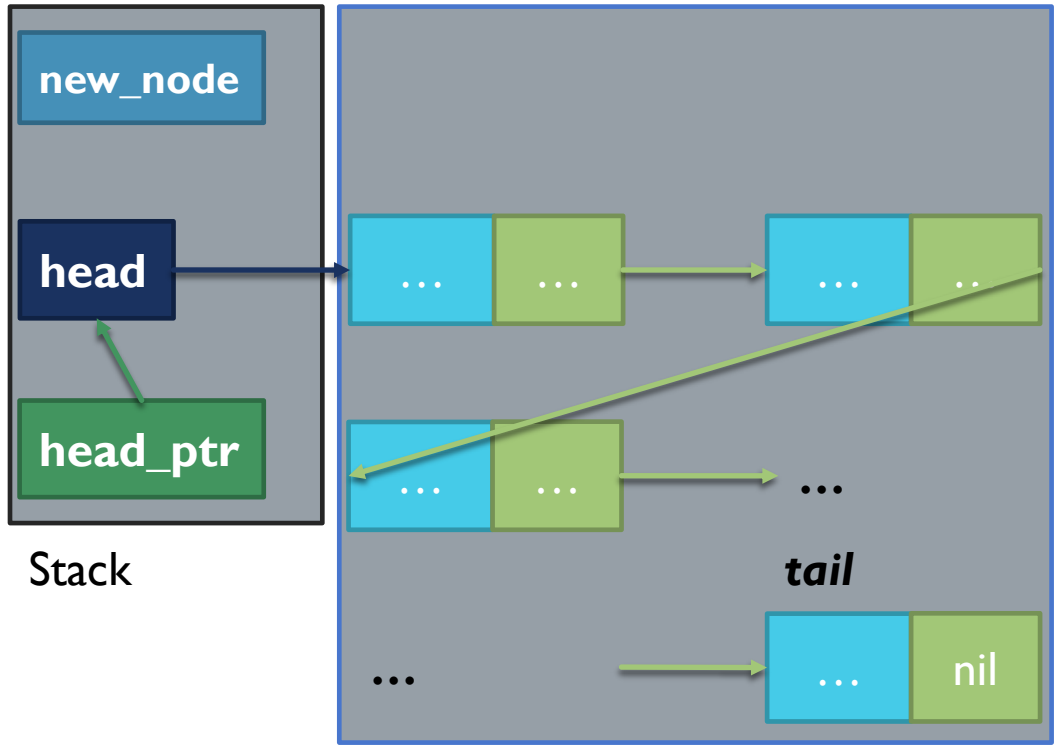


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the beginning of the list**

```c
/*
Append an item at the beginning of the Linked List
We need to change where the head points to, so we need a pointer to a pointer!!!
*/
void prepend(node_t **head_ptr, int val)
{
    node_t *new_node;
    new_node = malloc(sizeof(node_t));

    new_node->data = val;
    new_node->next = *head_ptr; //  Dereference head_ptr so RHS is just the head
    *head_ptr = new_node;    // Now the head needs to point to the new node
}
```

