Lecture 19

Pointers

CSE115: Computing Concepts

Arrays and Pointers

```
#include <stdio.h>

int main(void)
{
     char str[5] = {'H', 'E', 'L', 'L', 'O'};
     char *ptr = &str[0];
     printf(“ptr = %08x\n”, ptr);
     printf(“str = %08x\n”, str);
     return 0;
}
```

Arrays and Pointers

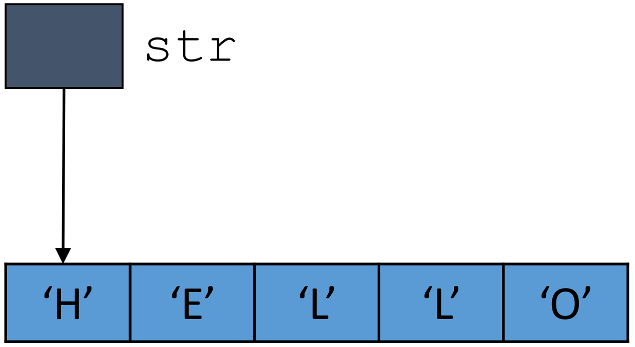
```
#include <stdio.h>

int main(void)
{
     char str[5] = {'H', 'E', 'L', 'L', 'O'};
     char *ptr = &str[0];
     printf(“ptr = %08x\n”, ptr);
     printf(“str = %08x\n”, str);
     return 0;
}
```

Output:

```
ptr = 0028ff17
str = 0028ff17
```

Arrays and Pointers

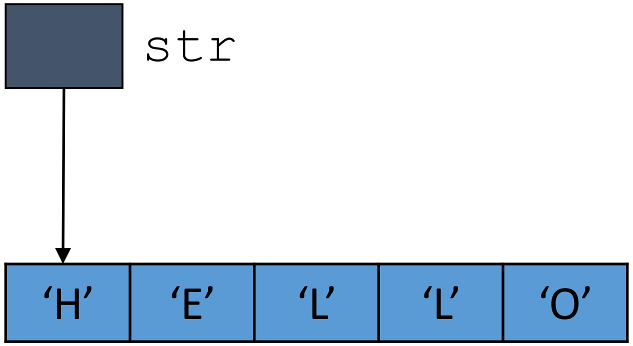
- The array name is basically the name of a pointer variable which contains the starting address of the array (address of the first element)



	address	content
	0x00000000	
	0x00000001	
	.	
	.	
str	0x180A96e7	0x180A96f3
	0x180A96e8	
	0x180A96e9	
	0x180A96f0	
	0x180A96f1	
	0x180A96f2	
	0x180A96f3	'H'
	0x180A96f4	'E'
	0x180A96f5	'L'
	0x180A96f6	'L'
	0x180A96f7	'O'
	.	
	.	

Arrays and Pointers

- The array name is basically the name of a pointer variable which contains the starting address of the array (address of the first element)

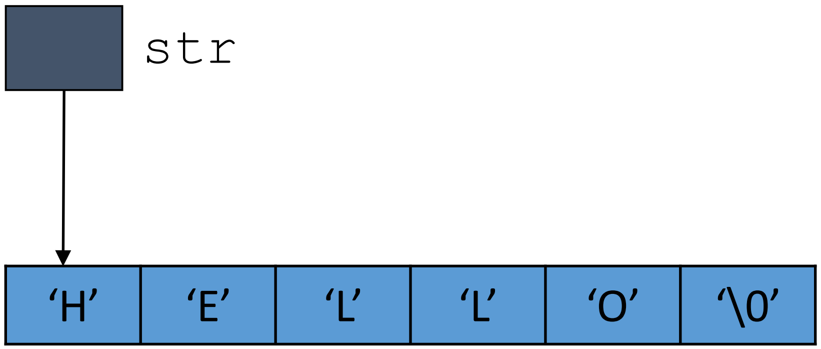


`c = str[2];` is equivalent to

$$c = *(\underset{\downarrow \atop \text{Base}}{str} + \underbrace{1 \times 2}_{\text{offset}});$$

	address	content
	0x00000000	
	0x00000001	
	.	
	.	
str	0x180A96e7	0x180A96f3
	0x180A96e8	
	0x180A96e9	
	0x180A96f0	
	0x180A96f1	
	0x180A96f2	
	0x180A96f3	‘H’
	0x180A96f4	‘E’
	0x180A96f5	‘L’
	0x180A96f6	‘L’
	0x180A96f7	‘O’
	.	
	.	

