Funcy Arrays

ESC101: Fundamentals of Computing

Purushottam Kar

Recap – 6 golden rules of functions

Recap – 6 golden rules of functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

Recap – 6 golden rules of functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

Recap – 6 golden rules of functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted

Recap – 6 golden rules of functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted

RULE 4: All values passed to a function get stored in a fresh variable inside that function

Recap – 6 golden rules of functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted

RULE 4: All values passed to a function get stored in a fresh variable inside that function

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

Recap – 6 golden rules of functions

RULE 1: When we give a variable as input, the value stored inside that variable gets passed as an argument

RULE 2: When we give an expression as input, the value generated by that expression gets passed as argument

RULE 3: In case of a mismatch b/w type of arg promised and type of arg passed, typecasting will be attempted

RULE 4: All values passed to a function get stored in a fresh variable inside that function

RULE 5: Value returned by a function can be used freely in any way values of that data-type could have been used

RULE 6: All clones share the memory address space

Passing Arrays to Functions

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to the first element of that array

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to the first element of that array

Thus, when we pass an array into a function, we are just sending a pointer, **not the entire array**

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to the first element of that array

Thus, when we pass an array into a function, we are just sending a pointer, **not the entire array**

CAREFUL: this has two important consequences

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to the first element of that array

Thus, when we pass an array into a function, we are just sending a pointer, **not the entire array**

CAREFUL: this has two important consequences

- The array will not get copied onto a new array inside the function since the only thing that will get copied is the address to the first element of the array

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to the first element of that array

Thus, when we pass an array into a function, we are just sending a pointer, **not the entire array**

CAREFUL: this has two important consequences

- The array will not get copied onto a new array inside the function since the only thing that will get copied is the address to the first element of the array
- If you make changes to the array inside the function, main() will see them

Passing Arrays to Functions

No new rules need to be learnt. Let us see how ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to the first element of that array

Thus, when we pass an array into a function, we are just sending a pointer, **not the entire array**

CAREFUL: this has two important consequences

- The array will not get copied onto a new array inside the function since the only thing that will get copied is the address to the first element of the array
- If you make changes to the array inside the function, main() will see them

Notice that when we pass an array to scanf (strings), scanf is able to change the values inside that array

Passing arrays to functions

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a pointer, if a clone uses the sizeof operator on an array passed to it, it will just get the value 8

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a pointer, if a clone uses the sizeof operator on an array passed to it, it will just get the value 8

Careful: this means that info about length of array must be passed separately to a clone

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a pointer, if a clone uses the sizeof operator on an array passed to it, it will just get the value 8

Careful: this means that info about length of array must be passed separately to a clone

```
void print2ndElement(int *b){
    printf("%d %d\n", b[1], sizeof(b));
}
int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a pointer, if a clone uses the sizeof operator on an array passed to it, it will just get the value 8

Careful: this means that info about length of array must be passed separately to a clone

```
void print2ndElement(int b[5]){
    printf("%d %d\n", b[1], sizeof(b));
}
int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```

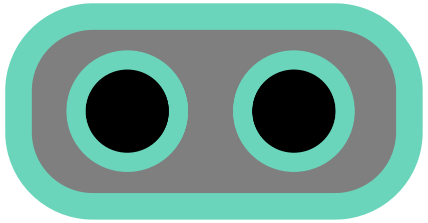
Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a pointer, if a clone uses the sizeof operator on an array passed to it, it will just get the value 8

Careful: this means that info about length of array must be passed separately to a clone

```
void print2ndElement(int b[5]){
    printf("%d %d\n", b[1], sizeof(b));
}
int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```



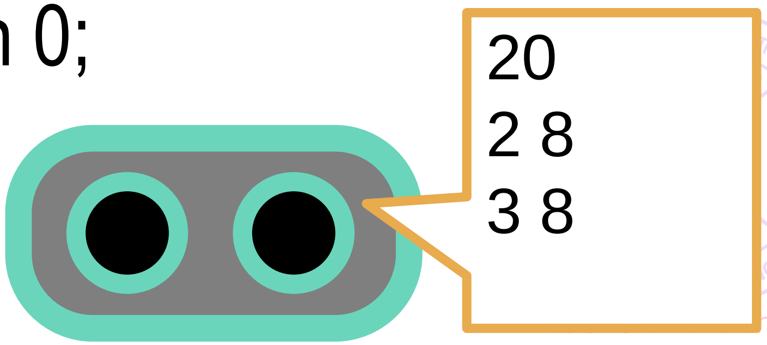
Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a pointer, if a clone uses the sizeof operator on an array passed to it, it will just get the value 8

Careful: this means that info about length of array must be passed separately to a clone

```c
void print2ndElement(int b[5]){
    printf("%d %d\n", b[1], sizeof(b));
}
int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```



```
20
2 8
3 8
```

Passing arrays to functions

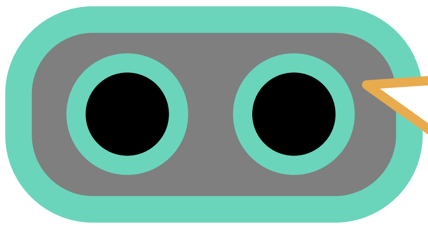
Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing ... clone uses the sizeof opera... on an array passed to it, it will just get the value 8

Careful: this means that info about length of array must be passed separately to a clone

```
void print2ndElement(int b[5]){
    printf("%d %d\n", b[1], sizeof(b));
}
int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```

&a[0] gets passed



```
20
2 8
3 8
```

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing a

&a[0] gets passed

clone uses the sizeof opera

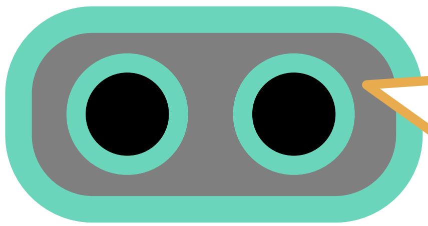
on an array passed to it, it will just get the value 8

Careful: this mea

&a[1] gets passed

about length of array must be passed separately to a clone

```
void print2ndElement(int b[5]){
    printf("%d %d\n", b[1], sizeof(b));
}
int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```



```
20
2 8
3 8
```

Passing arrays to functions

Two styles – both are exactly the same. Some people prefer one style and some prefer the other style

Note: since passing an array is just like passing clone uses the si on an array passed to it, it will just get the value 8

Careful: this mea about length of array must be passed separately to a clone

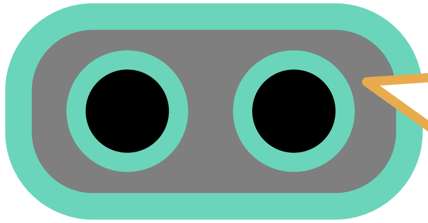
```
void print2ndElement(int b[5]){
    printf("%d %d\n", b[1], sizeof(b));
}

int main(void){
    int a[5] = {1,2,3,4,5};
    printf("%d\n", sizeof(a));
    print2ndElement(a);
    print2ndElement(a+1);
    return 0;
}
```

Recall that b[1] is the same as *(b+1)

&a[0] gets passed

&a[1] gets passed



```
20
2 8
3 8
```

Passing arrays to functions

Passing arrays to functions

No matter what the manner in which you declare function

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
```

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
```

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
void print2ndElement(int *b)
```

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
void print2ndElement(int *b)
```

In all cases, just pointer to first element of array gets passed

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
void print2ndElement(int *b)
```

In all cases, just pointer to first element of array gets passed

In second case you are telling Mr C that you will pass an array of 5 integers but he will throw that information away!

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
void print2ndElement(int *b)
```

In all cases, just pointer to first element of array gets passed

In second case you are telling Mr C that you will pass an array of 5 integers but he will throw that information away!

You can pass an int array of any length here. Just a pointer will get passed.

