C programming Language

Chapter 3

2. Dynamic Memory Allocation (DMA)

What is Dynamic Memory Allocation?

- **The problem:**
  Array definition: its size must be known at compilation time. The array, when used, may be either too small – not enough room for all the elements, or too big – so it's a waste of memory.
- **The solution:**
  Use Dynamic Memory Allocation (DMA):
  create the array at run-time, after determining the required number of elements.
- Dynamic memory allocation enables the programmer to:
  - Request exactly the required amount of memory.
  - Release the allocated memory when it is no longer needed.

malloc Function

- **malloc** function enables to allocate memory at run-time.
- Syntax:

```
malloc(number of requested bytes);
```

- Example:

```
int size;
printf(“how many integers?”);
scanf(“%d”, &size);
malloc(size*sizeof(int));
```

- **Comment: All the allocation functions use the <stdlib.h> library.**

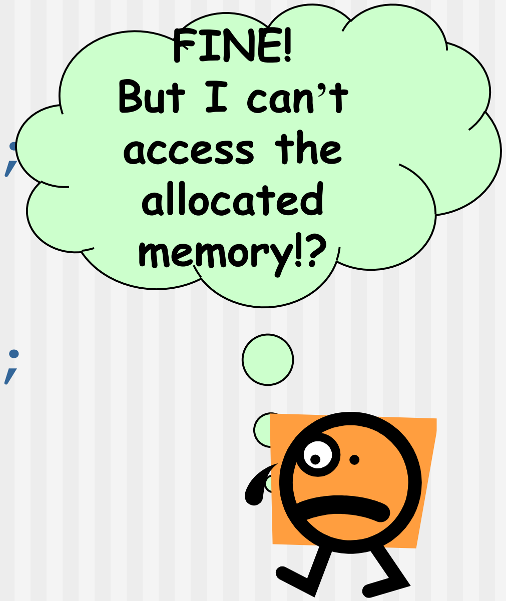
calloc Function

- **calloc** function enables to allocate memory at run time.
  - The allocated memory is initialized (cleared) with zeros.
- Syntax:

```
calloc(number of elements,
          size of each element);
```

Example:

```
int size;
printf(“how many integers?”);
scanf(“%d”, &size);
calloc(size, sizeof(int));
```



Allocation

- malloc and calloc return the first address of the allocated memory. Upon failure, they return NULL (0).
- We can save the return value in a pointer, and access the memory by using the operator * or operator [].
- For instance:

```
int size;
int *pointer;
printf(“how many integers?”);
scanf(“%d”, &size);
pointer = malloc(size*sizeof(int));
if(pointer != NULL)
      pointer[0] = 54;
```

DON’T WORRY!!! Pointers will help you!

Allocation

```
pointer = malloc(size*sizeof(int));
```

Problem: What type does malloc return?
int *, char *, float * ???

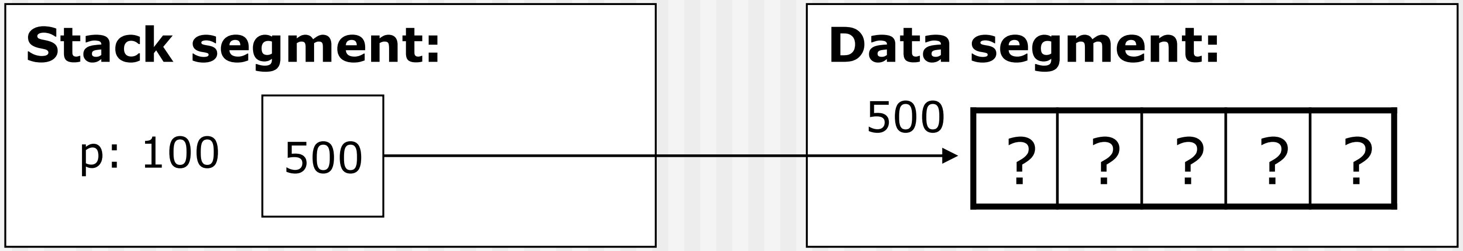
Answer: void * !!! void * may be changed to any pointer type by using casting.

```
pointer = (int *) malloc(size*sizeof(int));
```

Heap

- Where is the memory allocated?
- Reminder: we studied about the “stack segment” and “data segment”. Stack segment is dedicated to local variables. Allocated memory is not a local variable.
- Conclusion: dynamic memory is allocated in the data segment (heap).

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

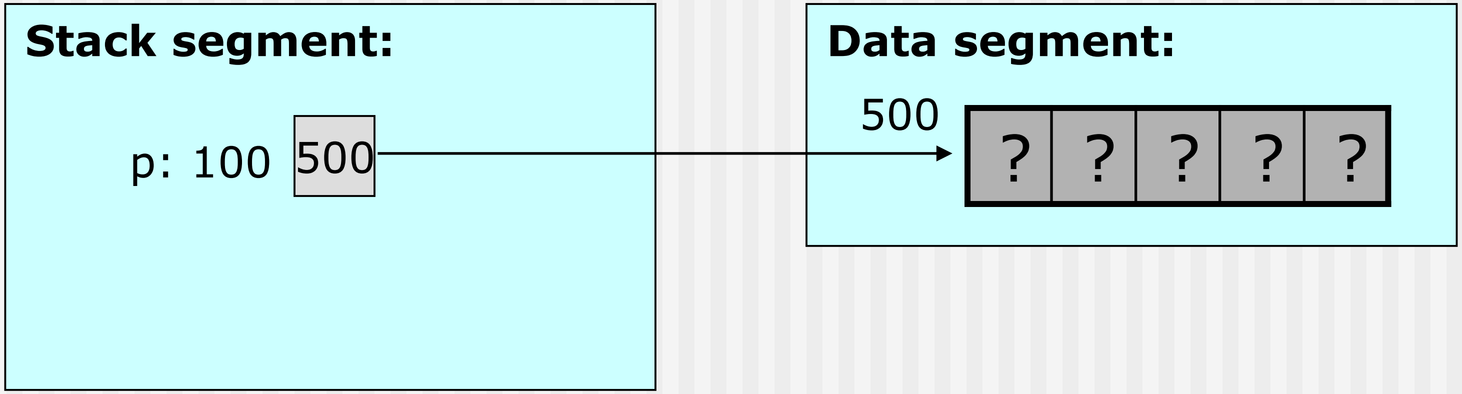


Comment

BE CAREFUL:

A pointer that points to allocated memory can be mistakenly assigned to another memory address. In this case, we might lose our connection to the previously allocated memory.

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

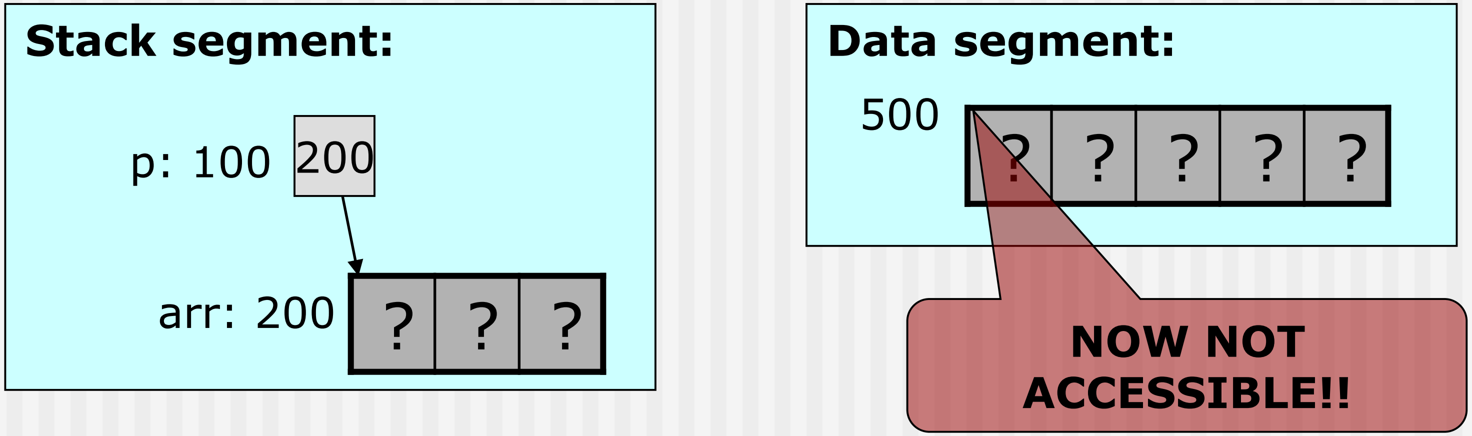


Comment

BE CAREFUL:

A pointer that points to allocated memory can be mistakenly assigned to another memory address. In this case, we might lose our connection to the previously allocated memory.

```
int *p = (int *) malloc(5*sizeof(int));
int arr[3];
p = arr;
```

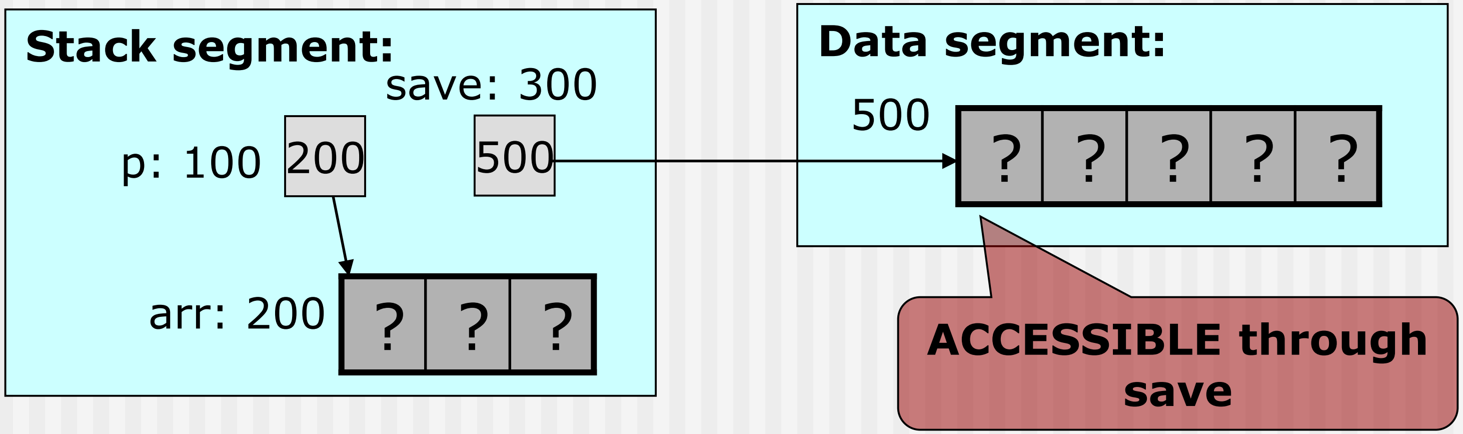


NOW NOT ACCESSIBLE!!

Comment

To avoid this problem, we can first save the address in another pointer:

```
int *save, *p = (int *) malloc(5*sizeof(int));
int arr[3];
save = p;
p = arr;
```



free Function

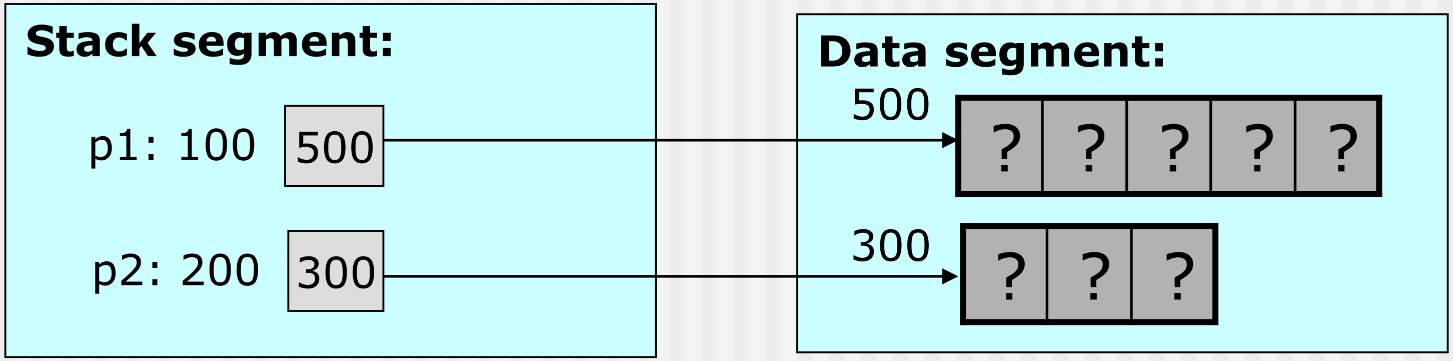
- Since dynamic memory is allocated in the data segment, it is not deleted when the end of a block is reached, but it’s the programmer responsibility to explicitly delete it.
- The free function gets the first address of an allocated memory, and frees it:
  - Syntax:

```
free(first_address);
```

Copyright Meir Kalech

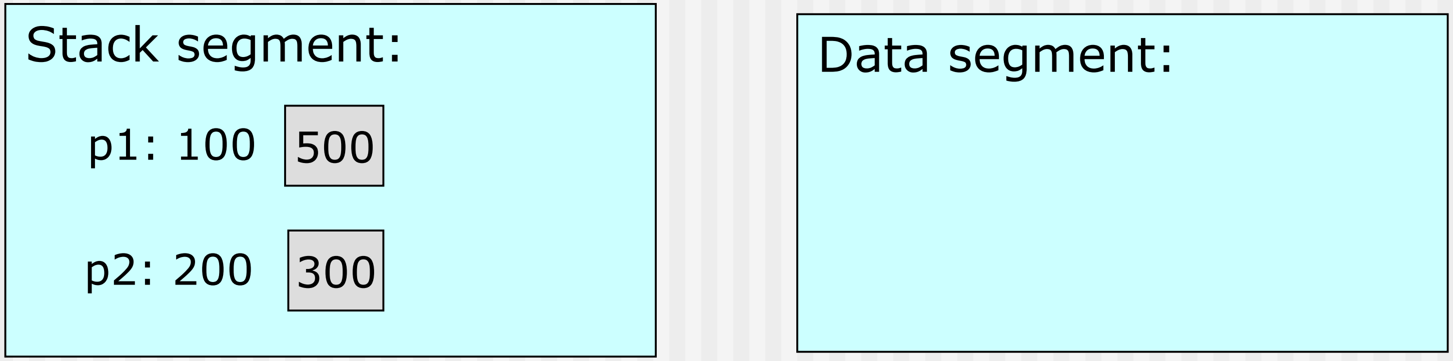
free Function

```
int *p1, *p2;
p1 = (int *) malloc(5*sizeof(int));
p2 = (int *) malloc(3*sizeof(int));
free(p2);
free(p1);
```



free Function

```
int *p1, *p2;
p1 = (int *) malloc(5*sizeof(int));
p2 = (int *) malloc(3*sizeof(int));
free(p2);
free(p1);
```



Stack segment:

p1: 100 500

p2: 200 300

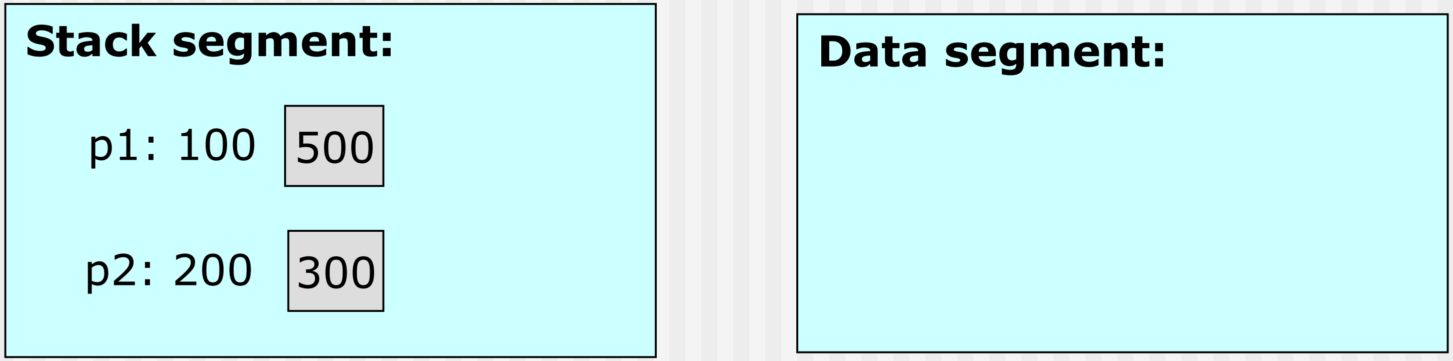
Data segment:

free Function

BE CAREFUL!!!