Arrays and Pointers

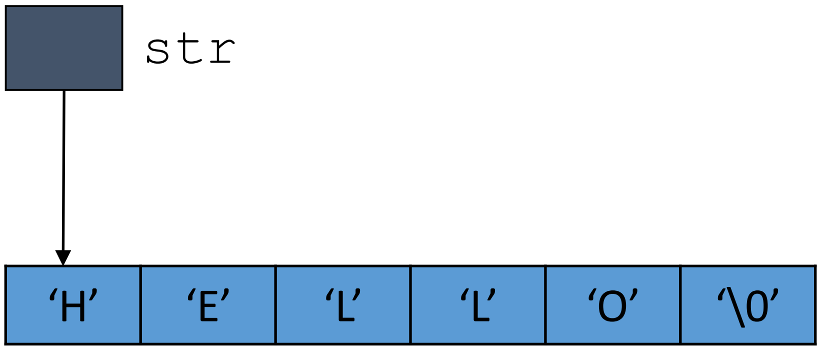


str

‘H’	‘E’	‘L’	‘L’	‘O’	‘\0’

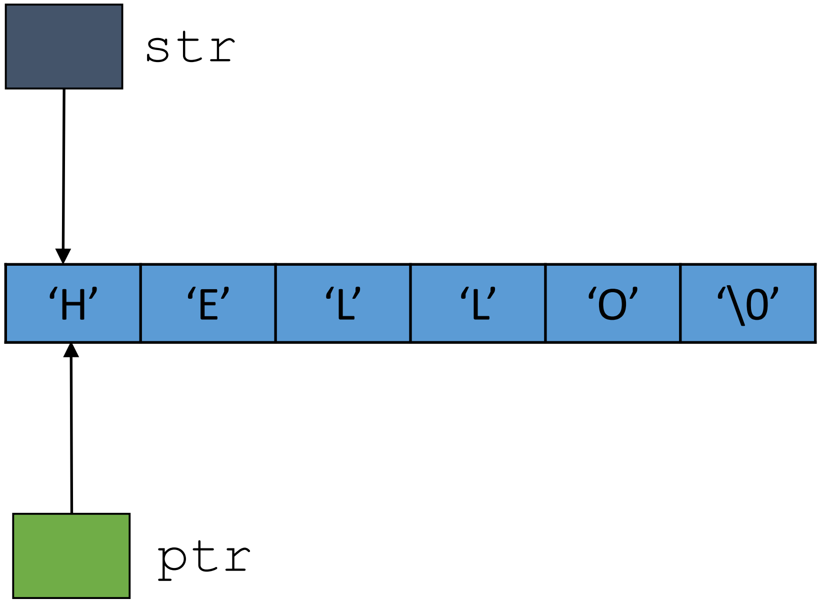
```
char str[6] = “HELLO";
```

Arrays and Pointers



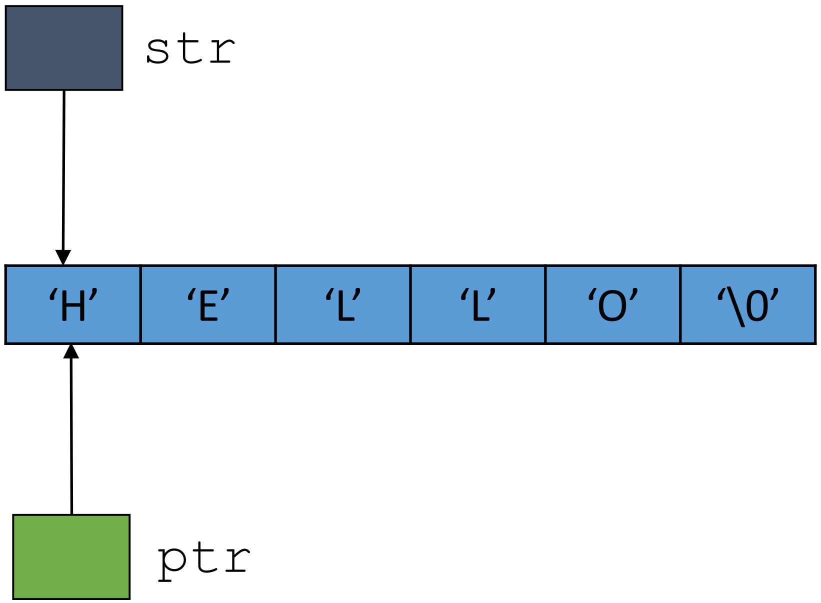
```
char str[6] = "HELLO";
char *ptr = str;
```

