Inception with Mr C

ESC101: Fundamentals of Computing

Purushottam Kar

Announcements

- Replacement lab October 6 (Sat) 2-5PM at NCL for sections B4, B5, B6, B13,
- No replacement lecture since not a “DoAA Saturday”
- Extra doubt clearing session after extra lab on Sat i.e. Oct 06 (Sat) 5-6PM CC-02.

BODMAS table has more members

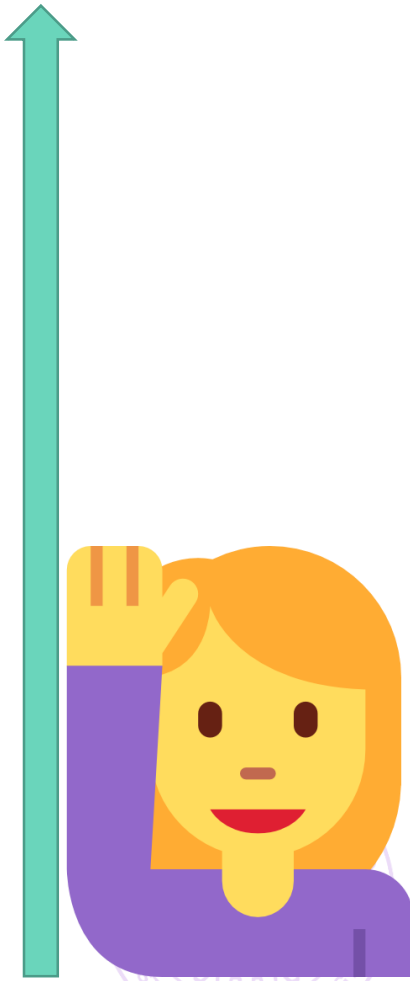
Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, *, &, sizeof	Right
Multiplication/division/ remainder	*, /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >, >=	Left
Relational	==, !=	Left
AND	&&	Left
OR	||	Left
Ternary Conditional	? :	Right
Assignment, Compound assignment	=, +=, -=, *=, /=, %=	Right

Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, *, &, sizeof	Right
Multiplication/division/ remainder	*, /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >, >=	Left
Relational	==, !=	Left
AND	&&	Left
OR	||	Left
Ternary Conditional	? :	Right
Assignment, Compound assignment	=, +=, -=, *=, /=, %=	Right

embers 3

Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, *, &, sizeof	Right
Multiplication/division/ remainder	*, /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >, >=	Left
Relational	==, !=	Left
AND	&&	Left
OR	||	Left
Ternary Conditional	? :	Right
Assignment, Compound assignment	=, +=, -=, *=, /=, %=	Right

HIGH PRECEDENCE



LOW PRECEDENCE

embers 3

Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, * &, sizeof	Right
Multiplication/division/ remainder	* /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >,	
Relational	==, !=	
AND	&&	
OR	||	Left
Ternary Conditional	? :	Right
Assignment, Compound assignment	=, +=, -=, *=, /=, %=	Right

HIGH PRECEDENCE

LOW PRECEDENCE

Be careful, * can act as multiplication operator as well as dereference operator

embers 3

Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, * &, sizeof	Right
Multiplication/division/ remainder	* /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >,	
Relational	==, !=	
AND	&&	
OR	||	L
Ternary Conditional	? :	Ri
Assignment, Compound assignment	=, +=, -=, *=, /=, %=	Ri

HIGH PRECEDENCE

LOW PRECEDENCE

Be careful, * can act as multiplication operator as well as dereference operator

```
int a = 10;
int *ptr = &a;
printf("%d", 3**ptr);
```

embers 3

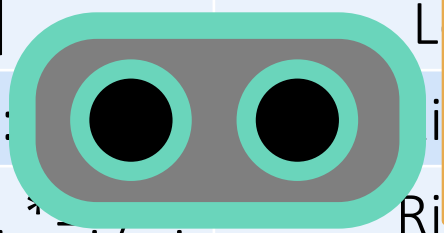
Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, * &, sizeof	Right
Multiplication/division/ remainder	* /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >,	
Relational	==, !=	
AND	&&	
OR	||	L
Ternary Conditional	? :	
Assignment, Compound assignment	=, +=, -=, %=	Right

HIGH PRECEDENCE

LOW PRECEDENCE

Be careful, * can act as multiplication operator as well as dereference operator

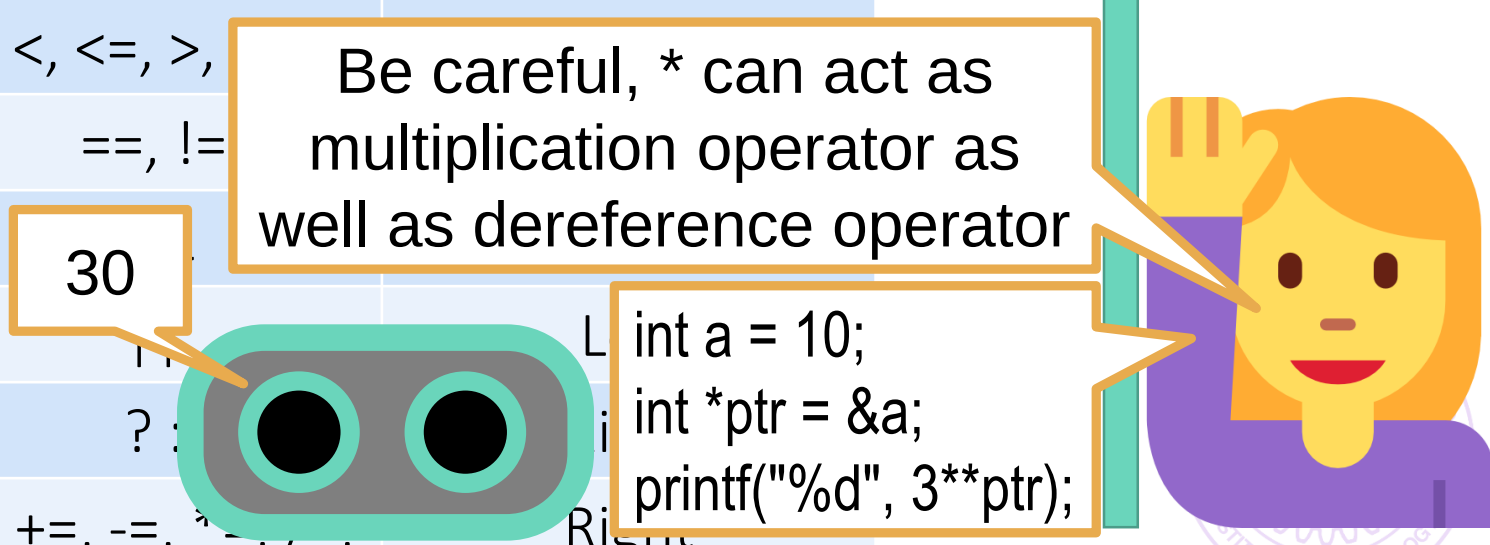
```
int a = 10;
int *ptr = &a;
printf("%d", 3**ptr);
```



Operator Name	Symbol/Sign	Associativity
Brackets (**array subscript**), Post increment/decrement	(), [] ++, --	Left
Unary negation, Pre-increment/decrement, NOT, (de)reference, sizeof	-, ++, --, !, * &, sizeof	Right
Multiplication/division/ remainder	* /, %	Left
Addition/subtraction	+, -	Left
Relational	<, <=, >,	
Relational	==, !=	
AND		
OR		L
Ternary Conditional	? :	
Assignment, Compound assignment	=, +=, -=, %=	Right

HIGH PRECEDENCE

LOW PRECEDENCE



On-demand memory allocation

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

On-demand memory allocation

- Allocate as much memory you need, when you need it ☺
- Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330
- Can adapt to changing user demands and work load
  - For example, Prutor has to deal with extremely variable workload

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

- For example, Prutor has to deal with extremely variable workload
- ~10-15 users simultaneously logged in during 12noon on weekdays

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

- For example, Prutor has to deal with extremely variable workload
- ~10-15 users simultaneously logged in during 12noon on weekdays
- ~150 users simultaneously logged in during Tue/Wed labs

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

- For example, Prutor has to deal with extremely variable workload
- ~10-15 users simultaneously logged in during 12noon on weekdays
- ~150 users simultaneously logged in during Tue/Wed labs
- ~250 users simultaneously logged in during lab exam

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

- For example, Prutor has to deal with extremely variable workload
- ~10-15 users simultaneously logged in during 12noon on weekdays
- ~150 users simultaneously logged in during Tue/Wed labs
- ~250 users simultaneously logged in during lab exam
- ~450 users simultaneously logged in the night before the exam

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

- For example, Prutor has to deal with extremely variable workload
- ~10-15 users simultaneously logged in during 12noon on weekdays
- ~150 users simultaneously logged in during Tue/Wed labs
- ~250 users simultaneously logged in during lab exam
- ~450 users simultaneously logged in the night before the exam

Need to include stdlib.h for dynamic memory allocation

On-demand memory allocation

Allocate as much memory you need, when you need it ☺

Avoid wastage of space – programs that take too much memory are often slow as well – details in CS330

Can adapt to changing user demands and work load

- For example, Prutor has to deal with extremely variable workload
- ~10-15 users simultaneously logged in during 12noon on weekdays
- ~150 users simultaneously logged in during Tue/Wed labs
- ~250 users simultaneously logged in during lab exam
- ~450 users simultaneously logged in the night before the exam

Need to include stdlib.h for dynamic memory allocation

- malloc(), calloc(), realloc(), free()

malloc – **m**emory **alloc**ation

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required and malloc allocates those many **consecutive** bytes and returns the address of (a pointer to) the first byte

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required and malloc allocates those many **consecutive** bytes and returns the address of (a pointer to) the first byte

Warning: allocated bytes filled with garbage

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required and malloc allocates those many **consecutive** bytes and returns the address of (a pointer to) the first byte

Warning: allocated bytes filled with garbage

Warning: if insufficient memory, NULL pointer returned

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required and malloc allocates those many **consecutive** bytes and returns the address of (a pointer to) the first byte

Warning: allocated bytes filled with garbage

Warning: if insufficient memory, NULL pointer returned

malloc has no idea if we are allocating an array of floats or chars – returns a void* pointer – typecast it yourself

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required and malloc allocates those many **consecutive** bytes and returns the address of (a pointer to) the first byte

Warning: allocated bytes filled with garbage

Warning: if insufficient memory, NULL pointer returned

malloc has no idea if we are allocating an array of floats or chars – returns a void* pointer – typecast it yourself

The allocated memory can be used safely as an array

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required and malloc allocates those many **consecutive** bytes and returns the address of (a pointer to) the first byte

Warning: allocated bytes filled with garbage

Warning: if insufficient memory, NULL pointer returned

malloc has no idea if we are allocating an array of floats or chars – returns a void* pointer – typecast it yourself

The allocated memory can be used safely as an array

```
int *ptr = (int*)malloc(10*sizeof(int));
*(ptr + 2) = 42; // *(ptr + 2) is the same as ptr[2]
printf("%d", ptr[2]); // prints 42
```

calloc – **c**ontiguous **alloc**ation

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

Use this if you actually want zero initialization

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

Use this if you actually want zero initialization

Syntax a bit different – instead of total number of bytes, we need to send it two things

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

Use this if you actually want zero initialization

Syntax a bit different – instead of total number of bytes, we need to send it two things

- length of array (number of elements in the array)

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

Use this if you actually want zero initialization

Syntax a bit different – instead of total number of bytes, we need to send it two things

- length of array (number of elements in the array)
- number of bytes per element

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

Use this if you actually want zero initialization

Syntax a bit different – instead of total number of bytes, we need to send it two things

- length of array (number of elements in the array)
- number of bytes per element

Sends back a NULL pointer if insufficient memory – **careful**!

calloc – **c**ontiguous **alloc**ation

A helpful version of malloc that initializes memory to 0 ☺

However, slower than malloc since time spent initializing

Use this if you actually want zero initialization

Syntax a bit different – instead of total number of bytes, we need to send it two things

- length of array (number of elements in the array)
- number of bytes per element

Sends back a NULL pointer if insufficient memory – **careful**!