Avoid freeing memory that is already freed (execution error)

```
int *p1, *p2;
p1 = (int *) malloc(5*sizeof(int));
p2 = (int *) malloc(3*sizeof(int));
free(p2);
free(p1);
free(p1);
```



Stack segment:

p1: 100 500

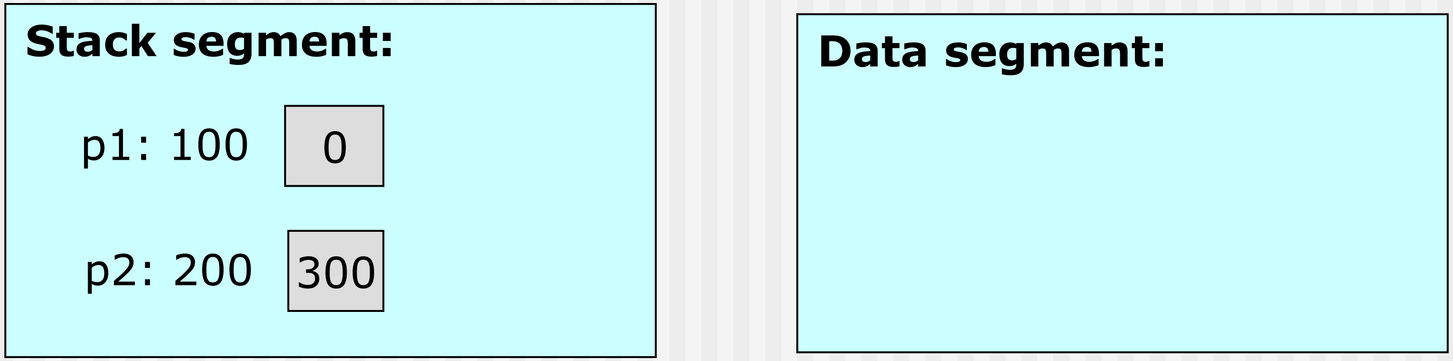
p2: 200 300

Data segment:

free Function

Initialize your pointers with NULL!
Now, there is no BUG.

```
int *p1, *p2
p1 = (int *) malloc(5*sizeof(int));
p2 = (int *) malloc(3*sizeof(int));
free(p2);
free(p1);
p1=NULL;
free(p1);
```



Stack segment:

p1: 100 0

p2: 200 300

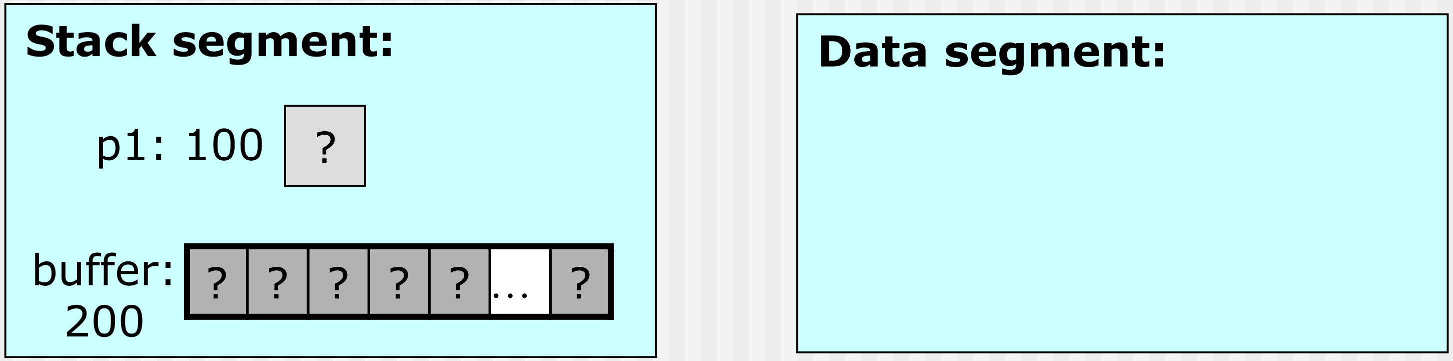
Data segment:

String Allocation

- For array allocation, the programmer gets the size from the user.
- On the other hand, for string allocation, it doesn’t make sense to ask the user for the size, since the meaning of a string size is its number of letters.
- Therefore, it is common to define a buffer (of length fit for a large string) to store the input string and allocate memory according to the size of the input string.

String Allocation

```
char *p1, buffer[30];
printf(“enter string”);
scanf(“%s”, buffer);
p1 = (char *) malloc(strlen(buffer)+1);
strcpy(p1, buffer);
free(p1);
```



Stack segment:

p1: 100 | ?

buffer: 200 | ? | ? | ? | ? | ? | … | ?

Data segment:

String Allocation

Input: “zion”

```
char *p1, buffer[30];
printf(“enter string”);
scanf(“%s”, buffer);
p1 = (char *) malloc(strlen(buffer)+1);
strcpy(p1, buffer);
free(p1);
```

Stack segment:

p1: 100 ?

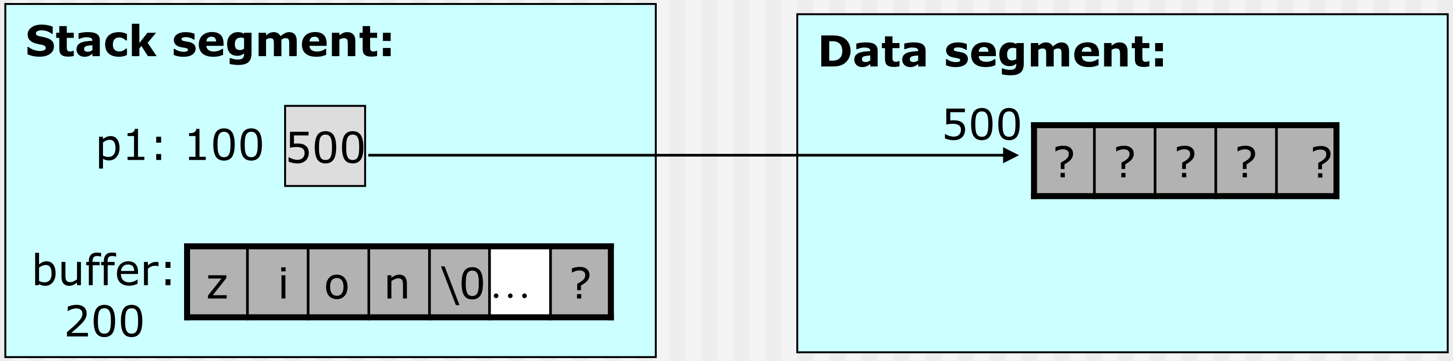
buffer: 200

z	i	o	n	\0	…	?

Data segment:

String Allocation

```
char *p1, buffer[30];
printf(“enter string”);
scanf(“%s”, buffer);
p1 = (char *) malloc(strlen(buffer)+1);
strcpy(p1, buffer);
free(p1);
```



Stack segment:

p1: 100 500

buffer: 200

z | i | o | n | \0 | … | ?