Arrays and Pointers



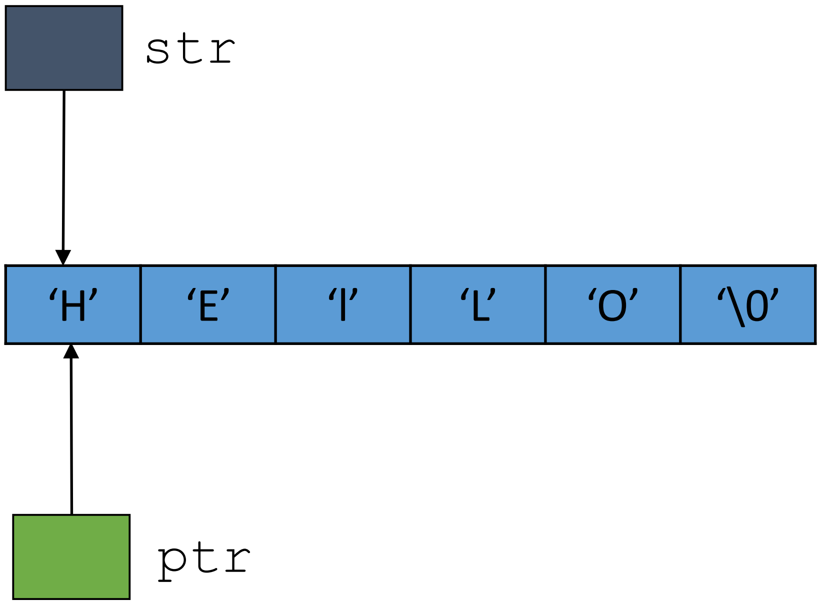
```
char str[6] = "HELLO";

char *ptr = str;
```

Arrays and Pointers



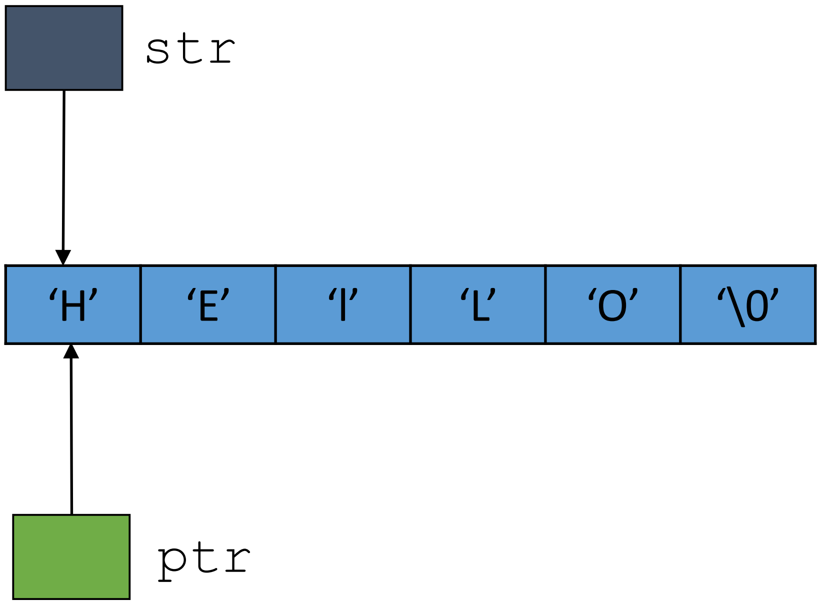
```
char str[6] = "HELLO";
char *ptr = str;
ptr[2] = ‘l’;
```

