Dynamic Memory Allocation

COP 3223C – Introduction to Programming with C

Fall 2025

Yancy Vance Paredes, PhD

Scenario

- Write a program that reads and stores data from a file
- The first line is **N** indicating the number of student test scores
- Afterward, **N** lines will follow
- Print the numbers in reverse order of how they appeared in the file

Sample Run

scores.txt

```
5
90
80
85
70
65
```

Sample Output

```
65
70
85
80
90
```

Challenge

Print the numbers in ascending order (i.e., least to greatest)

Sample Run

scores.txt

```
5
90
80
85
70
65
```

Sample Output

```
65
70
80
85
90
```

Discussion

- One important issue we need to consider in our solution is determining when the required information becomes available during the program's execution

- When working with arrays, their size must be specified in advance.

- However, in some cases, the size isn't known until the program is running (i.e., at runtime)

Automatic Memory Allocation /1

- The **compiler** determined how much memory to allocate for the variables we have declared
- The size **cannot** be changed after it is allocated
- When the program runs, local variables are stored in the **stack space**

Automatic Memory Allocation /2

- They stay there during their lifetime (recall: **variable scope**)

- It is **automatically deallocated** at the end of its lifetime

Discussion /1

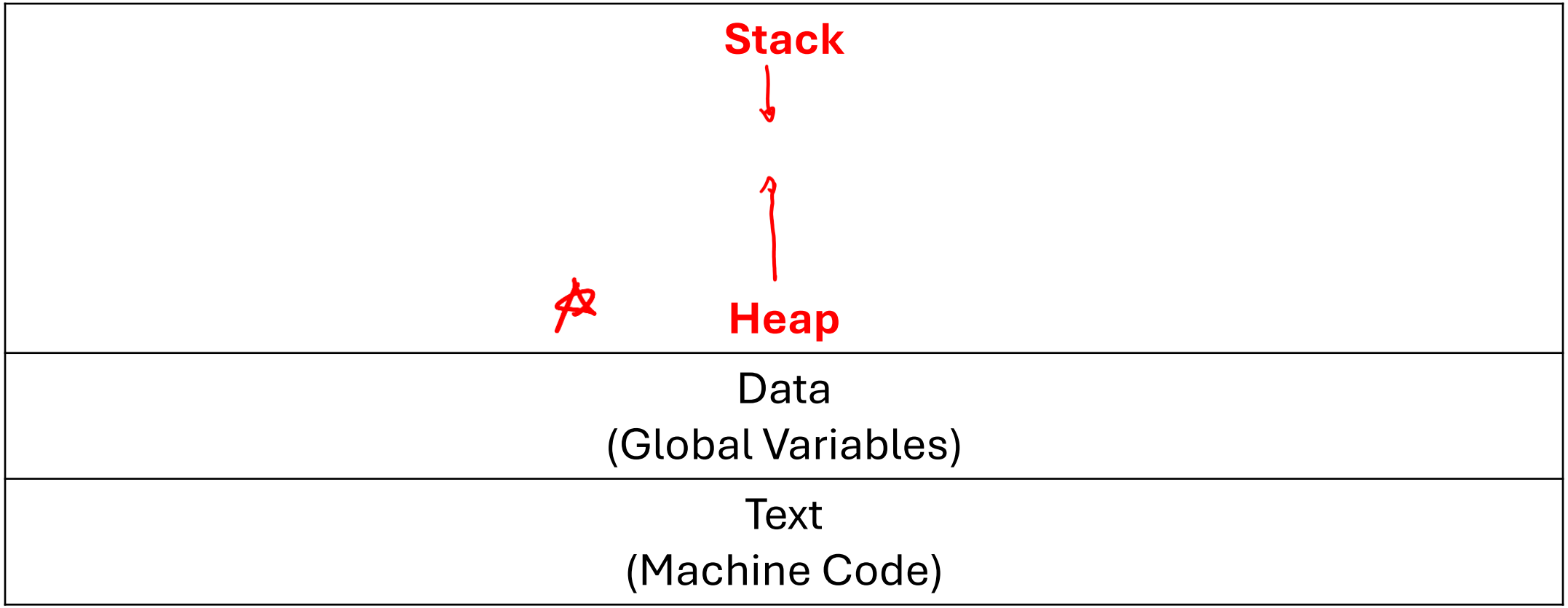
How about variables declared inside functions?

Discussion /2

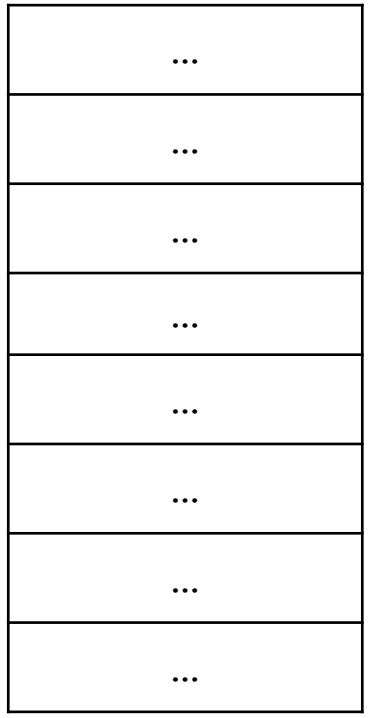
- There are instances in which you really don’t know how much memory you need

- Sometimes, you will only know the requirements during **runtime**

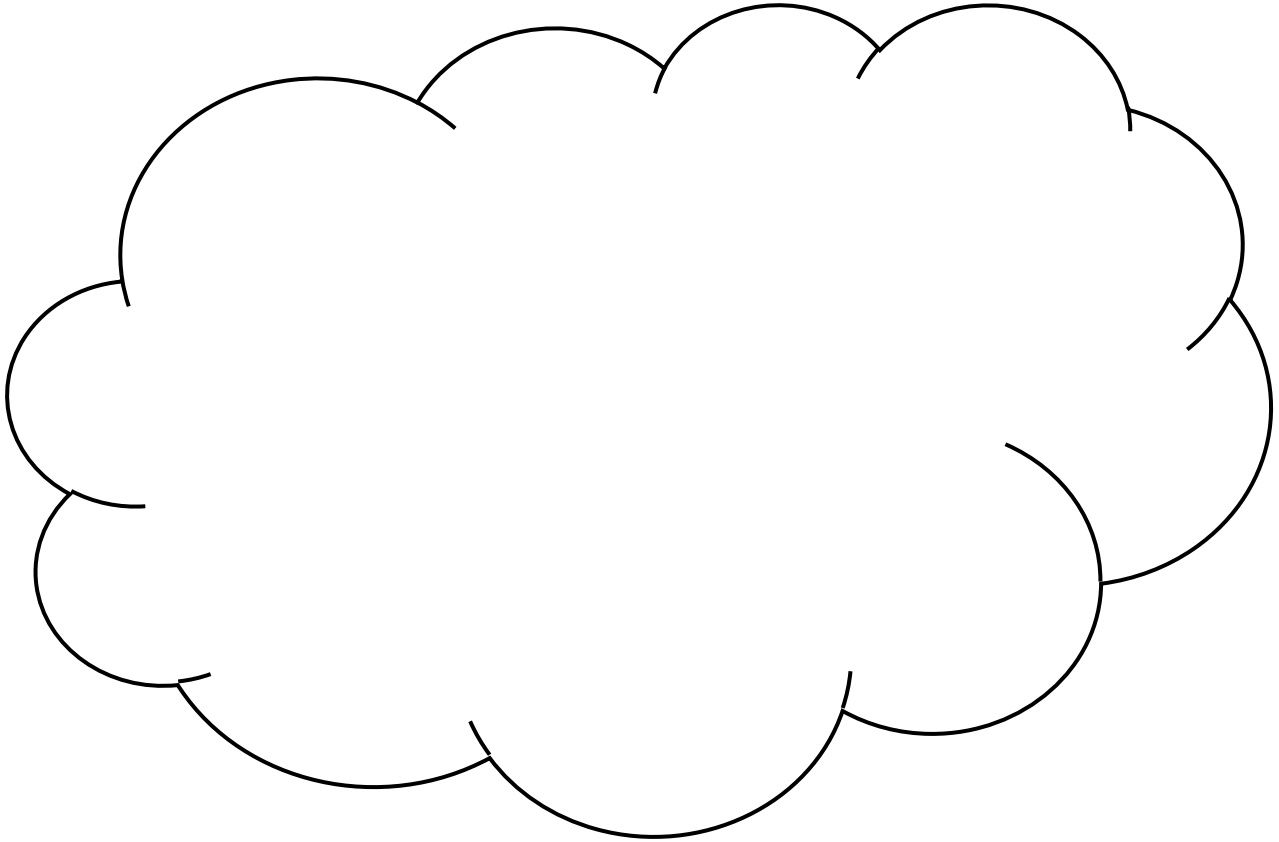
C Program Memory Layout (Segments)



Stack Space



Heap Space



Dynamic Memory Allocation

- Provides flexibility of specifying memory usage size during runtime
- Extra memory that we require will be from the **heap space**
- We simply indicate **how much** space we need
- Hopefully, it is **allocated** at runtime (it’s possible for it to fail)

The `sizeof` Operator

- Returns the size (in **bytes**) of a data type or even a variable

- Use this to indicate how much space you want to be allocated

For example:

Requesting for space /1

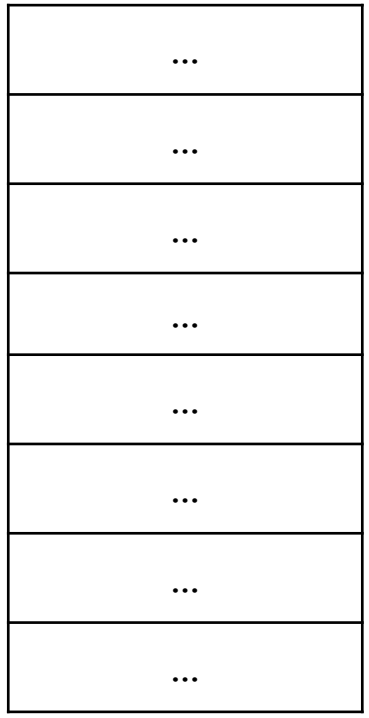
- Use the `malloc()` function defined under the `stdlib.h` file
- **Allocates a single block** of memory of a given size (in the heap)
- Returns either the **address of** the allocated space or the **`NULL`**
- Return type is `void*` (as in a `void` pointer; think of it as **generic**)

Requesting for space /2

Specify **how much space** is needed (i.e., size)

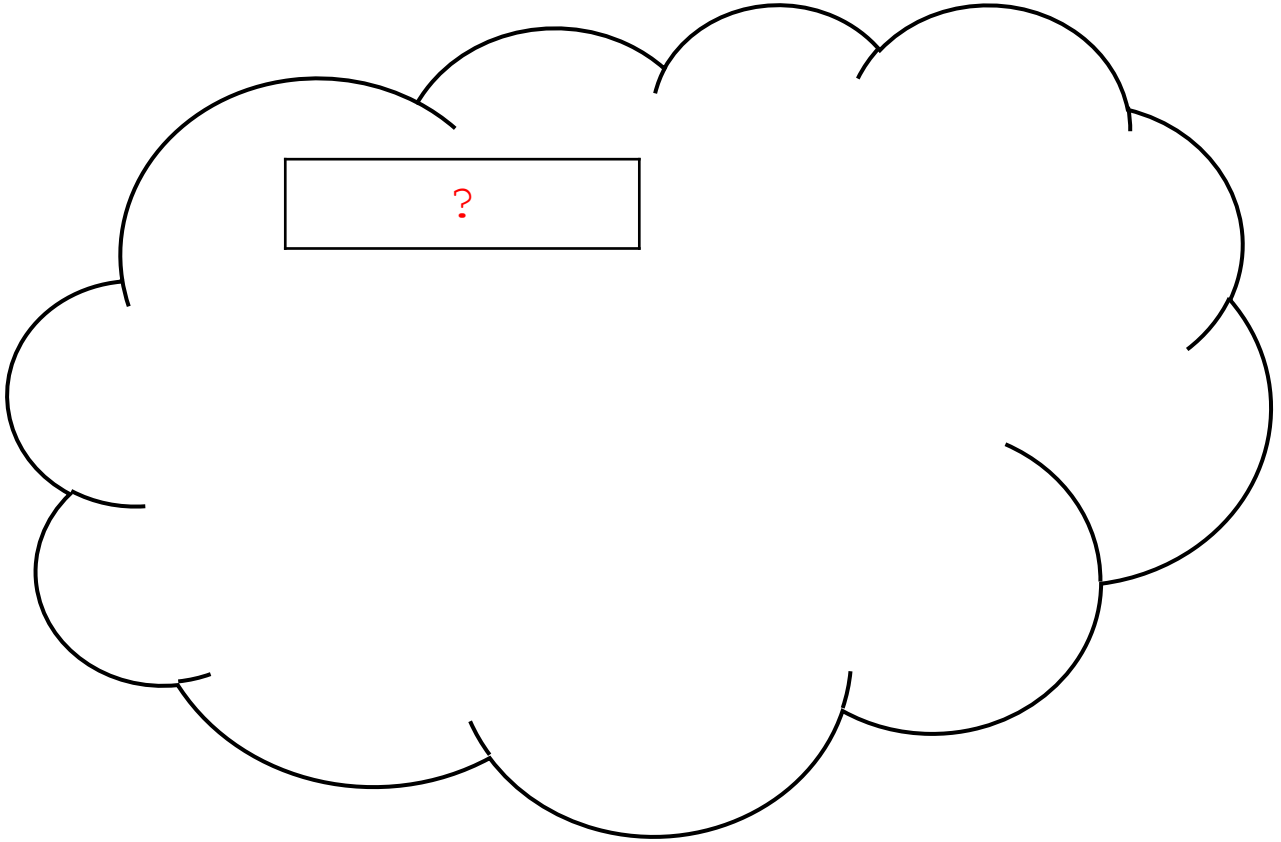
How?

Stack Space



malloc(sizeof(int));

Heap Space



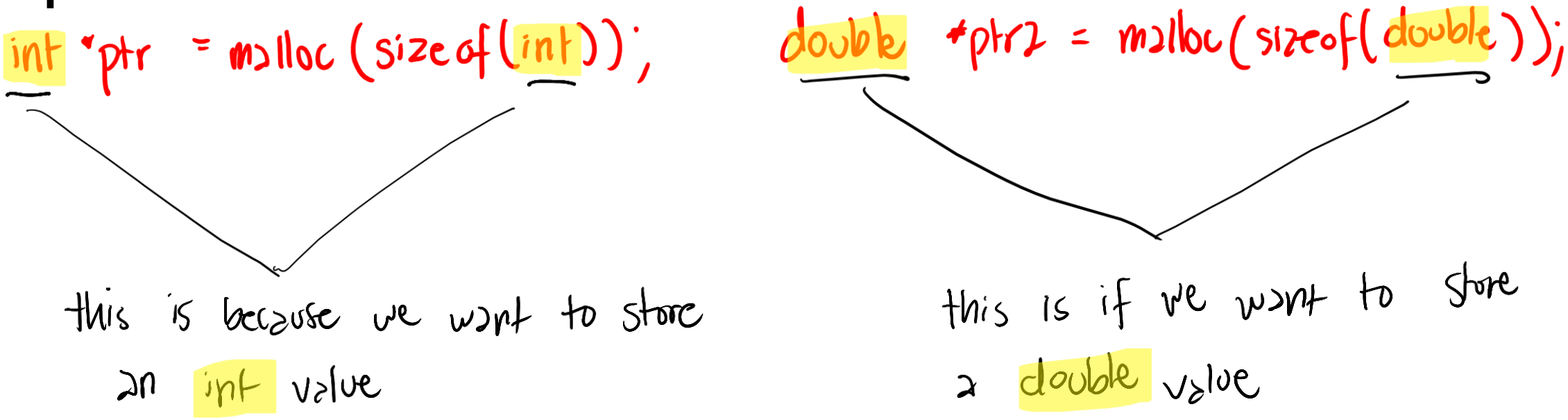
Notes

- We now have an **address** that `malloc()` provided
- However, the value at that location is **not initialized**
- Therefore, it contains a **garbage value**

Discussion

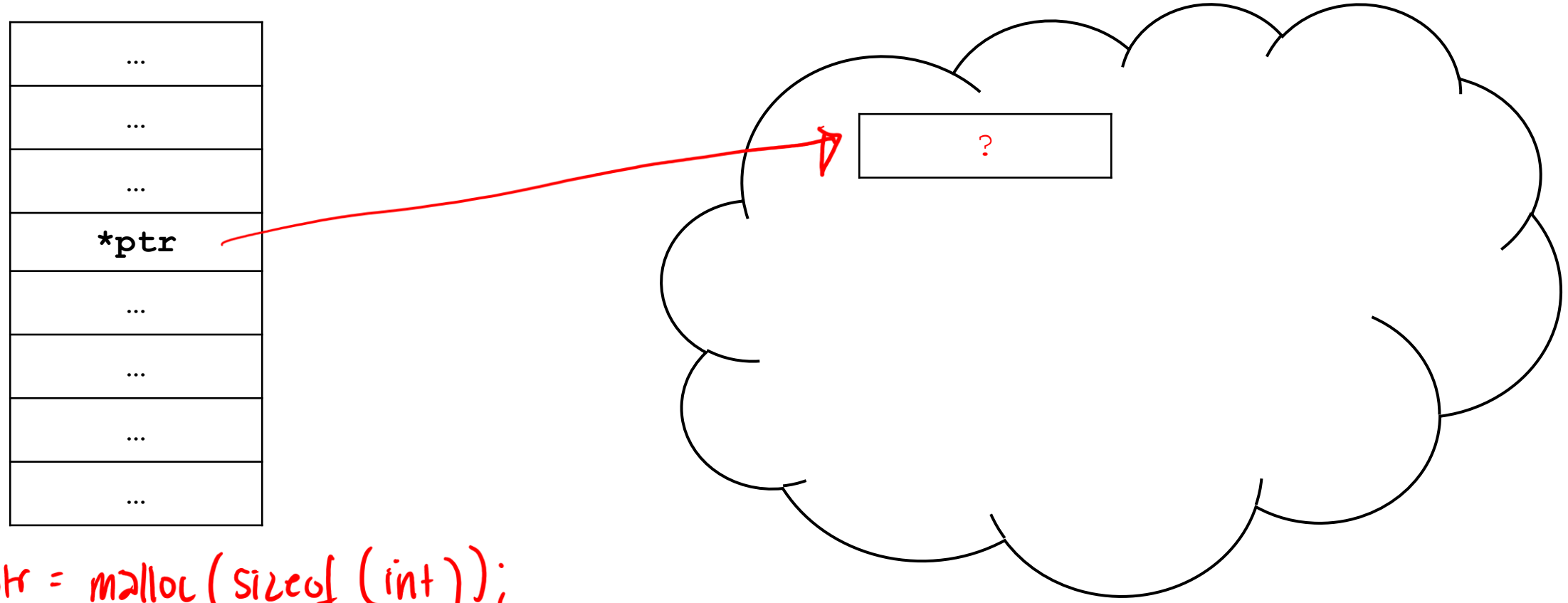
Because `malloc()` returns an **address**, what **kind of variable** do we need to remember it (or keep track of it for future use)?