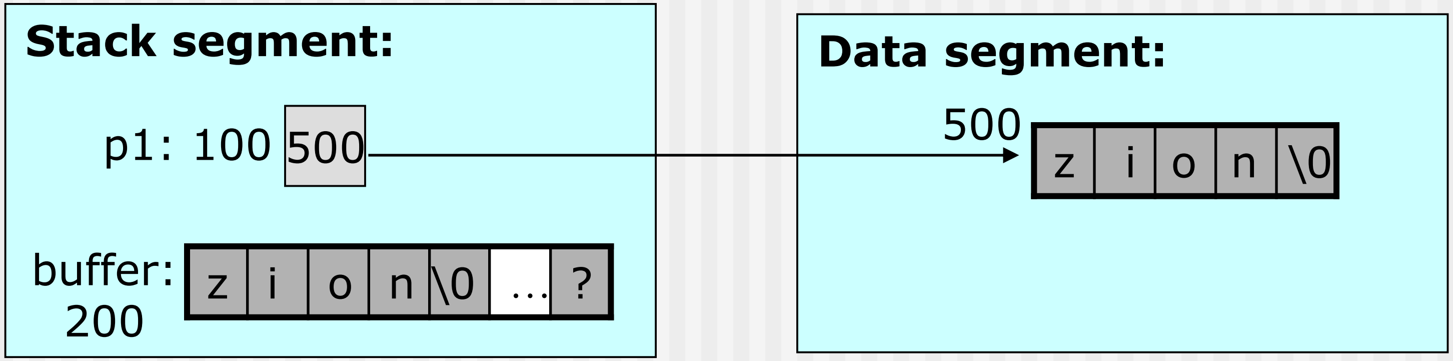
Data segment:

500

? | ? | ? | ? | ?

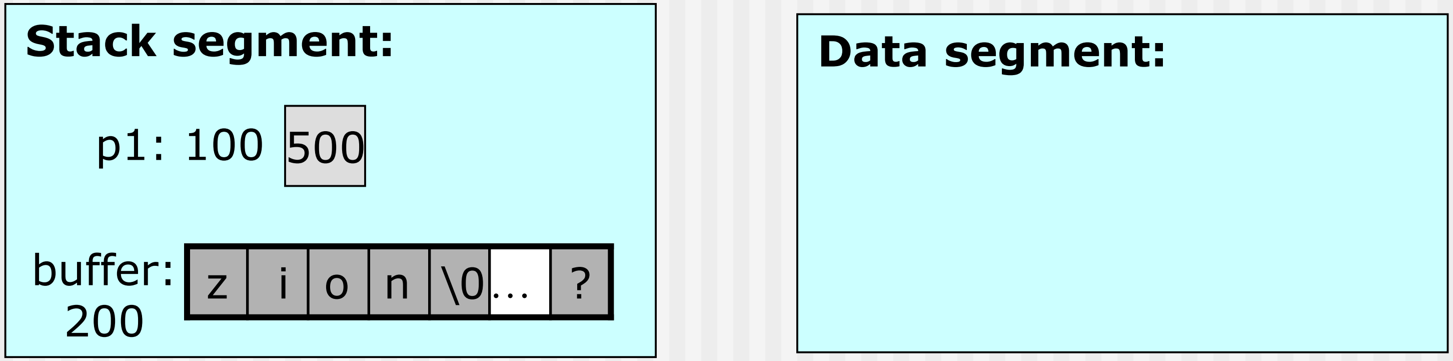
String Allocation

```
char *p1, buffer[30];
printf(“enter string”);
scanf(“%s”, buffer);
p1 = (char *) malloc(strlen(buffer)+1);
strcpy(p1, buffer);
free(p1);
```



String Allocation

```
char *p1, buffer[30];
printf(“enter string”);
scanf(“%s”, buffer);
p1 = (char *) malloc(strlen(buffer)+1);
strcpy(p1, buffer);
free(p1);
```



Stack segment:

p1: 100 | 500

buffer: 200 | z | i | o | n | \0 | … | ?

Data segment:

realloc Function

- One of the goals of dynamic memory allocation is to enable reallocation, namely, increase/decrease the old allocation size.
- There is reallocation function called **realloc**.
- Example:

```
int size, size2;
int *pointer;
printf(“how many integers?”);
scanf(“%d”, &size);
pointer = (int *) malloc(size*sizeof(int));
if(pointer == NULL)
        exit(1);
printf(“how many integers more?”);
scanf(“%d”, &size2);
pointer = (int *) realloc(pointer,(size+size2)*sizeof(int));
if(pointer == NULL)
        exit(1);
```

realloc Function

Some Comments:

1. The new size is the old + additional size.
2. The process:
   - If there is enough space
     - extend the old allocation.
   - Else
     - Allocate new size (old + additional).
     - Copy the old data to the new place.
     - Free the old allocation (should not be used anymore).
   - Return the first address of the allocation.
3. If the first parameter is NULL, then just malloc.

Array of Pointers

- The problem:
  Assume we want to read a list of 3 names.
  The names differ in length but the maximum name length is 23.
- First solution:

```
char arr[3][23];
```

A list of names is an array of strings. Since a string is an array of char, we have a 2D array of char.
Can cause waste of memory!!!

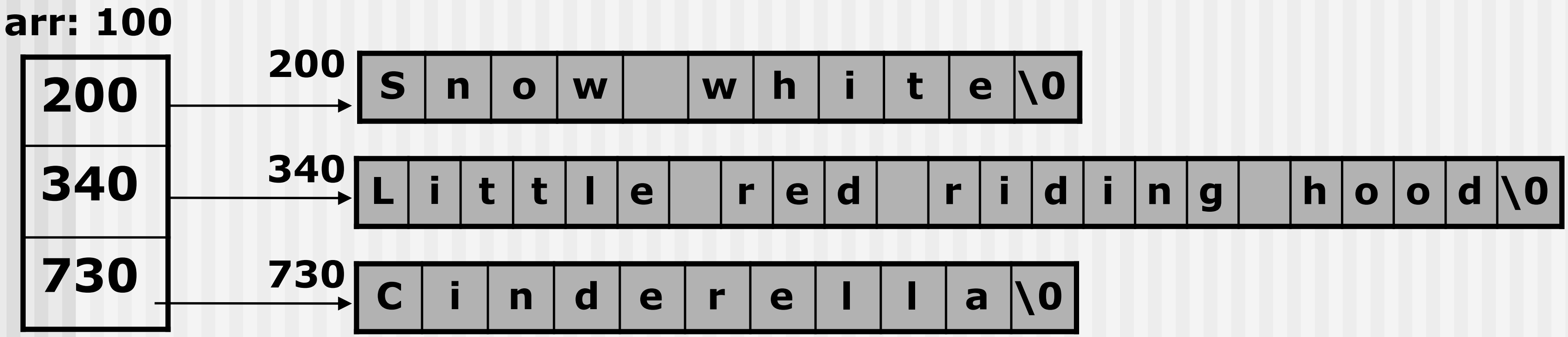
100	S	n	o	w		w	h	i	t	e	\0												
123	L	i	t	t	l	e		r	e	d		r	i	d	i	n	g		h	o	o	d	\0
146	C	i	n	d	e	r	e	l	l	a	\0												

Array of Pointers

- Second solution:

  ```
  char *arr[3];
  ```

  arr is array of 3 elements, where each of them is a pointer to char.
  Now each element can point to a different char array of a different size.



arr: 100

200

340

730

200 | S | n | o | w | | w | h | i | t | e | \0

340 | L | i | t | t | l | e | | r | e | d | | r | i | d | i | n | g | | h | o | o | d | \0

730 | C | i | n | d | e | r | e | l | l | a | \0

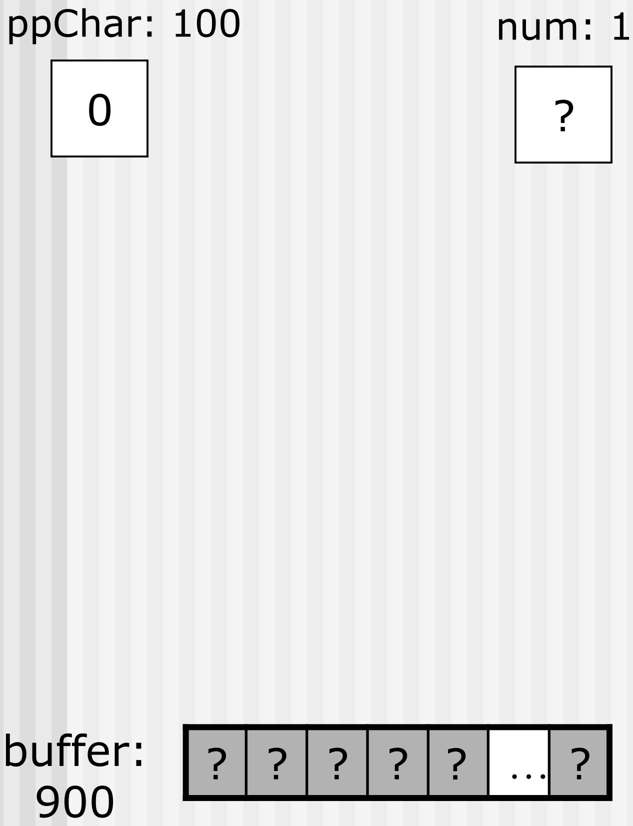
Copyright Meir Kalech

Pointer to Pointer

- Sometimes the number of rows (names) is unknown ahead of time.
- In this case, we should allocate both the rows as well as the elements of each vector:
  1. Allocate array of pointers.
  2. Allocate each pointer in the array.