Arrays and Pointers



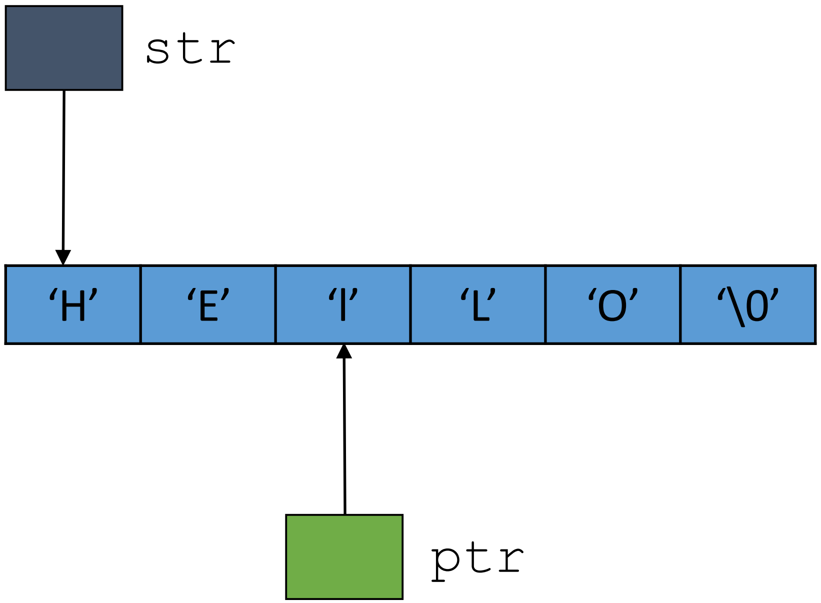
```
char str[6] = "HELLO";

char *ptr = str;

ptr[2] = ‘l’;
```

Arrays and Pointers



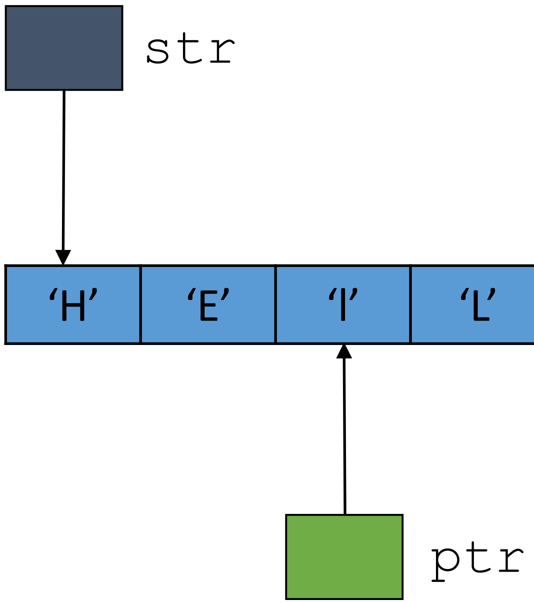
```
char str[6] = "HELLO";
char *ptr = str;
ptr[2] = ‘l’;
ptr = ptr + 2;
```

Arrays and Pointers



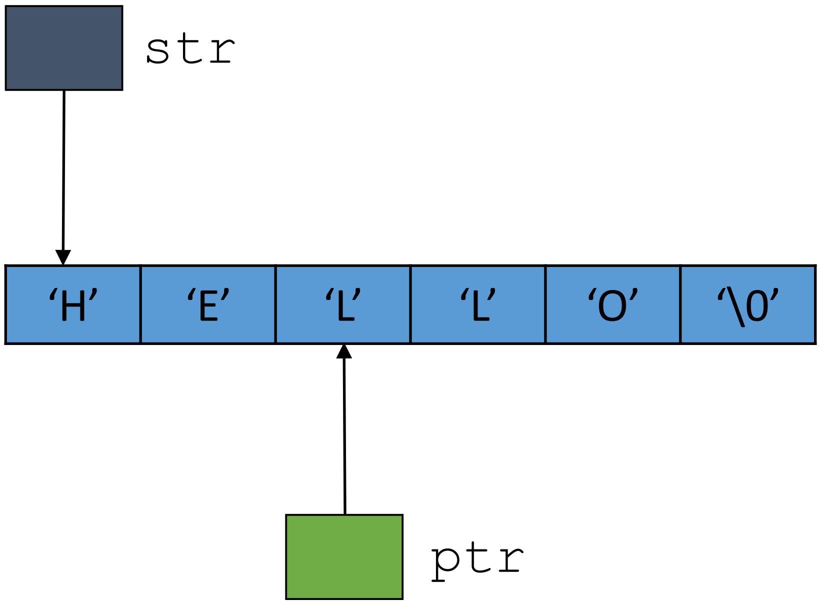
```
char str[6] = "HELLO";
char *ptr = str;
ptr[2] = ‘l’;
ptr = ptr + 2;
```