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
void print2ndElement(int *b)
```

In all cases, just pointer to first element of array gets passed

In second case you are telling Mr C that you will pass an array of 5 integers but he will throw that information away!

- You can pass an int array of any length here. Just a pointer will get passed.

This is why you must always pass the actual length of array as a separate argument

Passing arrays to functions

No matter what the manner in which you declare function

```
void print2ndElement(int b[])
void print2ndElement(int b[5])
void print2ndElement(int *b)
```

In all cases, just pointer to first element of array gets passed

In second case you are telling Mr C that you will pass an array of 5 integers but he will throw that information away!

You can pass an int array of any length here. Just a pointer will get passed.

This is why you must always pass the actual length of array as a separate argument

```
void print2ndElement(int b[], int arrLength)
```

Declaring arrays inside functions

Declaring arrays inside functions

We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc

Declaring arrays inside functions

We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc

However, all these (including arrays) get destroyed when the clone dies –are called *local variables* of the function

Declaring arrays inside functions

- We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc
- However, all these (including arrays) get destroyed when the clone dies –are called *local variables* of the function
  - If you declare a double variable inside a function, it will be destroyed when the function returns

Declaring arrays inside functions

- We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc
- However, all these (including arrays) get destroyed when the clone dies –are called *local variables* of the function
  - If you declare a double variable inside a function, it will be destroyed when the function returns
  - If you declare a static float array inside a function, it will be destroyed when the function returns

Declaring arrays inside functions

We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc

However, all these (including arrays) get destroyed when the clone dies –are called *local variables* of the function

- If you declare a double variable inside a function, it will be destroyed when the function returns
- If you declare a static float array inside a function, it will be destroyed when the function returns

… except dynamically declared arrays (i.e. those declared using malloc/calloc/realloc) – these are not destroyed ☺

Declaring arrays inside functions

We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc

However, all these (including arrays) get destroyed when the clone dies –are called *local variables* of the function

- If you declare a double variable inside a function, it will be destroyed when the function returns
- If you declare a static float array inside a function, it will be destroyed when the function returns

… except dynamically declared arrays (i.e. those declared using malloc/calloc/realloc) – these are not destroyed ☺

- Advantage: can exploit this to return arrays from a function

Declaring arrays inside functions

We can declare and use new variables inside functions – char, float, long, int, even arrays of int, char etc

However, all these (including arrays) get destroyed when the clone dies –are called *local variables* of the function

- If you declare a double variable inside a function, it will be destroyed when the function returns
- If you declare a static float array inside a function, it will be destroyed when the function returns

… except dynamically declared arrays (i.e. those declared using malloc/calloc/realloc) – these are not destroyed ☺

- <u>Advantage</u>: can exploit this to return arrays from a function
- <u>Disadvantage</u>: can cause memory leaks if you are not careful – remember to free arrays before a function exits if the array no longer needed

Returning Arrays from Functions

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array

WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

Returning Arrays from Functions

- To return an array, simply return the address of the first element of the array – as simple as that ☺
  - **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array
- **WARNING**: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!
  - If the main function tries to read a destroyed array – SEGFAULT!

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array

WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array

WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

- <u>Advantage</u>: can return as many values as you want ☺

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array

WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

- Advantage: can return as many values as you want ☺
- Disadvantage: all those values have to be of that same type

Returning Arrays from Functions

To return an array, simply return the address of the first element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to the first element of that array

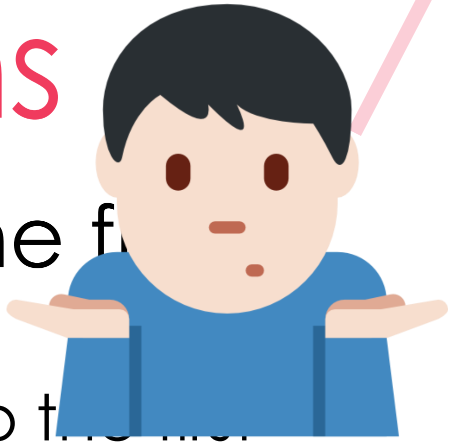
WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

- Advantage: can return as many values as you want ☺
- Disadvantage: all those values have to be of that same type
- Disadvantage: can only return one array ☹

Returning Arrays from Functions



To return an array, simply return the address of the f
element of the array – as simple as that ☺

Rule **5** (of pointers): name of an array is nothing but a pointer to t
element of that array

WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

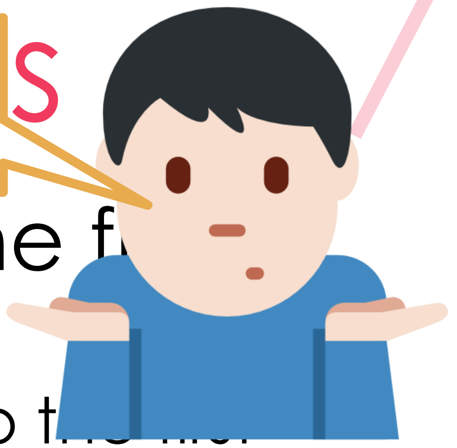
<u>Advantage</u>: can return as many values as you want ☺

<u>Disadvantage</u>: all those values have to be of that same type

<u>Disadvantage</u>: can only return one array ☹

Returning Arr s

What if I want to return multiple arrays?



To return an array, simply return the address of the f
element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to t
  element of that array

WARNING: return only those arrays that have been malloced/calloced/realloced. Do not return statically declared arrays – they are destroyed!

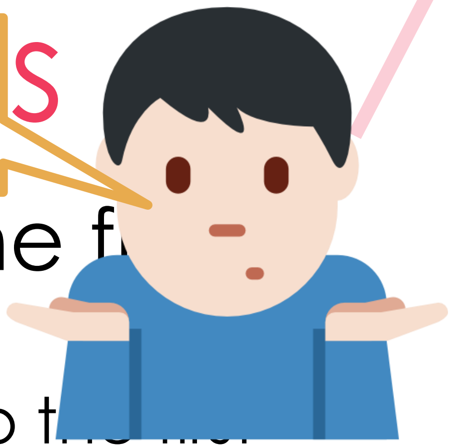
- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

- Advantage: can return as many values as you want ☺
- Disadvantage: all those values have to be of that same type
- Disadvantage: can only return one array ☹

Returning Arr s

What if I want to return multiple arrays?



To return an array, simply return the address of the f element of the array – as simple as that ☺

- **Rule 5** (of pointers): name of an array is nothing but a pointer to t element of that array

WARNING: return only those arrays that have bee malloced/calloced/realloced. Do not return stati declared arrays – they are destroyed!

- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

- Advantage: can return as many values as you want ☺
- Disadvantage: all those values have to be of that same type
- Disadvantage: can only return one array ☹

Returning Arr s

To return an array, simply return the address of the f
element of the array – as simple as that ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to t
element of that array

WARNING: return only tho
malloced/calloced/realloced. Do not return stati
declared arrays – they are destroyed!

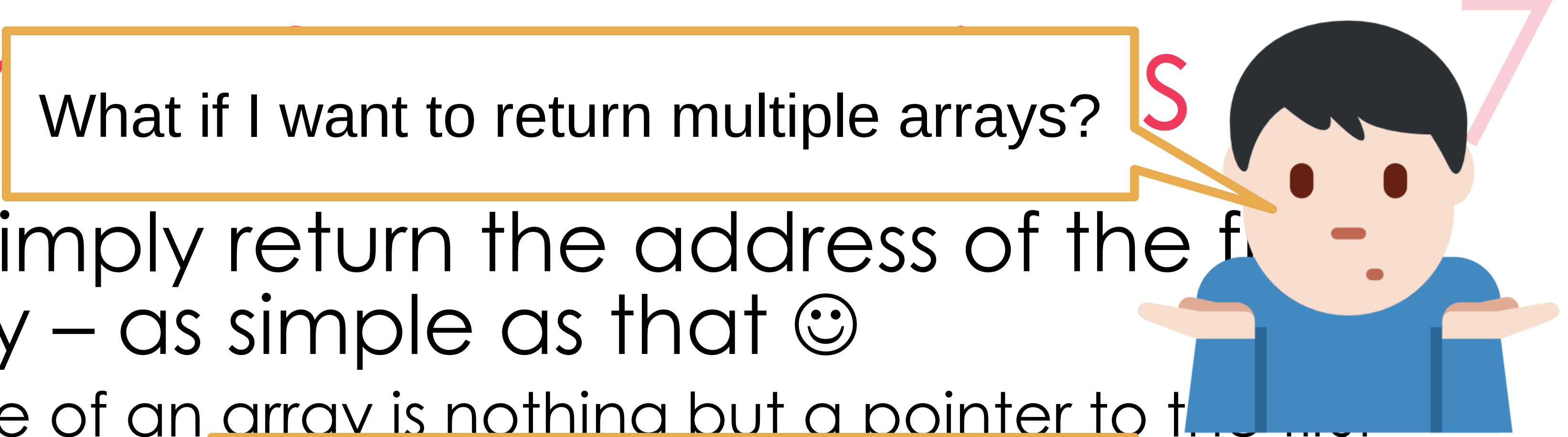
If the main function tries to read a destroyed array – SEGFAULT!

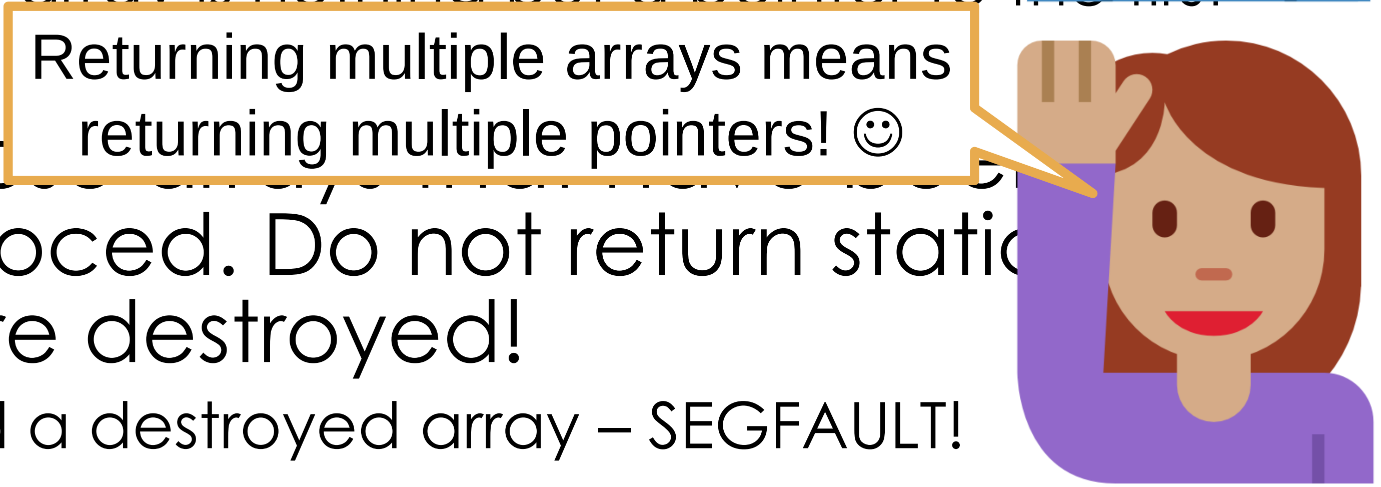
Gives us another trick of returning multiple values from a function – simply return an array

<u>Advantage</u>: can return as many values as you want ☺

<u>Disadvantage</u>: all those values have to be of that same type

<u>Disadvantage</u>: can only return one array ☹




Returning multiple arrays means returning multiple pointers! ☺

Returning Arr s

To return an array, simply return the address of the f element of the array – as simple as that ☺

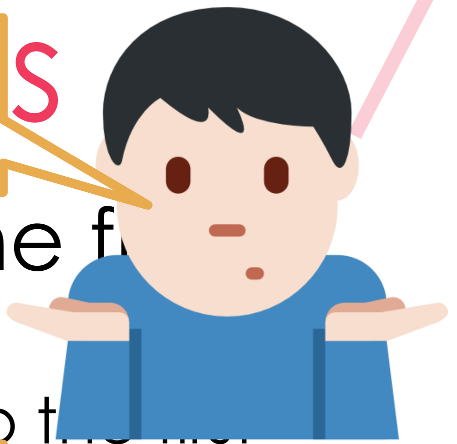
- **Rule 5** (of pointers): name of an array is nothing but a pointer to t element of that array

WARNING: return only tho malloced/calloced/realloced. Do not return stati declared arrays – they are destroyed!

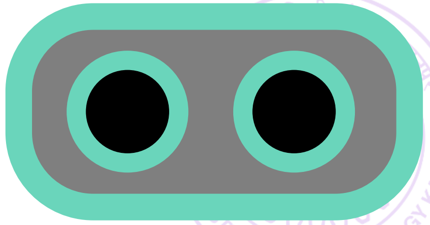
- If the main function tries to read a destroyed array – SEGFAULT!

Gives us another trick of returning multiple values from a function – simply return an array

- <u>Advantage</u>: can return as many values as you want ☺
- <u>Disadvantage</u>: all those values have to be of that same type
- <u>Disadvantage</u>: can only return one array ☹



Returning multiple arrays means returning multiple pointers! ☺



Returning Arr s

To return an array, simply return the address of the f
element of the array – as simple as that ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to t
element of that array

WARNING: return only tho
malloced/calloced/realloced. Do not return static
declared arrays – they are destroyed!

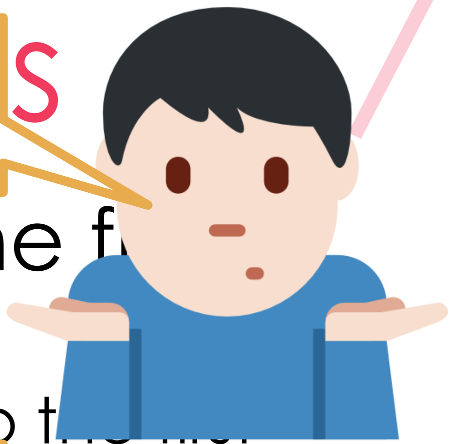
If the main function tries to read a destroyed array – T!

Gives us another trick of returning multipl es from a
function – simply return an array

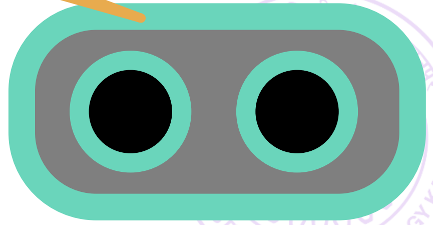
<u>Advantage</u>: can return as many values as you want ☺

<u>Disadvantage</u>: all those values have to be of that same type

<u>Disadvantage</u>: can only return one array ☹



Returning multiple arrays means returning multiple pointers! ☺



Returning Arr s

To return an array, simply return the address of the f
element of the array – as simple as that ☺

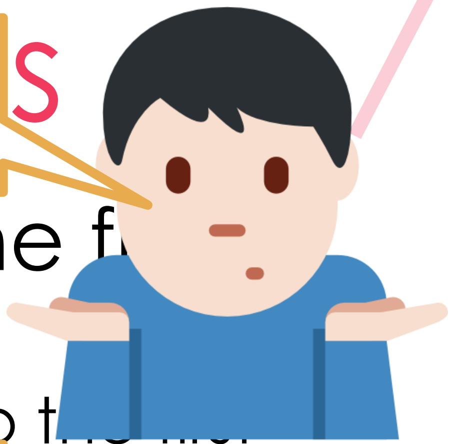
- **Rule 5** (of pointers): name of an array is nothing but a pointer to th
element of that array

WARNING: return only tho
malloced/calloced/realloced. Do not return static
declared arrays – they are destroyed!

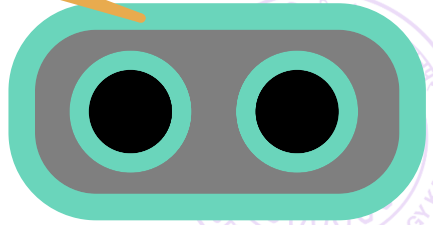
- If the main function tries to read a destroyed array – T!

Gives us another trick of returning multipl es from a function – simply return an array

- Advantage: can return as many values as you want ☺
- Disadvantage: all those values have to be of that same type
- Disadvantage: can only return one array ☹

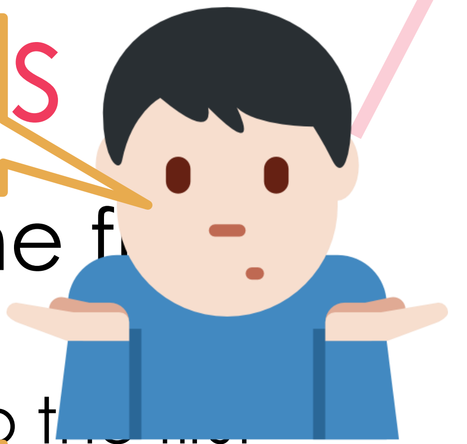


Returning multiple arrays means returning multiple pointers! ☺



Returning Arr s

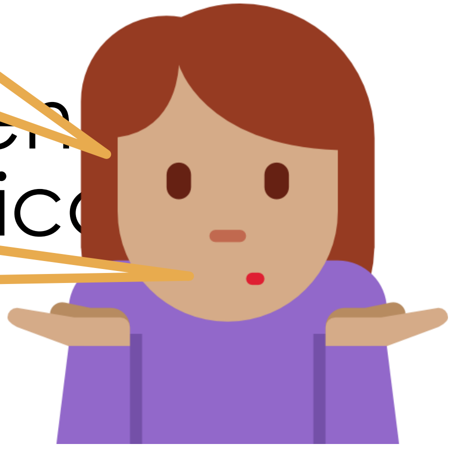
What if I want to return multiple arrays?



To return an array, simply return the address of the f
element of the array – as simple as that ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to t
element of that array

Returning multiple arrays means returning multiple pointers! ☺

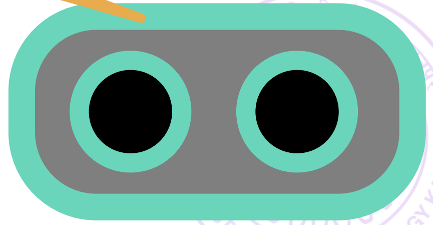


How do I return multiple pointers?

WARNING: return only tho en
malloced/calloced/real ic
declared arrays – they a

If the main function tries to read a destroyed array – T!

Correct!



Gives us another trick of returning multipl es from a function – simply return an array

<u>Advantage</u>: can return as many values as you want ☺

<u>Disadvantage</u>: all those values have to be of that same type

<u>Disadvantage</u>: can only return one array ☹

Returning Arr s

What if I want to return multiple arrays?

To return an array, simply return the address of the f
element of the array – as simple as that ☺

Rule 5 (of pointers): name of an array is nothing but a pointer to t
element of that array

Returning multiple arrays means returning multiple pointers! ☺

WARNING: return only tho
malloced/calloced/rea
declared arrays – they a

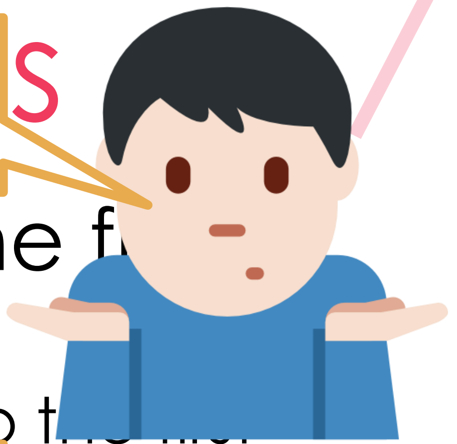
How do I return multiple pointers?

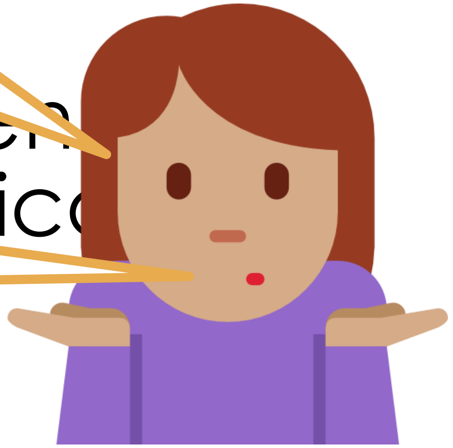
If the main function tries to read a destroyed array – T!

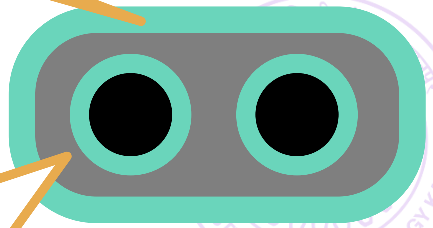
Correct!

Gives us another trick of returning multi es from a
fu

Using the same tricks we learnt till now to return multiple variables
Trick 1: malloc an array of pointers and return that array
Trick 2: ask the calling function to tell you the address where a pointer variable is stored so that you can modify the address (this will require passing a pointer to a pointer to the function)







Passing 2D arrays as inputs

Passing 2D arrays as inputs

More care required for 2D arrays

Passing 2D arrays as inputs

More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

Passing 2D arrays as inputs

More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

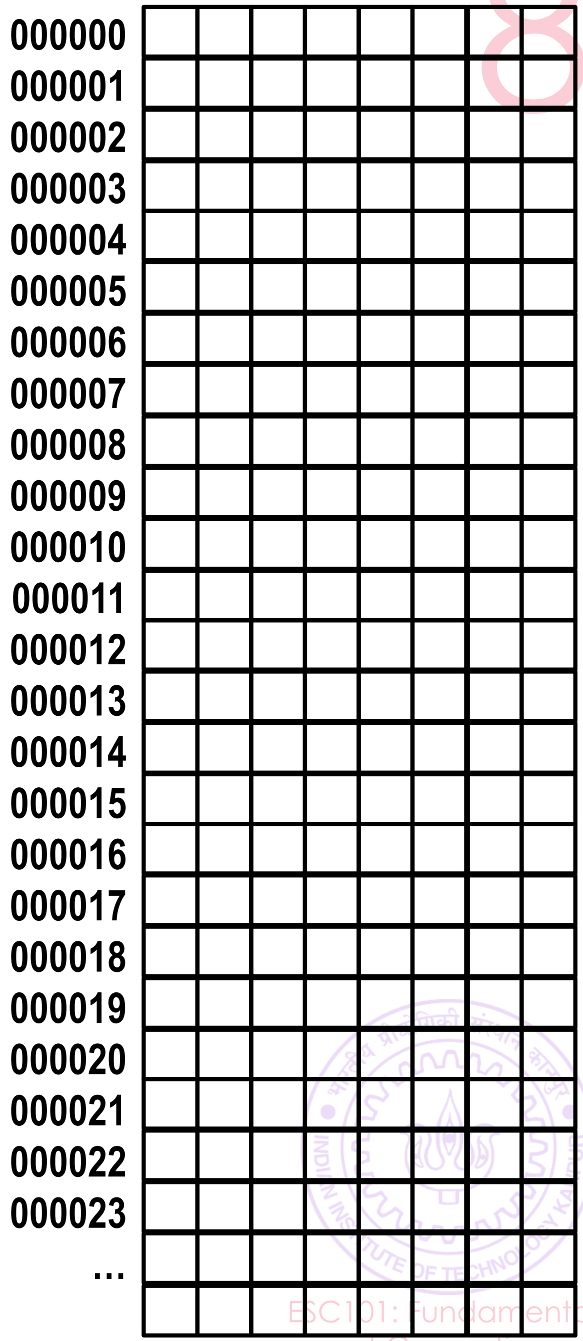
```
char str[3][5];
```

Passing 2D arrays as inputs

More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