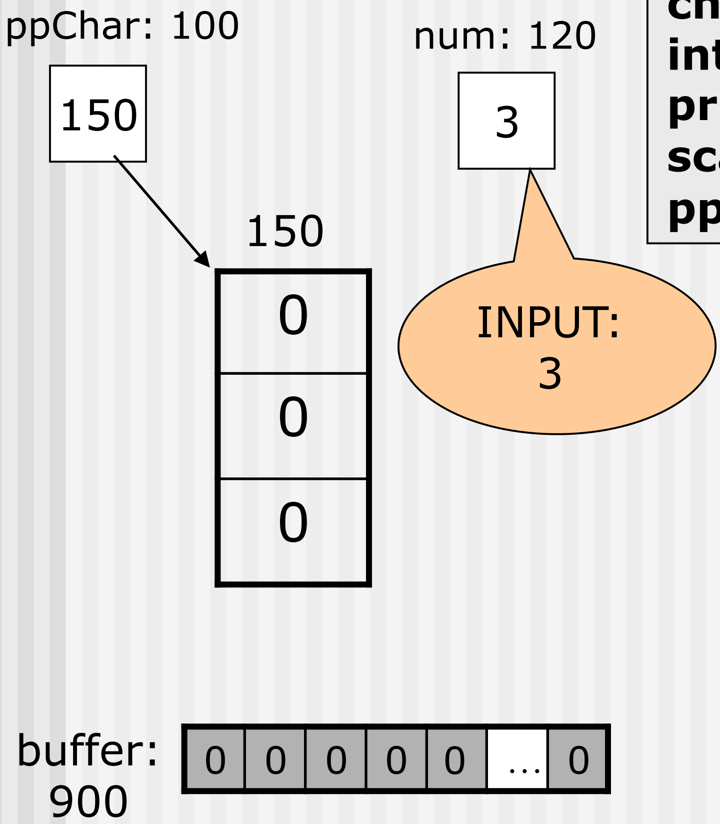
Pointer to Pointer

```
char **ppChar = NULL, buffer[30];
int num;
```

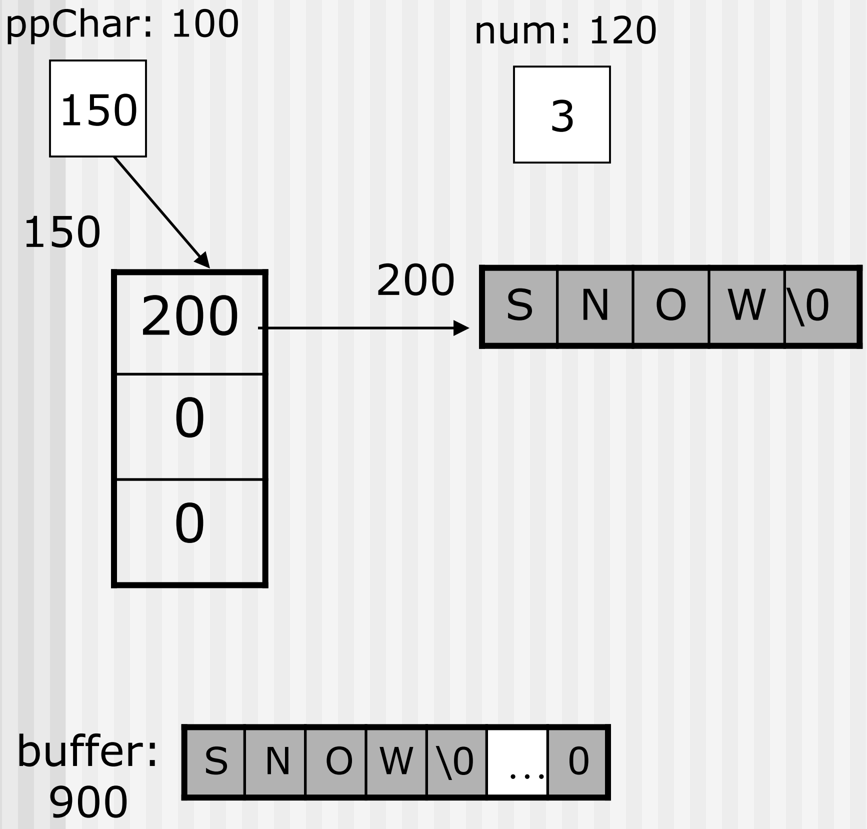


Pointer to Pointer

```
char **ppChar = NULL,  buffer[30];
int num;
printf(“how many names?”);
scanf(“%d”, &num);
ppChar = (char **) calloc(num, sizeof(char *));
```