Arrays and Pointers



```
char str[6] = "HELLO";
char *ptr = str;
ptr[2] = ‘l’;
ptr = ptr + 2;
*ptr = ‘L’;
```

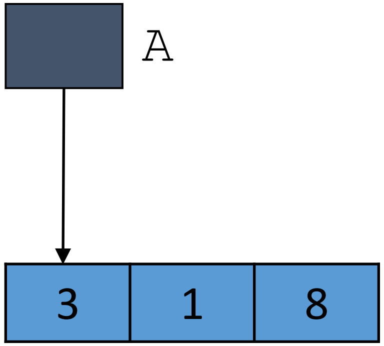
Arrays and Pointers



```
char str[6] = "HELLO";

char *ptr = str;

ptr[2] = ‘l’;

ptr = ptr + 2;

*ptr = ‘L’;
```

Arrays and Pointers

- `int A[3] = {3, 1, 8};`



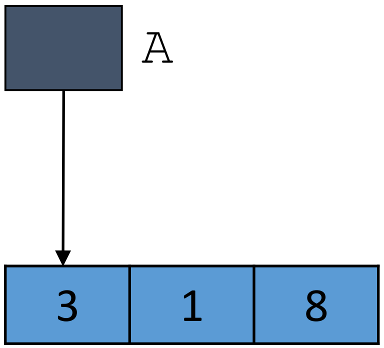
`i = A[2];` is equivalent to

$$i = *(\underset{\downarrow \atop \text{Base}}{A} + \underbrace{4 \times 2}_{\text{offset}});$$

	address	content
	0x00000000	
	0x00000001	
	.	
	.	
A	0x180A96e7	0x180A96f3
	0x180A96e8	
	0x180A96e9	
	0x180A96f0	
	0x180A96f1	
	0x180A96f2	
	0x180A96f3	3
	0x180A96f4	
	0x180A96f5	
	0x180A96f6	
	0x180A96f7	1
	0x180A96f8	
	0x180A96f9	
	0x180A96fA	
	0x180A96fB	8
	0x180A96fC	
	0x180A96fD	
	0x180A96fE	

Arrays and Pointers

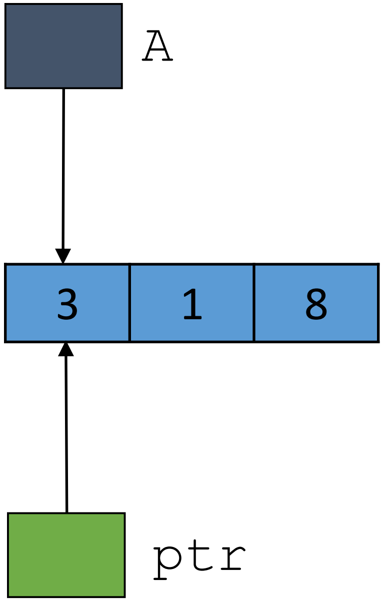
- `int A[3] = {3, 1, 8};`



```
int *ptr = A;
```

Arrays and Pointers

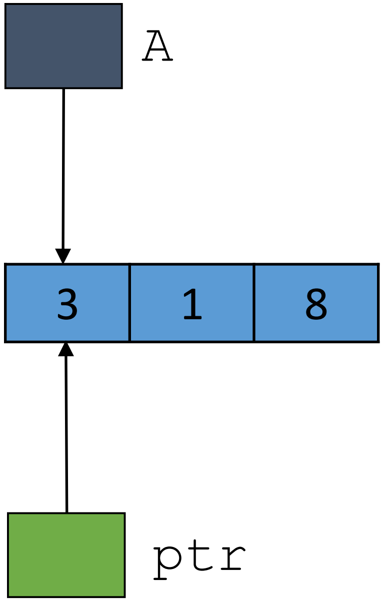
- `int A[3] = {3, 1, 8};`



```
int *ptr = A;
```