Need to typecast the pointer returned by calloc too!

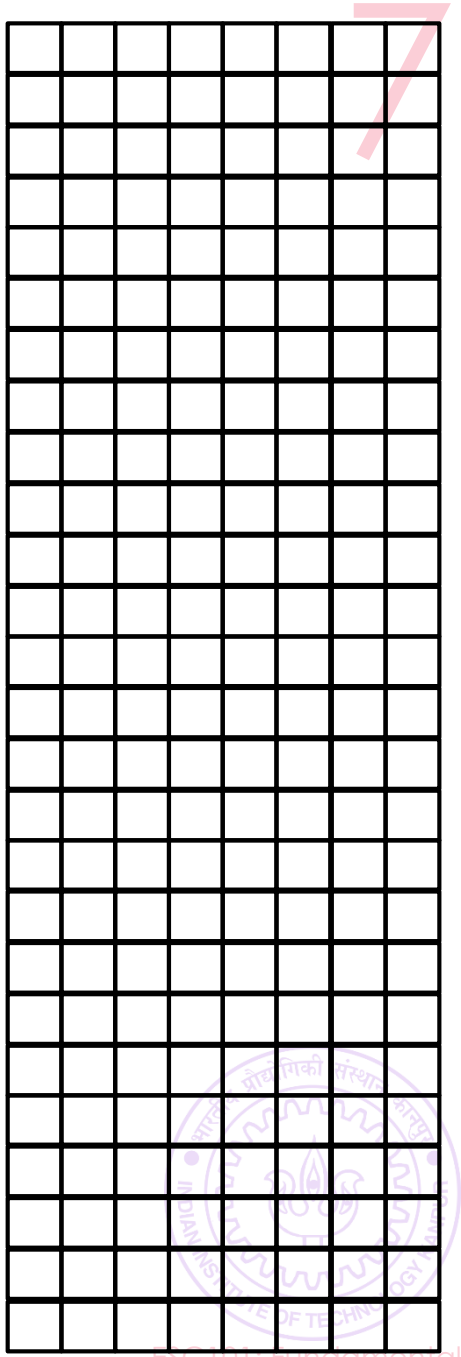
Memory leaks

Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

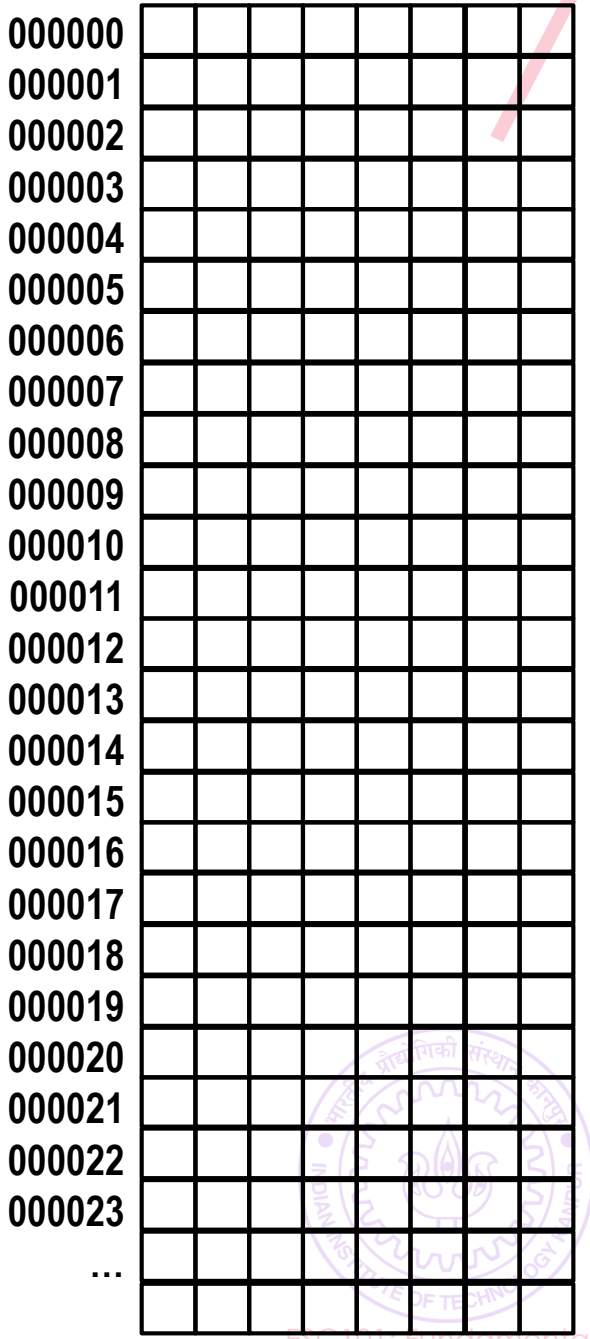
Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹



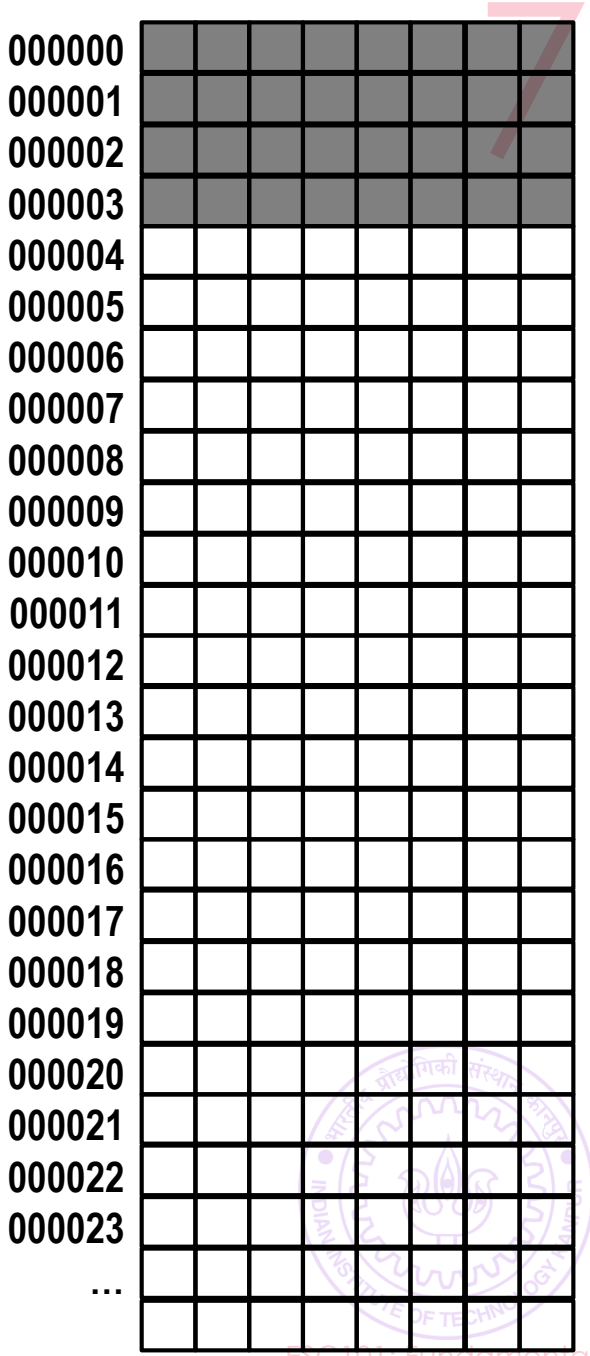
Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹



Memory leaks

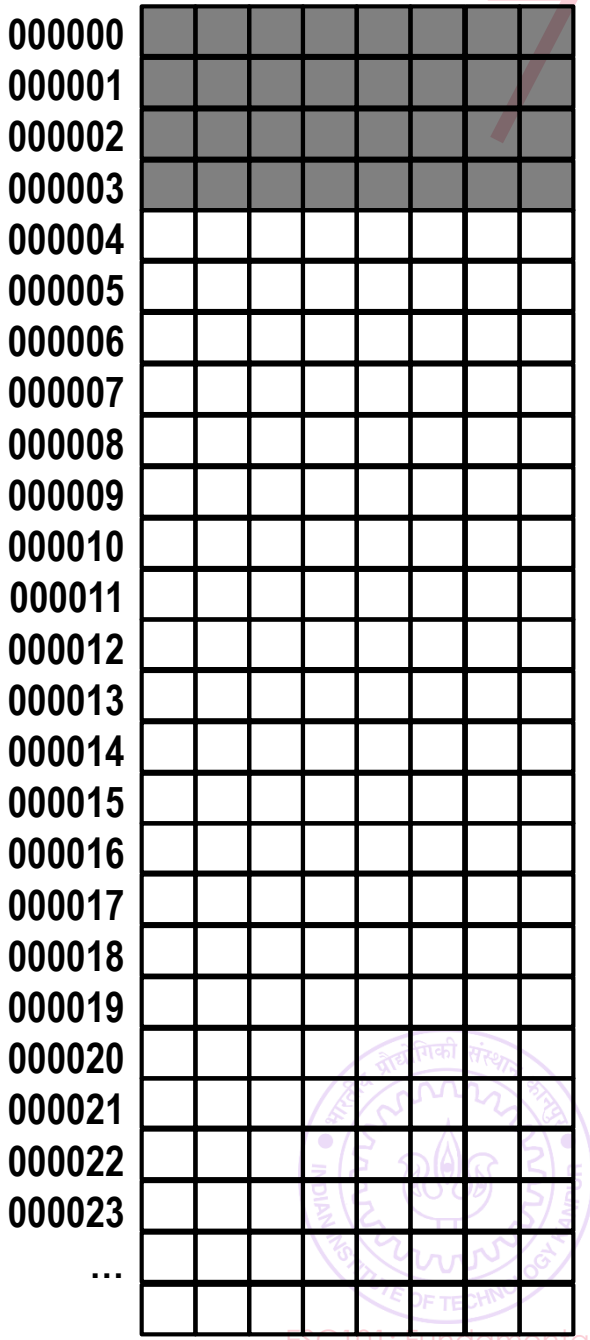
Situation where memory allocated earlier becomes unusable and blocked ☹