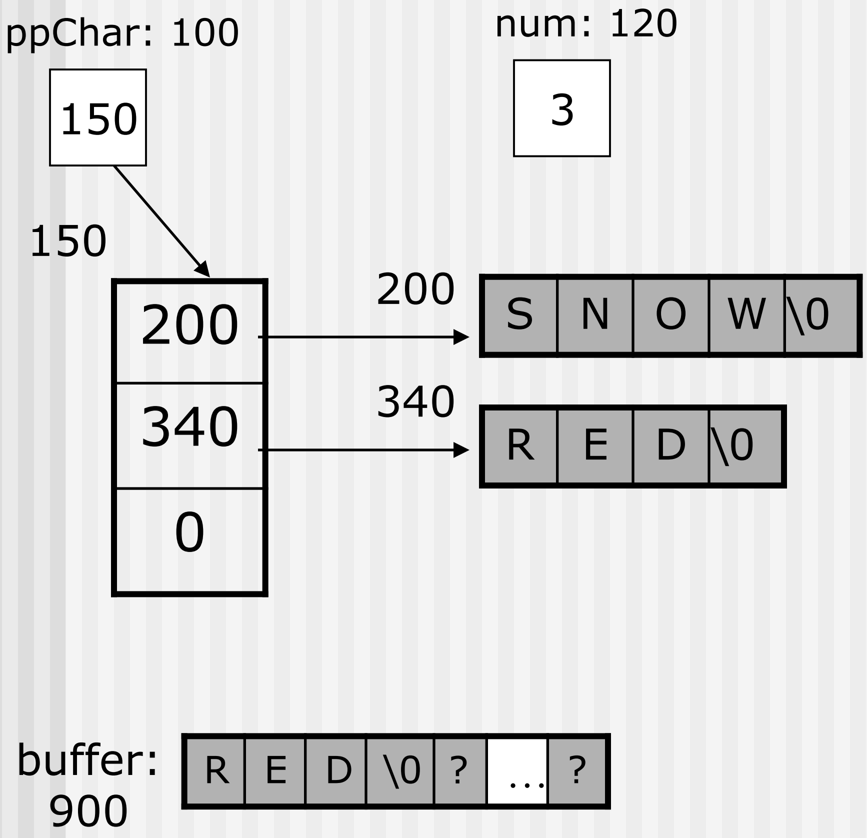


Pointer to Pointer



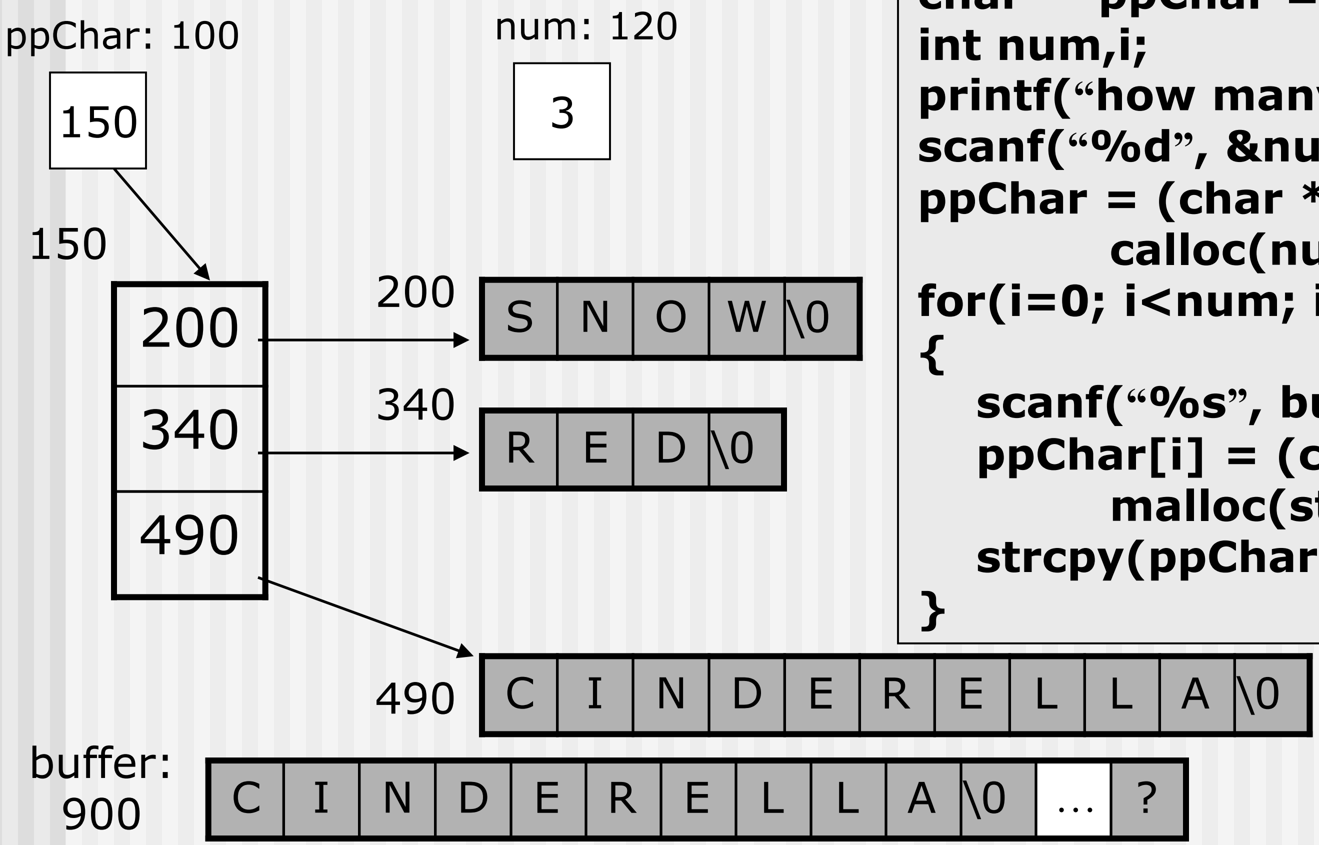
```
char **ppChar = NULL, buffer[30];
int num,i;
printf("how many names?");
scanf("%d", &num);
ppChar = (char **)
          calloc(num, sizeof(char *));
for(i=0; i<num; i++)
{
   scanf("%s", buffer);
   ppChar[i] = (char *)
          malloc(strlen(buffer)+1);
   strcpy(ppChar[i], buffer);
}
```

Pointer to Pointer



```
char **ppChar = NULL, buffer[30];
int num,i;
printf(“how many names?”);
scanf(“%d”, &num);
ppChar = (char **)
          calloc(num, sizeof(char *));
for(i=0; i<num; i++)
{
   scanf(“%s”, buffer);
   ppChar[i] = (char *)
          malloc(strlen(buffer)+1);
   strcpy(ppChar[i], buffer);
}
```

Pointer to Pointer



ppChar: 100

150

num: 120

3

150

200

340

490

200

S N O W \0

340

R E D \0

490

C I N D E R E L L A \0

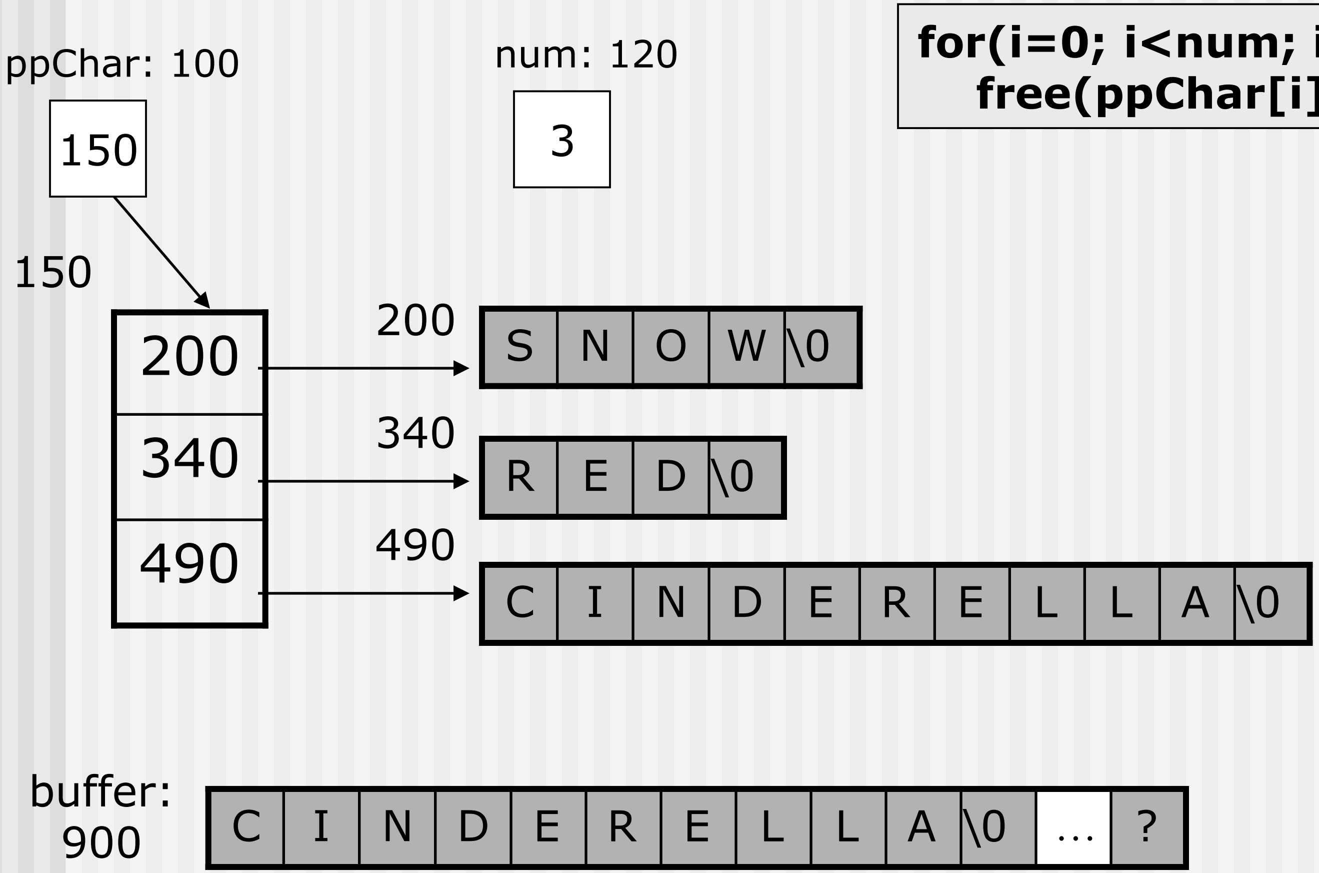
buffer: 900

C I N D E R E L L A \0 … ?

```
char** ppChar = NULL, buffer[30];
int num,i;
printf("how many names?");
scanf("%d", &num);
ppChar = (char **)
            calloc(num, sizeof(char *));
for(i=0; i<num; i++)
{
    scanf("%s", buffer);
    ppChar[i] = (char *)
            malloc(strlen(buffer)+1);
    strcpy(ppChar[i], buffer);
}
```