```
char str[3][5];
```

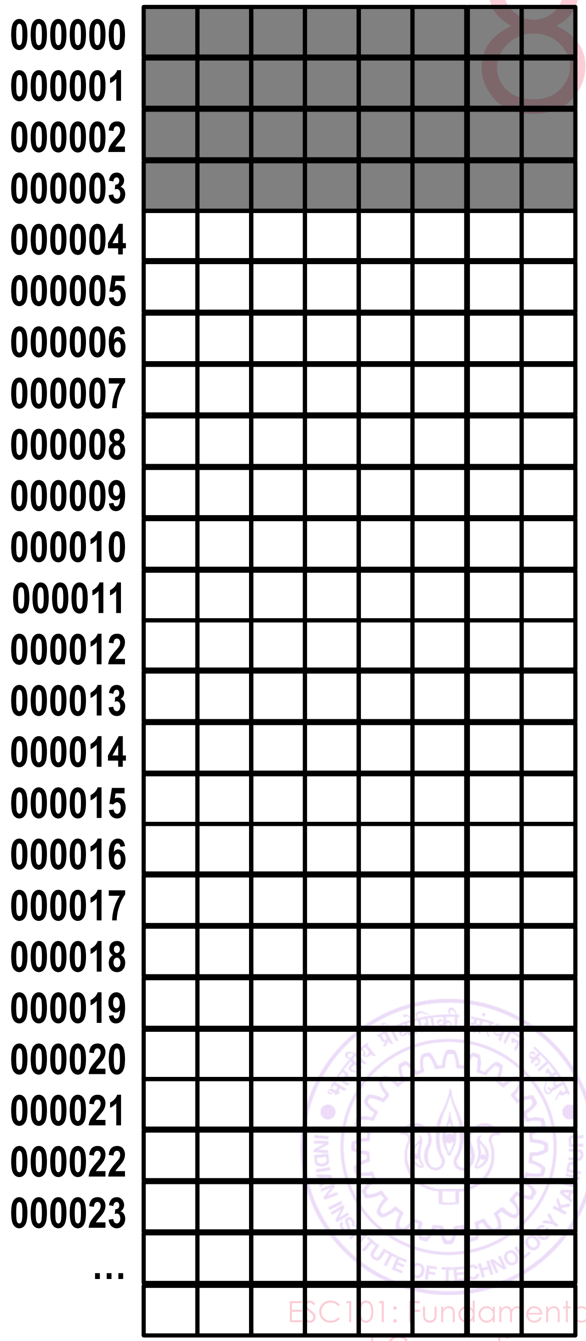


Passing 2D arrays as inputs

More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