A **pointer**!

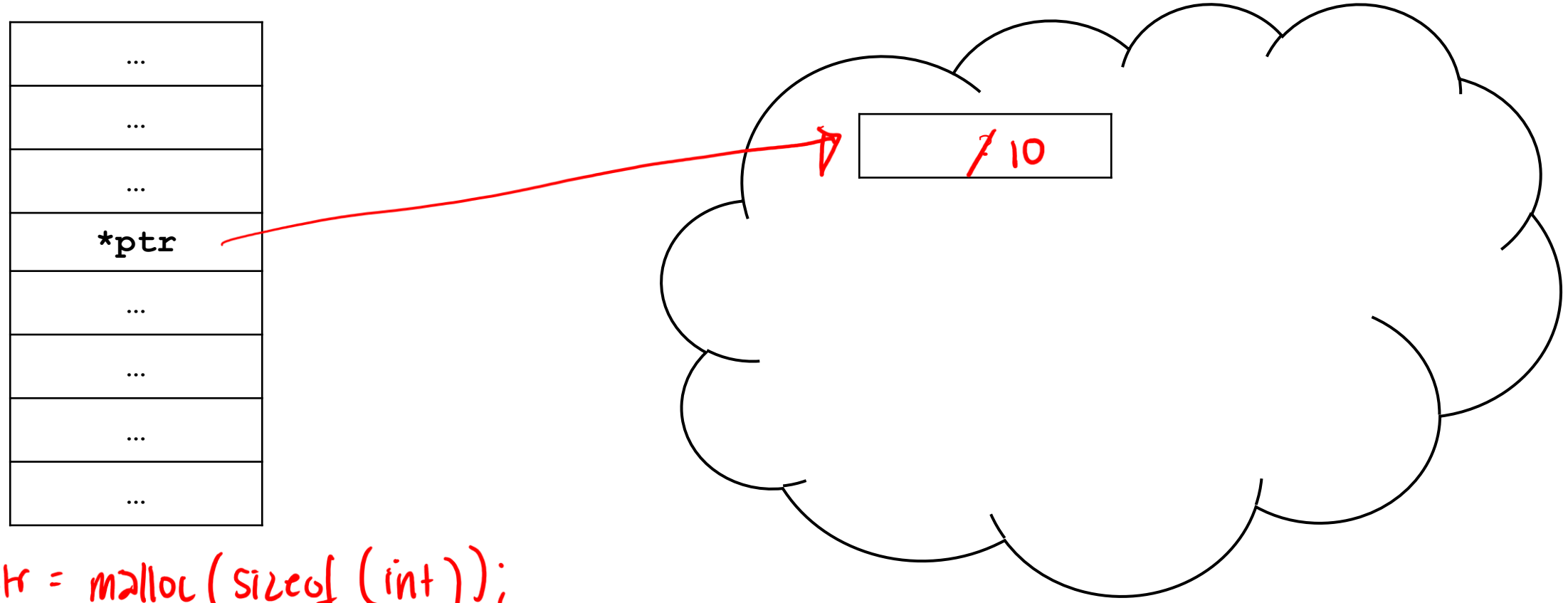


Stack Space

Heap Space




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




```
int *ptr = malloc(sizeof(int));
*ptr = 10;
```

Remember

When you’re done with the **dynamic memory**, it needs to be **deallocated** (freed; released)

Why?

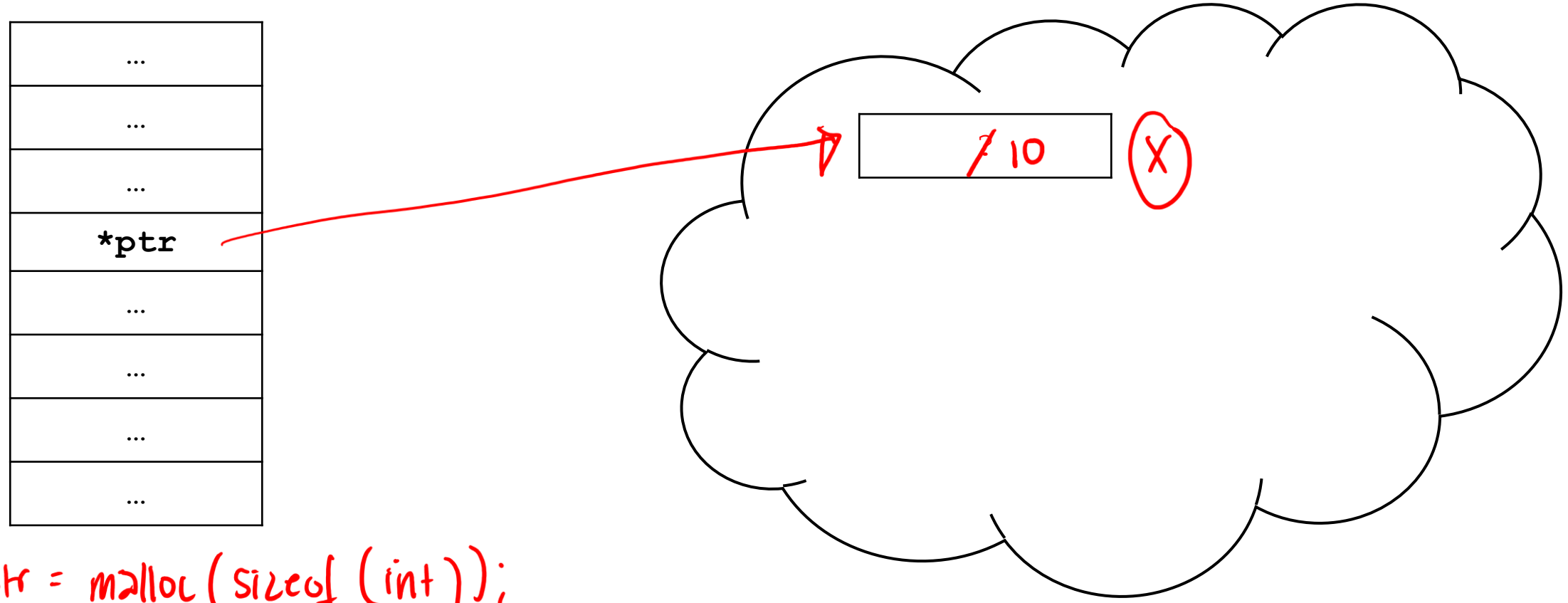
The `free()` Function

- Used to release a dynamically allocated memory

- Indicate the **address** in the memory that needs to be freed

Stack Space

Heap Space




```
int *ptr = malloc(sizeof(int));
*ptr = 10;
free(ptr);
*ptr = 20;
```

memory access violation! Segmentation Fault!

Notes

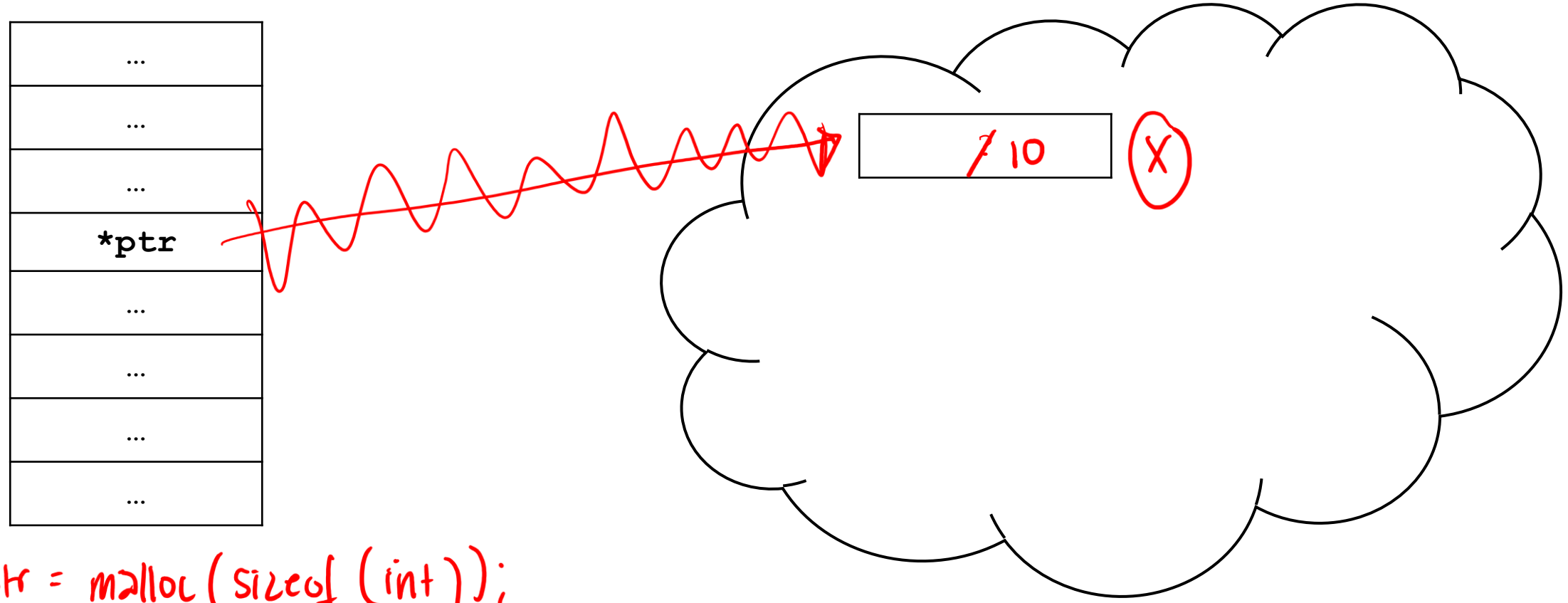
Rule of thumb is that each `malloc()` call should have its corresponding `free()` call

Discussion /1

- The `free()` **does not do** anything to the value at that location
- It only **labels** (or marks) the memory location **as free**
- Tells the operating system (OS) that this location is free
- Passing a freed location to `free()` leads to undefined behavior

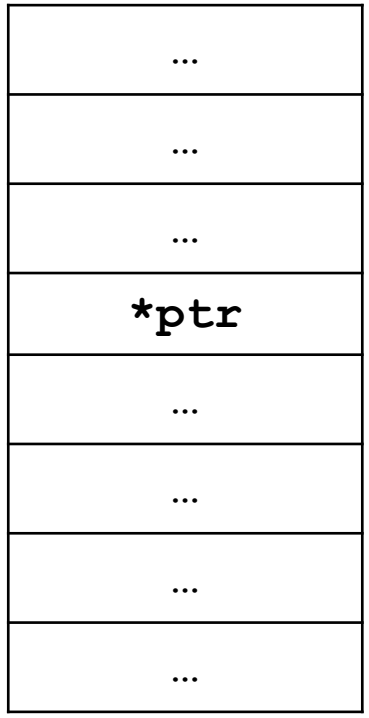
Discussion /2

- In your code, you could accidentally “reuse” the memory location after being freed
- **Best practice** to “forget” that address (i.e., discard)
- Set your pointer to `NULL`
- **Example:** `ptr = NULL;`

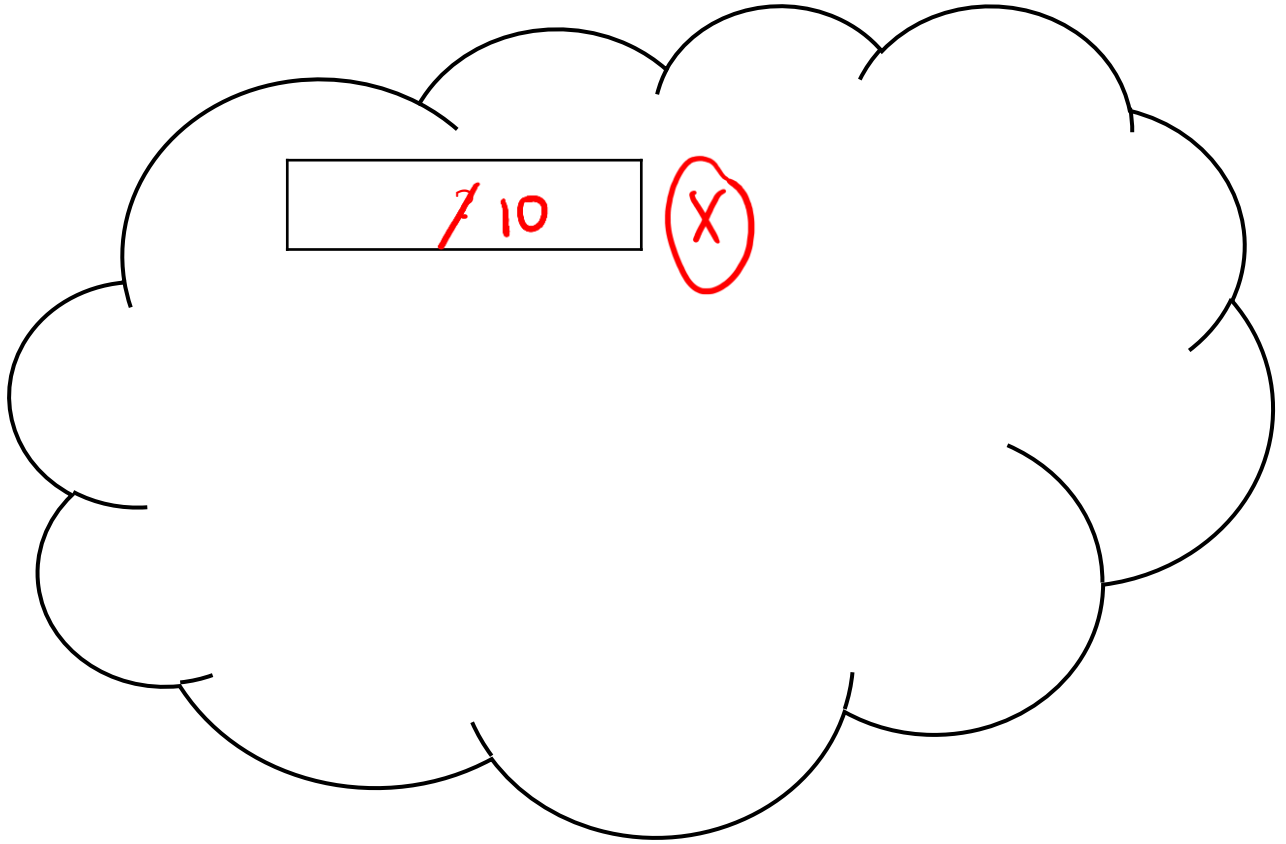



```
int *ptr = malloc(sizeof(int));
*ptr = 10;
free(ptr);
ptr = NULL;
```

Stack Space



Heap Space




```
int *ptr = malloc(sizeof(int));
*ptr = 10;
free(ptr);
ptr = NULL;
```

Your Turn!

- Dynamically allocate memory that could store a single `int` value
- Assign this to an `int` pointer variable called `num_ptr`

Caution: Memory Leak

- Allocated memory **no longer needed** and was **not released**

- In short, a **`malloc()`** that does not have a corresponding **`free()`**

Notes

If you are interested, you can explore the tool **valgrind** to help you detect any memory leaks on your C program

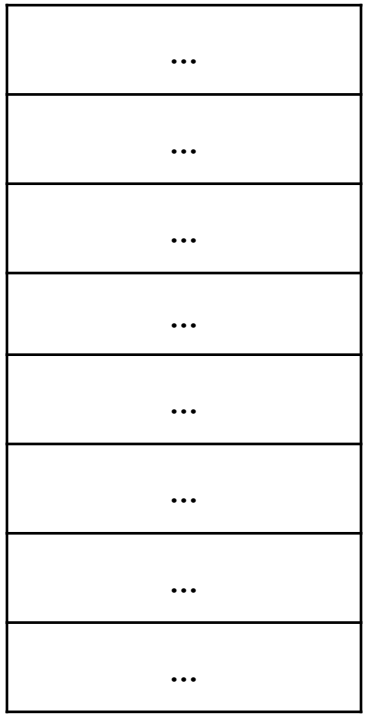
Discussion

- Why is this an issue?