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

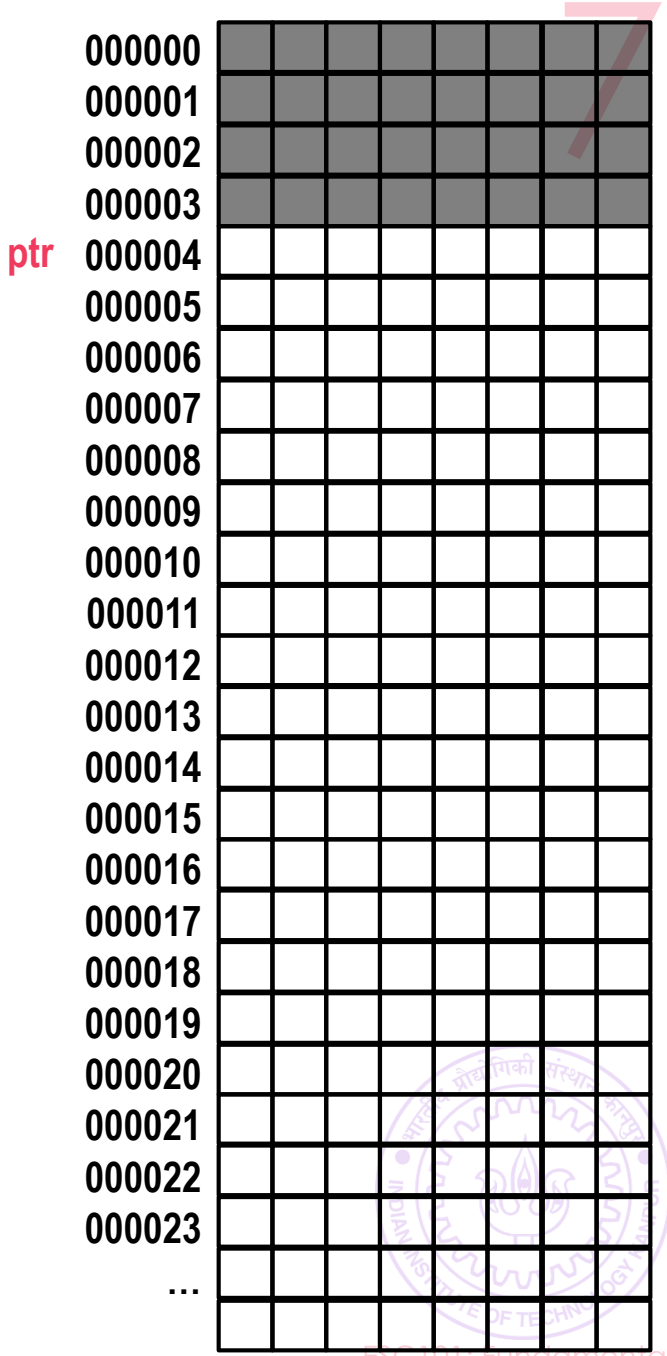
```
int *ptr;
```



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

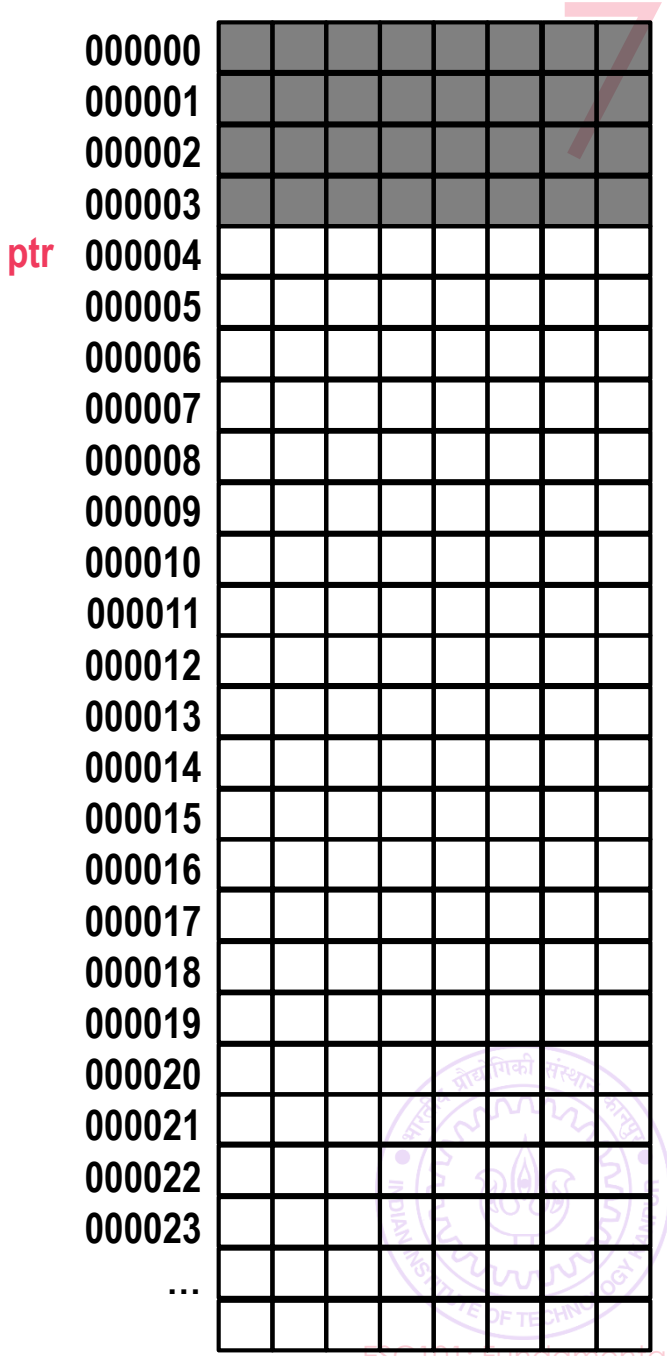
```
int *ptr;
```



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

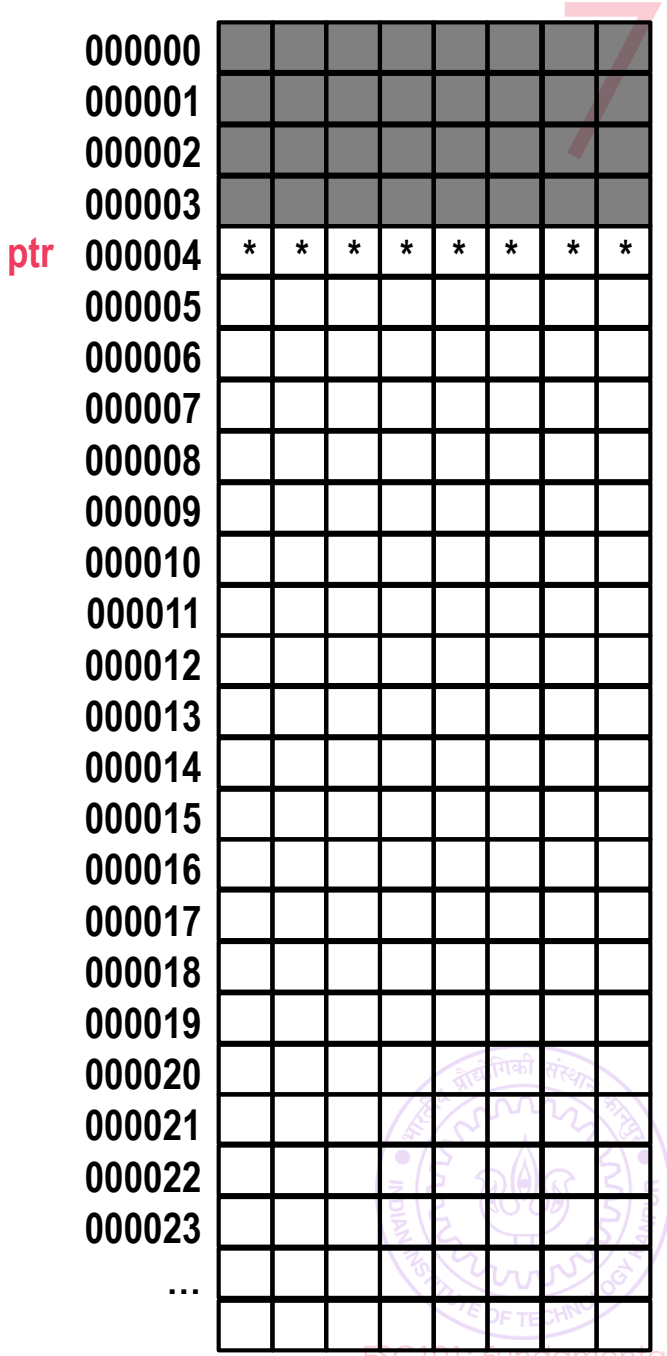
int *ptr; // may contain a junk address now



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

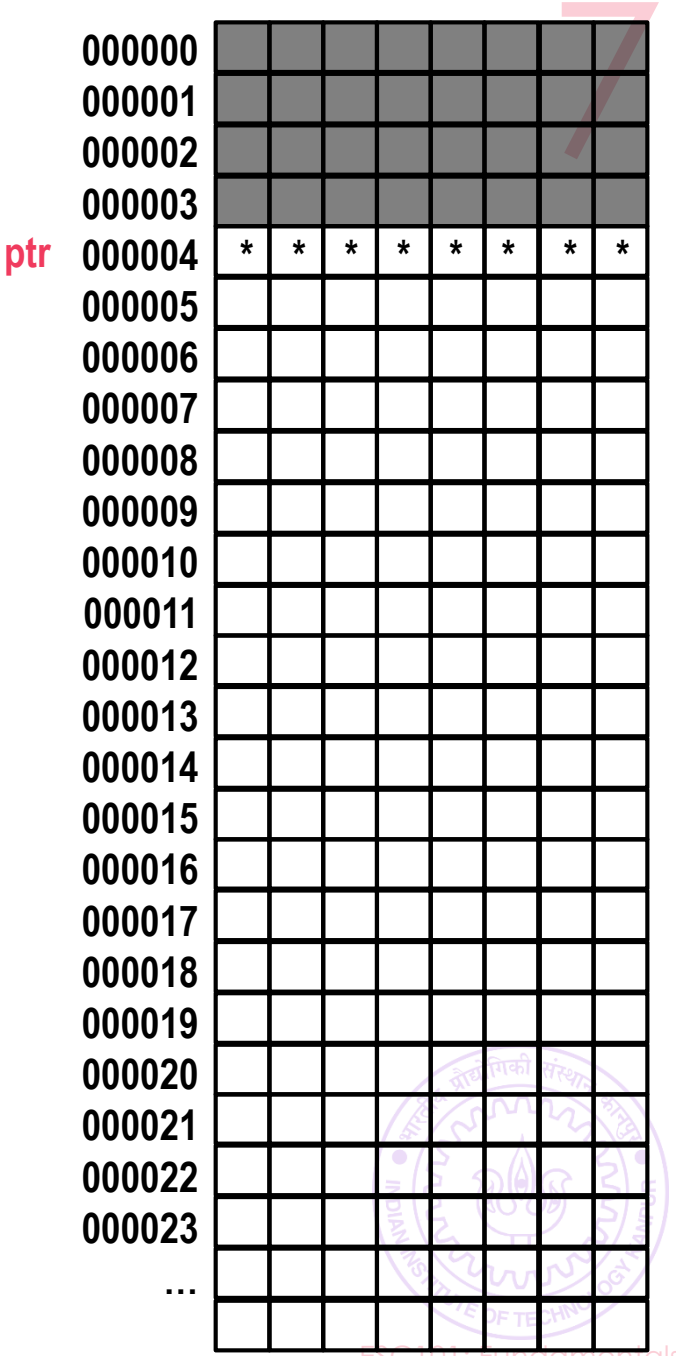
int *ptr; // may contain a junk address now