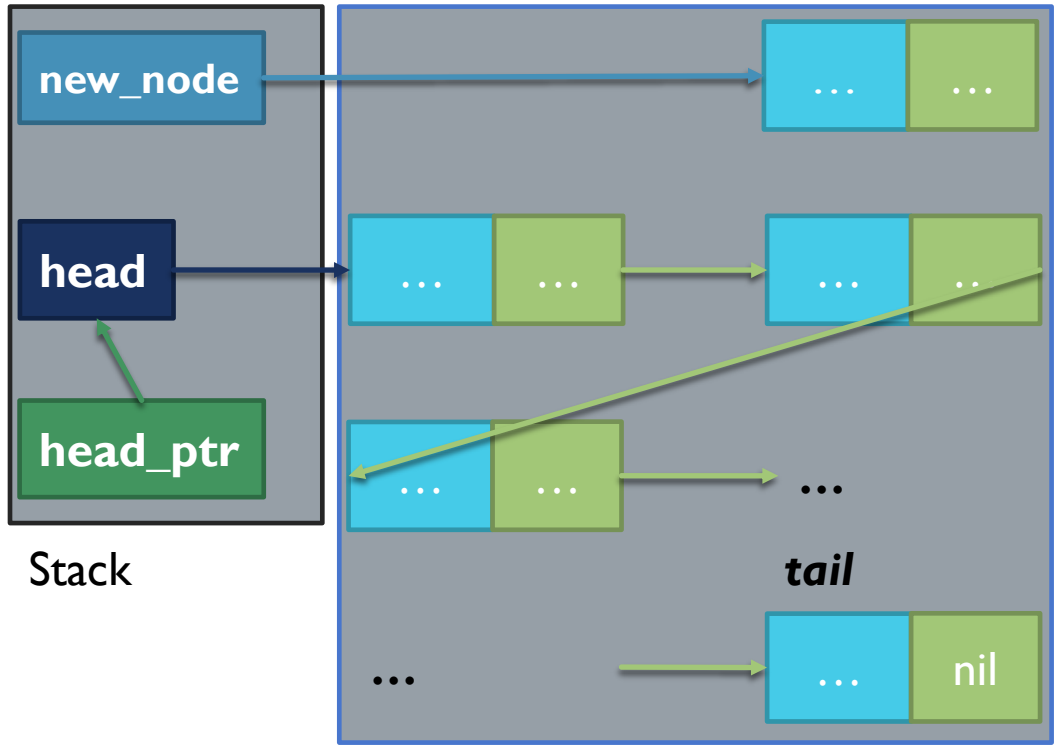


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the beginning of the list**

```c
/*
Append an item at the beginning of the Linked List
We need to change where the head points to, so we need a pointer to a pointer!!!
*/
void prepend(node_t **head_ptr, int val)
{
    node_t *new_node;
    new_node = malloc(sizeof(node_t));

    new_node->data = val;
    new_node->next = *head_ptr; //  Dereference head_ptr so RHS is just the head
    *head_ptr = new_node;    // Now the head needs to point to the new node
}
```