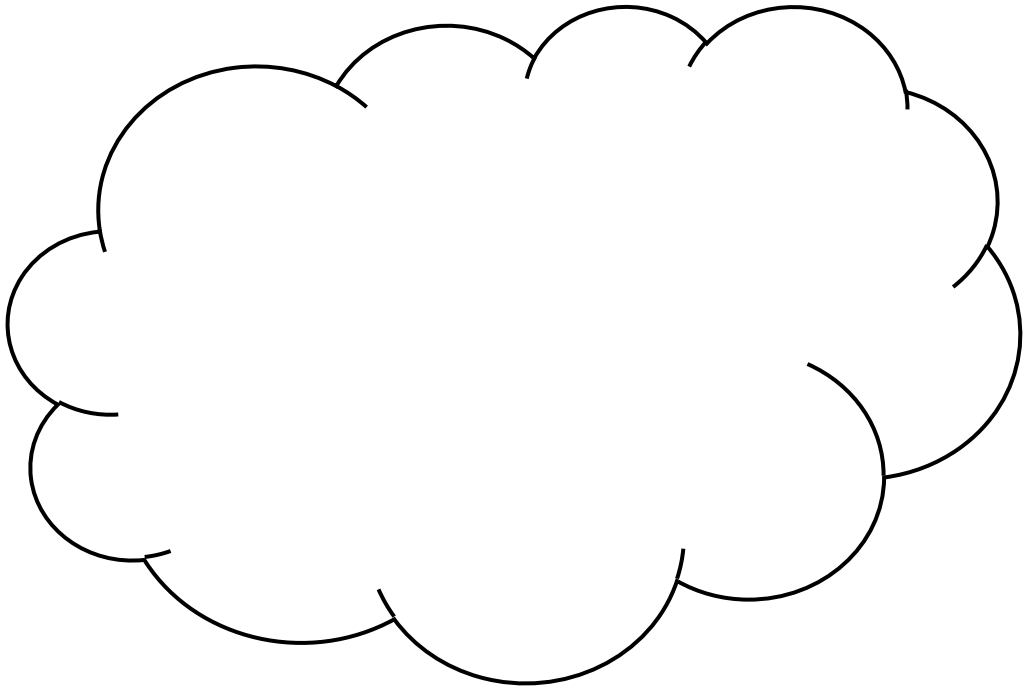
- Think about long-running programs such as server programs

```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;




// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```

```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;




// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```

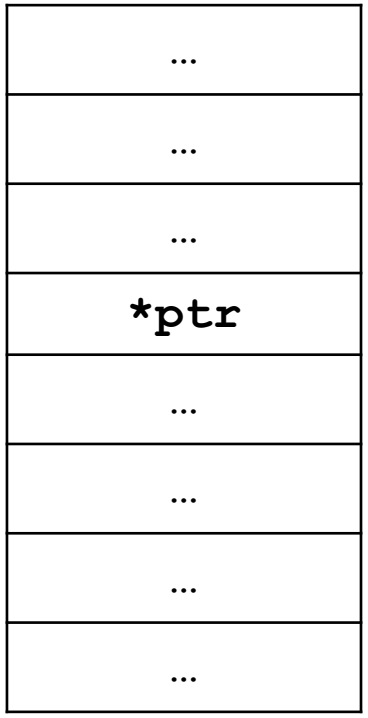


Stack Space

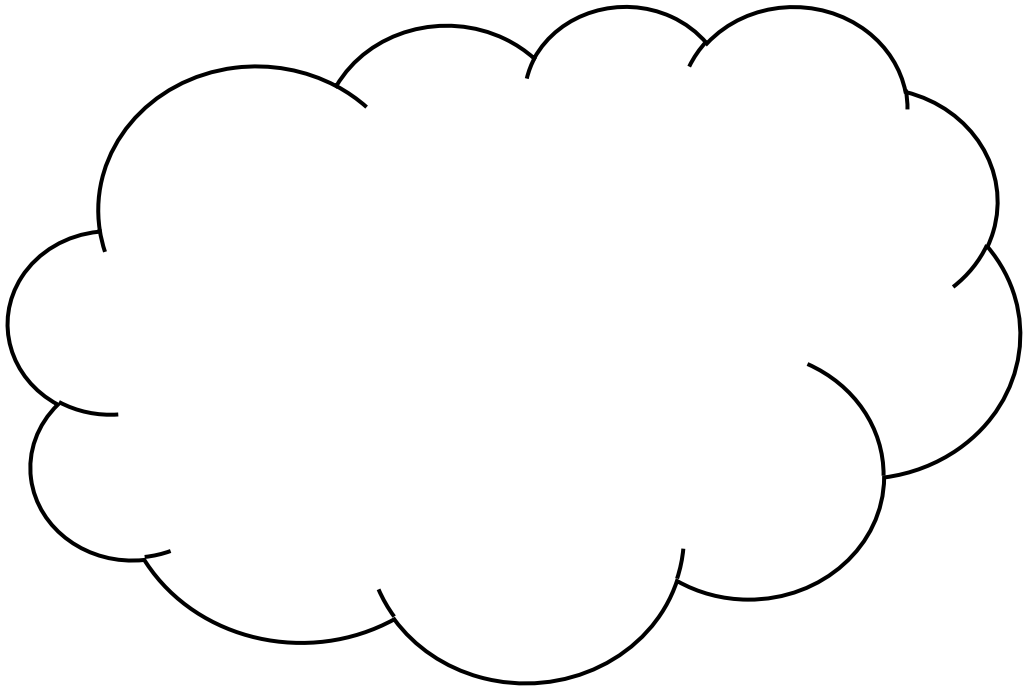


Heap Space

```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;




// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```

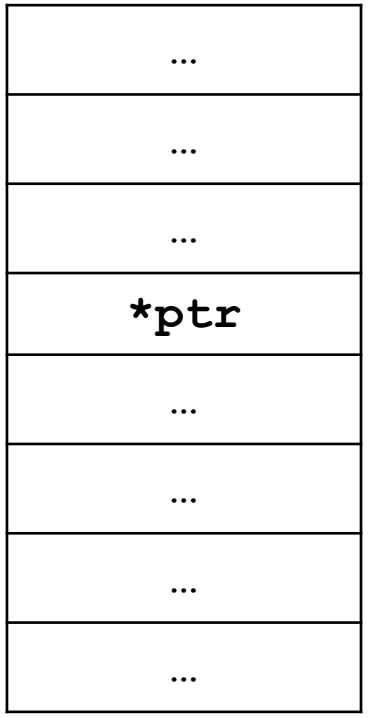


Stack Space

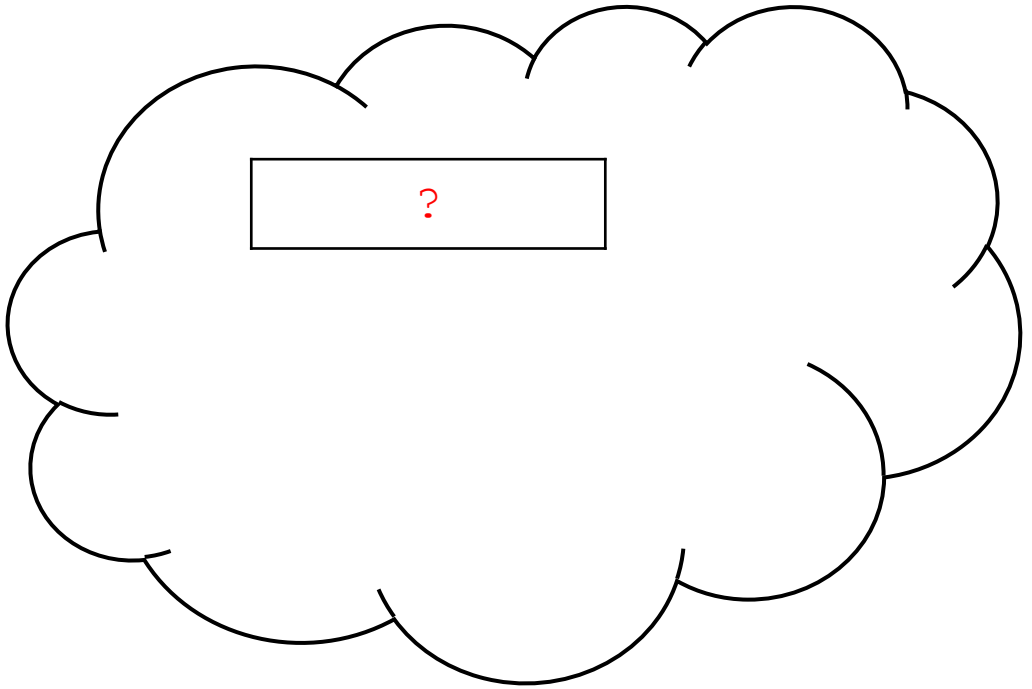


Heap Space

```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;



// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```

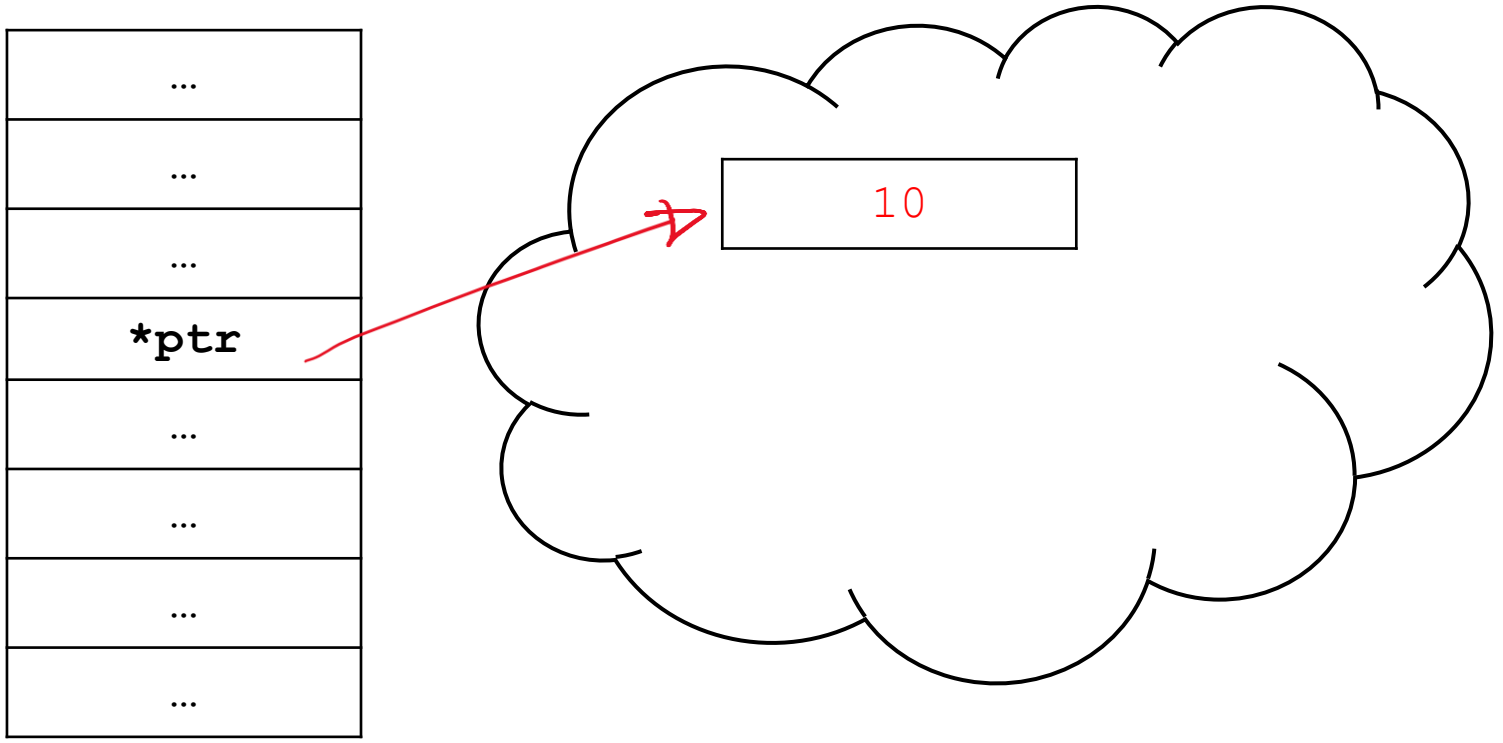


Stack Space



Heap Space

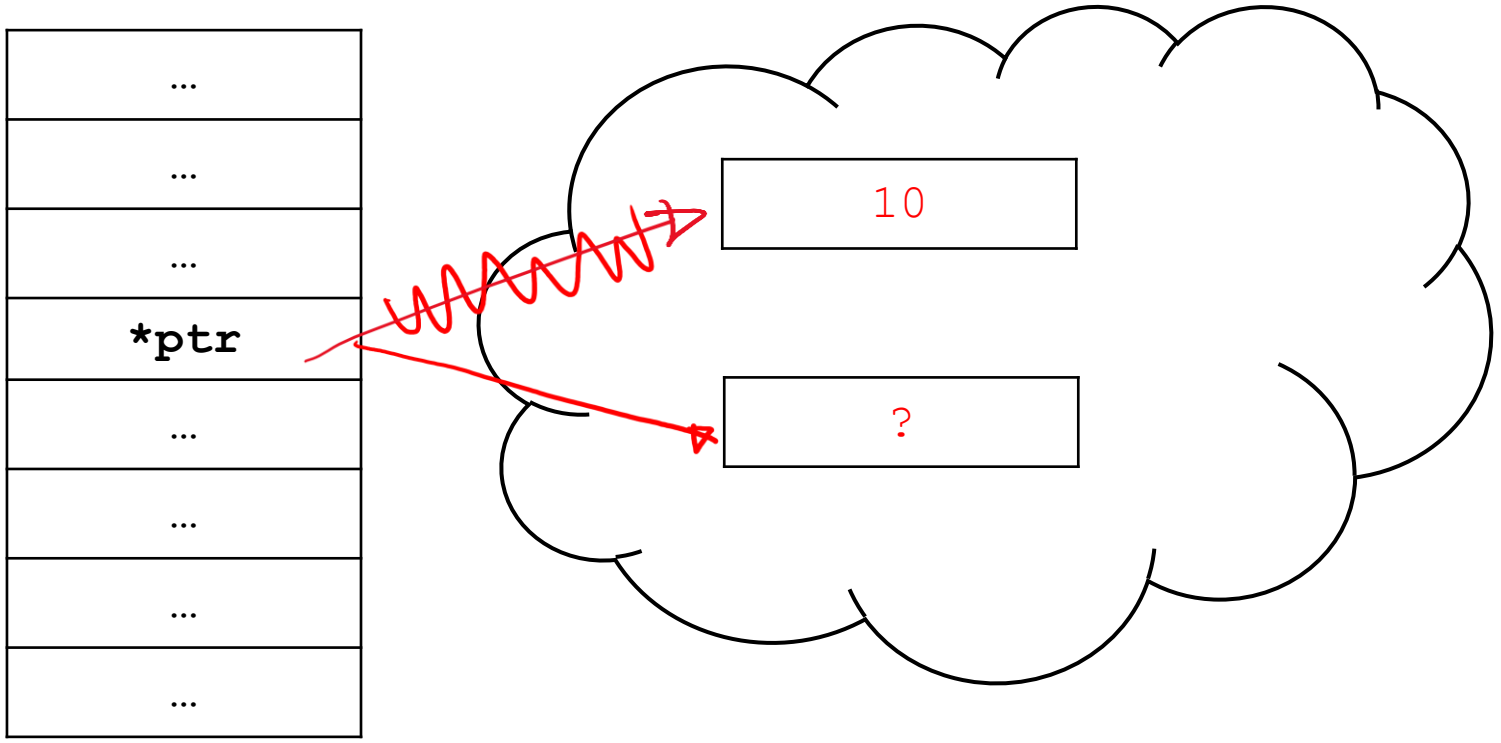
```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;



// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```



Stack Space **Heap Space**

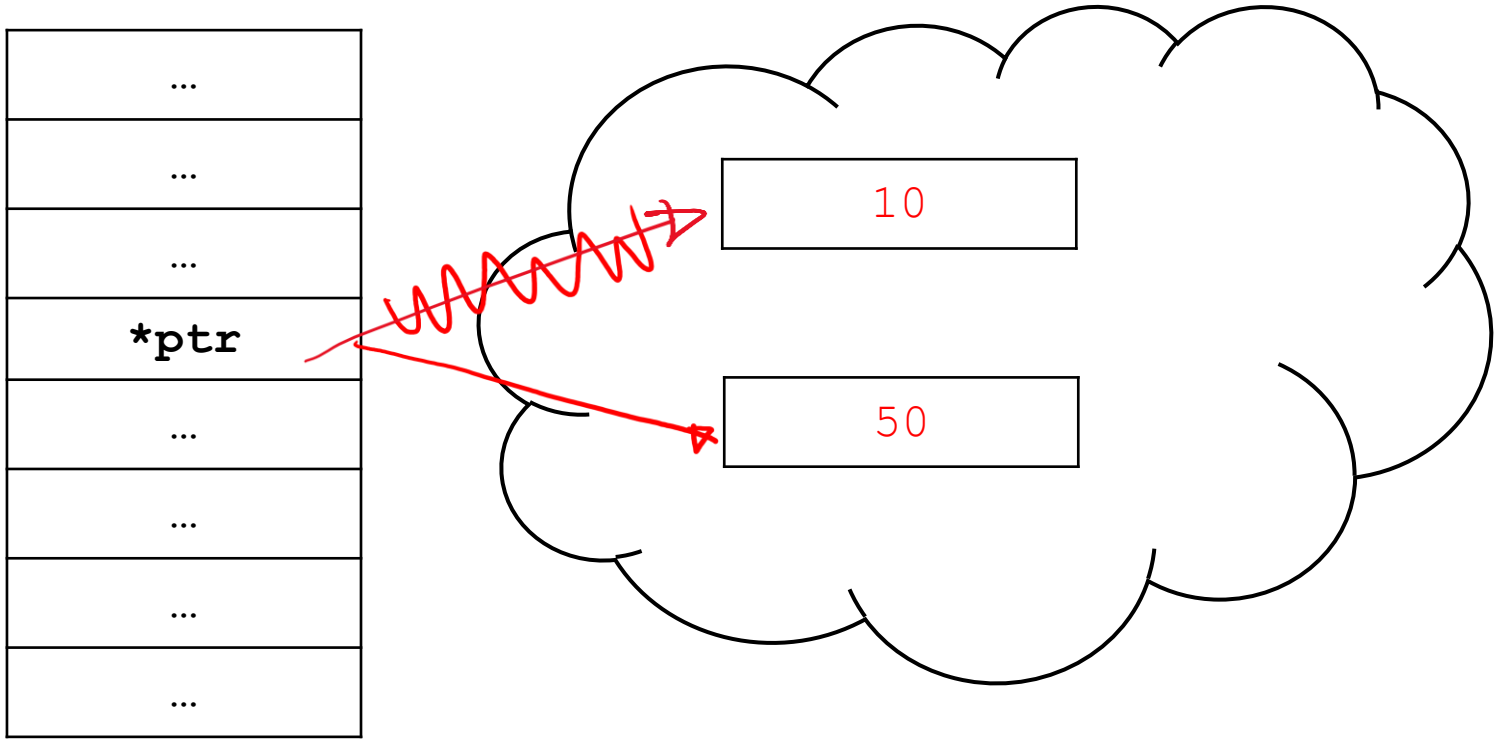
```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;




// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```

The previous location became “unreachable”



Stack Space **Heap Space**

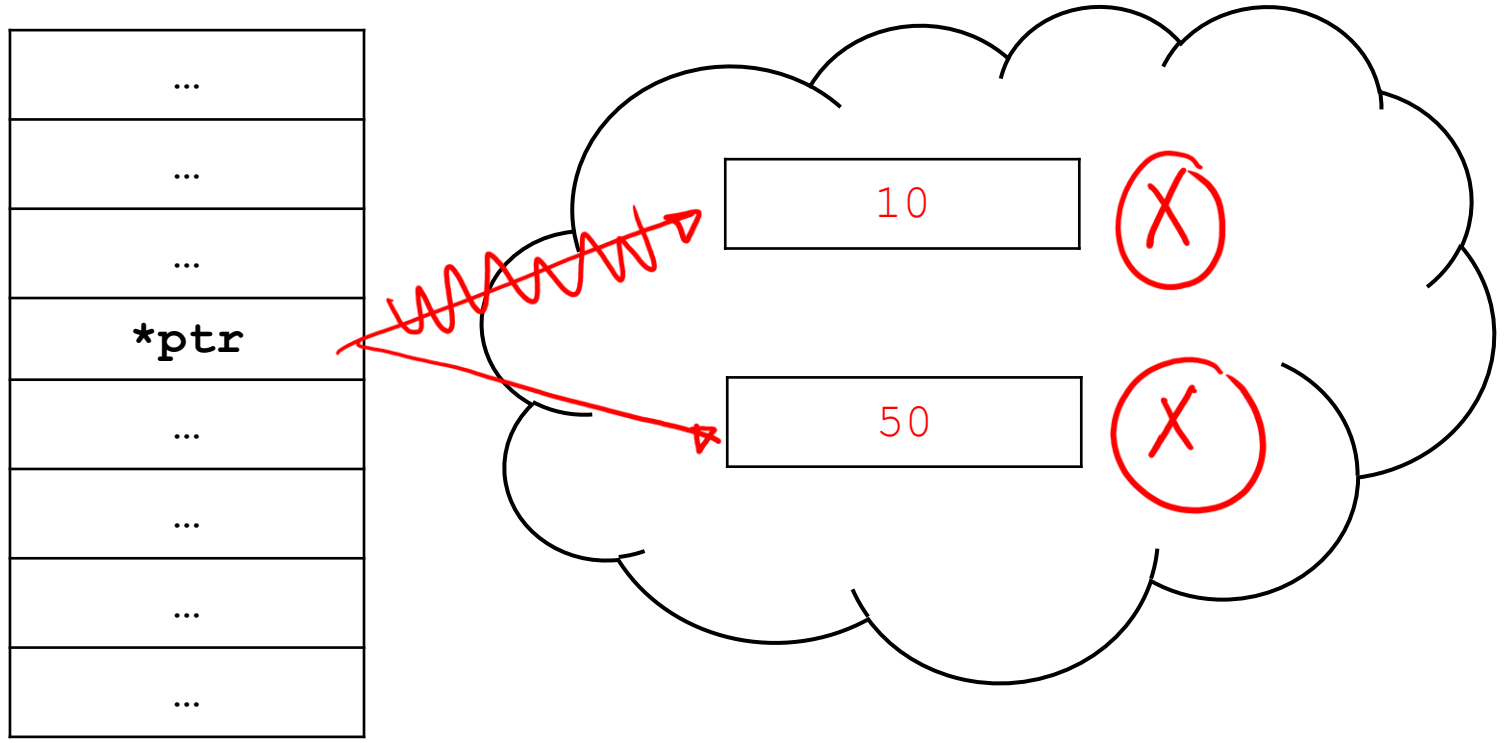
```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;




// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;
```



Stack Space **Heap Space**

```
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;

// free first memory
free(ptr);

// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;

// free first memory
free(ptr);
```

```c
int *ptr;

// first memory allocation
ptr = malloc(sizeof(int));

// set the value
*ptr = 10;

// free first memory
free(ptr);

// second memory allocation
ptr = malloc(sizeof(int));

*ptr = 50;

// free second memory
free(ptr);
```



Stack Space **Heap Space**

Discussion

- We have seen how to allocate a single variable, how about “several”?

- Can we do the same for an array?

Dynamically Allocated Arrays /1

- The same concept can be applied when you want to dynamically allocate an array
- We simply indicate how many elements this array will hold
- How?

Dynamically Allocated Arrays /2

- The idea is to do the same thing except that you want to indicate that this memory block should be able to hold **`N`** elements

- Recall that each element uses up **`sizeof(T)`** bytes

- So, let's do the Math!

Practice

- Ask the user to enter a number **`N`**
- Afterward, dynamically allocate an array of int with size **`N`**
- Lastly, deallocate the array

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

int main(void) {
    int N;

    // ask the user how many numbers to store
    printf("How Many: ");
    scanf("%d", &N);

    // dynamically allocate an array with size N
    int *arr = malloc(sizeof(int) * N);


    // deallocate the array
    free(arr);

    return 0;
}
```

Accessing Elements of the Array

- Despite a different way of allocating the array, the way how to access an element is still the same

- How do you set the value of the first element? Second element?

Practice

- Set the first and second elements of the array to 10 and 20

- Confirm by printing these

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

int main(void) {
    int N;

    // ask the user how many numbers to store
    printf("How Many: ");
    scanf("%d", &N);

    // dynamically allocate an array with size N
    int *arr = malloc(sizeof(int) * N);

    // set first element to 10 and second to 20
    arr[0] = 10;
    arr[1] = 20;

    // confirm if values were stored
    printf("%d %d\n", arr[0], arr[1]);

    // deallocate the array
    free(arr);

    return 0;
}
```

Recall: Pointer Arithmetic /1

- Allows you to move around in memory through pointers
- Useful when you are working with dynamically allocated arrays

Recall: Pointer Arithmetic /2

Given **`ptr`** is a pointer to type **`T`**, then:

`ptr+1` is the address that is `sizeof(T)` bytes after `ptr`

`ptr-1` is the address that is `sizeof(T)` bytes before `ptr`

Typically, you can add or subtract other whole numbers (be careful)

Recall: Pointer Arithmetic /3

How do we **access** the value at this location?

We use the * (dereference operator)

```
*(ptr+1)
```

→ this is equivalent to ptr[1] if ptr is a dynamic array

Note the order of precedence!

```
*ptr+1
```

so, *(ptr+i) == ptr[i]

(ptr+i) == &ptr[i]

Practice

- Modify the code so that it uses the pointer arithmetic
- Do this for both printing and reading

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

int main(void) {
    int N;

    // ask the user how many numbers to store
    printf("How Many: ");
    scanf("%d", &N);

    // dynamically allocate an array with size N
    int *arr = malloc(sizeof(int) * N);

    // pointer arithmetic way of setting elements
    for(int i = 0; i < N; i++) {
        scanf("%d", arr+i);
    }

    // pointer arithmetic way of accessing elements
    for(int i = 0; i < N; i++) {
        printf("#%d: %d\n", i+1, *(arr+i));
    }

    // deallocate the array
    free(arr);

    return 0;
}
```

Code Tracing Practice

In the following code tracing slides, assume the following:

- An **`int`** takes 4 bytes of space
- A pointer to **`int`** takes 8 bytes of space

Code Tracing

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5
6      int *a = malloc(sizeof(int));
7
8      int *b = malloc(sizeof(int) * 10);
9
10     int **c = malloc(sizeof(int*) * 20);
11
12     int d[10];
13
14     int e[5][4];
15
16     printf("%d\n", sizeof(a));   // 8
17     printf("%d\n", sizeof(b));   // 8
18     printf("%d\n", sizeof(c));   // 8
19     printf("%d\n", sizeof(d));   // 40
20     printf("%d\n", sizeof(e));   // 80
21
22     return 0;
23 }
```

these are all pointers thus size is 8 regardless

Code Tracing

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

int main(void) {

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

    int *b = malloc(sizeof(int) * 10);

    int **c = malloc(sizeof(int*) * 20);

    int d[10];

    int e[5][4];

    foo(a);
    foo(b);
    foo(d);

    return 0;
}


void foo(int *ptr) {
    printf("%d\n", sizeof(ptr));
}
```

Code Tracing

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

int main(void) {

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

    int *b = malloc(sizeof(int) * 10);

    int **c = malloc(sizeof(int*) * 20);

    int d[10];

    int e[5][4];

    bar(d);

    return 0;
}


void bar(int arr[]) {
    printf("%d\n", sizeof(arr));
}
```

int *arr

this decays into a pointer

8

Notes

- When you pass an array to a function, it generally decays into a pointer to its first element
- So, in the perspective of the function, it is dealing with a pointer, not an array
- Therefore, when designing functions, you need to pass the size information as well!

Discussion

- No intrinsic difference between a pointer to a single value and a pointer to the first element of an array.

- Treating a pointer as a "dynamic array" is based on context and convention, not on any metadata or type information.

More Helper Functions /1

- Define a function called **`create_array(N)`** that returns a dynamic array of integers with size **`N`**.
- Set all the elements of this array to **0**
- What will the function prototype look like?
- How do we call this function?

```
int * create_array(int N);
```

More Helper Functions /2

- Say you want to print all the contents of this array, define a function called **`print_array(arr, N)`**.

- What will the function prototype look like?

- We will need the information about the size!

```
void print_array(int *arr, int N);
```

More Helper Functions /3

- Of course, we want to ensure that we eventually deallocate this array, so we need to define a function called **`destroy_array(arr)`**.

- This function simply frees the memory used by the array.

```
void destroy_array(int *arr);
```

Practice

- Define the create_array() function
- Define the print_array() function
- Define the destroy_array() function
- Call all from the main function

Your Turn!

Solve the scenario posed at the beginning of the slide deck. There will be **`N`** numbers which is indicated in the first line of the input file. It will be followed by **`N`** test scores. Your program should print the numbers in reverse order of how they appeared in the file.

Sample Run

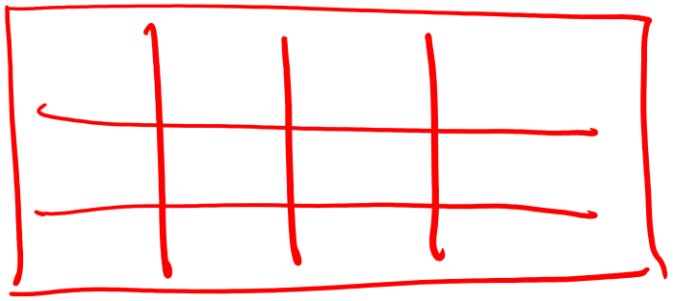
scores.txt

```
5
90
80
85
70
65
```

Sample Output

```
65
70
85
80
90
```

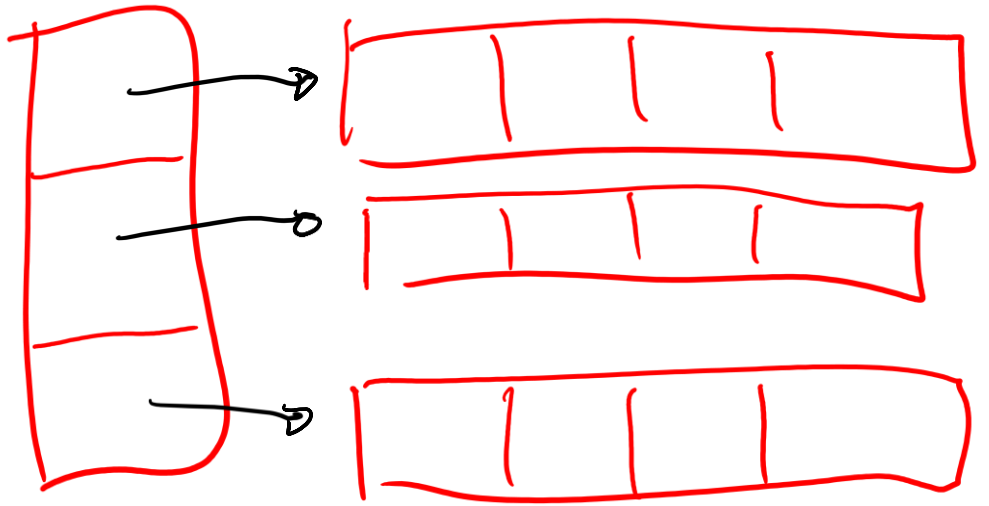
2D Dynamic Arrays



- How do we create a 2D array of integers?

- Say, we want to ask the user how many rows and columns

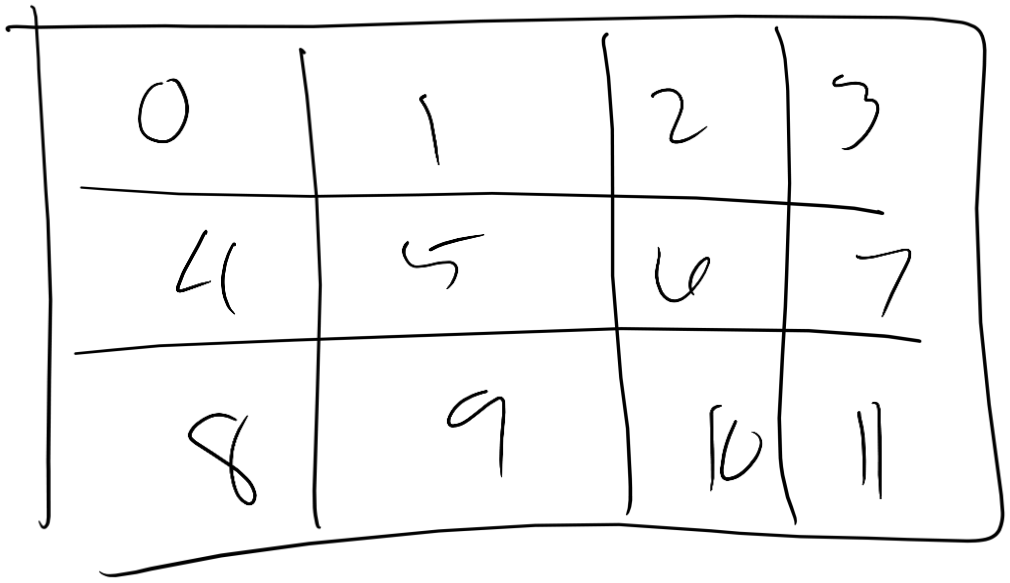
```
        3      4
double arr[rows][cols];
```