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

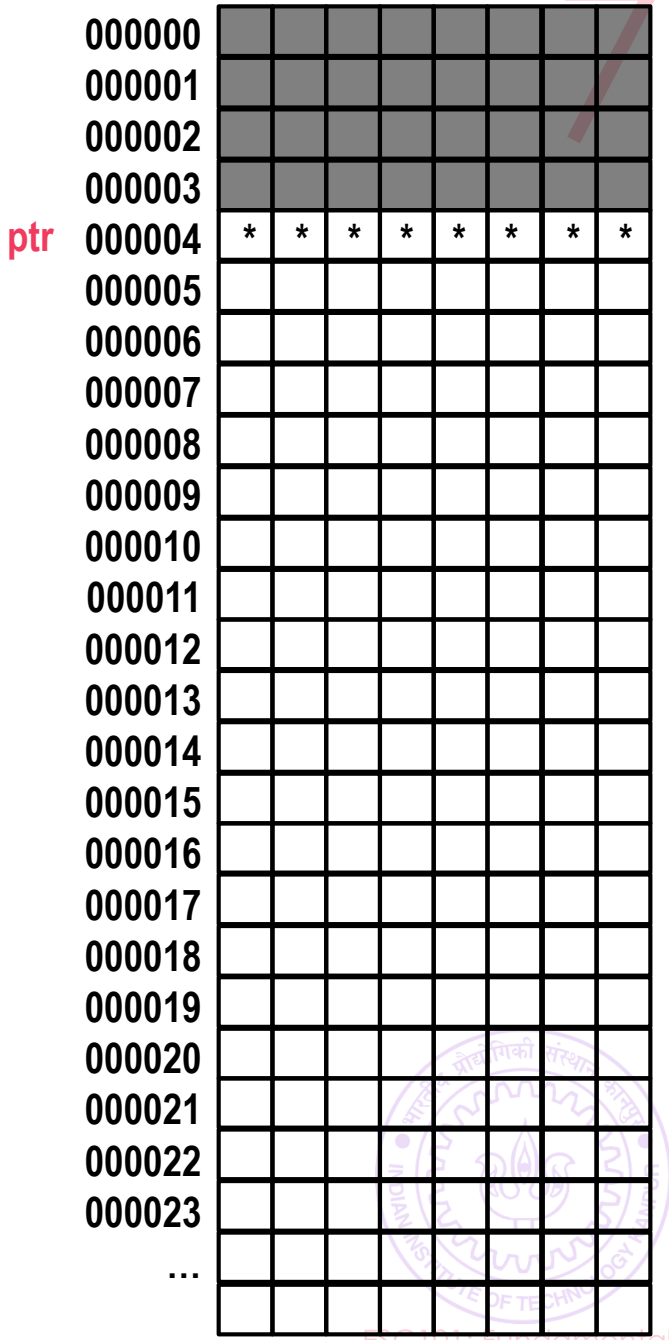
```
int *ptr; // may contain a junk address now
```



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

int *ptr; // may contain a junk address now

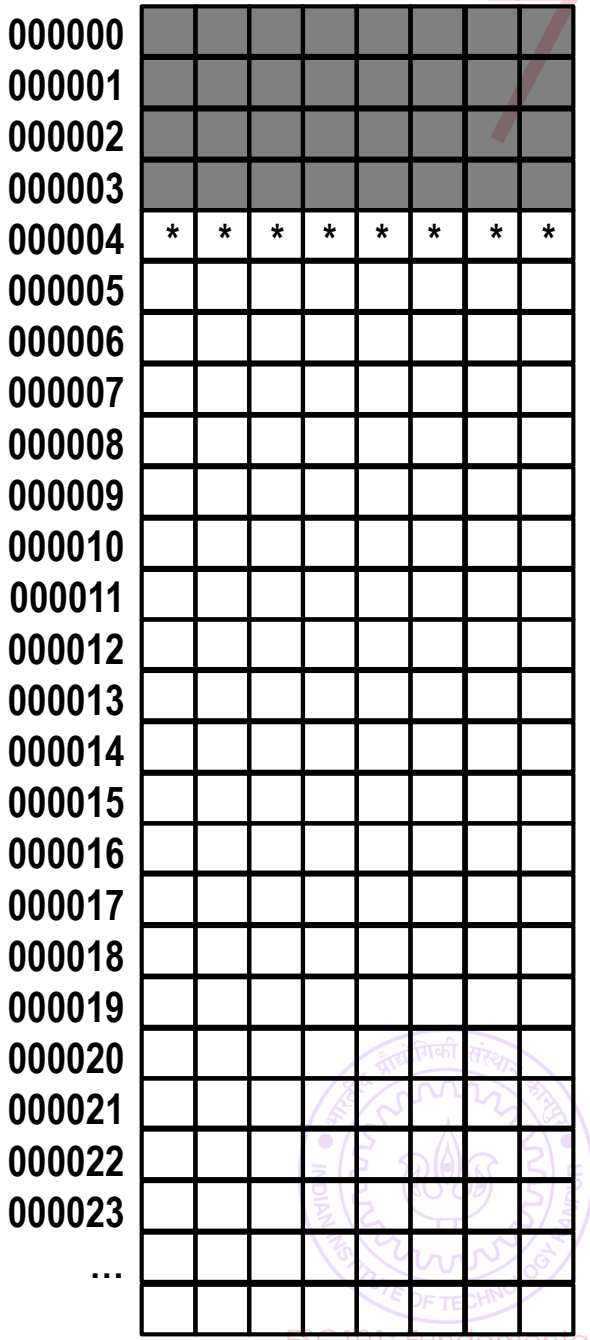


ptr will take 8 bytes to store –
sorry for not drawing accurately

Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
```

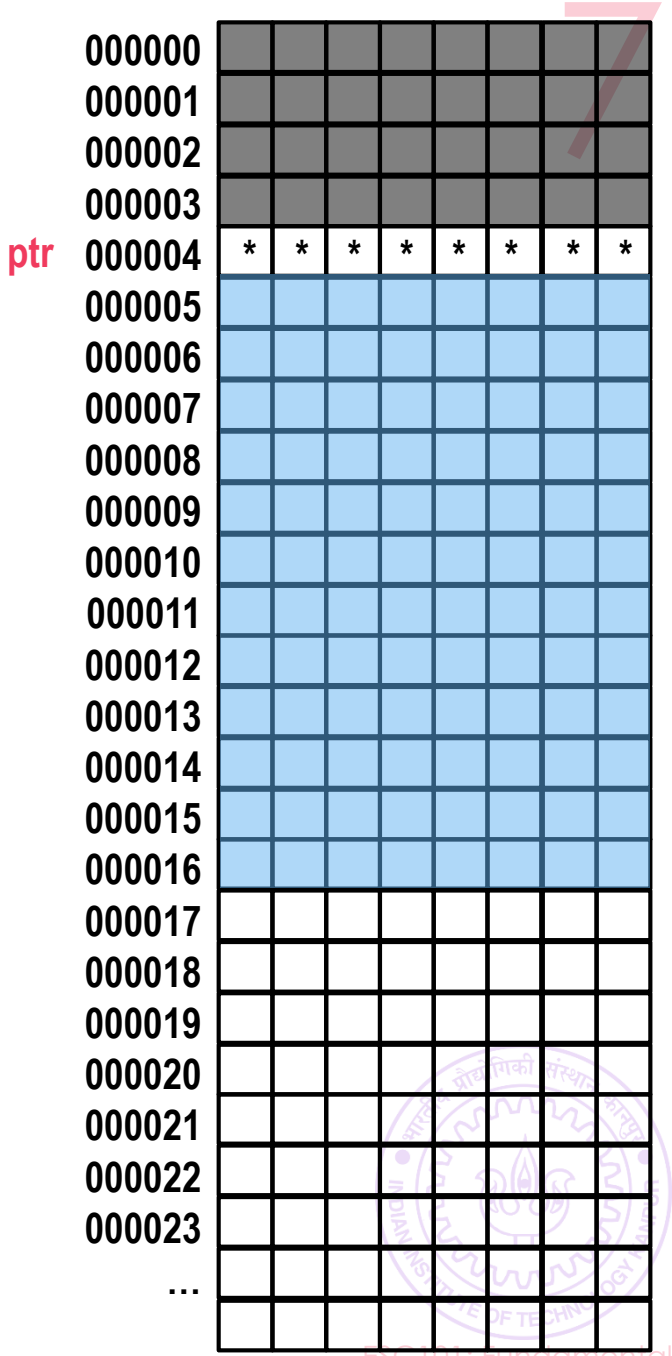


ptr will take 8 bytes to store –
sorry for not drawing accurately

Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
```