Pointer to Pointer - free

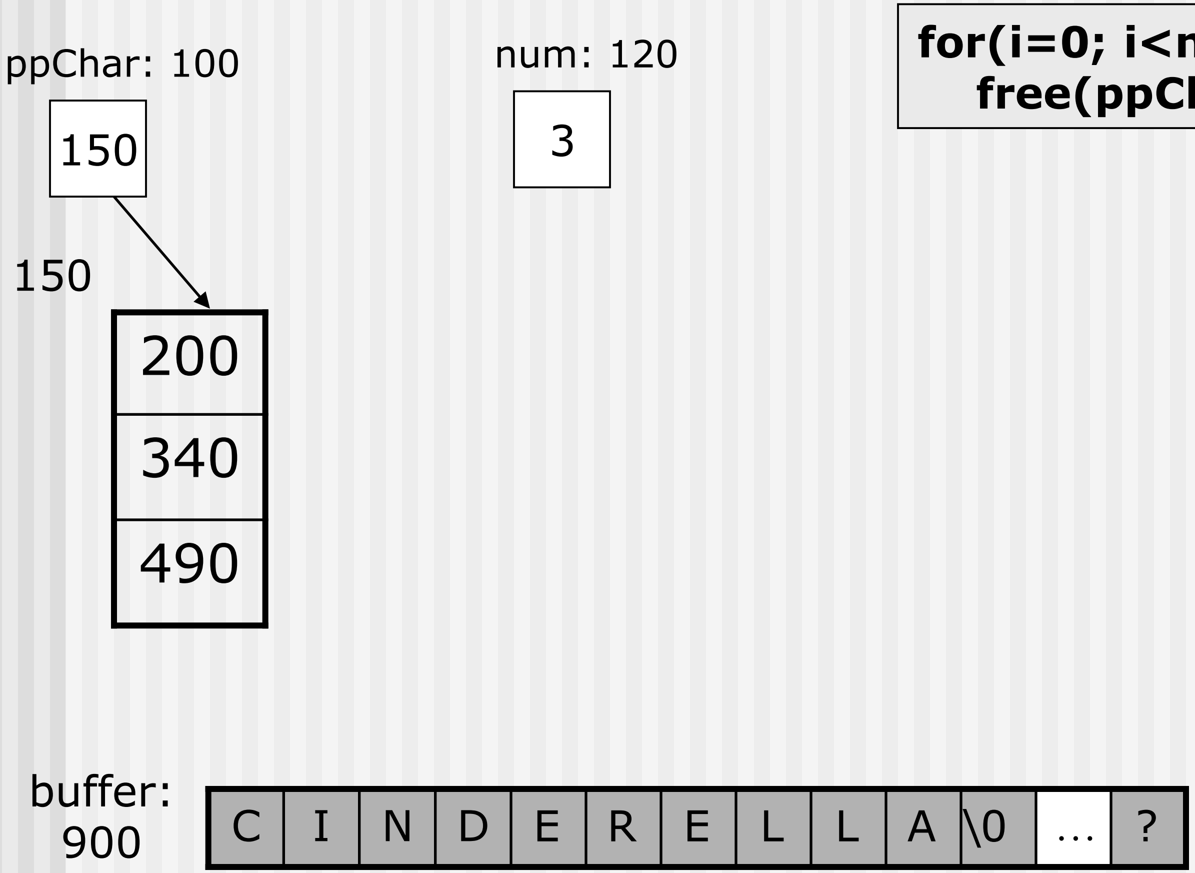
```
for(i=0; i<num; i++)
   free(ppChar[i]);
```



Copyright Meir Kalech

Pointer to Pointer - free

```
for(i=0; i<num; i++)
    free(ppChar[i]);
```



ppChar: 100

150

num: 120

3

150

200

340

490

buffer: 900

C	I	N	D	E	R	E	L	L	A	\0	…	?

Pointer to Pointer - free

```
for(i=0; i<num; i++)
    free(ppChar[i]);
free(ppChar);
```

ppChar: 100

150

num: 120

3

buffer:
900

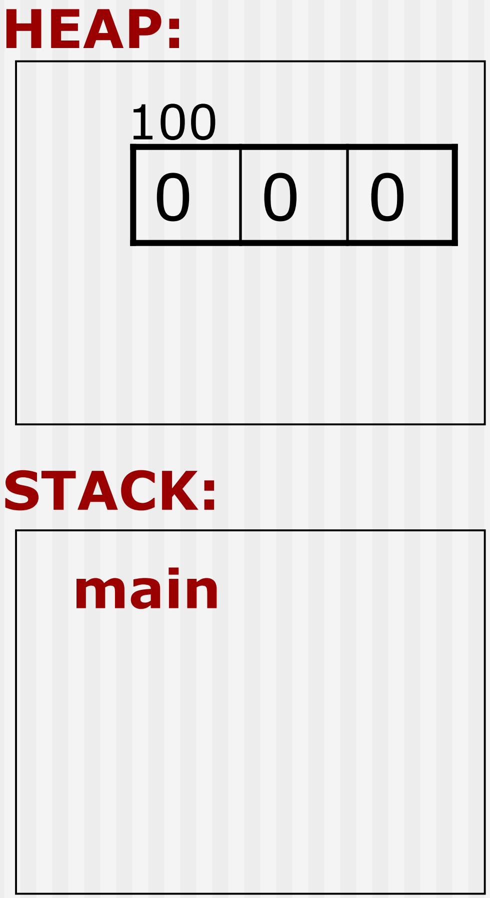
C	I	N	D	E	R	E	L	L	A	\0	…	?

Dynamic Memory Allocation - Function

- How to allocate memory in a function?
- Problem: indeed DMA lifetime does not depend on the function, but the scope and lifetime of the pointer to the memory is only in the function!
- So how to use the allocated memory also outside the function?
- Solution:
  - Return the pointer from the function.
  - Pass the pointer by address.

Problem

```
void func()
{
   int *p;
   p = (int *) calloc(3,sizeof(int));
}
void main()
{
   func();
   // how to use p here???
}
```



HEAP:

100

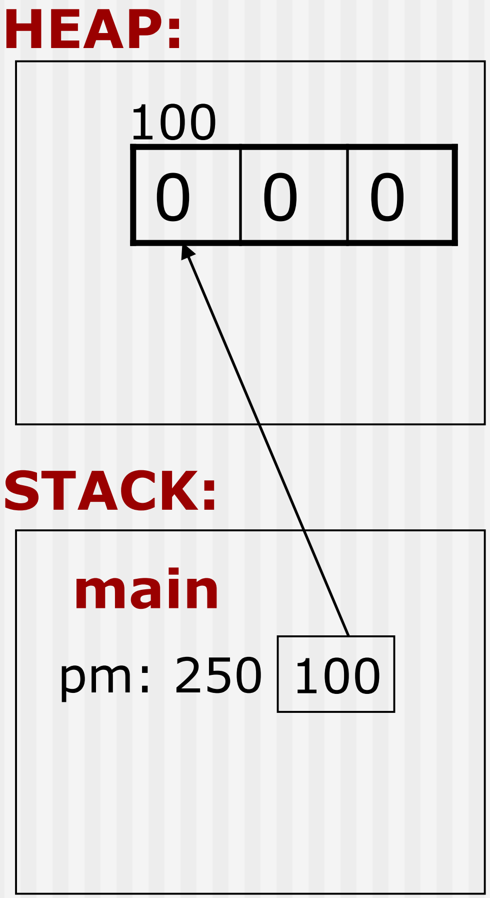
0 0 0

STACK:

main

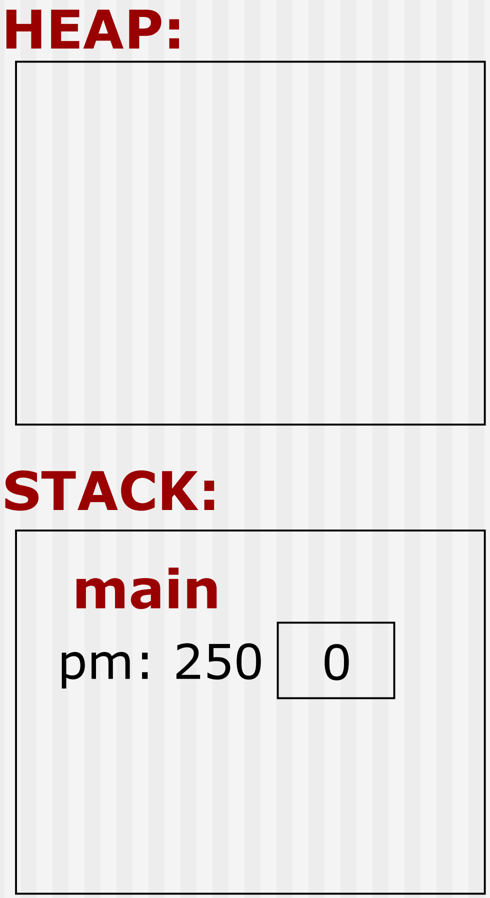
Return Address

```
int *func()
{
   int *p;
   p = (int *) calloc(3,sizeof(int));
   return p;
}
void main()
{
   int *pm = func();
}
```