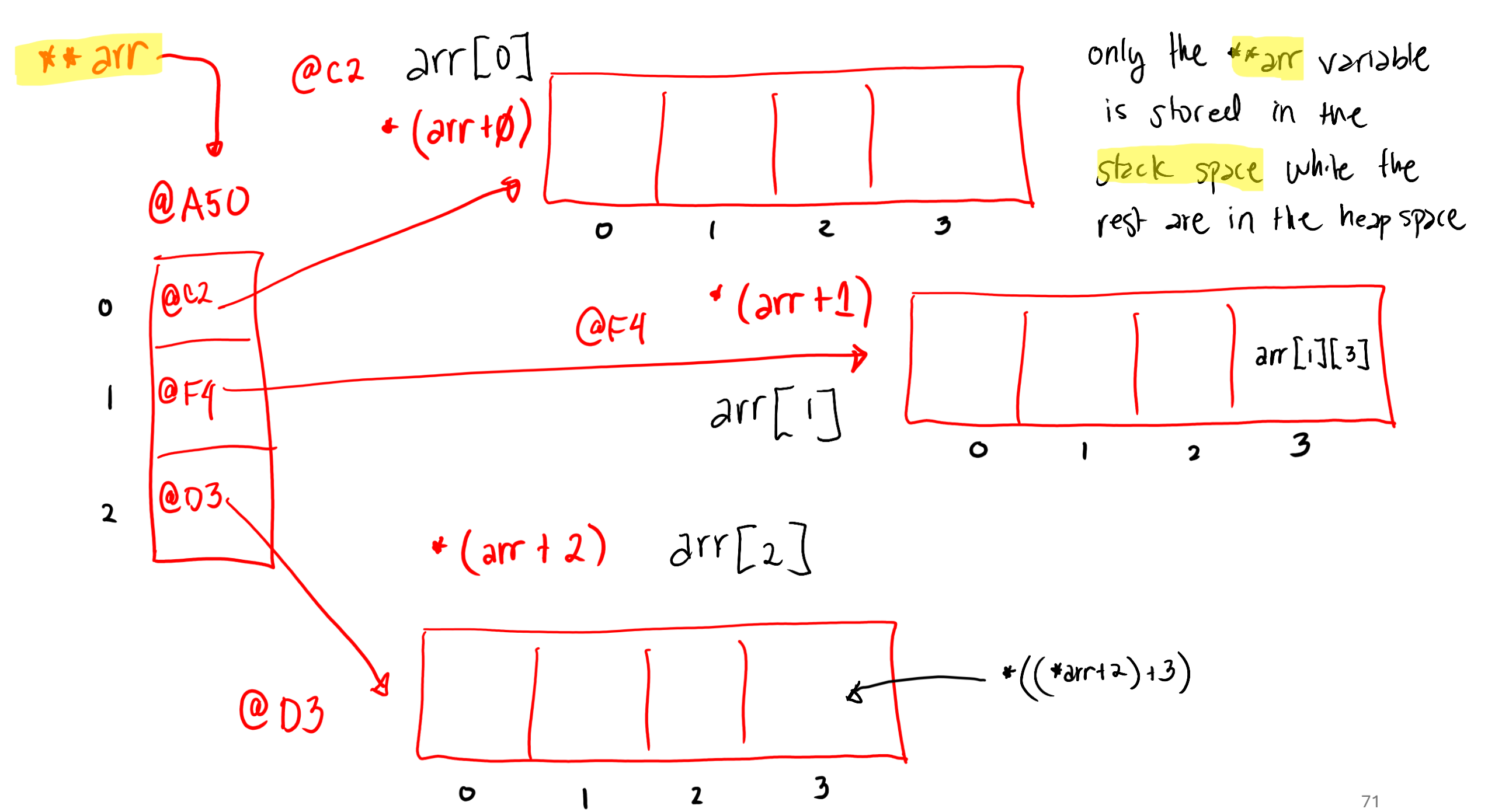
Practice

- Ask the user to enter the dimension of the 2D array (e.g., 3 rows and 4 columns)

- Afterward, populate each element



Refer to Webcourses for the C Code: Dynamic 2D Array

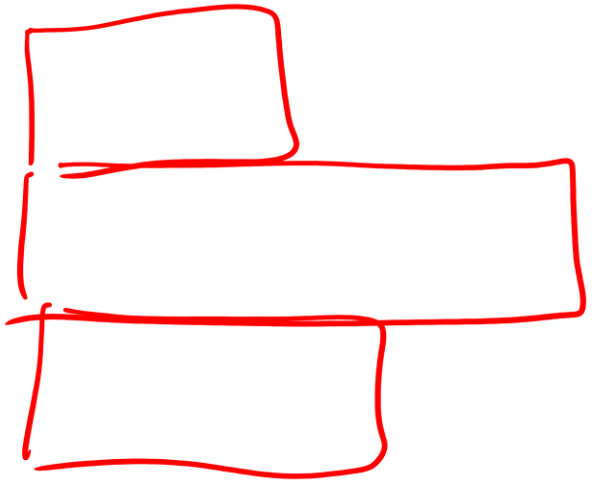


only the **arr variable is stored in the stack space while the rest are in the heap space

Discussion

- How do we deallocate this 2D array?

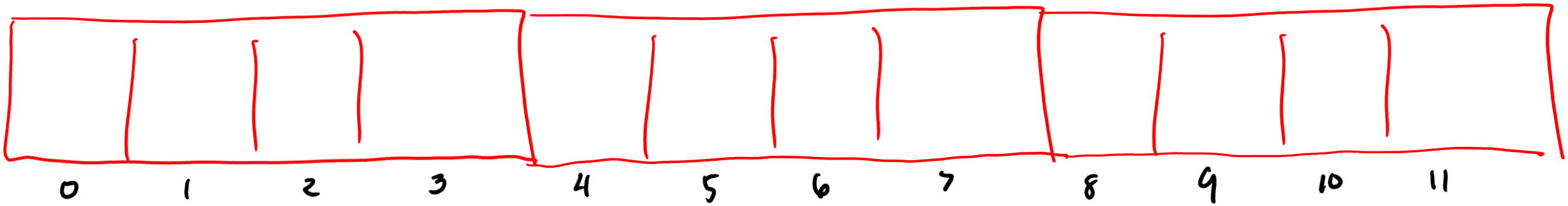
- Because we individually allocate arrays for each “row”, it possible for us to end up with a **jagged 2D array**, where some rows have lesser columns than others



2D Dynamic Arrays

- There is another approach to creating a 2D array
- However, this approach simply creates a 1D array but with number of elements that is **`rows * cols`**
- In short, this is a single block of memory but gives an illusion of being a 2D array

Refer to Webcourses for the C Code: Dynamic 2D Array (Other Approach)



rows = 3

cols = 4

allocated 12 contiguous spaces

indices

row 0: 0, 1, 2, 3

row 1: 4, 5, 6, 7

row 2: 8, 9, 10, 11

Notes

- The first approach is often more straightforward compared to the second approach

- One difference between the two approaches is the number of times you would have to call the **`malloc()`** function

- Consequently, this also affects the number of times you would have to call the **`free()`** function

Your Turn! /1

Define a function **`create_2d_array(r,c)`** that creates a 2D dynamic array with given rows and columns. Initialize all the values of all the elements to 0. It returns a pointer to this array.

```
int ** create_2d_array(int r, int c) {



}
```

Your Turn! /2

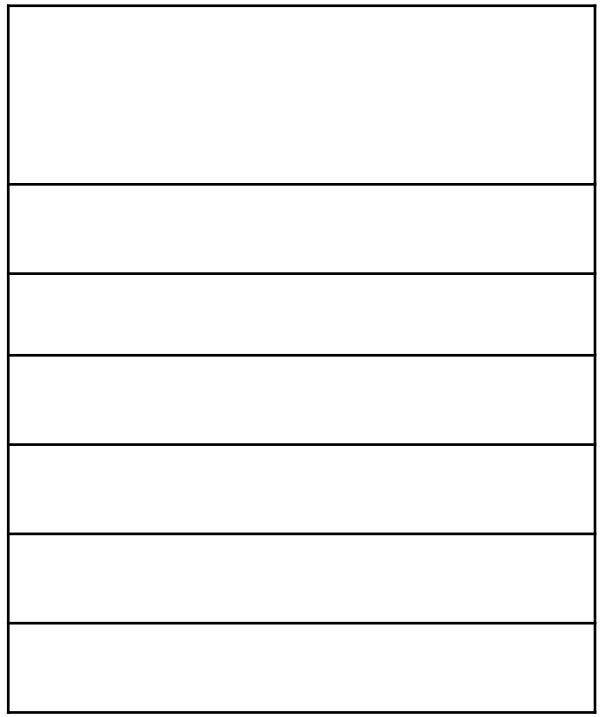
Define a function **`destroy_2d_array(arr)`** that deallocates all dynamic memory allocated to the array. It accepts a pointer to the array.

```
void destroy_2d_array(int *arr) {




}
```

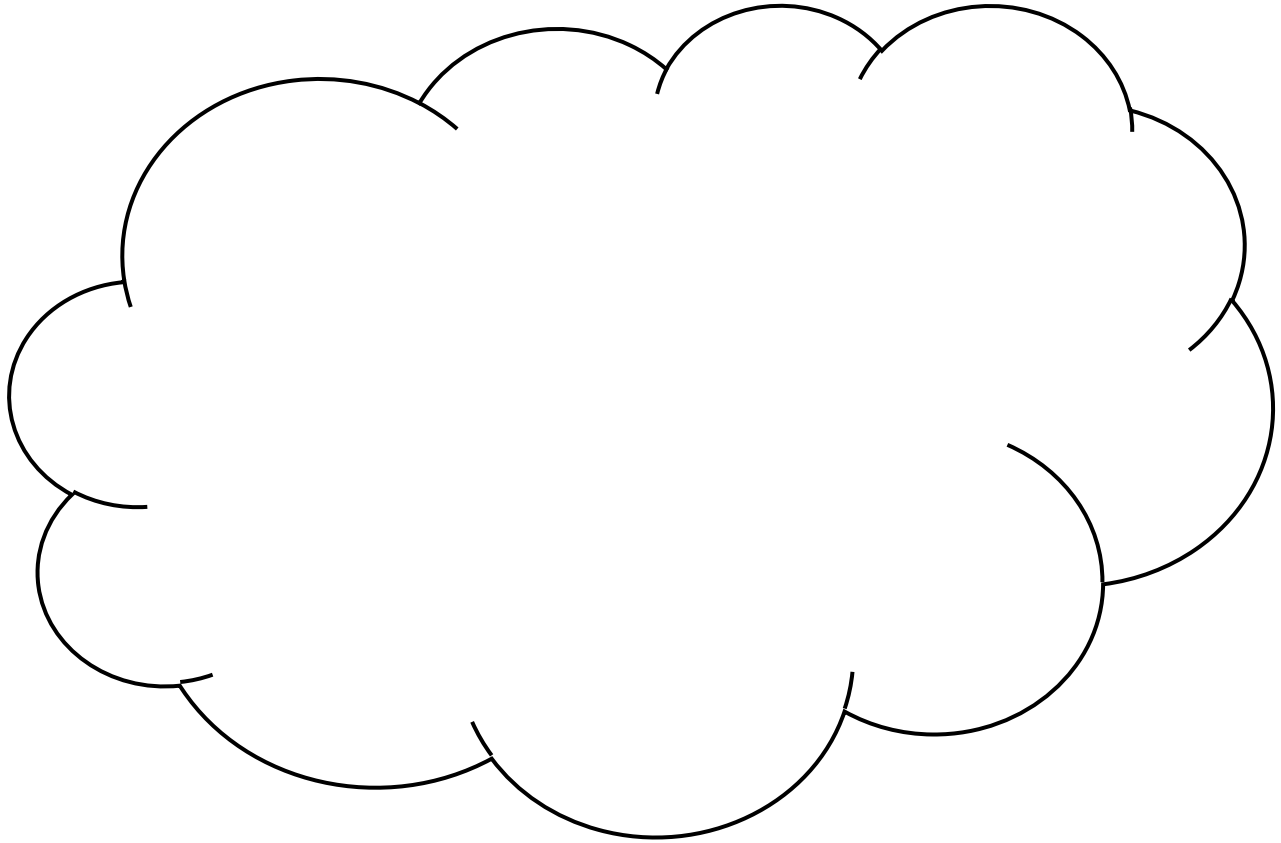
Dynamic Structs

Just like our primitive data types, we can also create pointers to our user-defined data types

Stack Space



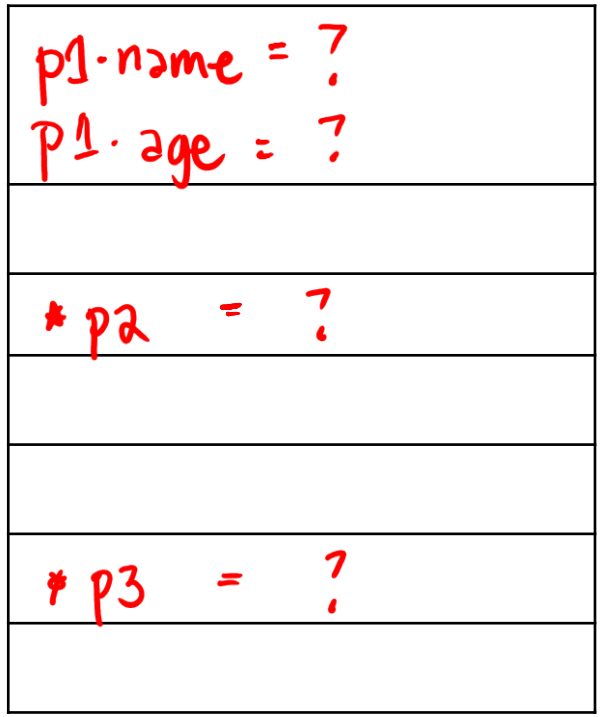
Heap Space



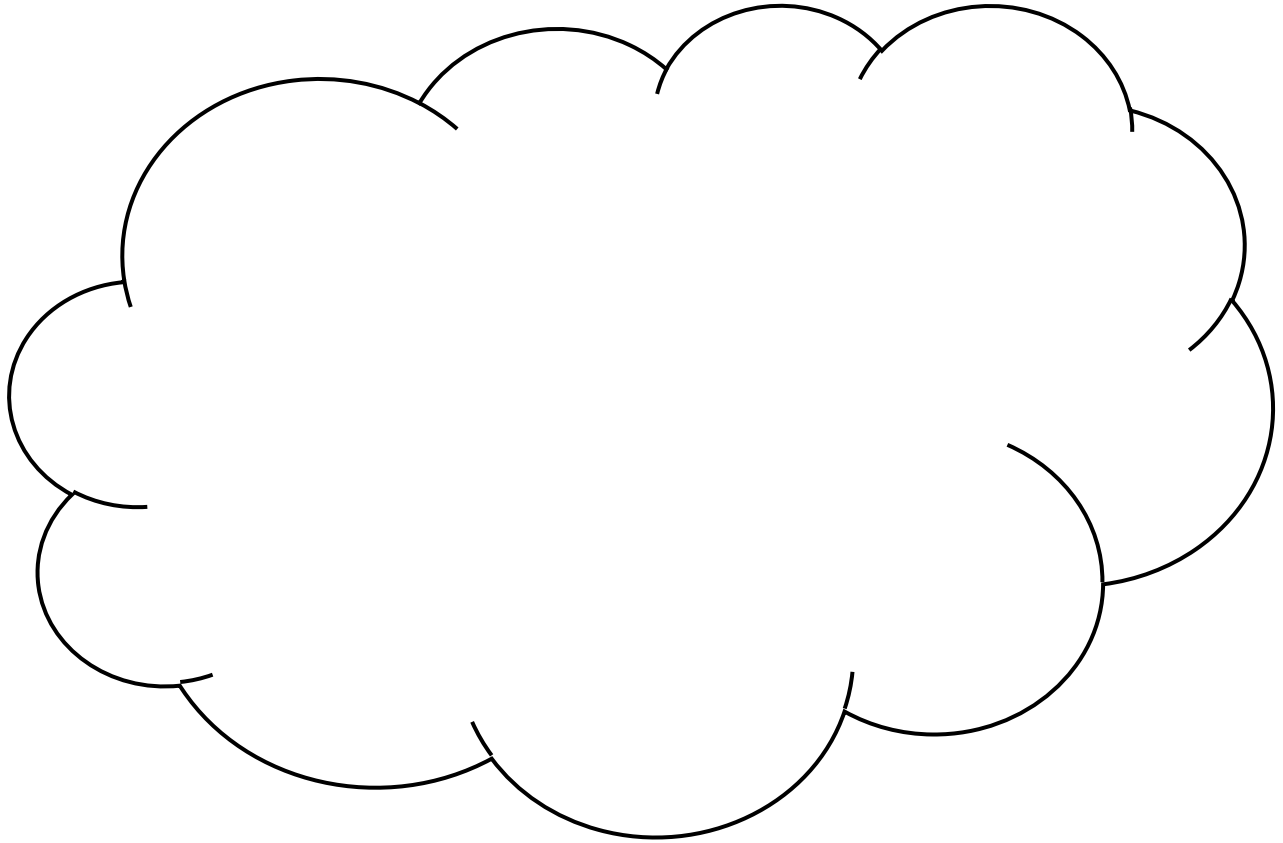
Code Comprehension

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define WORD_SIZE 256

typedef struct Person_s {
    char name[WORD_SIZE];
    int age;
} Person;

int main(void) {
    Person p1;

    Person *p2, *p3;


    return 0;
}
```

Stack Space



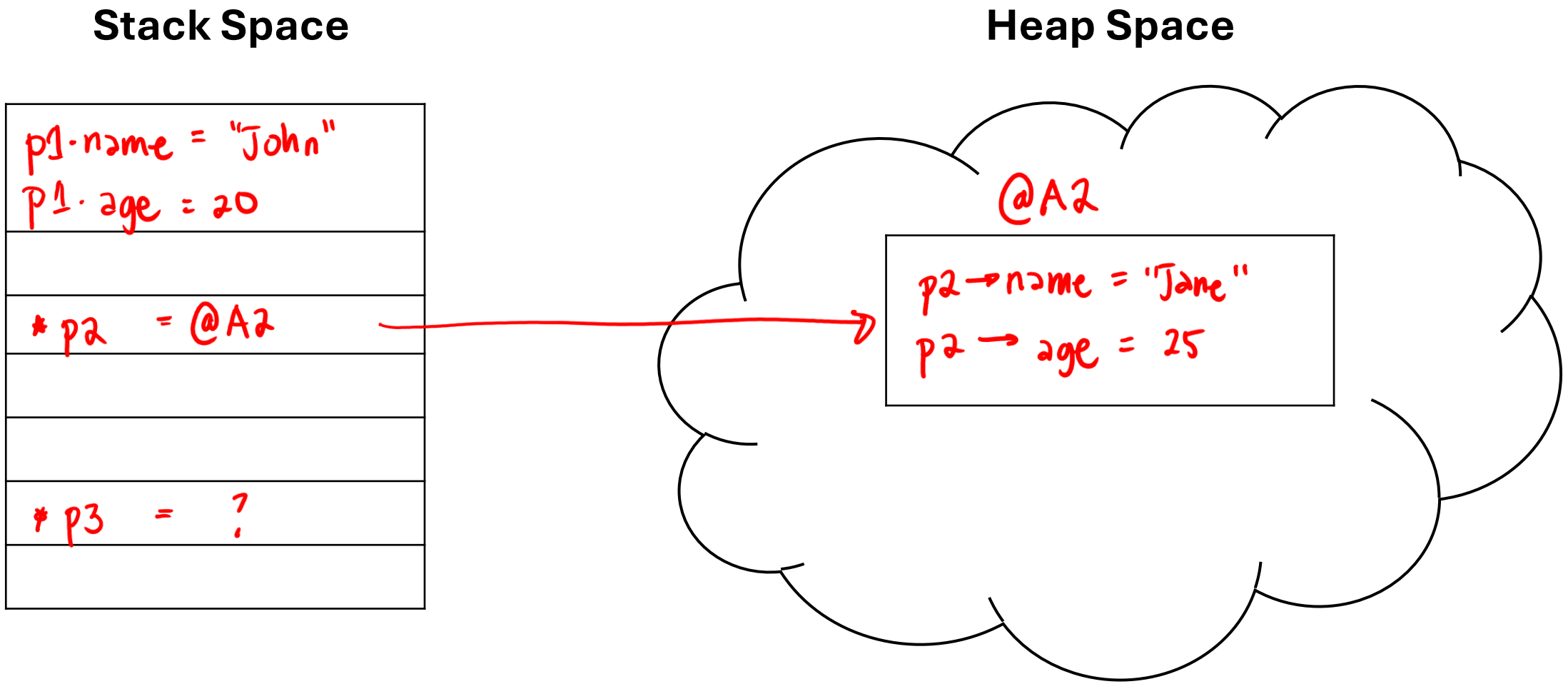
Heap Space



Code Completion