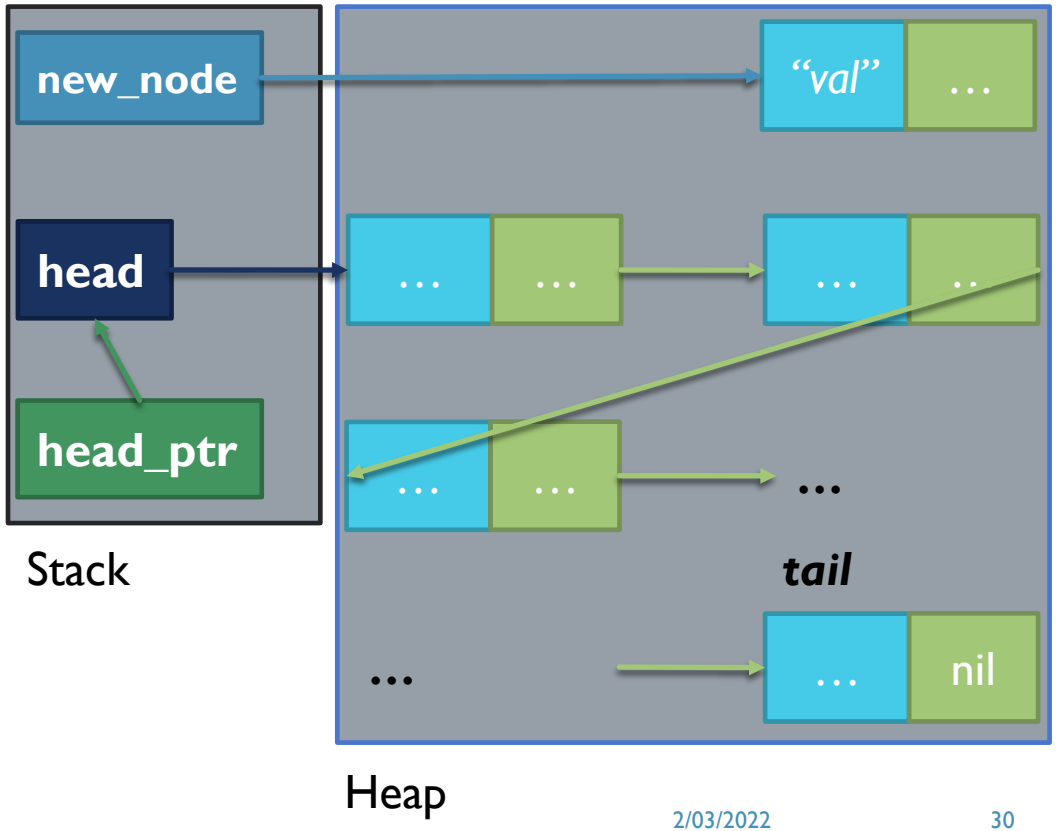


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the beginning of the list**

```c
/*
Append an item at the beginning of the Linked List
We need to change where the head points to, so we need a pointer to a pointer!!!
*/
void prepend(node_t **head_ptr, int val)
{
    node_t *new_node;
    new_node = malloc(sizeof(node_t));

    new_node->data = val;
    new_node->next = *head_ptr; //  Dereference head_ptr so RHS is just the head
    *head_ptr = new_node;    // Now the head needs to point to the new node
}
```

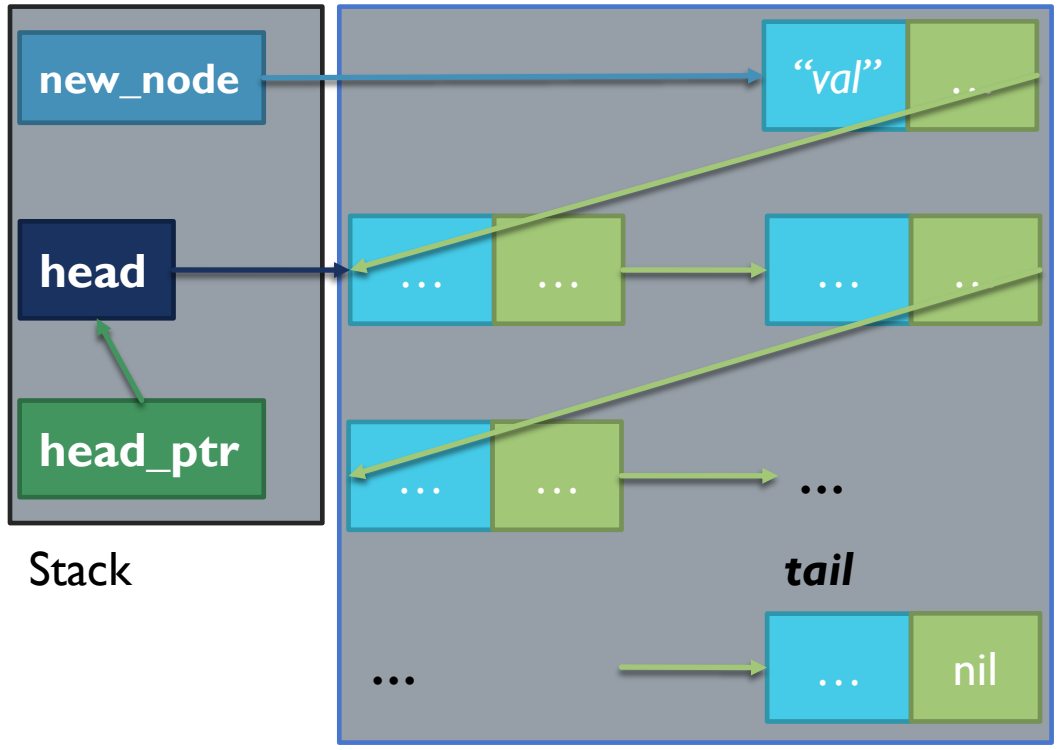


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the beginning of the list**

```c
/*
Append an item at the beginning of the Linked List
We need to change where the head points to, so we need a pointer to a pointer!!!
*/
void prepend(node_t **head_ptr, int val)
{
    node_t *new_node;
    new_node = malloc(sizeof(node_t));

    new_node->data = val;
    new_node->next = *head_ptr; // Dereference head_ptr so RHS is just the head
    *head_ptr = new_node;    // Now the head needs to point to the new node
}
```