Arrays and Pointers

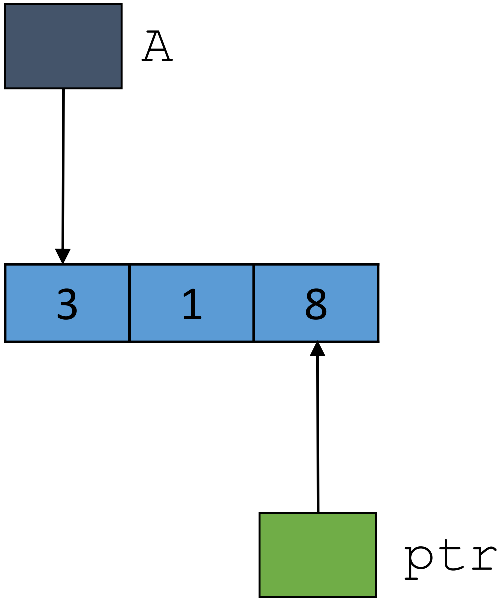
- `int A[3] = {3, 1, 8};`



```
int *ptr = A;
ptr = ptr + 2;
```

Arrays and Pointers

- `int A[3] = {3, 1, 8};`



```
int *ptr = A;
ptr = ptr + 2;
```

Dynamic Memory Allocation

Dynamic memory allocation is used to obtain and release memory during program execution. Up until this point we reserved memory at compile time using declarations.

You have to be careful with dynamic memory allocation. It operates at a low-level, you will often find yourself having to do a certain amount of work to manage the memory it gives you.

To use the functions discussed here, you must include the stdlib.h header file.

Four Dynamic Memory Allocation Functions:

- Allocate memory - malloc(), calloc(), and realloc()
- Free memory - free()

malloc()

To allocate memory, use

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

- Takes number of bytes to allocate as argument.
- Use sizeof to determine the size of a type.
- Returns pointer of type void *. A void pointer may be assigned to any pointer.
- If no memory available, returns NULL.

e.g.

```
char *line;
int linelength = 100;
line = (char*)malloc(linelength);
```

malloc()

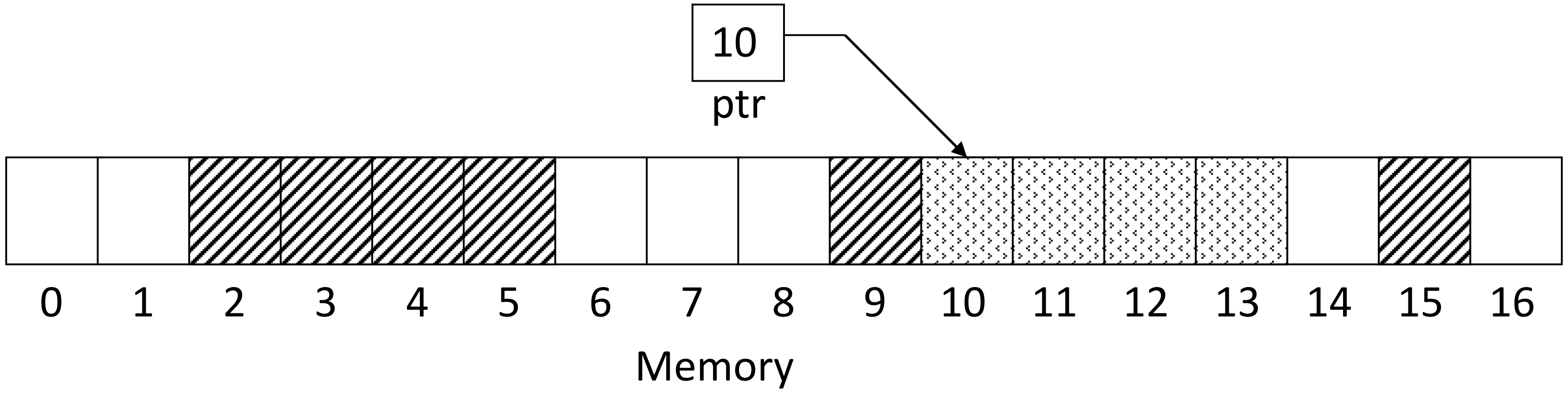
- Prototype: `void *malloc(size_t size);`
  - function searches memory for ***size*** contiguous free bytes
  - function returns the address of the first byte
  - programmers responsibility to not lose the pointer
  - programmers responsibility to not write into area past the last byte allocated
- Example:

```
char *ptr;
ptr = malloc(4);  // new allocation
```