```
// answer 1
strcpy(p1.name, "John");
p1.age = 20;

// answer 2
p2 = malloc(sizeof(Person));
strcpy(p2->name, "Jane");
p2->age = 25;
```

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define WORD_SIZE 256

typedef struct Person_s {
    char name[WORD_SIZE];
    int age;
} Person;

int main(void) {
    Person p1;

    // set name and age of p1 to John & 20
    // see answer 1

    Person *p2, *p3;

    // set name and age of p2 to Jane & 25
    // see answer 2

    return 0;
}
```


Stack Space
Heap Space
p1.name = "John"
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2→name = "Jane"
p2→age = 25

Code Completion

```
// answer
p3 = malloc(sizeof(Person) * 10);
// approach 1 using pointers
strcpy(p3->name, "Bob");
p3->age = 30;
// approach 2 using array notation
strcpy(p3[0].name, "Bob");
p3[0].age = 30;
```

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #define WORD_SIZE 256
5
6  typedef struct Person_s {
7      char name[WORD_SIZE];
8      int age;
9  } Person;
10
11 int main(void) {
12     Person p1;
13
14     // set name and age of p1 to John & 20
15
16
17     Person *p2, *p3;
18
19     // set name and age of p2 to Jane & 25
20
21
22
23     // create an array of Persons with size 10
24     // set the name and age of the first Person
25     // to Bob & 30 ; assign the array to p3
26
27     // see answer
28
29
30
31     return 0;
32 }
```

Code Completion

```
34      // write the function call to introduce
35      // so that the 3 Persons can introduce
36      // themselves
37
38      introduce(&p1);
39      introduce(p2);
40      introduce(p3);
41
42      return 0;
43  }
44
45
46  void introduce(Person *p) {
47      printf("My name is %s. I am %d years old.\n", p->name, p->age);
48  }
```

→ array notation

or `introduce(&p3[0]);`

→ expects an address

Notes

- What if for **`p3`**, we want to set the name and age of the second Person in the array?

- How will that affect the assignment statements?

- How about the call to the **`introduce()`** function?

Code Completion (Array Notation)

```
26      // create an array of Persons with size 10
27      // set the name and age of the second Person
28      // to Bob & 30 ; assign the array to p3
29      p3 = malloc(sizeof(Person) * 10);
30
31      strcpy(p3[1].name , "Bob");
32
33      p3[1].age = 30;
34
35      // write the function call to introduce
36      // so that the 3 Persons can introduce
37      // themselves
38
39      introduce(&p3[1]);
40
41
42
```

Code Completion (Pointer Notation) /1

```
26    // create an array of Persons with size 10
27    // set the name and age of the second Person
28    // to Bob & 30 ; assign the array to p3
29    p3 = malloc(sizeof(Person) * 10);
30
31
32
33
34
35    // write the function call to introduce
36    // so that the 3 Persons can introduce
37    // themselves
38
39
40
41
42
```

strcpy((p3+1)→name, "Bob");

(p3+1)→age = 30;

introduce((p3+1));

↳ this is the address of the array plus offset of 1 (so index 1), which is still an address

Code Completion (Pointer Notation) /2

```
26    // create an array of Persons with size 10
27    // set the name and age of the second Person
28    // to Bob & 30 ; assign the array to p3
29    p3 = malloc(sizeof(Person) * 10);
30
31
32
33
34
35    // write the function call to introduce
36    // so that the 3 Persons can introduce
37    // themselves
38
39
40
41
42
```

strcpy(*(p3+1).name, "Bob");

*(p3+1).age = 30;

} notice if you want to use the dereference operator instead!

introduce((p3+1));

↳ this is the address of the array plus offset of 1 (so index 1), which is still an address

Structs with Dynamic Members

- Previously we learned about defining structures
- We defined the Person struct which keeps track of a person's name and age
- However, because the name is a string (array of `char`), we need to specify its size

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define WORD_SIZE 256

typedef struct Person_s {
    char name[WORD_SIZE];
    int age;
} Person;
```

Discussion

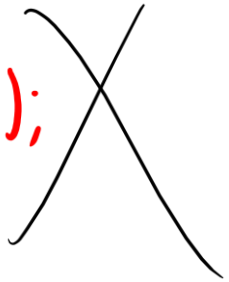
- Recall that a **string** is an array of characters
- Now that we can create an array dynamically, what do you think is the implication?
- Can we now relax our assumptions about the length of the input strings?

Code Completion

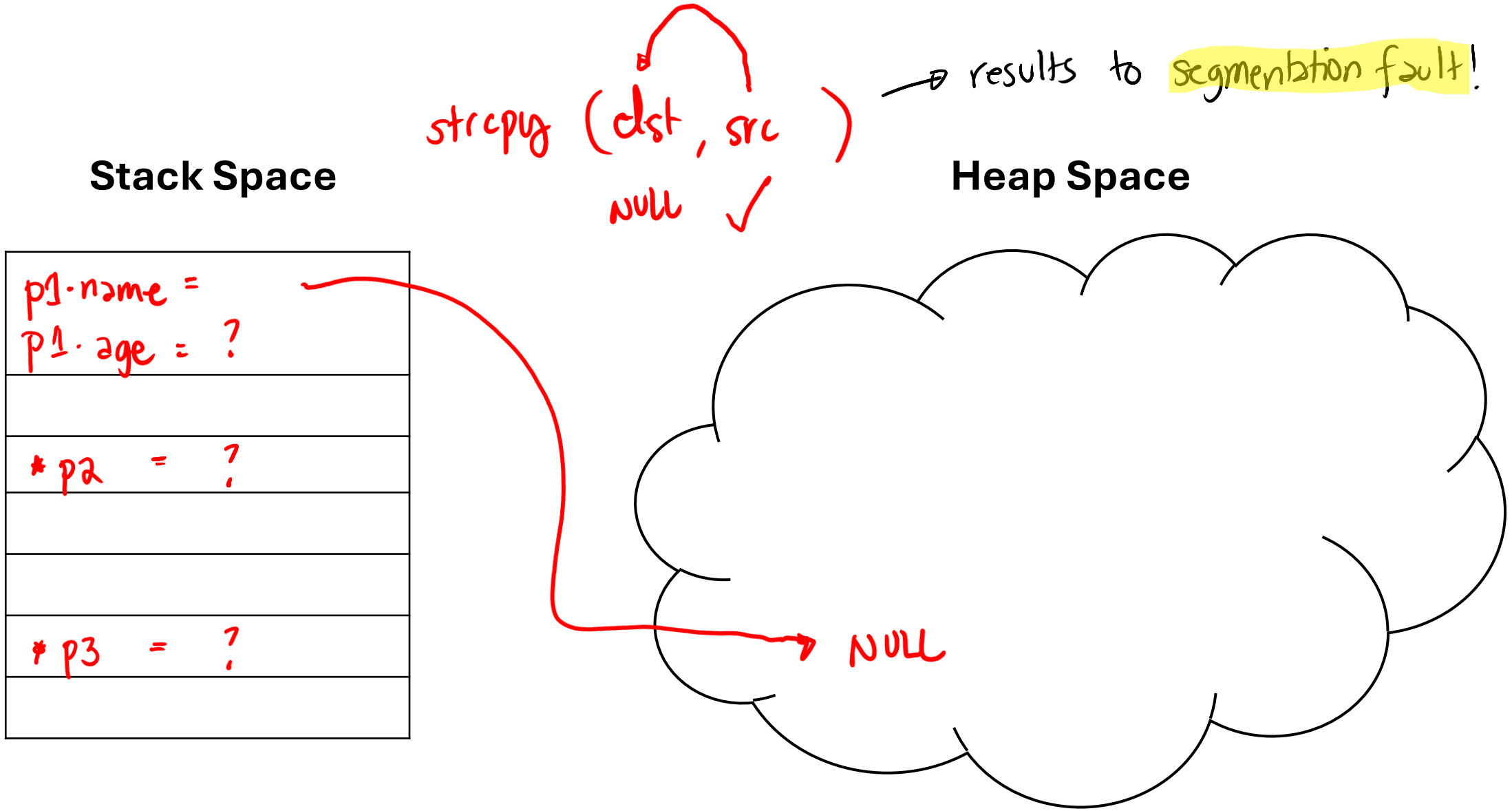
// wrong answer 1

strcpy(p1.name, "John");

p1.age = 20;



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

typedef struct Person_s {
    char *name;
    int age;
} Person;

int main(void) {
    Person p1;

    // set name and age of p1 to John & 20

    // see wrong answer 1

    Person *p2, *p3;

    // set name and age of p2 to Jane & 25

    return 0;
}
```


strcpy (dst, src)
NULL ✓
→ results to segmentation fault!
Stack Space
Heap Space
p1.name =
p1.age = ?
*p2 = ?
*p3 = ?
NULL

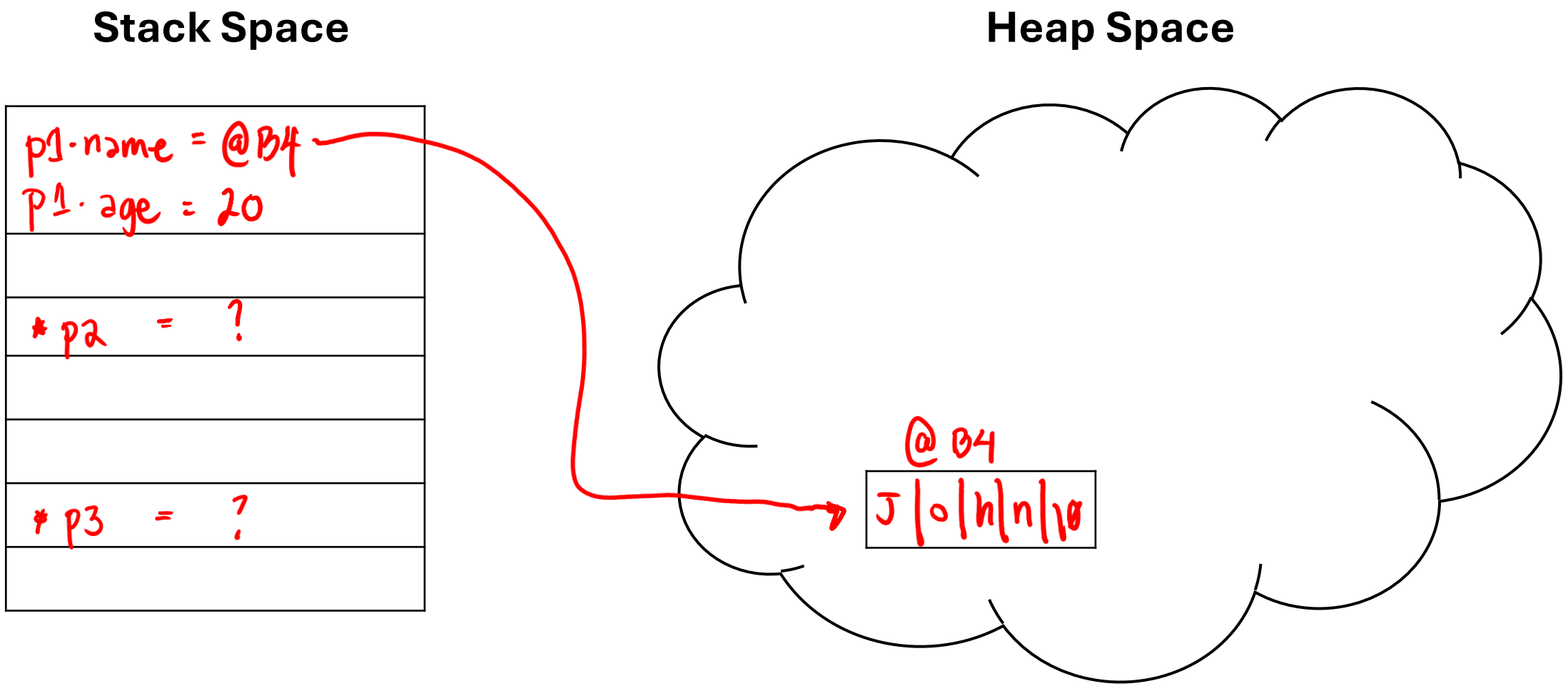
Code Completion

// answer 1

```
p1.name = malloc(sizeof(char)*256);
strcpy(p1.name, "John");
p1.age = 20;
```

we just assume that the max length is 255 the extra 1 is for the null character

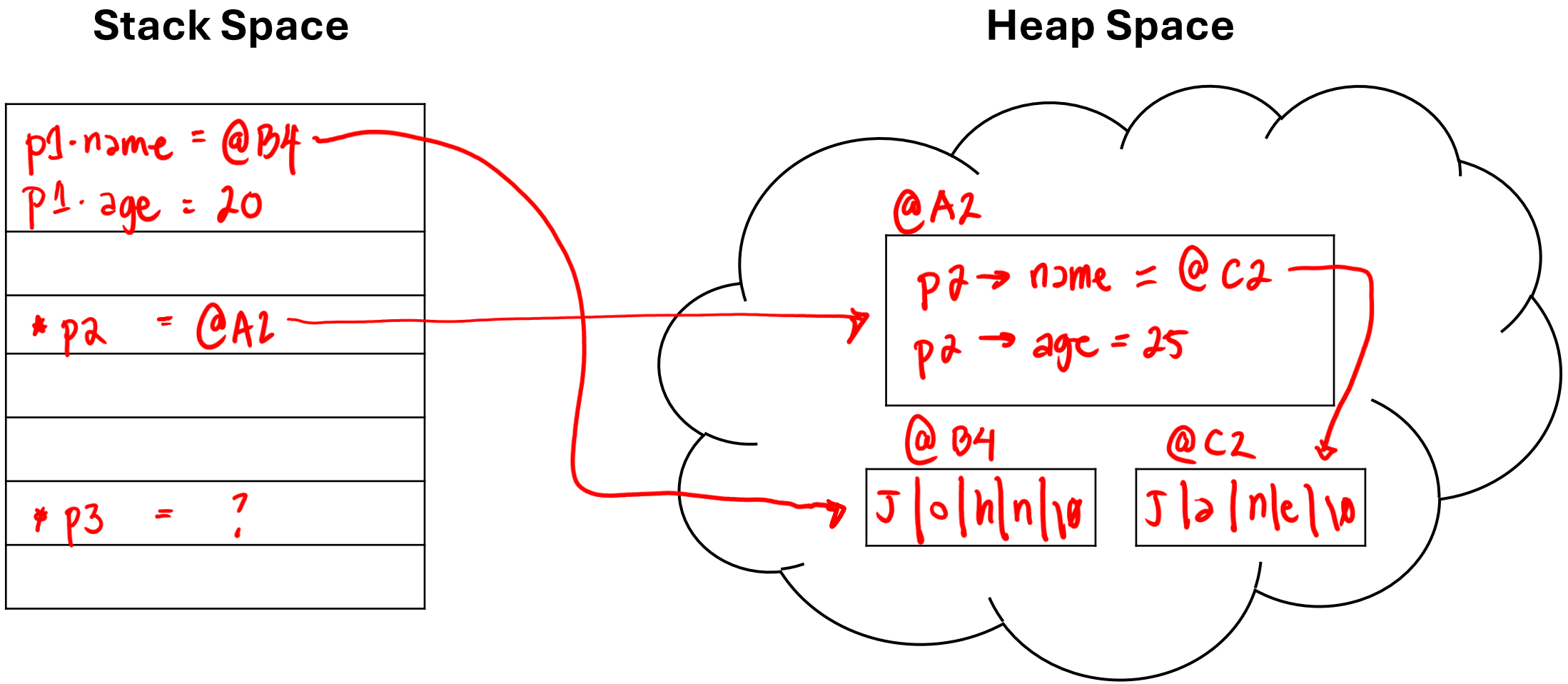
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct Person_s {
    char *name;
    int age;
} Person;

int main(void) {
    Person p1;

    // set name and age of p1 to John & 20
    // see answer 1

    Person *p2, *p3;

    // set name and age of p2 to Jane & 25

    return 0;
}
```


Stack Space
Heap Space
p1.name = @B4
p1.age = 20
*p2 = ?
*p3 = ?
@04
J|o|h|n|\0

Code Completion

```
// answer 1
p1.name = malloc(sizeof(char)*256));
strcpy(p1.name, "John");
p1.age = 20;

// answer 2
p2 = malloc(sizeof(Person));
p2->name = malloc(sizeof(char)*256));
strcpy(p2->name, "Jane");
p2->age = 25;
```