```
char str[3][5];
```

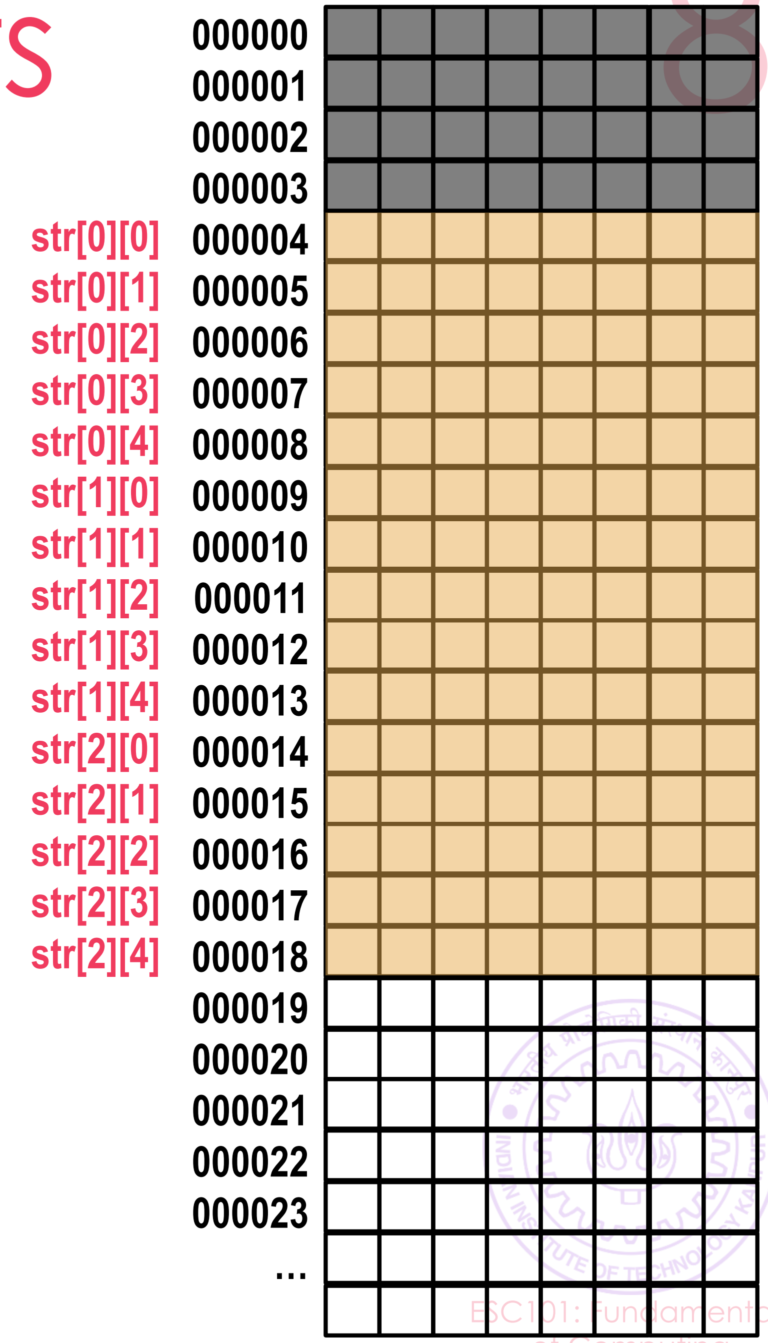


Passing 2D arrays as inputs

More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

```
char str[3][5];
```



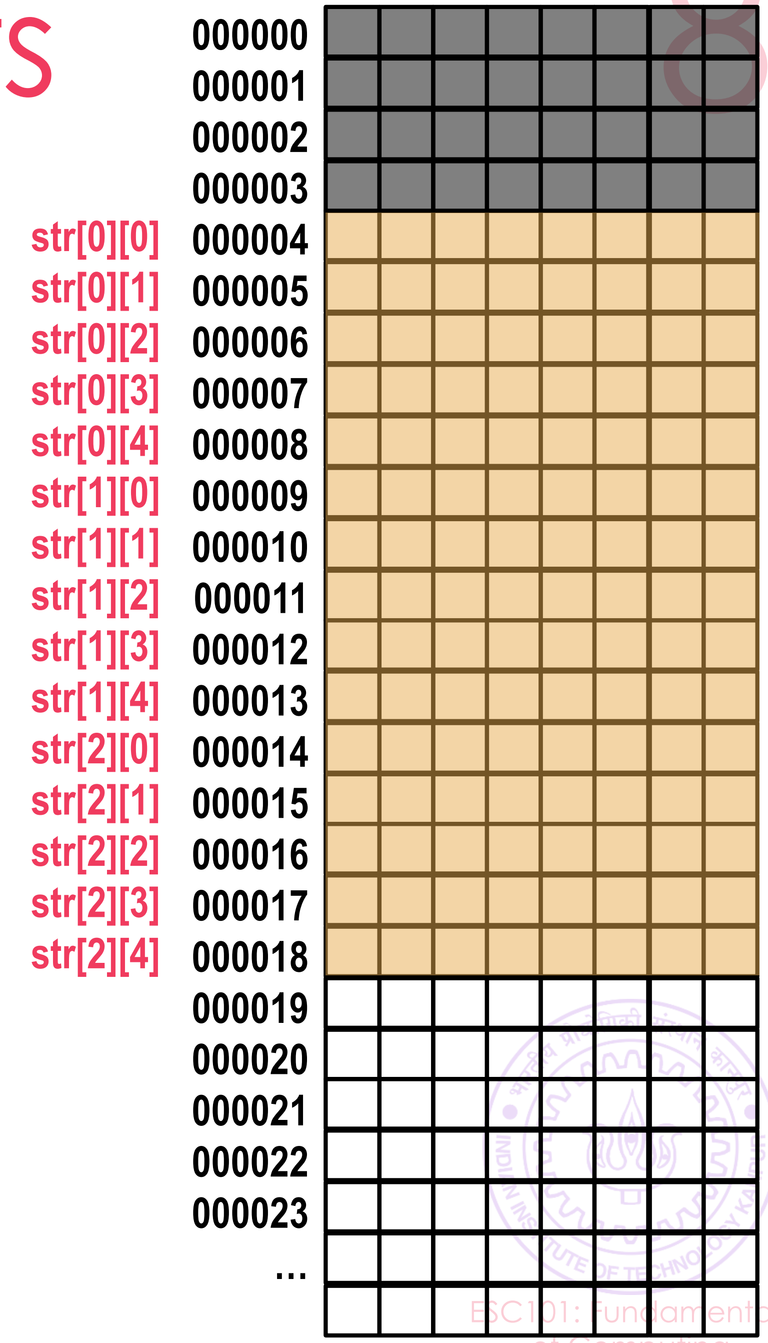
Passing 2D arrays as inputs

More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