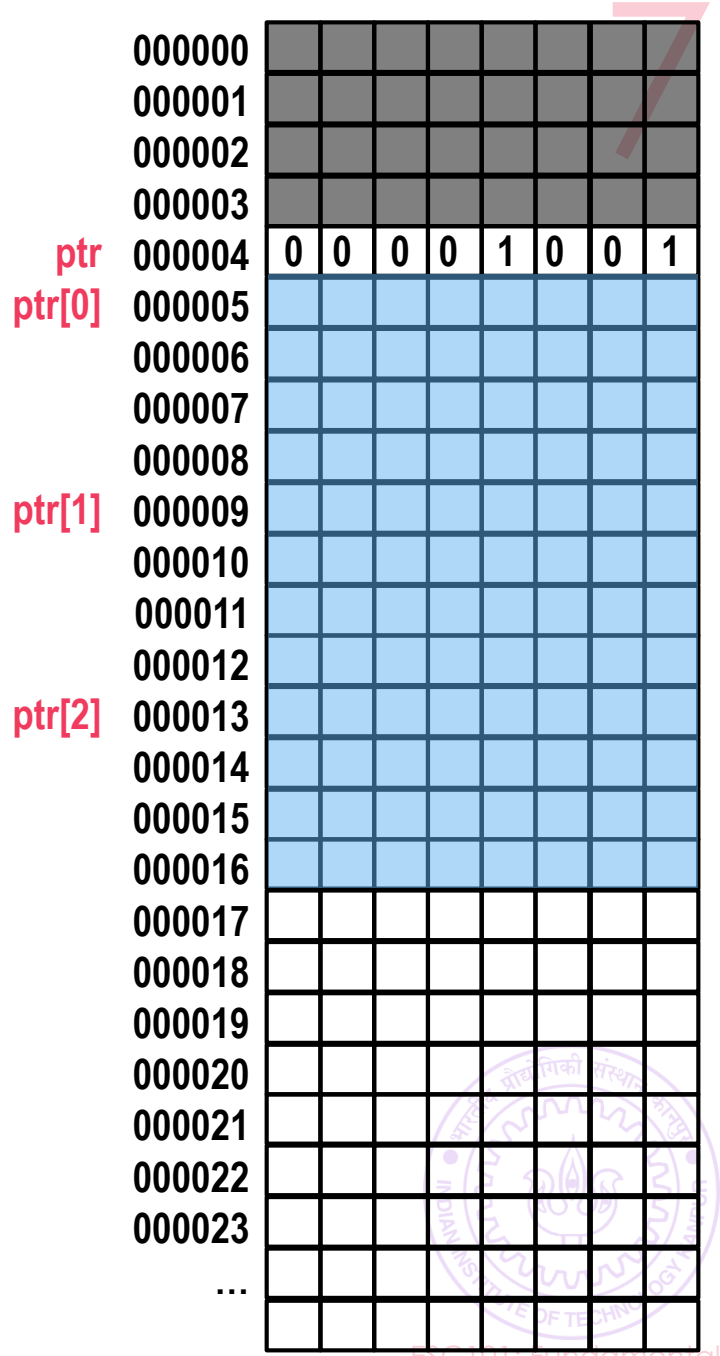


ptr will take 8 bytes to store –
sorry for not drawing accurately

Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
```

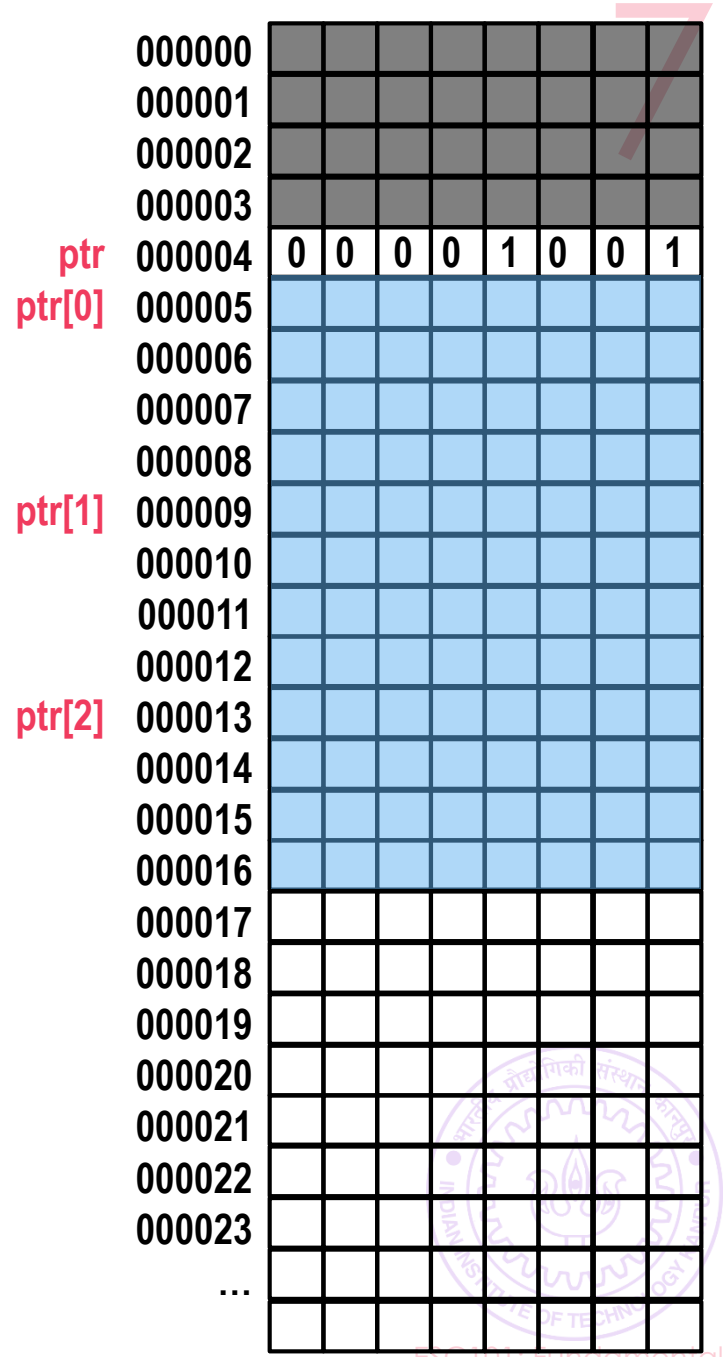


ptr will take 8 bytes to store –
sorry for not drawing accurately

Memory leaks

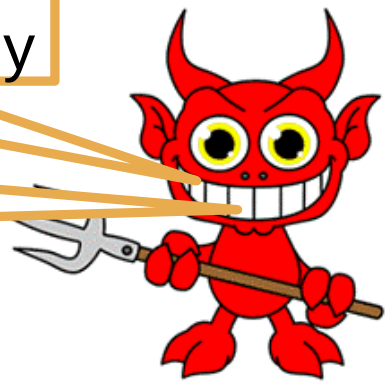
Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
```



ptr will take 8 bytes to store – sorry for not drawing accurately

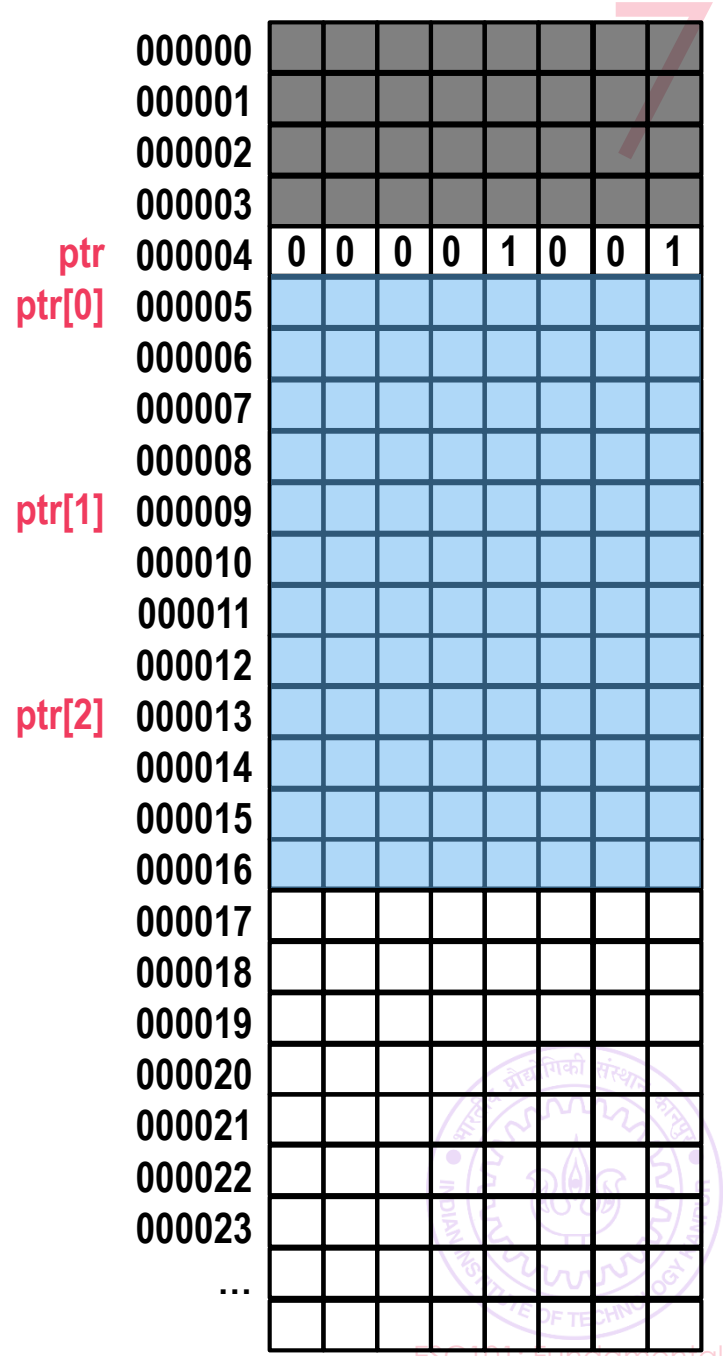
Actually even this is not accurate ☹ – details in CS220/330/335 ☺