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

typedef struct Person_s {
    char *name;
    int age;
} Person;

int main(void) {
    Person p1;

    // set name and age of p1 to John & 20
    // see answer 1

    Person *p2, *p3;

    // set name and age of p2 to Jane & 25
    // see answer 2

    // notice that we also need to dynamically
    // allocate name for p2 as well

    return 0;
}
```


Stack Space
Heap Space
p1.name = @B4
p1.age = 20
* p2 = @A2
* p3 = ?
@A2
p2 → name = @C2
p2 → age = 25
@B4
J|o|h|n|\0
@C2
J|a|n|e|\0

Your Turn!

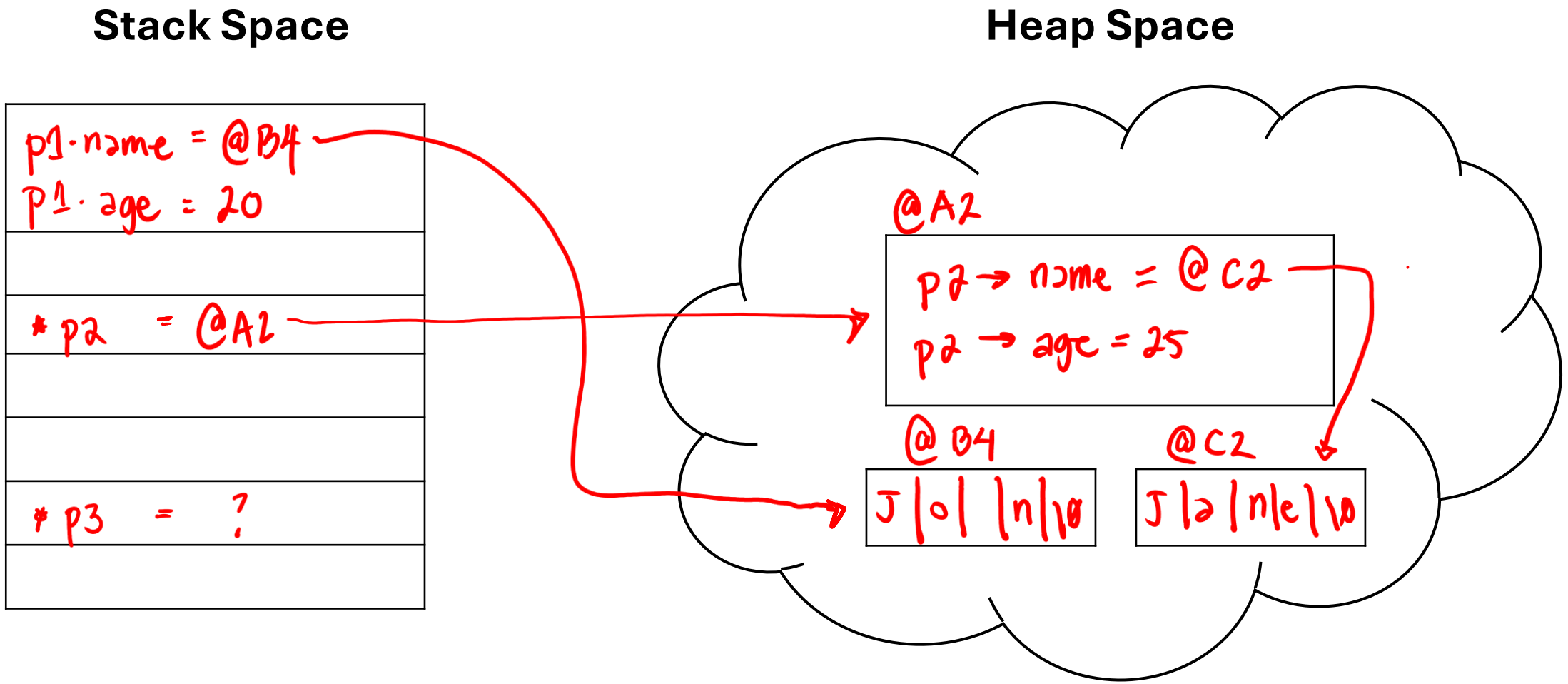
- Similar to the previous slides, dynamically allocate a Person variable and set the attributes

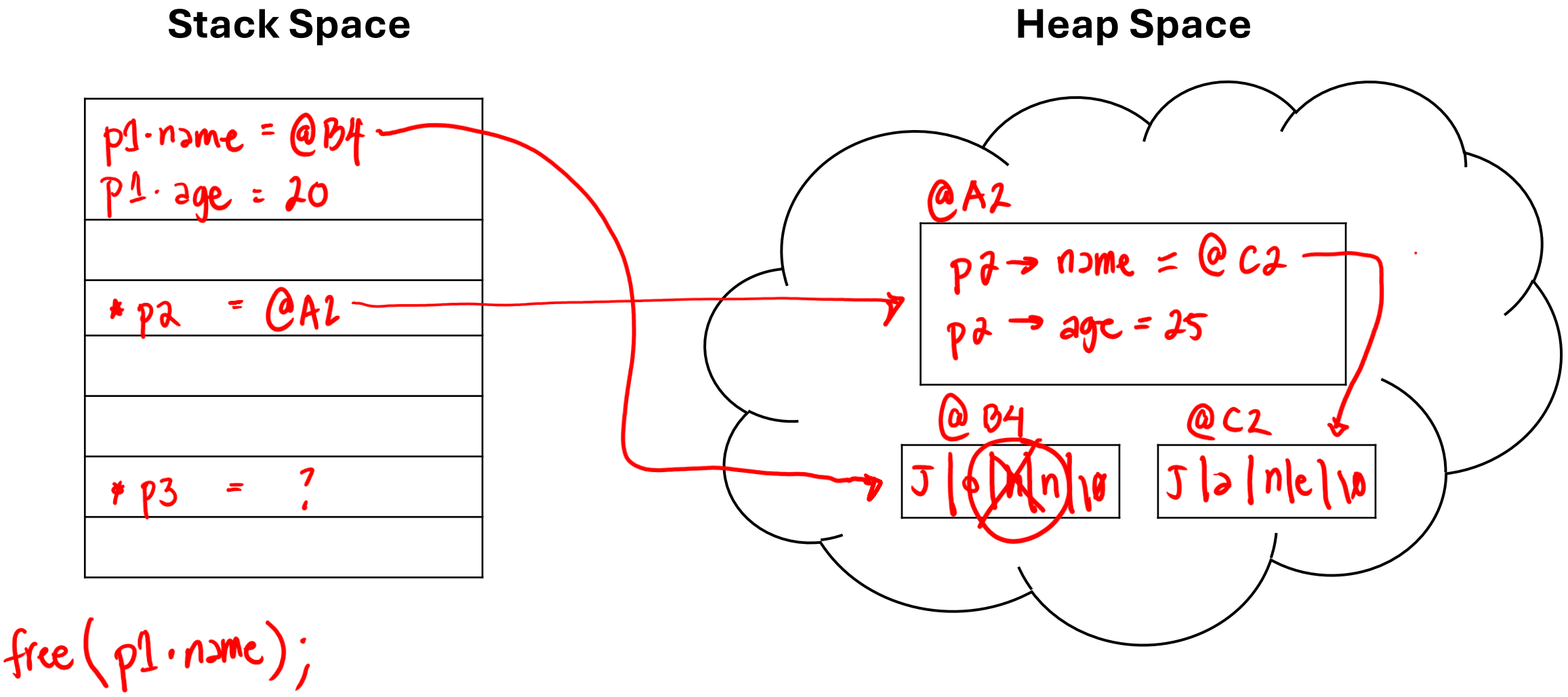
- However, this time, get the information from the user instead of hard-coding the values

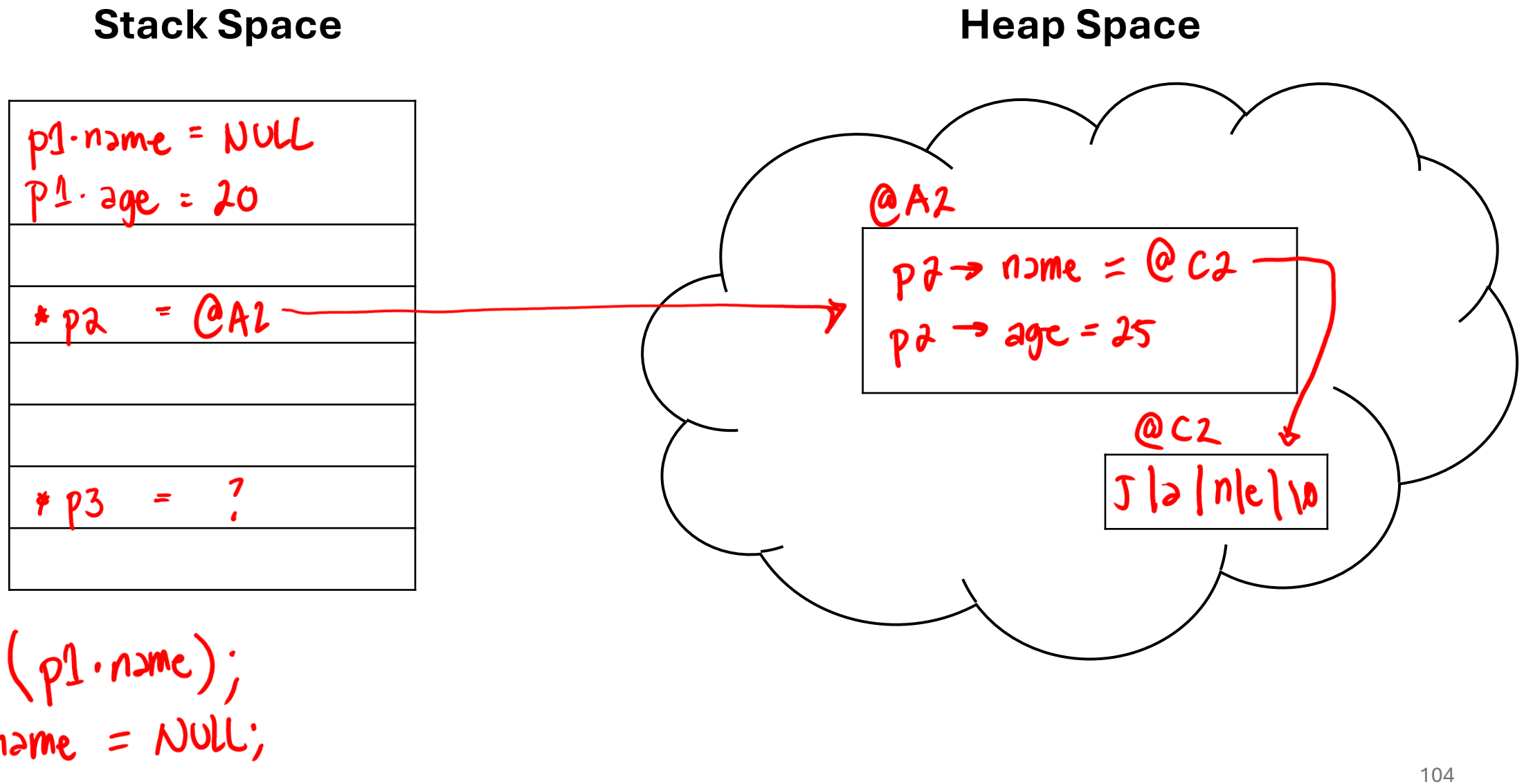
Deallocating Memory

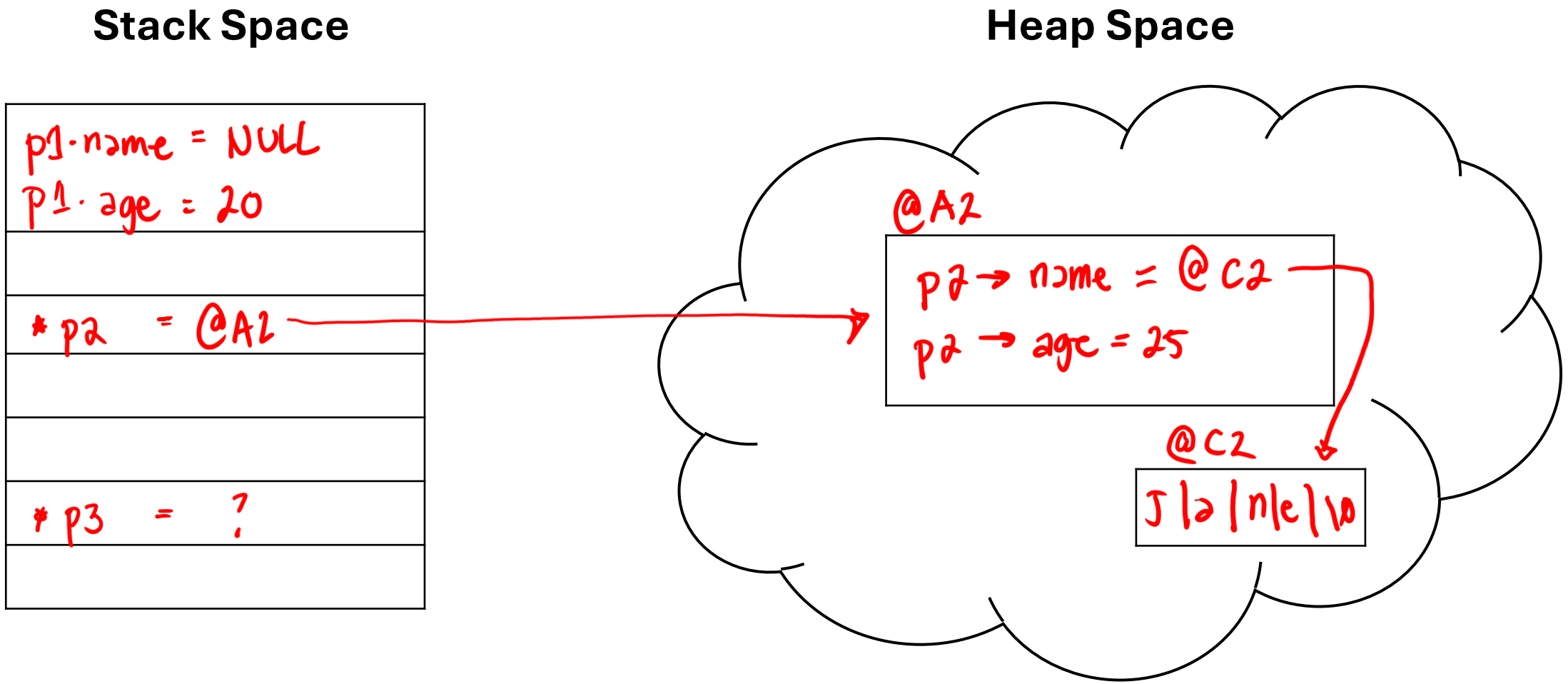
- How do we deallocate the space used by **`p2`**?

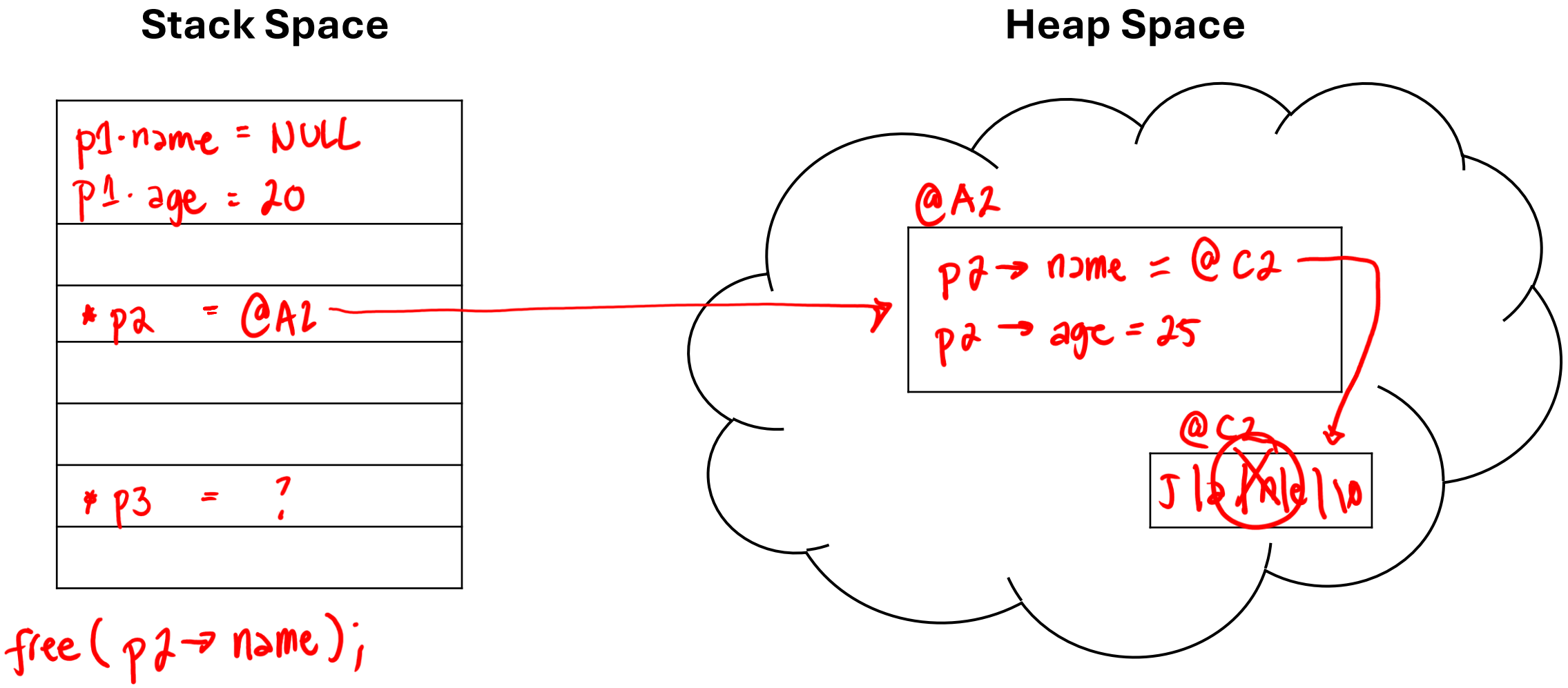
- Notice that even if **`p1`** was not dynamically allocated, it has a member (or field) that was dynamically allocated?

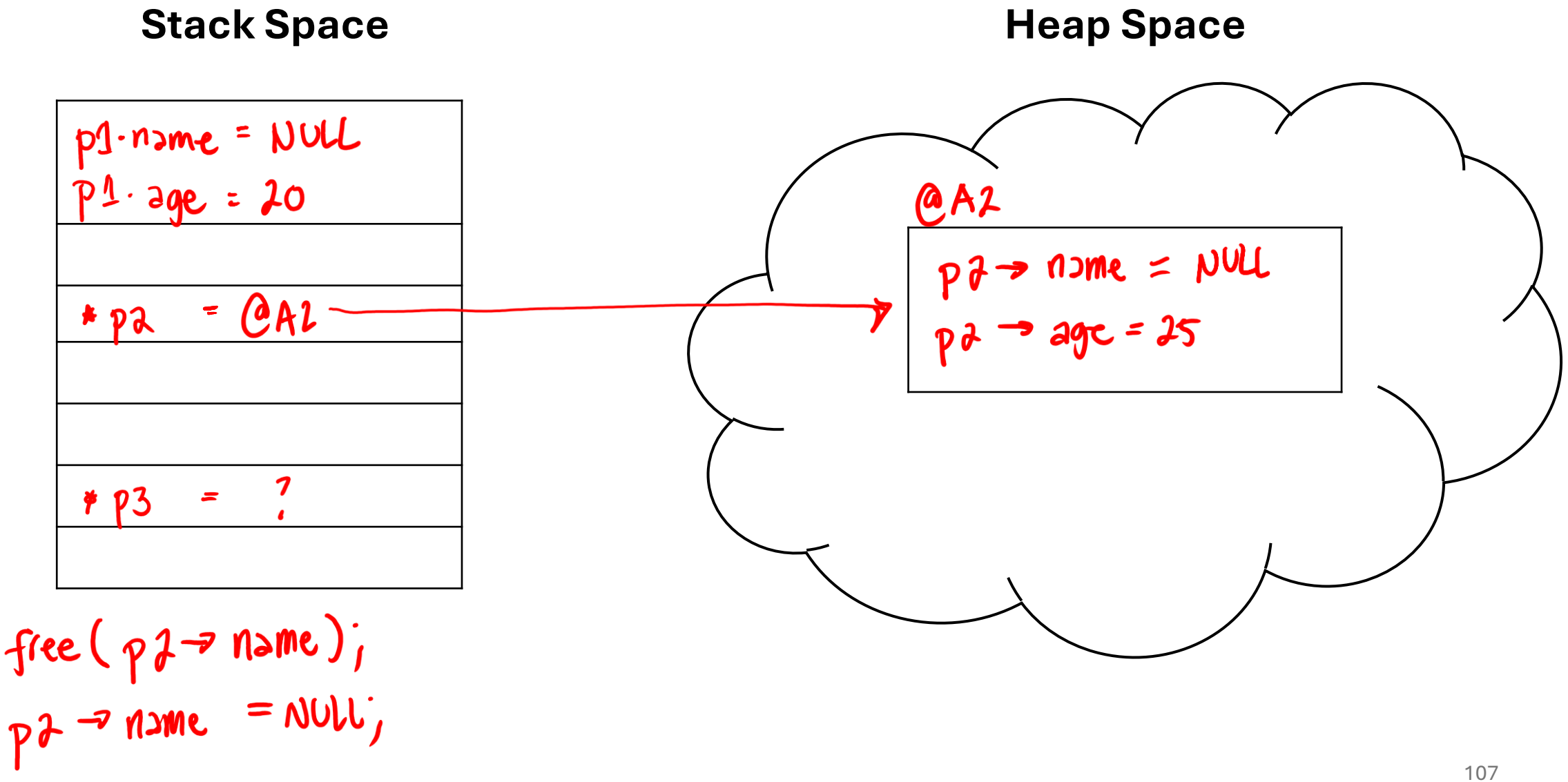

Stack Space
Heap Space
p1.name = @B4
p1.age = 20
* p2 = @A2
* p3 = ?
@A2
p2 → name = @C2
p2 → age = 25
@B4
J|o| |n|\0
@C2
J|a|n|e|\0

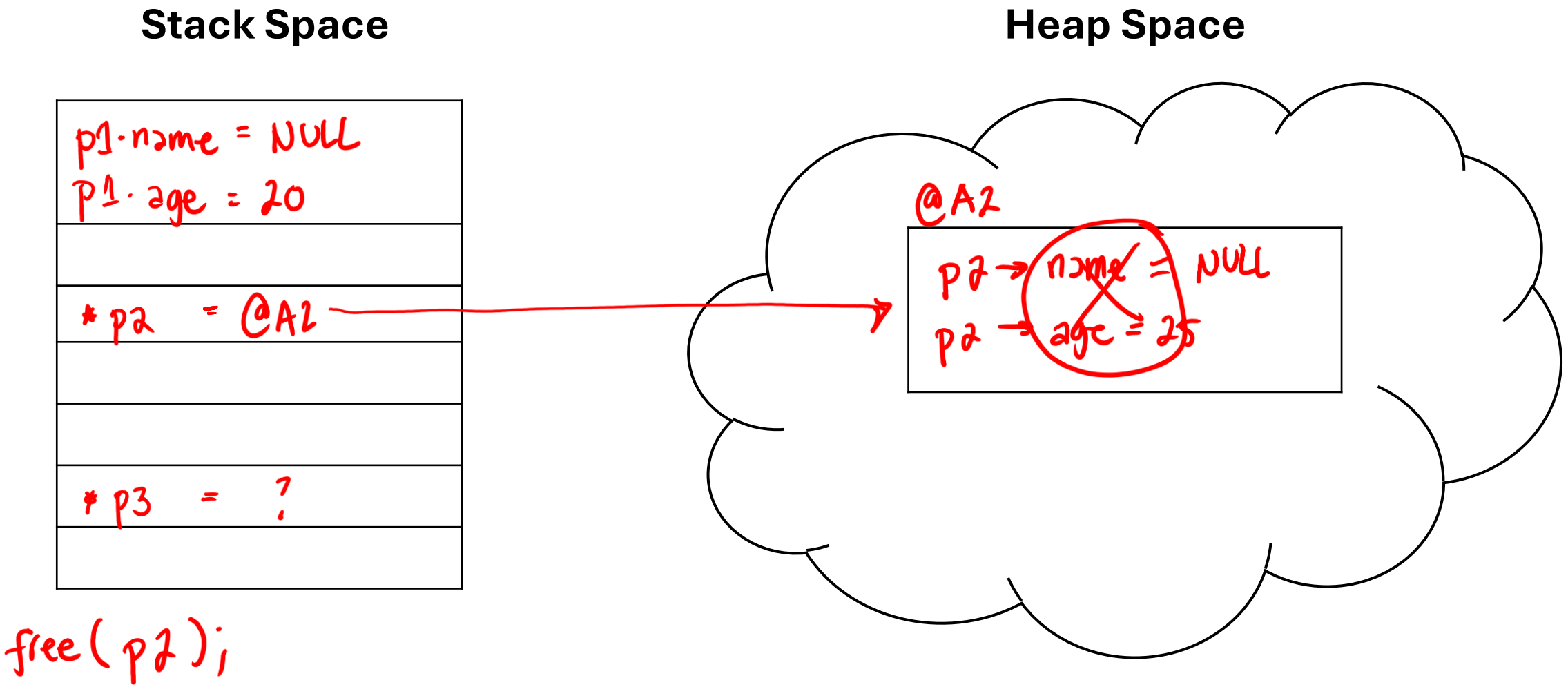

Stack Space
Heap Space
p1.name = @B4
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2→name = @C2
p2→age = 25
@B4
@C2
J|o|h|n|\0
J|a|n|e|\0
free(p1.name);


Stack Space
Heap Space
p1.name = NULL
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2 → name = @C2
p2 → age = 25
@C2
J|a|n|e|\0
free(p1.name);
p1.name = NULL;


Stack Space
Heap Space
p1.name = NULL
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2 → name = @C2
p2 → age = 25
@C2
J|a|n|e|\0


Stack Space
Heap Space
p1.name = NULL
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2 → name = @C2
p2 → age = 25
@C2
J l e l o
free(p2→name);


Stack Space
Heap Space
p1.name = NULL
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2→name = NULL
p2→age = 25
free(p2→name);
p2→name = NULL;


Stack Space
Heap Space
p1.name = NULL
p1.age = 20
*p2 = @A2
*p3 = ?
@A2
p2 → name = NULL
p2 → age = 25
free(p2);

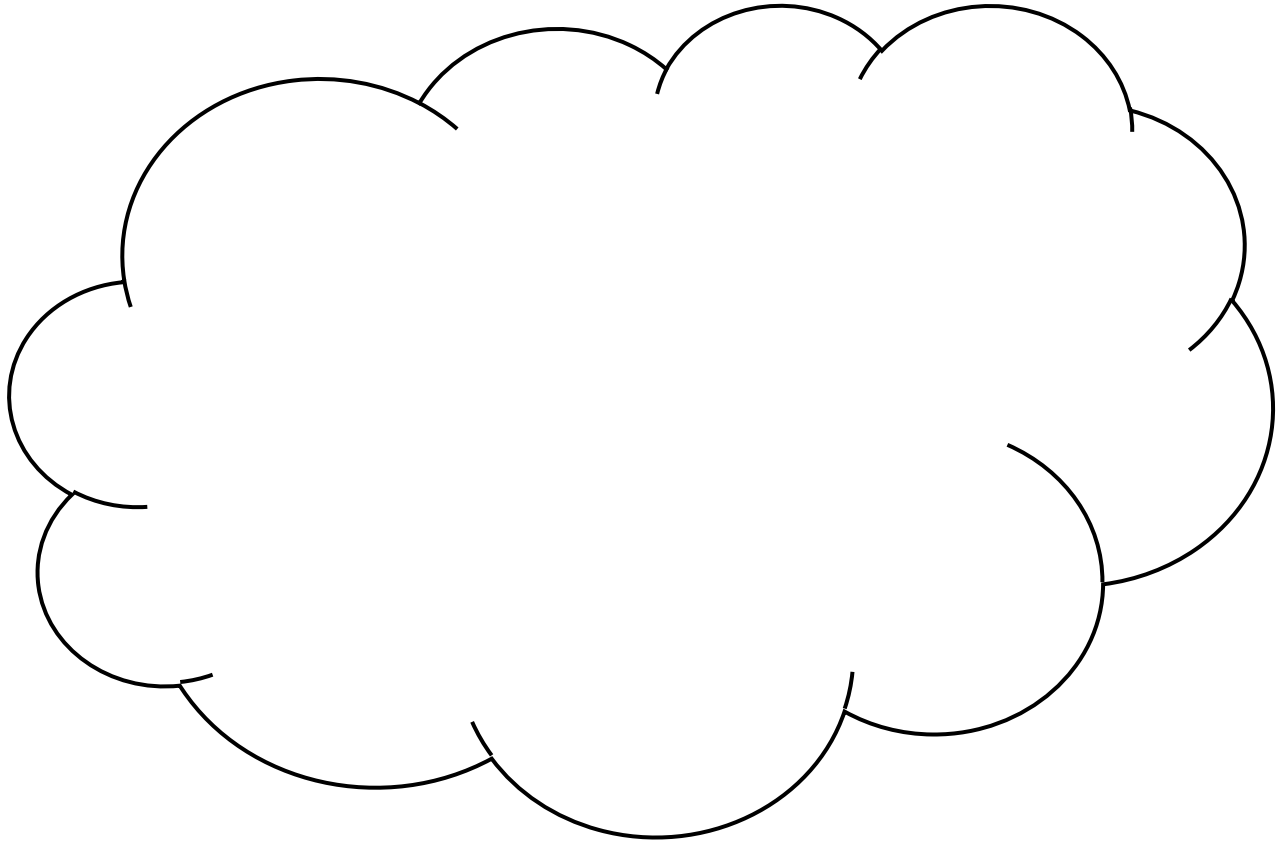
Stack Space

p1.name = NULL p1.age = 20
*p2 = NULL
*p3 = ?

free(p2);

p2 = NULL;

Heap Space



Discussion /1

- If our struct has a dynamic member, then, you should also allocate space to it
- Otherwise, you will end up with a **segmentation fault**
- Additionally, this dynamic member needs to be deallocated once you are done with it

Discussion /2

- This is where your constructors and destructors will come handy
- For now, we didn’t see the benefit of dynamically allocating a string because we still had to specify the size when we called **`malloc()`**
- However, think of it this way, what if you want to define a function that automatically sets the Person’s name and age based on the parameters it received

Your Turn! /1

Define a function **`create_person(n, a)`** that dynamically allocates a single Person. It sets the name and the age based on the arguments it received. Afterward, return a pointer to this Person.