```
char str[3][5];
```

This means, in order to access str[1][0], we need to skip 5 elements



Passing 2D arrays as inputs

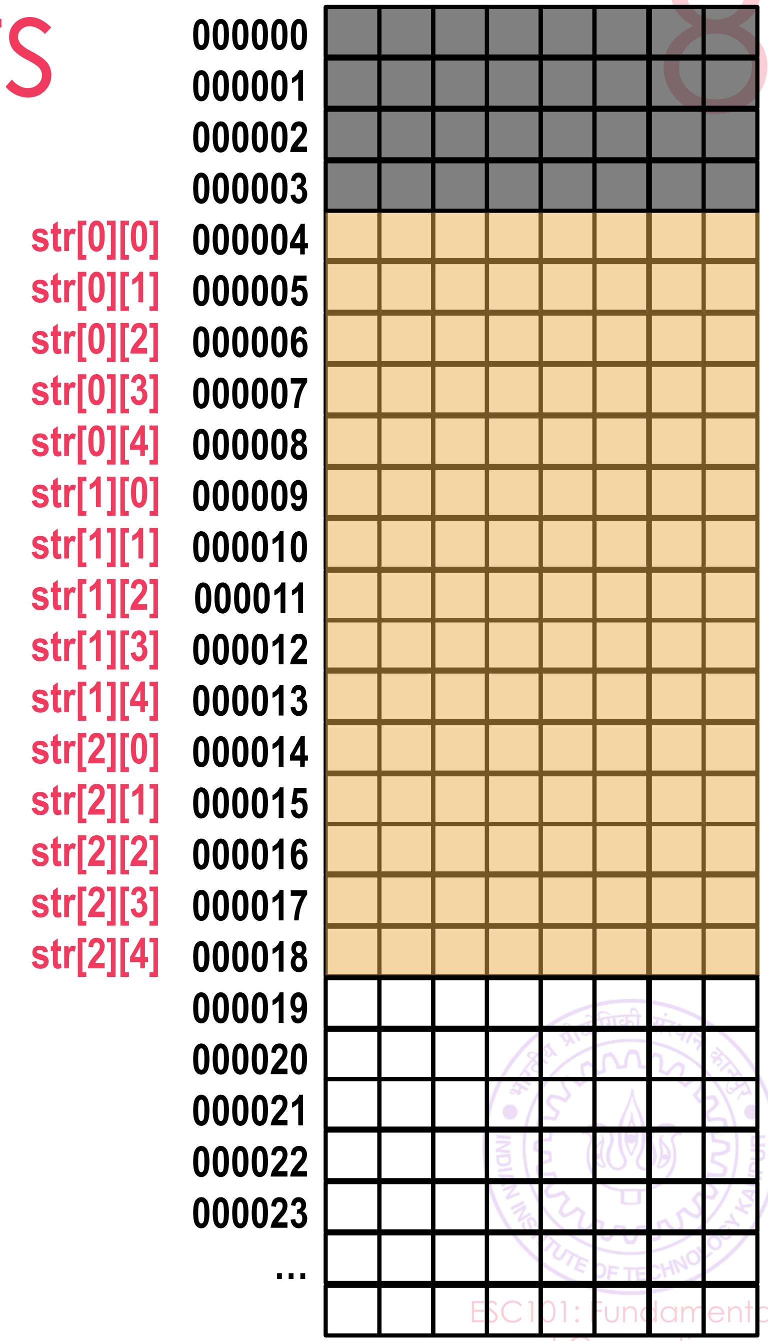
More care required for 2D arrays

Recall 2D arrays are stored as 1D arrays

char str[3][5];

This means, in order to access str[1][0], we need to skip 5 elements

To do so we need to know how many elements are there in each row



Passing 2D arrays as inputs

More care required for 2D arrays

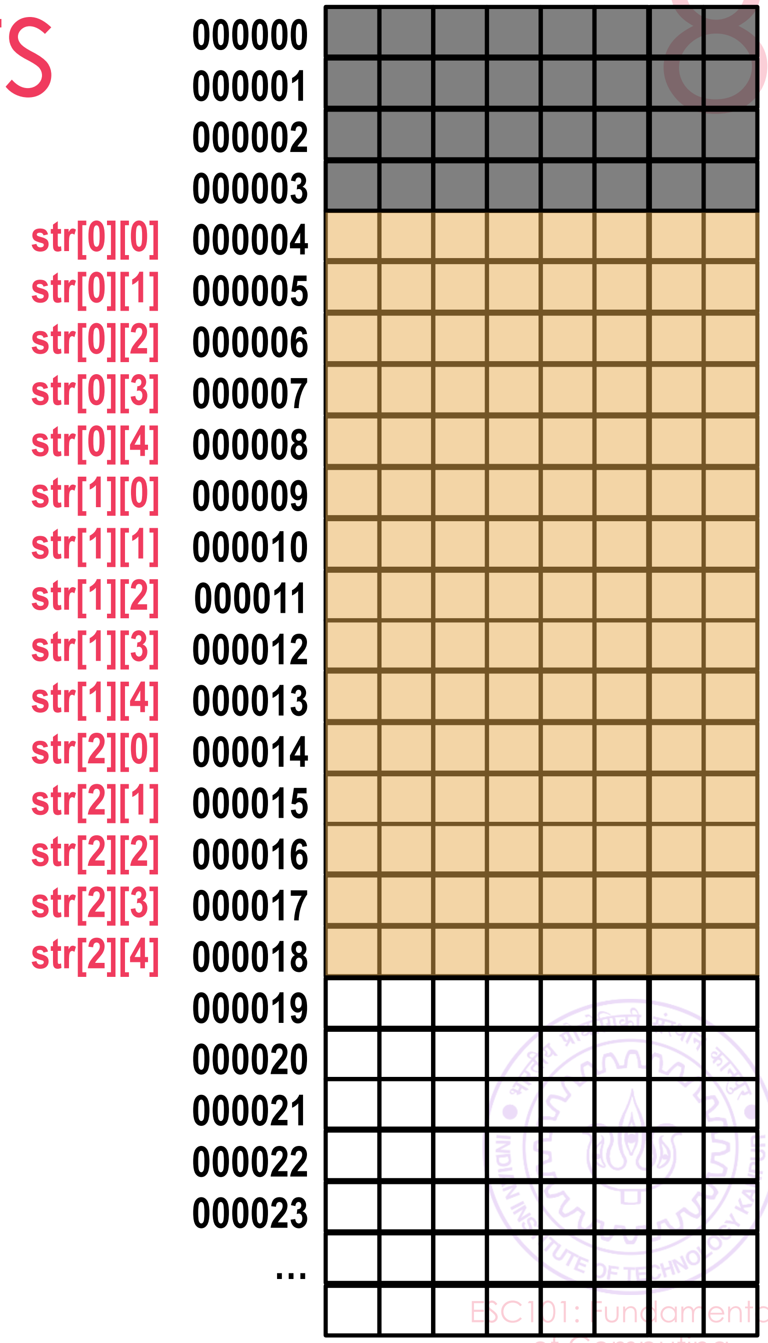
Recall 2D arrays are stored as 1D arrays

```
char str[3][5];
```

This means, in order to access str[1][0], we need to skip 5 elements

To do so we need to know how many elements are there in each row

If passing a 2D array to a clone, must tell that clone this information



Passing 2D arrays to functions

Passing 2D arrays to functions

Case 1: both the number of rows and the number of columns are fixed

Passing 2D arrays to functions

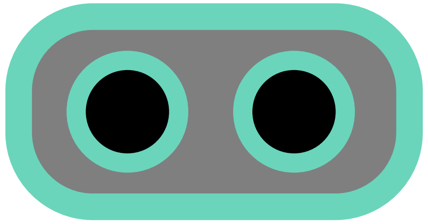
Case 1: both the number of rows and the number of columns are fixed

```
void access2D(char str[3][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 1, 0);
    return 0;
}
```

Passing 2D arrays to functions

Case 1: both the number of rows and the number of columns are fixed

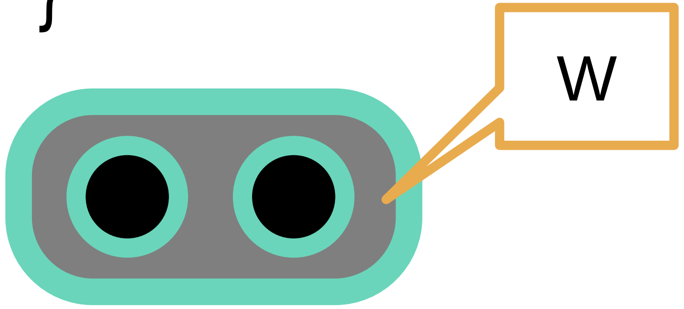
```
void access2D(char str[3][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 1, 0);
    return 0;
}
```



Passing 2D arrays to functions

Case 1: both the number of rows and the number of columns are fixed

```
void access2D(char str[3][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 1, 0);
    return 0;
}
```

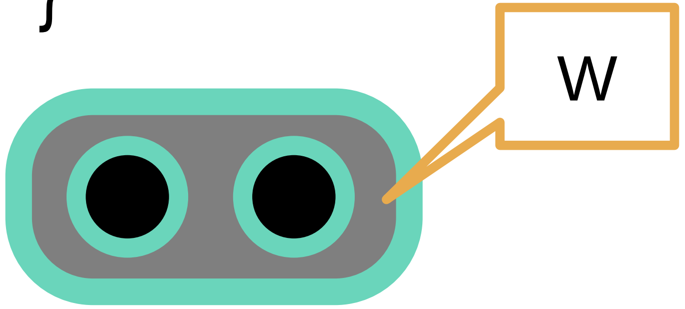


Passing 2D arrays to functions

Case 1: both the number of rows and the number of columns are fixed

Notice that our usual way of accessing array elements works here just fine!