Key

▨ previously allocated

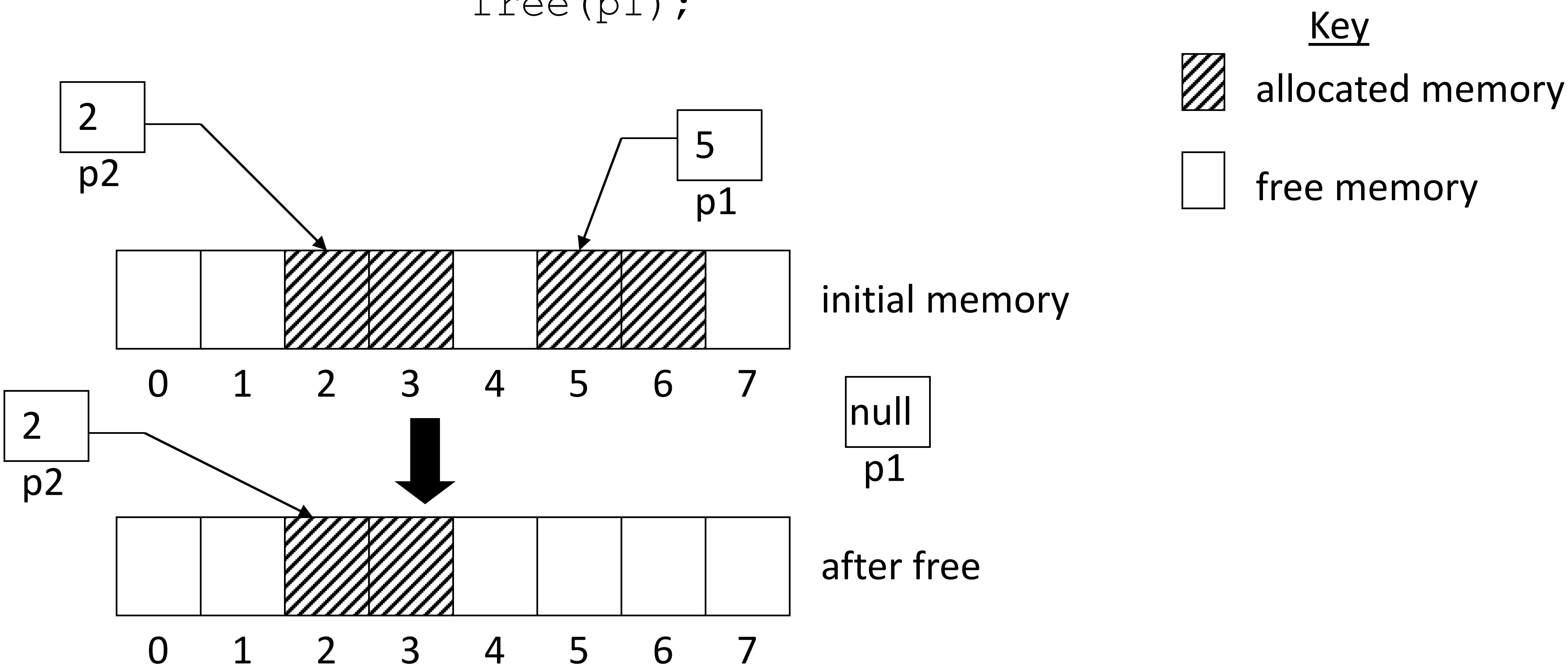
░ new allocation



`free()`

- Prototype: `void free(void *ptr);`
  - releases the area pointed to by ptr
  - ptr must not be null
    - trying to free the same area twice will generate an error
- Example:

`free(p1);`



Example

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

int main()
{
   char *str;

   /* Memory allocation */
   str = (char *) malloc(15);

   strcpy(str, "KungFu");
   printf("String = %s\n", str);
   strcat(str, "Panda");
   printf("String = %s\n", str);

   /* Memory deallocation */
   free(str);

   return(0);
}
```