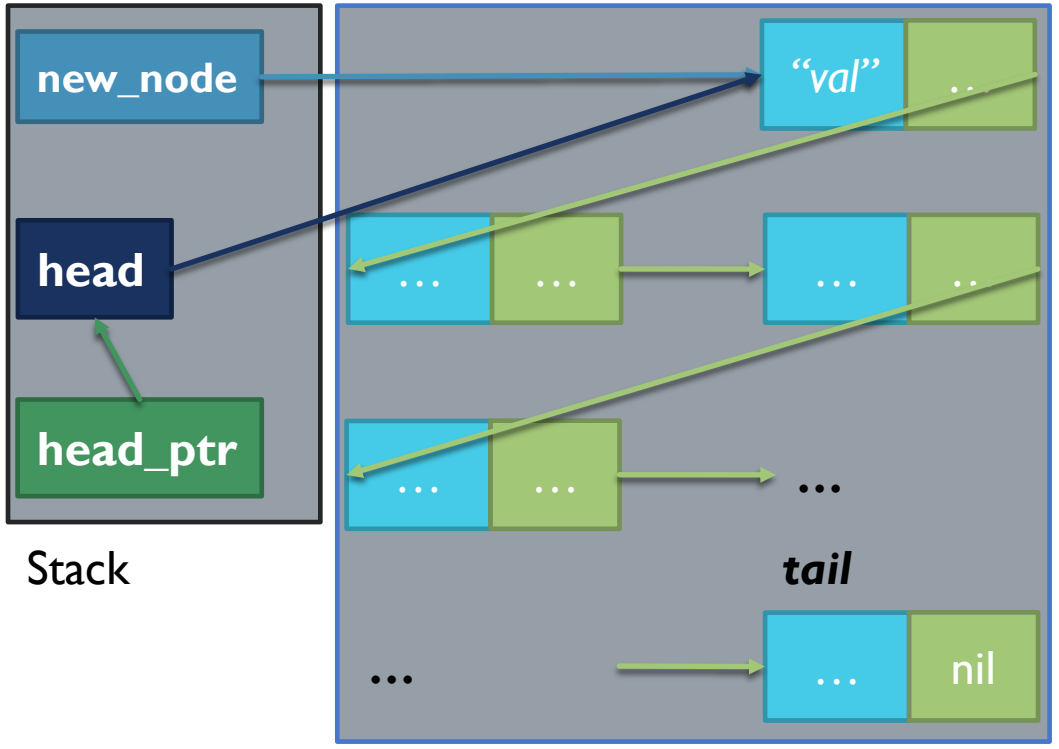


EXAMPLE USAGE – LINKED LISTS

- **Append an element at the beginning of the list**

```c
/*
Append an item at the beginning of the Linked List
We need to change where the head points to, so we need a pointer to a pointer!!!
*/
void prepend(node_t **head_ptr, int val)
{
    node_t *new_node;
    new_node = malloc(sizeof(node_t));

    new_node->data = val;
    new_node->next = *head_ptr; //  Dereference head_ptr so RHS is just the head
    *head_ptr = new_node; // Now the head needs to point to the new node
}
```