```
void access2D(char str[3][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 1, 0);
    return 0;
}
```



Passing 2D arrays to functions

Passing 2D arrays to functions

Case 2: number of rows unknown but number of columns are fixed

Passing 2D arrays to functions

Case 2: number of rows unknown but number of columns are fixed

```
void access2D(char str[][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 2, 1);
    return 0;
}
```

Passing 2D arrays to functions

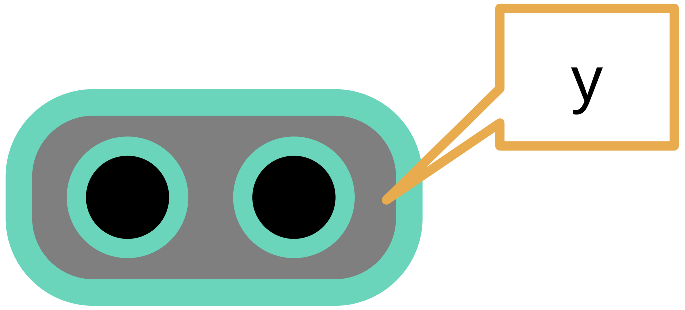
Case 2: number of rows unknown but number of columns are fixed

```
void access2D(char str[][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 2, 1);
    return 0;
}
```

Passing 2D arrays to functions

Case 2: number of rows unknown but number of columns are fixed

```
void access2D(char str[][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 2, 1);
    return 0;
}
```

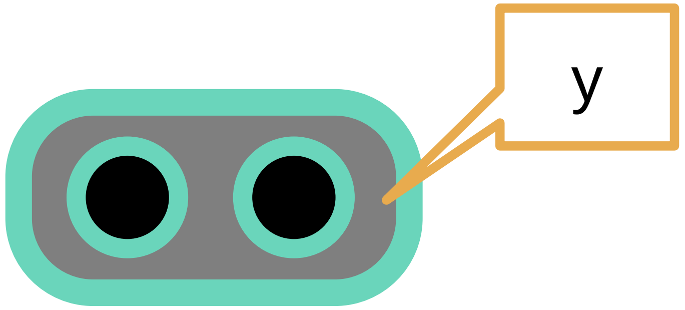


Passing 2D arrays to functions

Case 2: number of rows unknown but number of columns are fixed

Notice that our usual way of accessing array elements still works here just fine!

```
void access2D(char str[][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 2, 1);
    return 0;
}
```



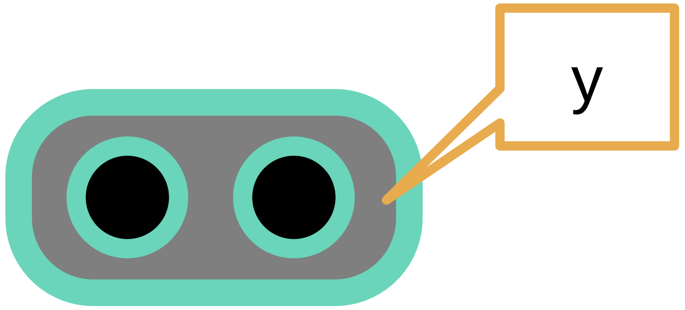
Passing 2D arrays to functions

Case 2: number of rows unknown but number of columns are fixed

Notice that our usual way of accessing array elements still works here just fine!

Note that specifying number of rows is not needed at all!

```
void access2D(char str[][5], int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    access2D(str, 2, 1);
    return 0;
}
```



Passing 2D arrays to functions

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Works since internally Mr C stores all 2D arrays as nothing but 1D arrays

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Works since internally Mr C stores all 2D arrays as nothing but 1D arrays

```
void access2D(char* str, int i, int j, int c){
    // c gives num of columns i.e. num of
    // elements in each row
    printf("%c", *(str + c * i + j));
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    char *ptr = &str[0][0];
    int numCols = 5;
    access2D(str, 1, 2, numCols);
    return 0;
}
```

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Works since internally Mr C stores all 2D arrays as nothing but 1D arrays

```
void access2D(char* str, int i, int j, int c){
    // c gives num of columns i.e. num of
    // elements in each row
    printf("%c", *(str + c * i + j));
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    char *ptr = &str[0][0];
    int numCols = 5;
    access2D(str, 1, 2, numCols);
    return 0;
}
```

Passing 2D arrays to functions

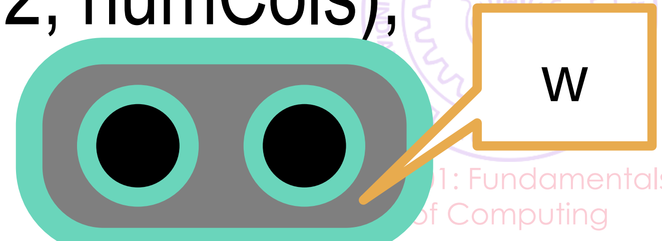
Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Works since internally Mr C stores all 2D arrays as nothing but 1D arrays

```
void access2D(char* str, int i, int j, int c){
    // c gives num of columns i.e. num of
    // elements in each row
    printf("%c", *(str + c * i + j));
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    char *ptr = &str[0][0];
    int numCols = 5;
    access2D(str, 1, 2, numCols);
    return 0;
}
```



Passing 2D arrays to functions

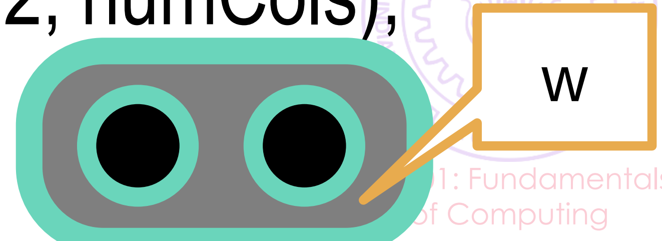
Cas oth num. of rows an nber of columns are unkn n ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Works since internally Mr C stores all 2D arrays as nothing but 1D arrays

```
void access2D(char* str, int i, int j, int c){
    // c gives num of columns i.e. num of
    // elements in each row
    printf("%c", *(str + c * i + j));
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    char *ptr = &str[0][0];
    int numCols = 5;
    access2D(str, 1, 2, numCols);
    return 0;
}
```



Passing

If accessing row index 0 column index 2, simply skip 2 elements of first row. If accessing row index 1 column index 2, first skip 5 elements of first row and then skip 2 elements of 2nd row

Cas

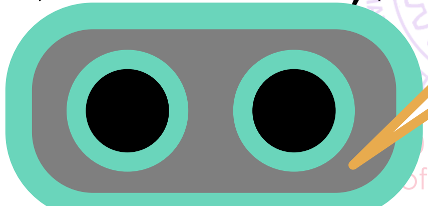
and nber of columns are unkn own ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 1: treat 2D array as a 1D array and do the indexing yourself ☺

Works since internally Mr C stores all 2D arrays as nothing but 1D arrays