Pass Address

```
void func(int *p)
{
   p = (int *) calloc(3,sizeof(int));
}
void main()
{
   int* pm = NULL;
   func(pm);
   // how to use p here???
}
```



HEAP:

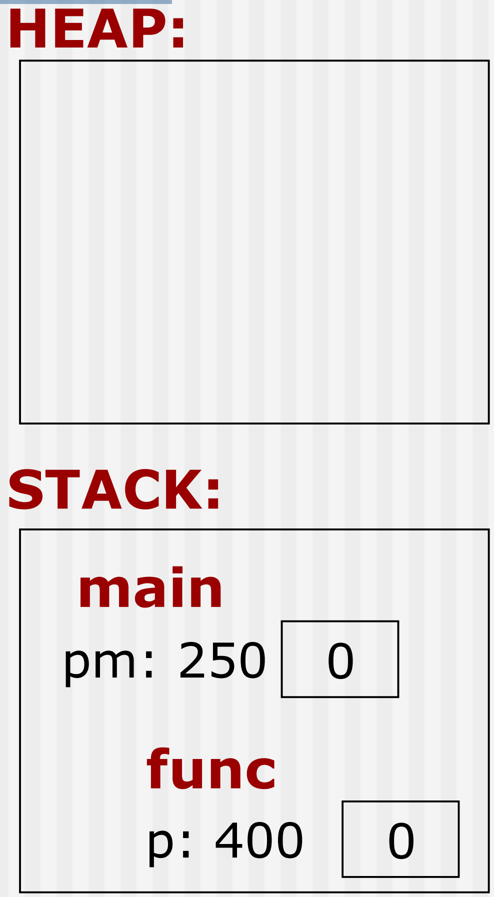
STACK:

main

pm: 250 0

Pass Address

```
void func(int *p)
{
   p = (int *) calloc(3,sizeof(int));
}
void main()
{
   int *pm = NULL;
   func(pm);
   // how to use p here???
}
```



HEAP:

STACK:

main

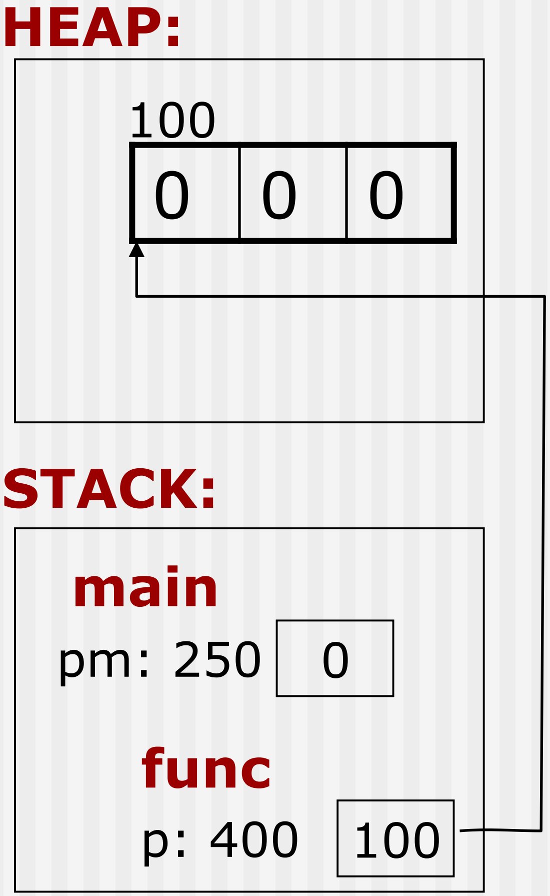
pm: 250 | 0

func

p: 400 | 0

Pass Address

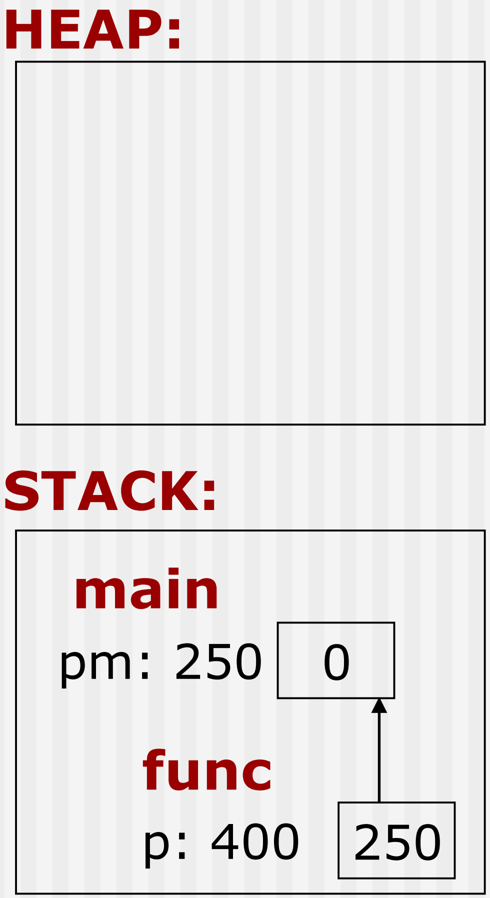
```
void func(int *p)
{
   p = (int *) calloc(3,sizeof(int));
}
void main()
{
   int *pm = NULL;
   func(pm);
}
```



Copyright Meir Kalech

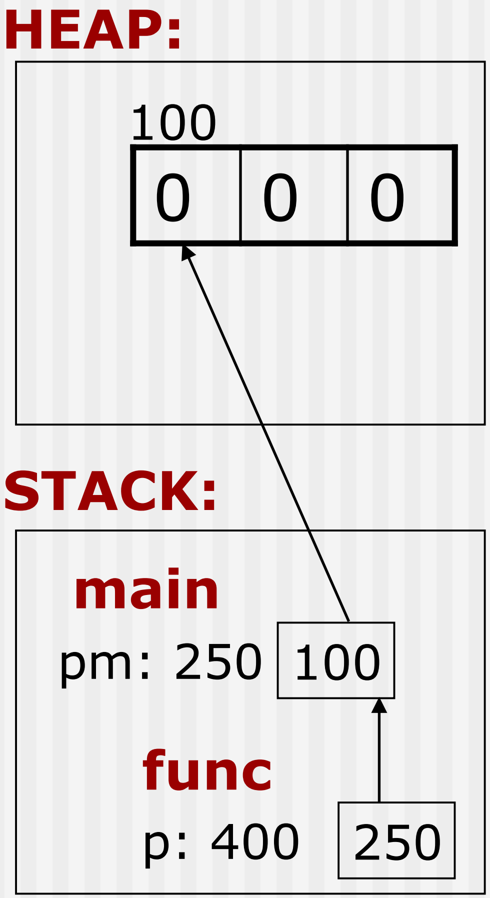
Pass Address – Right Way

```
void func(int **p)
{
   *p = (int *) calloc(3,sizeof(int));
}
void main()
{
   int *pm = NULL;
   func(&pm);
}
```



Pass Address – Right Way

```
void func(int **p)
{
   *p = (int *) calloc(3,sizeof(int));
}
void main()
{
   int *pm = NULL;
   func(&pm);
}
```



HEAP:

100

0 | 0 | 0

STACK:

main

pm: 250 | 100

func

p: 400 | 250

If a parameter (pointer) is changed in a function (“p =“), it must be passed by address.
If its pointed **value** is changed (“p[i] =“), it can be passed by value.