EXAMPLE USAGE – LINKED LISTS

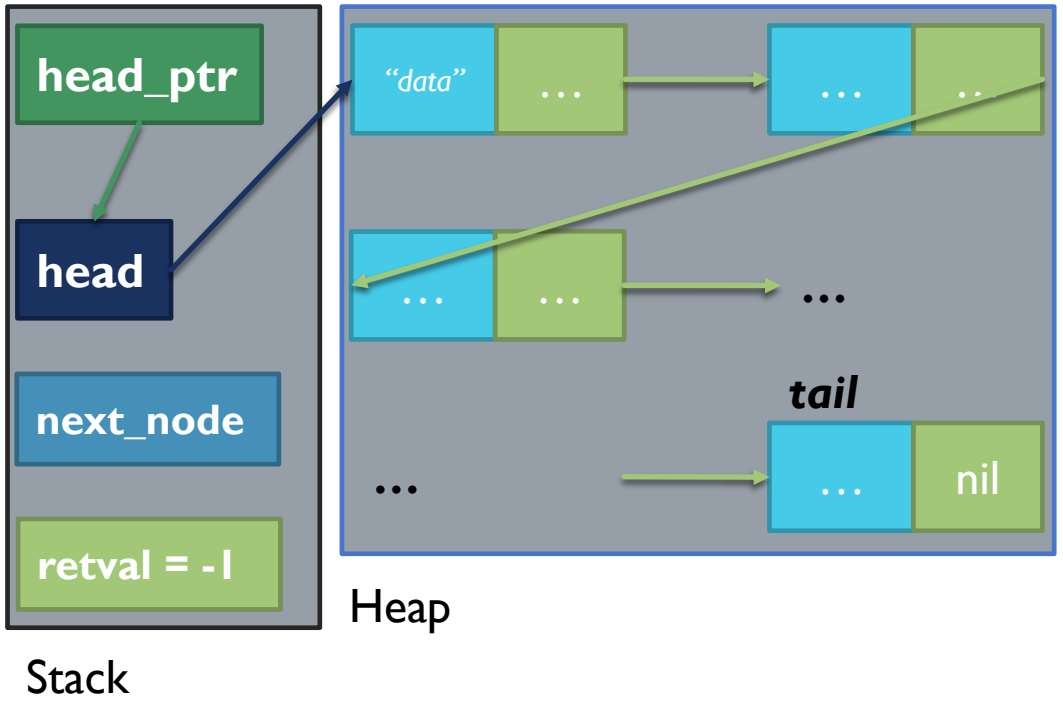
- **Remove first element of the list**

```c
/*
Removes the Head node (remember to keep track of head)
*/
int pop(node_t **head_ptr)
{
    int retval = -1;
    node_t *next_node = NULL;

    //  Checks if the Linked List only has 1 node (head itself)
    if (*head_ptr == NULL)
    {
        return -1;
    }

    next_node = (*head_ptr)->next;
    retval = (*head_ptr)->data; //  Retrives the data in the node being removed
    free(*head_ptr);    //  Remember to free the memory pointed by the initial head
    *head_ptr = next_node;  //  Assign the new node as the head

    return retval;
}
```