```
c){
    // c gives num of columns i.e. num of
    // elements in each row
    printf("%c", *(str + c * i + j));
}
int main(){
    char str[3][5] = {"Hi","Wow","Bye"};
    char *ptr = &str[0][0];
    int numCols = 5;
    access2D(str, 1, 2, numCols);
    return 0;
}
```



Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 2: create array of arrays

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn’t know how to access 2nd row elements

Trick 2: create array of arrays

Advantage: hassle free indexing

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 2: create array of arrays

- Advantage: hassle free indexing
- Disadvantage: write code for malloc ☺

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 2: create array of arrays

- Advantage: hassle free indexing
- Disadvantage: write code for malloc ☺
- This works since going to the second row does not require knowing how many elements are there in first row

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

Here, Mr C doesn't know how to access 2nd row elements

Trick 2: create array of arrays

- Advantage: hassle free indexing
- Disadvantage: write code for malloc ☺
- This works since going to the second row does not require knowing how many elements are there in first row
- In case of arrays of arrays, every row has a separate pointer pointing to its first element – see last week's lecture

Passing 2D arrays to functions

Case 3: both num. of rows and number of columns are unknown ☹

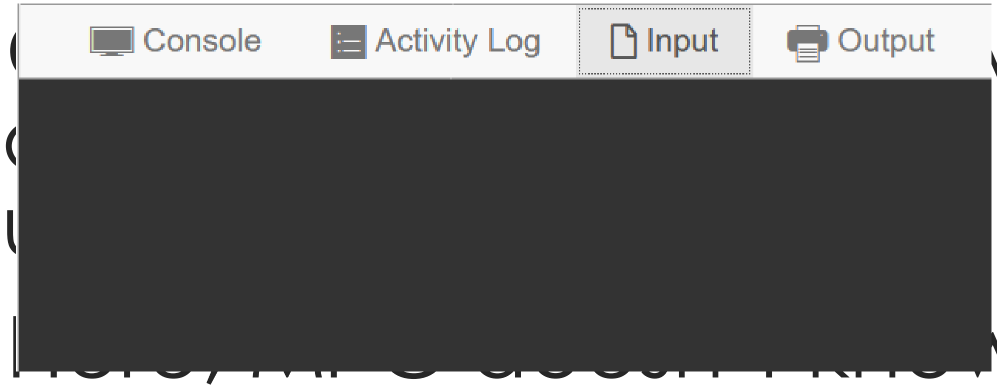
Here, Mr C doesn't know how to access 2nd row elements

Trick 2: create array of arrays

- Advantage: hassle free indexing
- Disadvantage: write code for malloc ☺
- This works since going to the second row does not require knowing how many elements are there in first row
- In case of arrays of arrays, every row has a separate pointer pointing to its first element – see last week's lecture

```
void access2D(char** str, int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char **str = (char**)malloc(3*sizeof(char*));
    int i;
    for(i = 0; i < 3; i++){
        str[i] = (char*)malloc(5*sizeof(char));
        gets(str[i]);
    }
    access2D(str, 1, 1);
    return 0;
}
```

Passing 2D arrays to functions



ws

are

w how to access 2nd row elements

Trick 2: create array of arrays

Advantage: hassle free indexing

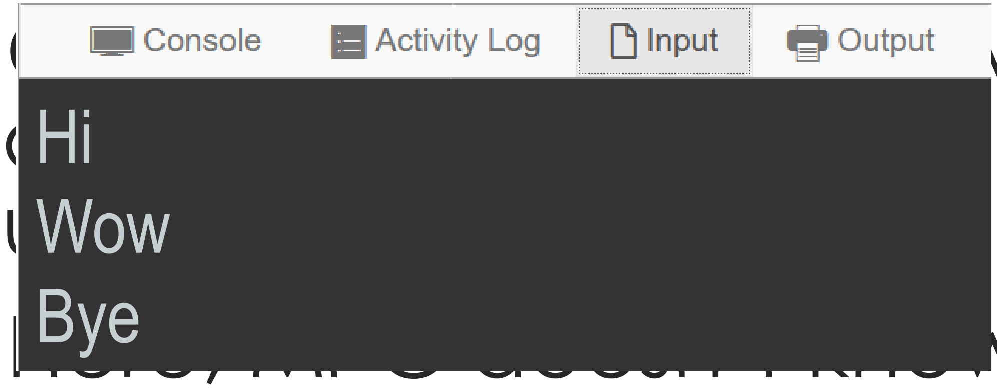
Disadvantage: write code for malloc ☺

This works since going to the second row does not require knowing how many elements are there in first row

In case of arrays of arrays, every row has a separate pointer pointing to its first element – see last week's lecture

```
void access2D(char** str, int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char **str = (char**)malloc(3*sizeof(char*));
    int i;
    for(i = 0; i < 3; i++){
        str[i] = (char*)malloc(5*sizeof(char));
        gets(str[i]);
    }
    access2D(str, 1, 1);
    return 0;
}
```

Passing 2D arrays to functions



```
Hi
Wow
Bye
```

ws are

v how to access 2nd row elements

Trick 2: create array of arrays

<u>Advantage</u>: hassle free indexing

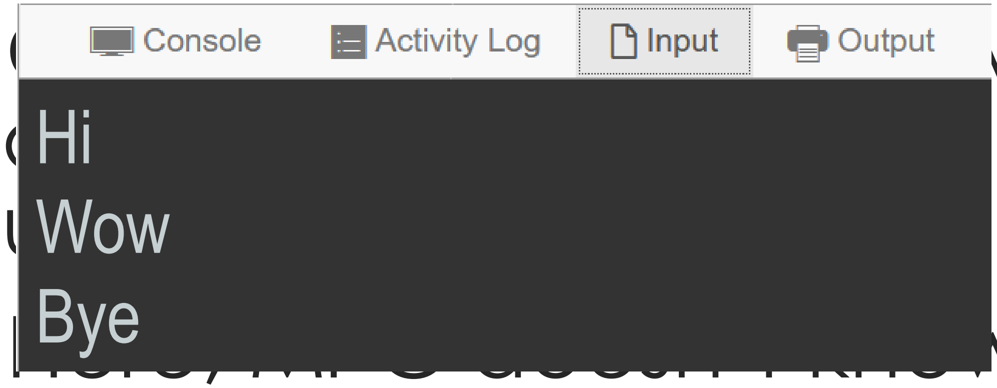
<u>Disadvantage</u>: write code for malloc ☺

This works since going to the second row does not require knowing how many elements are there in first row

In case of arrays of arrays, every row has a separate pointer pointing to its first element – see last week's lecture

```
void access2D(char** str, int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char **str = (char**)malloc(3*sizeof(char*));
    int i;
    for(i = 0; i < 3; i++){
        str[i] = (char*)malloc(5*sizeof(char));
        gets(str[i]);
    }
    access2D(str, 1, 1);
    return 0;
}
```

Passing 2D arrays to functions



```
Hi
Wow
Bye
```

ws
are

v how
to access 2nd row elements

Trick 2: create array of arrays

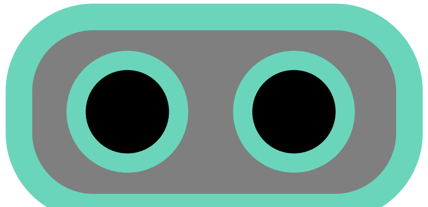
<u>Advantage</u>: hassle free indexing

<u>Disadvantage</u>: write code for malloc ☺

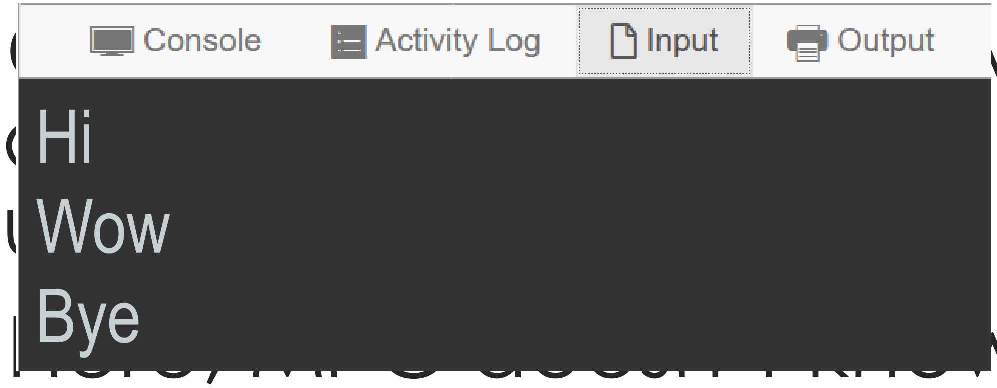
This works since going to the second row does not require knowing how many elements are there in first row

In case of arrays of arrays, every row has a separate pointer pointing to its first element – see last week's lecture

```
void access2D(char** str, int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char **str = (char**)malloc(3*sizeof(char*));
    int i;
    for(i = 0; i < 3; i++){
        str[i] = (char*)malloc(5*sizeof(char));
        gets(str[i]);
    }
    access2D(str, 1, 1);
    return 0;
}
```



Passing 2D arrays to functions



```
Hi
Wow
Bye
```

ws

are

v how
to access 2nd row elements

Trick 2: create array of arrays

Advantage: hassle free indexing

Disadvantage: write code for malloc ☺

This works since going to the second row does not require knowing how many elements are there in first row

In case of arrays of arrays, every row has a separate pointer pointing to its first element – see last week's lecture

```
void access2D(char** str, int i, int j){
    printf("%c", str[i][j]);
}
int main(){
    char **str = (char**)malloc(3*sizeof(char*));
    int i;
    for(i = 0; i < 3; i++){
        str[i] = (char*)malloc(5*sizeof(char));
        gets(str[i]);
    }
    access2D(str, 1, 1);
    return 0;
}
```