Memory leaks

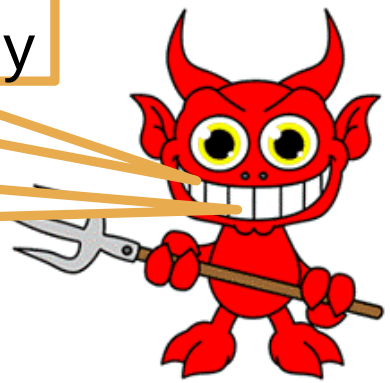
Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
…
```



ptr will take 8 bytes to store – sorry for not drawing accurately

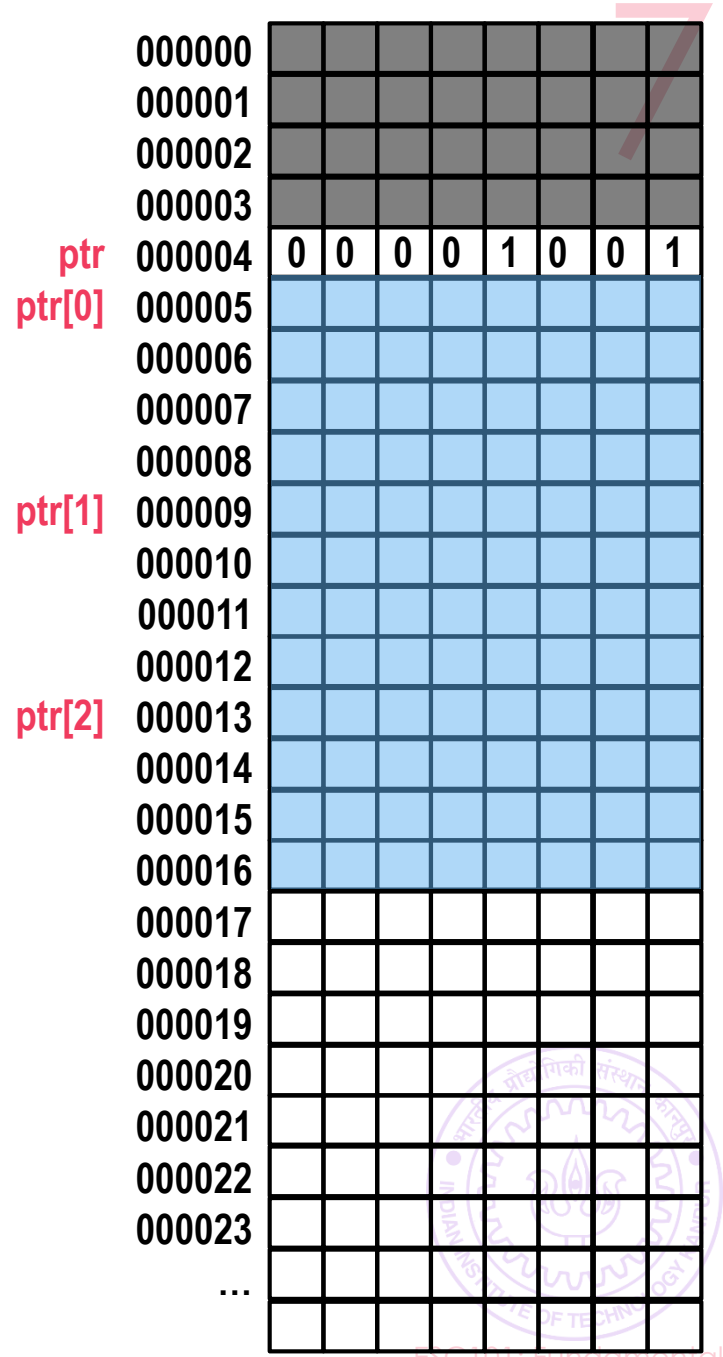
Actually even this is not accurate ☹ – details in CS220/330/335 ☺



Memory leaks

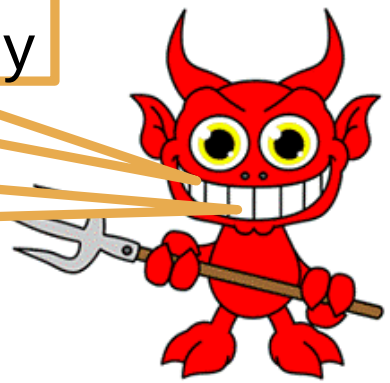
Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
…
ptr = (int*)malloc(2 * sizeof(int));
```



ptr will take 8 bytes to store – sorry for not drawing accurately

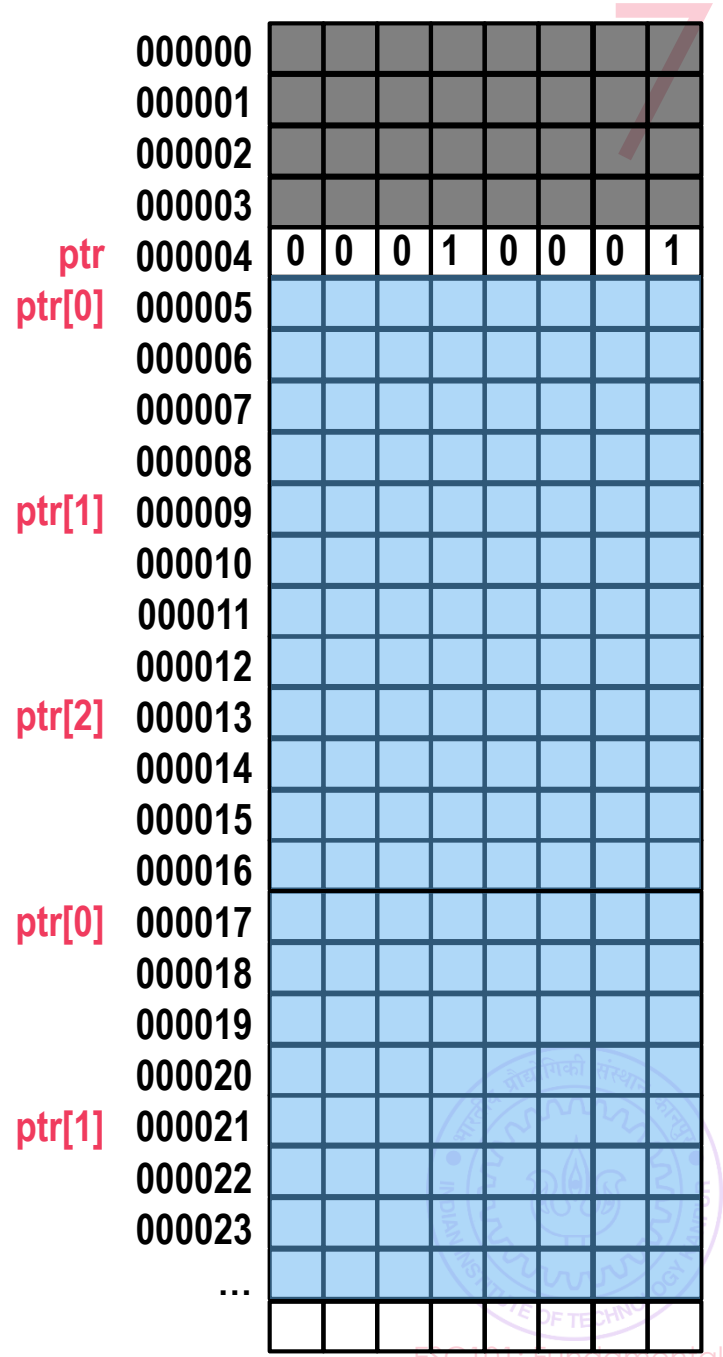
Actually even this is not accurate ☹ – details in CS220/330/335 ☺



Memory leaks

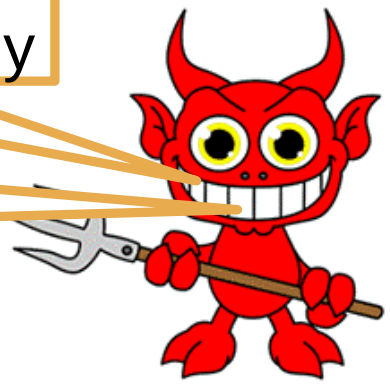
Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
…
ptr = (int*)malloc(2 * sizeof(int));
```



ptr will take 8 bytes to store – sorry for not drawing accurately

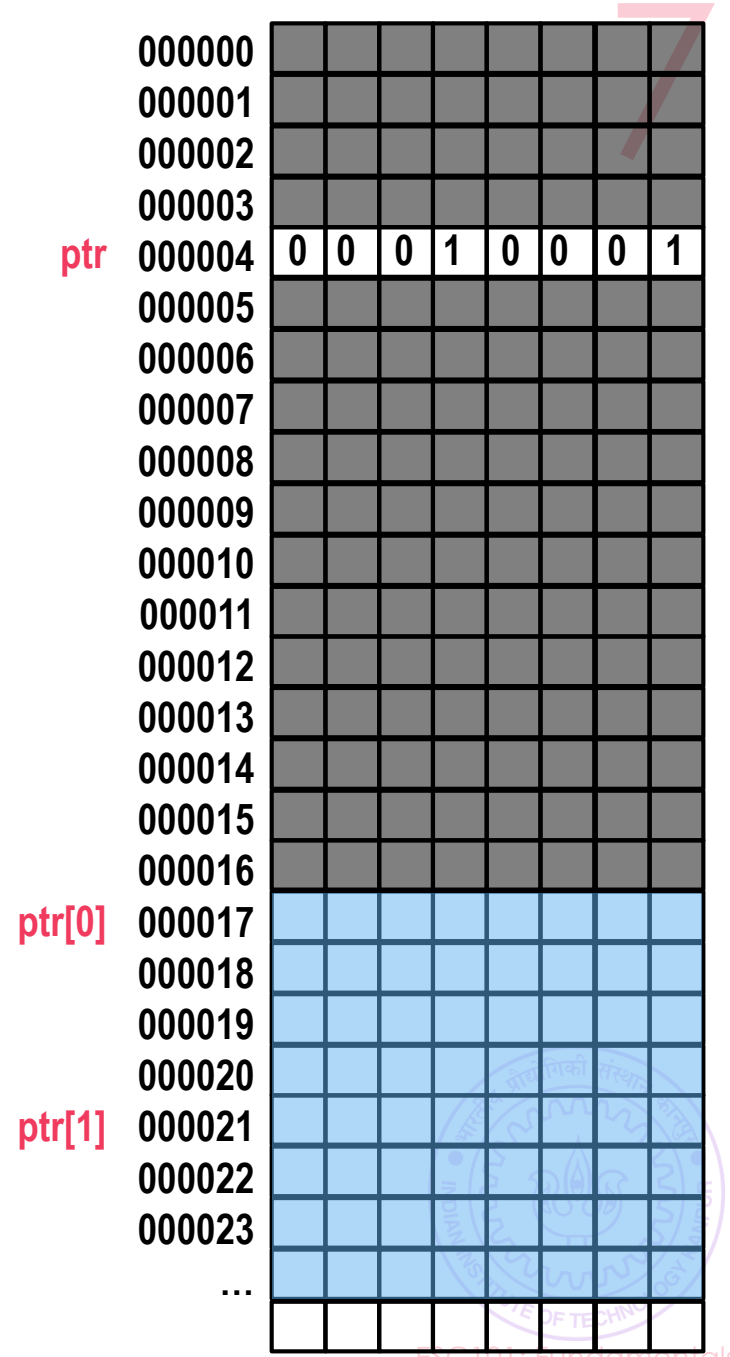
Actually even this is not accurate ☹ – details in CS220/330/335 ☺



Memory leaks

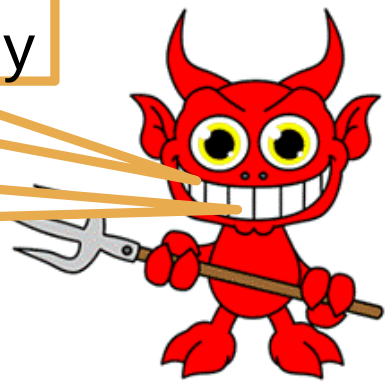
Situation where memory allocated earlier becomes unusable and blocked ☹