```
Person *create_person(char *n, int a) {



}
```

Your Turn! /2

Define a function **`destroy_person(p)`** that deallocates all dynamic memory allocated to a Person. It accepts a pointer to a Person.

```
void destroy_person(Person *p) {




}
```

Structures with Dynamic Components

Given the following, how do you set the **`name`** of a **`Person`**?

```
typedef struct Person_s {
    char *name;
    int age;
} Person;
```

Dynamically Allocating Structures

- In this course, the structures we will be dealing with will mostly be dynamically allocated

- Why? We want to pass data efficiently!

Visualizing /1

How do you set the age of the 0-th element?

A dynamically allocated array of structs

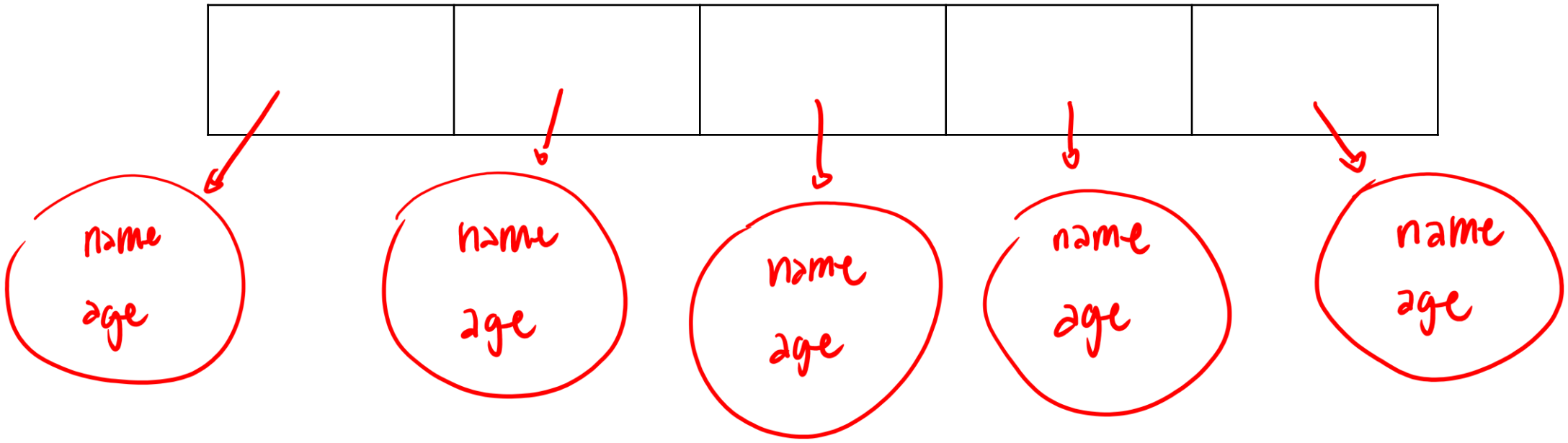
```
Person *p = malloc(sizeof(Person) * 5);
```

Visualizing /2

How do you set the age of the 0-th element?

- A dynamically allocated array of pointers to structs

```
Person **p = malloc(sizeof(Person*) * 5);
```



Final Notes /1

- We have seen that it was possible to dynamically allocate memory during **runtime**.

- A good use case involves dynamically allocating an array where the size cannot be determined during **compile time**, unless we impose certain assumptions.

Final Notes /2

- However, it is still possible for it to run out of space. For example, you allocated memory so that it can hold 10 elements. Later on, you realized that you needed to store 15 elements.

- Recall, once an array is created, it cannot “grow” anymore. Therefore, we ran out of space.

- How do we proceed then?

Final Notes /3

- We can potentially create a bigger array and transfer all the elements from the old one (smaller) to the new one (bigger).
- We deallocate the small array and maintain only the big array.
- This behavior that you need to implement behind the scenes gives rise to what we call as **dynamic array**.

Final Notes /4

- There are also other functions that you can use instead such as **`realloc()`** which could “resize” your dynamic memory.

- Additionally, it is important to note that dynamically allocated memory space must eventually be deallocated (or freed), otherwise you will end up with a **memory leak**.

Final Notes /5

- One of the best practices to ensure this is prevented is to define constructor and destructor helper functions that perform these housekeeping operations.

- Lastly, when dealing with structs that have dynamic members, you must also ensure that these are deallocated first.

Questions?