EXAMPLE USAGE – LINKED LISTS

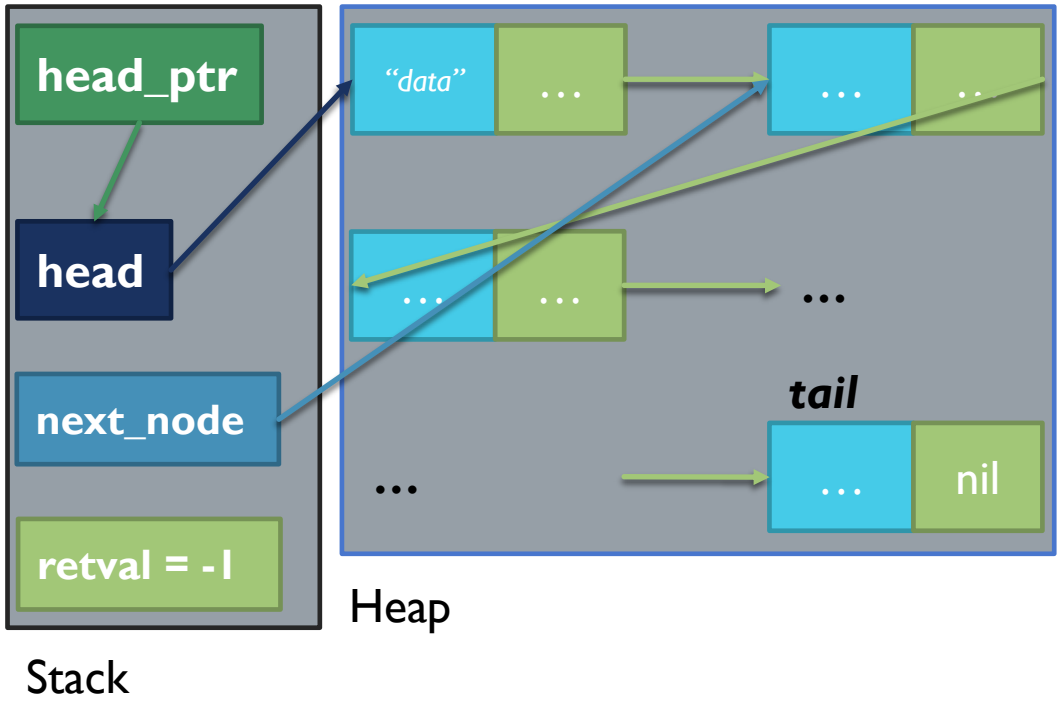
- **Remove first element of the list**

```c
/*
Removes the Head node (remember to keep track of head)
*/
int pop(node_t **head_ptr)
{
    int retval = -1;
    node_t *next_node = NULL;

    //  Checks if the Linked List only has 1 node (head itself)
    if (*head_ptr == NULL)
    {
        return -1;
    }

    next_node = (*head_ptr)->next;
    retval = (*head_ptr)->data; //  Retrives the data in the node being removed
    free(*head_ptr);    //  Remember to free the memory pointed by the initial head
    *head_ptr = next_node;  //  Assign the new node as the head

    return retval;
}
```



EXAMPLE USAGE – LINKED LISTS

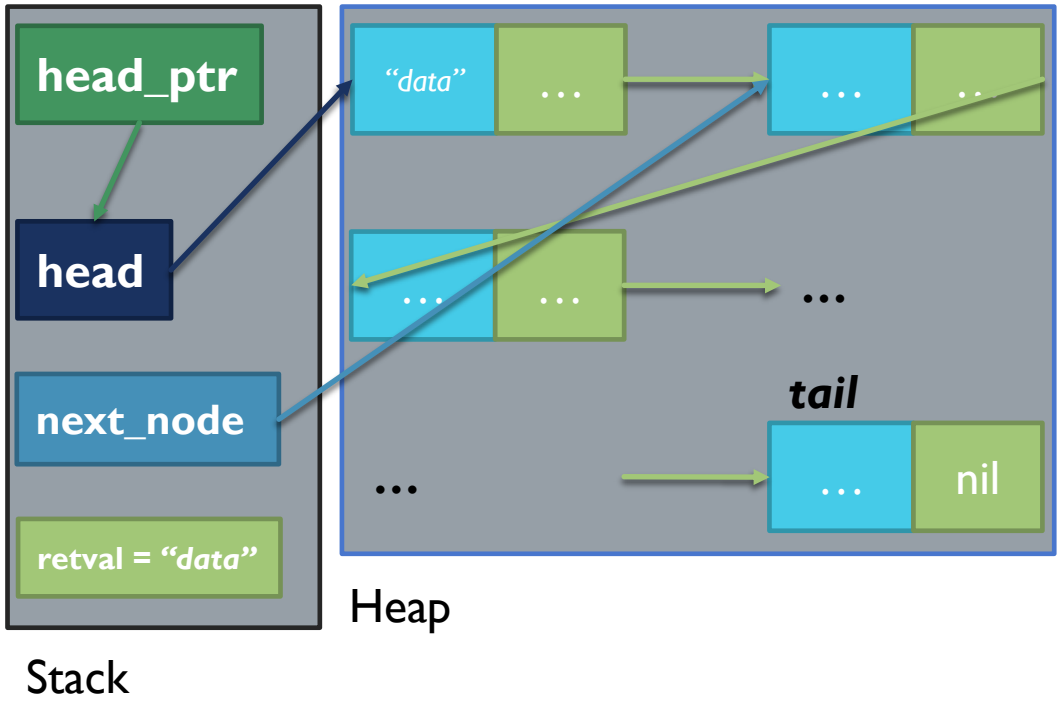
- **Remove first element of the list**

```c
/*
Removes the Head node (remember to keep track of head)
*/
int pop(node_t **head_ptr)
{
    int retval = -1;
    node_t *next_node = NULL;

    //  Checks if the Linked List only has 1 node (head itself)
    if (*head_ptr == NULL)
    {
        return -1;
    }

    next_node = (*head_ptr)->next;
    retval = (*head_ptr)->data;  // Retrives the data in the node being removed
    free(*head_ptr);    //  Remember to free the memory pointed by the initial head
    *head_ptr = next_node;  //  Assign the new node as the head

    return retval;
}
```