```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
…
ptr = (int*)malloc(2 * sizeof(int));
```



ptr will take 8 bytes to store – sorry for not drawing accurately

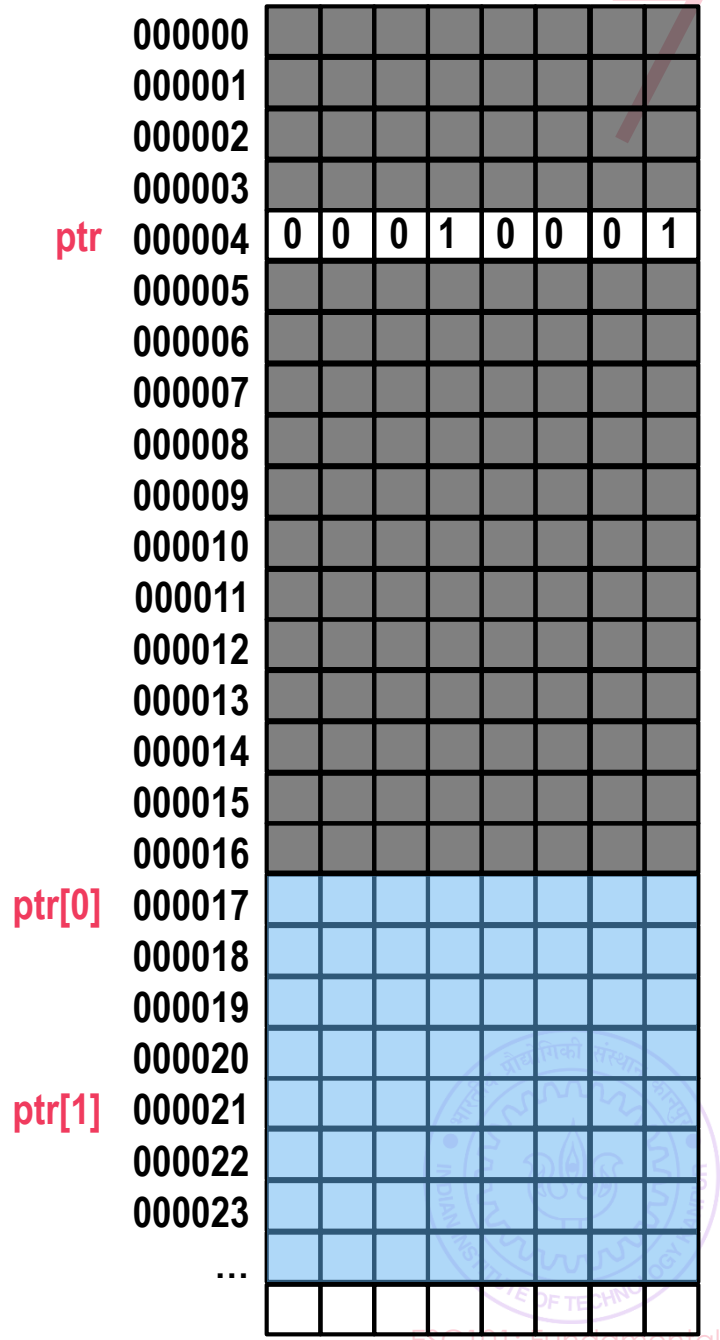
Actually even this is not accurate ☹ – details in CS220/330/335 ☺



Memory leaks

Situation where memory allocated earlier becomes unusable and blocked ☹

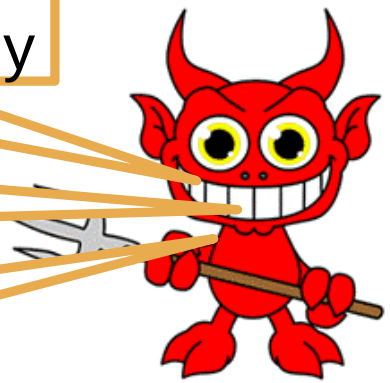
```
int *ptr; // may contain a junk address now
ptr = (int*)malloc(3 * sizeof(int));
…
ptr = (int*)malloc(2 * sizeof(int));
```



ptr will take 8 bytes to store – sorry for not drawing accurately

Actually even this is not accurate ☹ – details in CS220/330/335 ☺

If you keep losing memory like this, soon your program may crash!



free

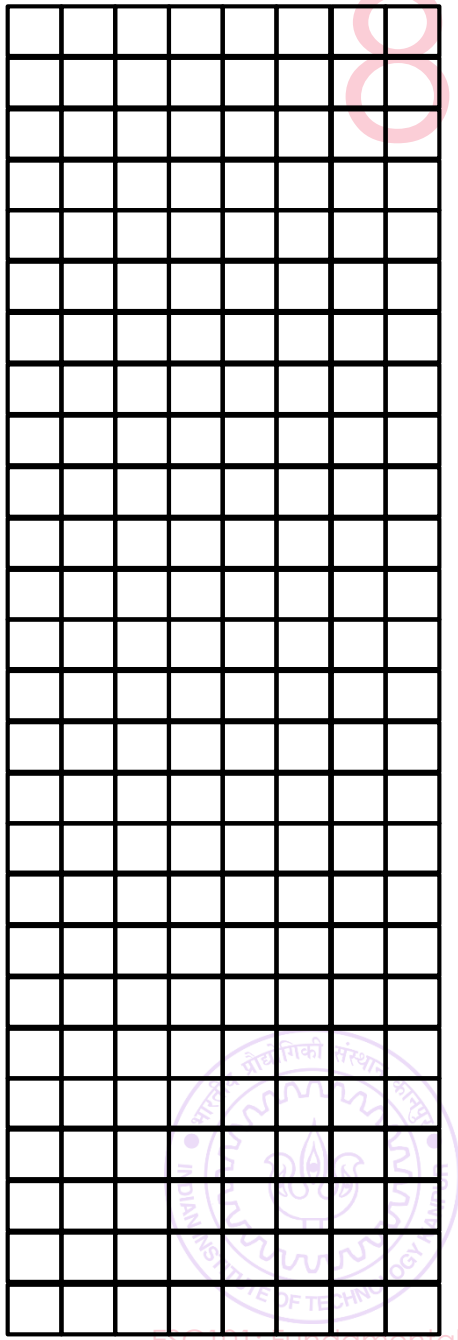
भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

free

Used to deallocate memory allocated using malloc/calloc/realloc

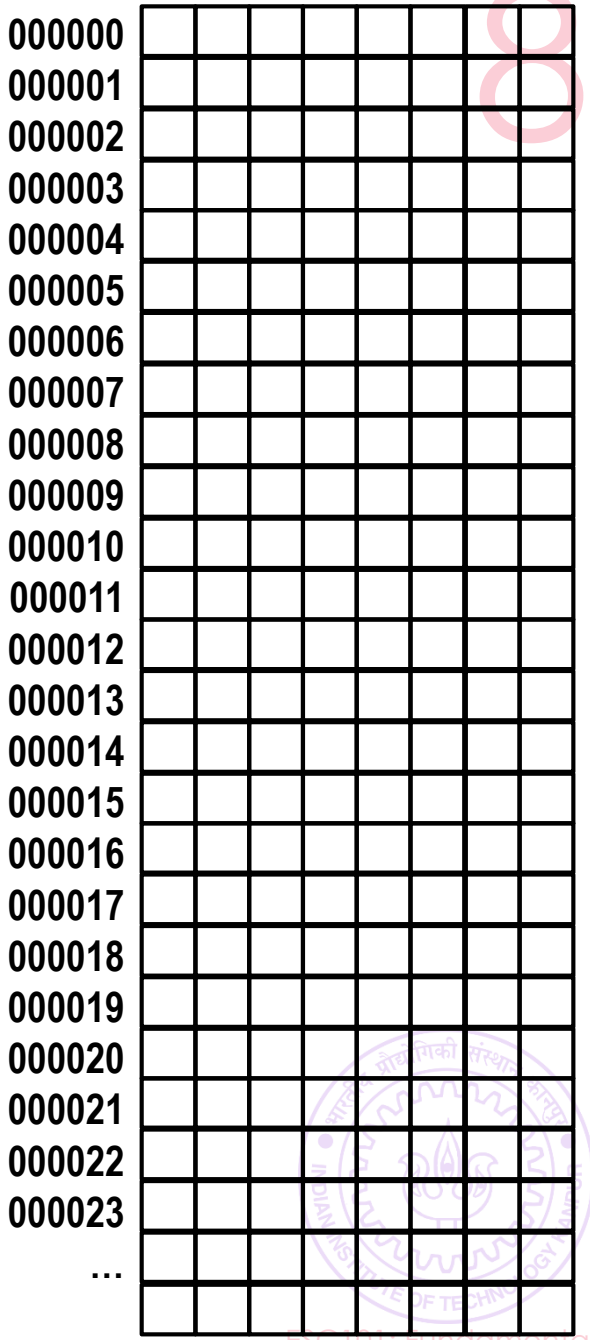
free

Used to deallocate memory allocated using malloc/calloc/realloc



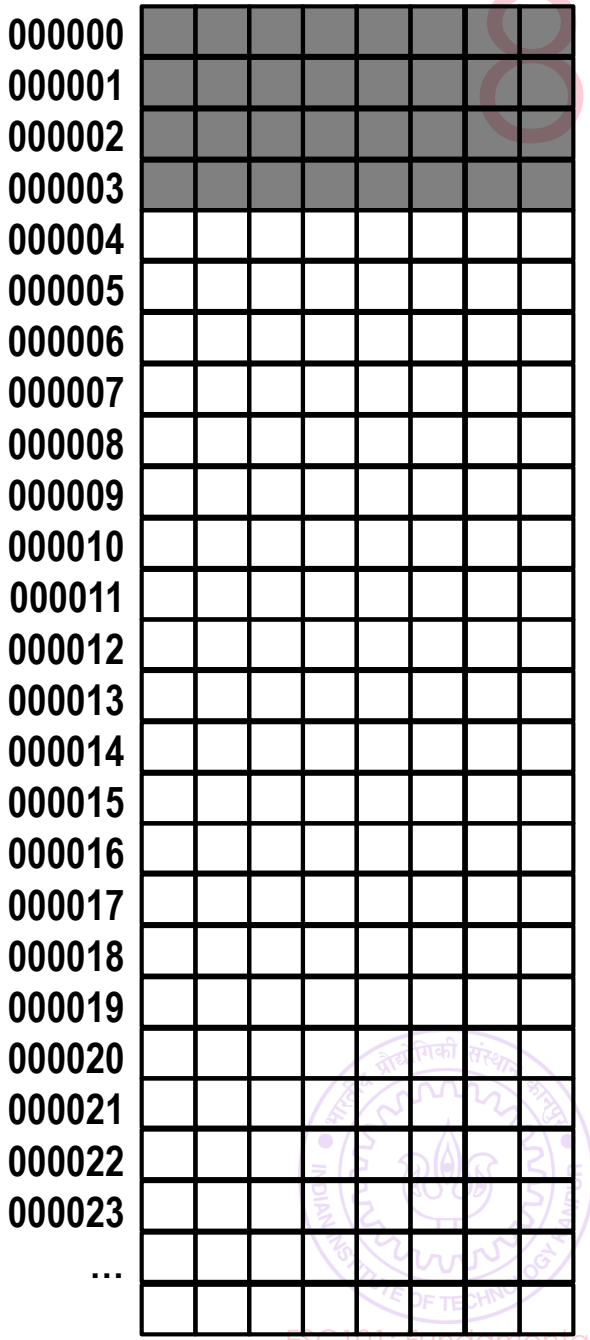
free

Used to deallocate memory allocated using malloc/calloc/realloc



free

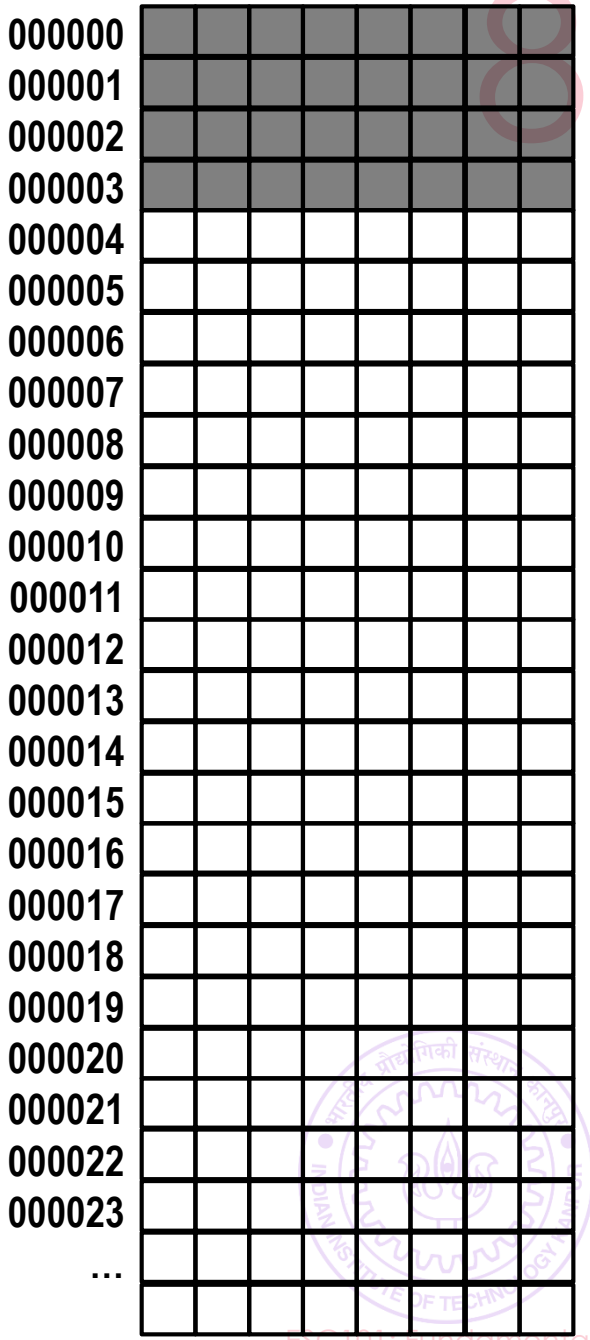
Used to deallocate memory allocated using malloc/calloc/realloc



free

Used to deallocate memory allocated using malloc/calloc/realloc

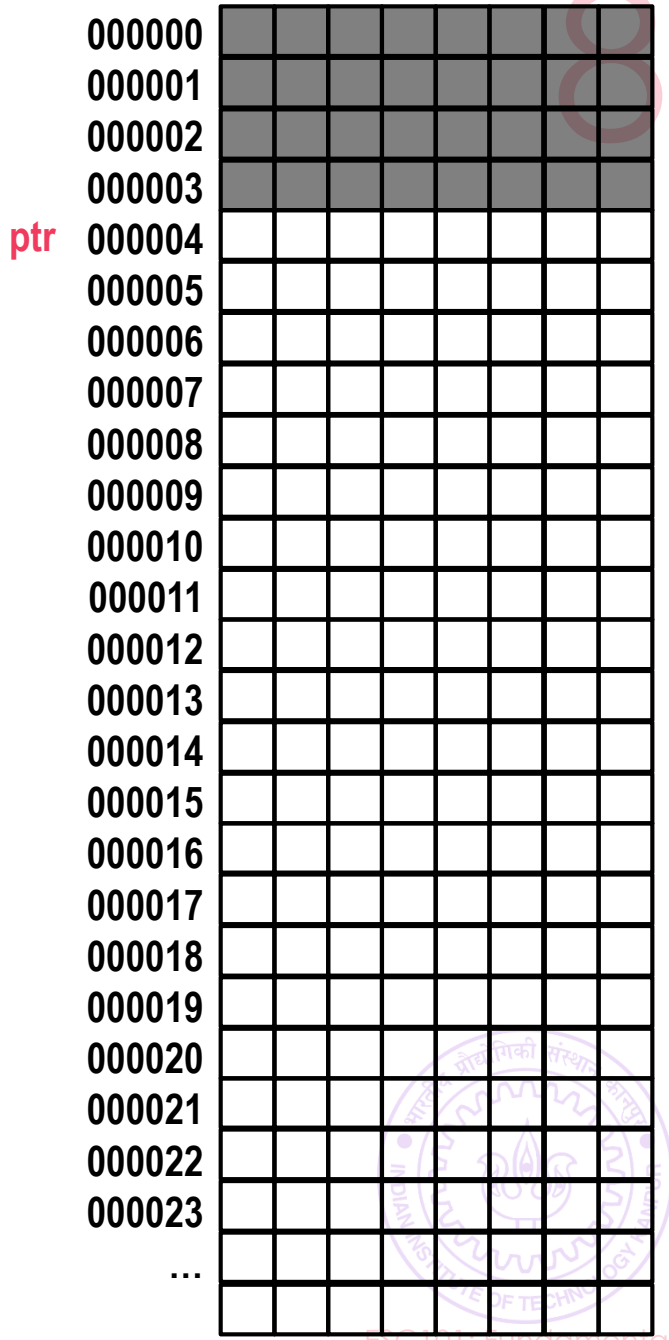
```
int *ptr;
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