EXAMPLE USAGE – LINKED LISTS

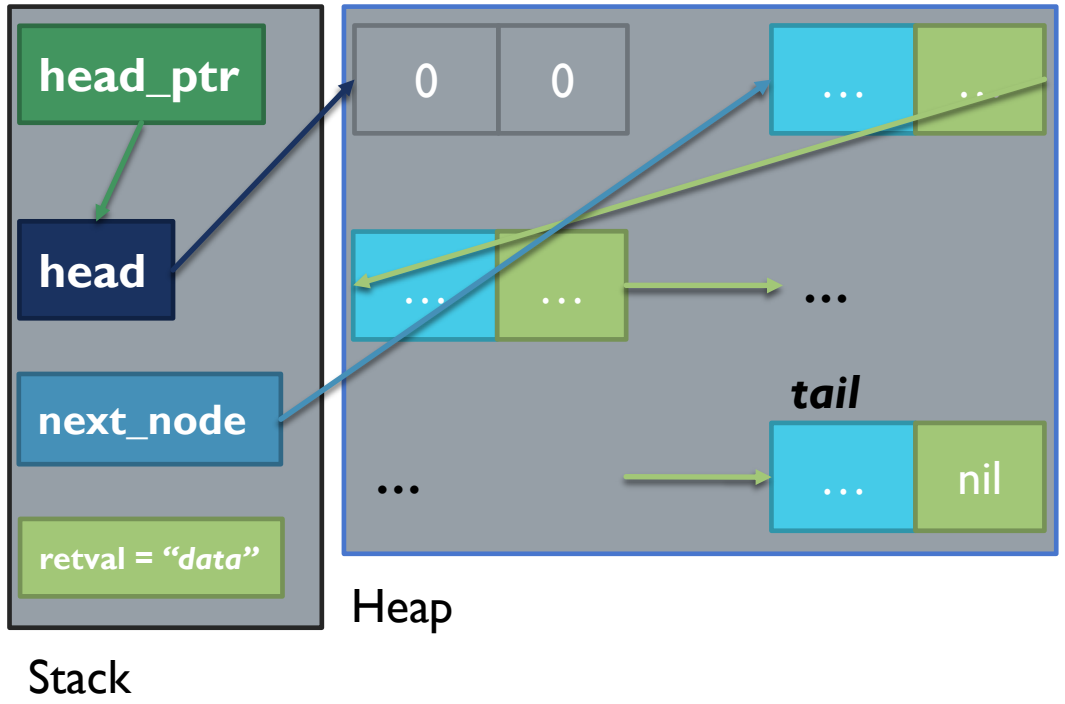
- **Remove first element of the list**

```c
/*
Removes the Head node (remember to keep track of head)
*/
int pop(node_t **head_ptr)
{
    int retval = -1;
    node_t *next_node = NULL;

    //  Checks if the Linked List only has 1 node (head itself)
    if (*head_ptr == NULL)
    {
        return -1;
    }

    next_node = (*head_ptr)->next;
    retval = (*head_ptr)->data; //  Retrives the data in the node being removed
    free(*head_ptr);            //  Remember to free the memory pointed by the initial head
    *head_ptr = next_node;  //  Assign the new node as the head

    return retval;
}
```



EXAMPLE USAGE – LINKED LISTS

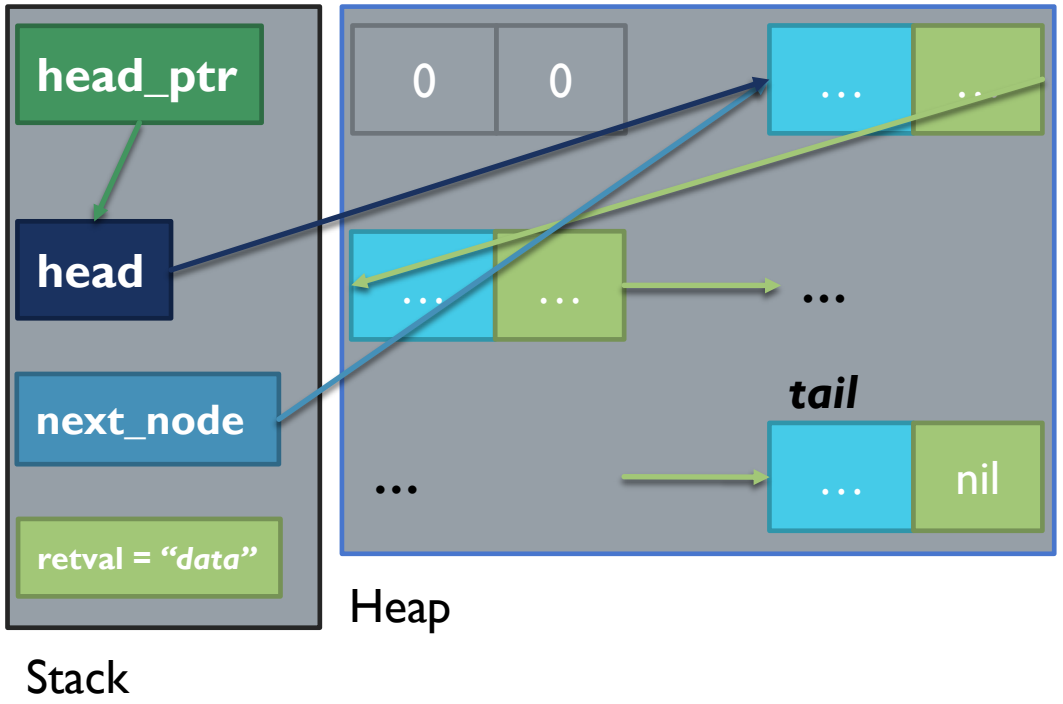
- **Remove first element of the list**

```c
/*
Removes the Head node (remember to keep track of head)
*/
int pop(node_t **head_ptr)
{
    int retval = -1;
    node_t *next_node = NULL;

    //  Checks if the Linked List only has 1 node (head itself)
    if (*head_ptr == NULL)
    {
        return -1;
    }

    next_node = (*head_ptr)->next;
    retval = (*head_ptr)->data; //  Retrives the data in the node being removed
    free(*head_ptr);    //  Remember to free the memory pointed by the initial head
    *head_ptr = next_node;  //  Assign the new node as the head

    return retval;
}
```



CONCLUSION

- Codes shown in slides available on myCourses
- References
  - https://www.learn-c.org/en/Linked_lists
  - https://www.tutorialspoint.com/cprogramming/index.htm
  - https://github.com/dmeger/COMP206_Fall2018_Lectures_Public
- Additional Resources
  - Visualize code – https://pythontutor.com/c.html#mode=edit
  - Common Mistakes – https://www.acodersjourney.com/top-20-c-pointer-mistakes/

THANK YOU!

Q&A