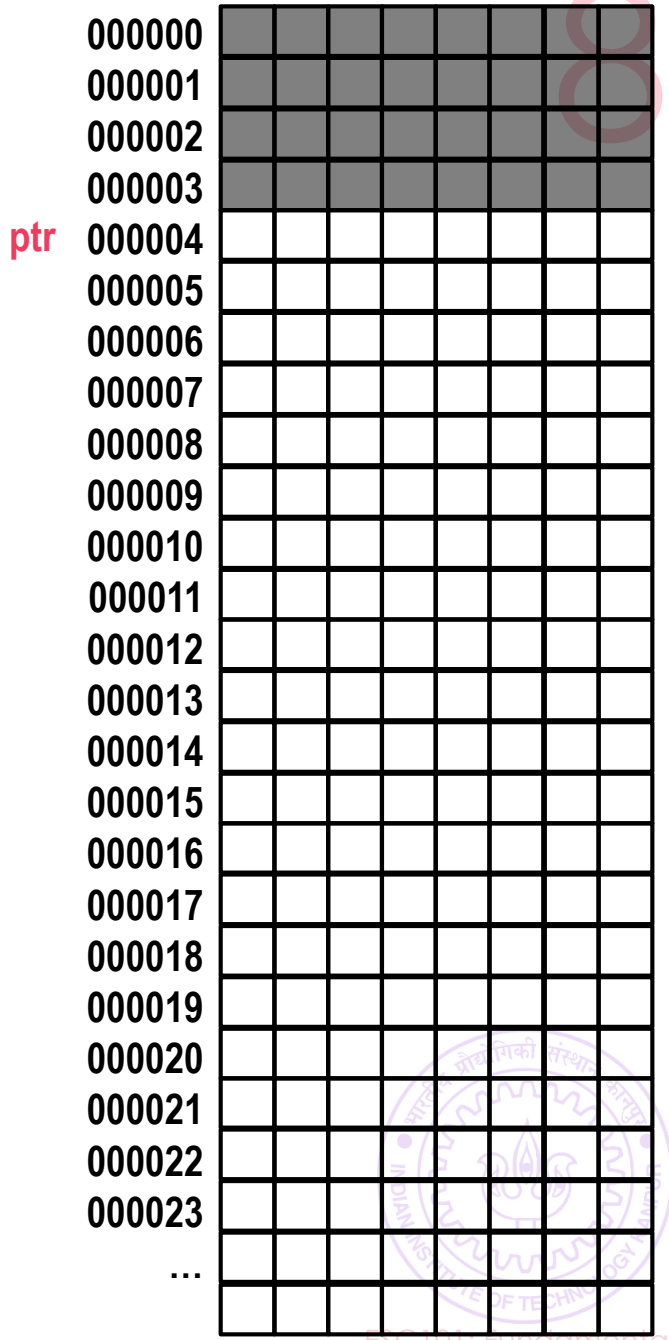
```
int *ptr;
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

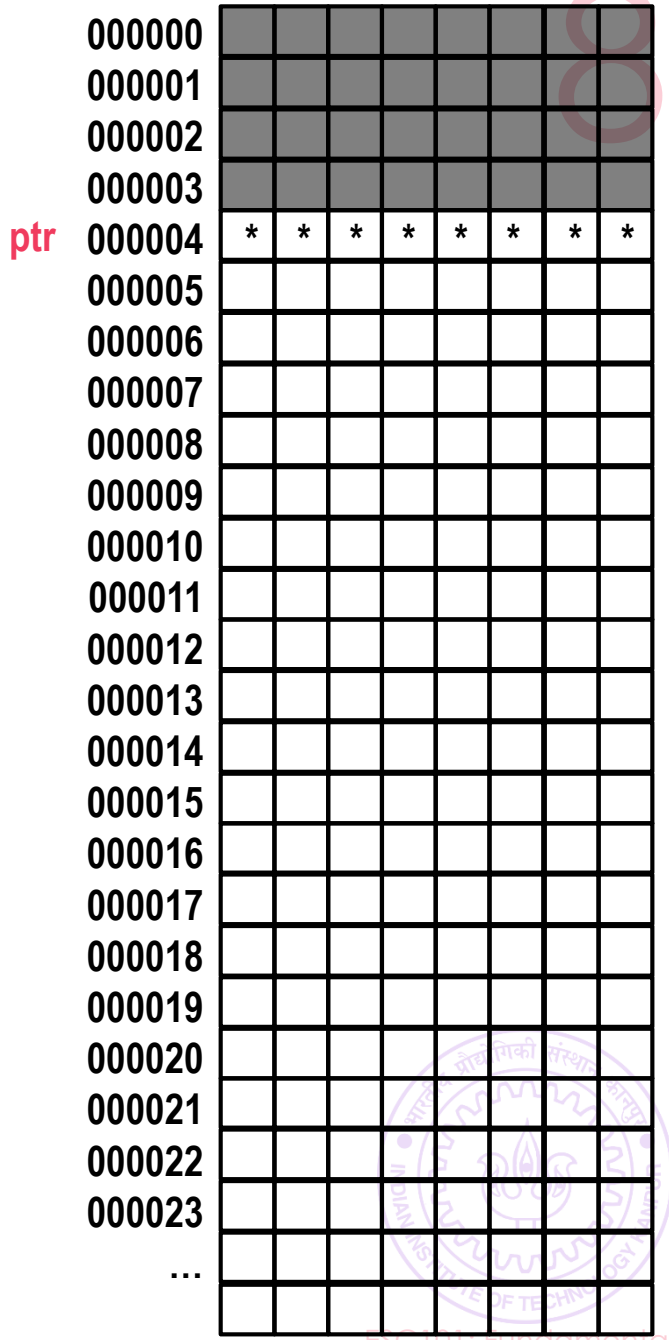
int *ptr; // may contain a junk address now



free

Used to deallocate memory allocated using malloc/calloc/realloc

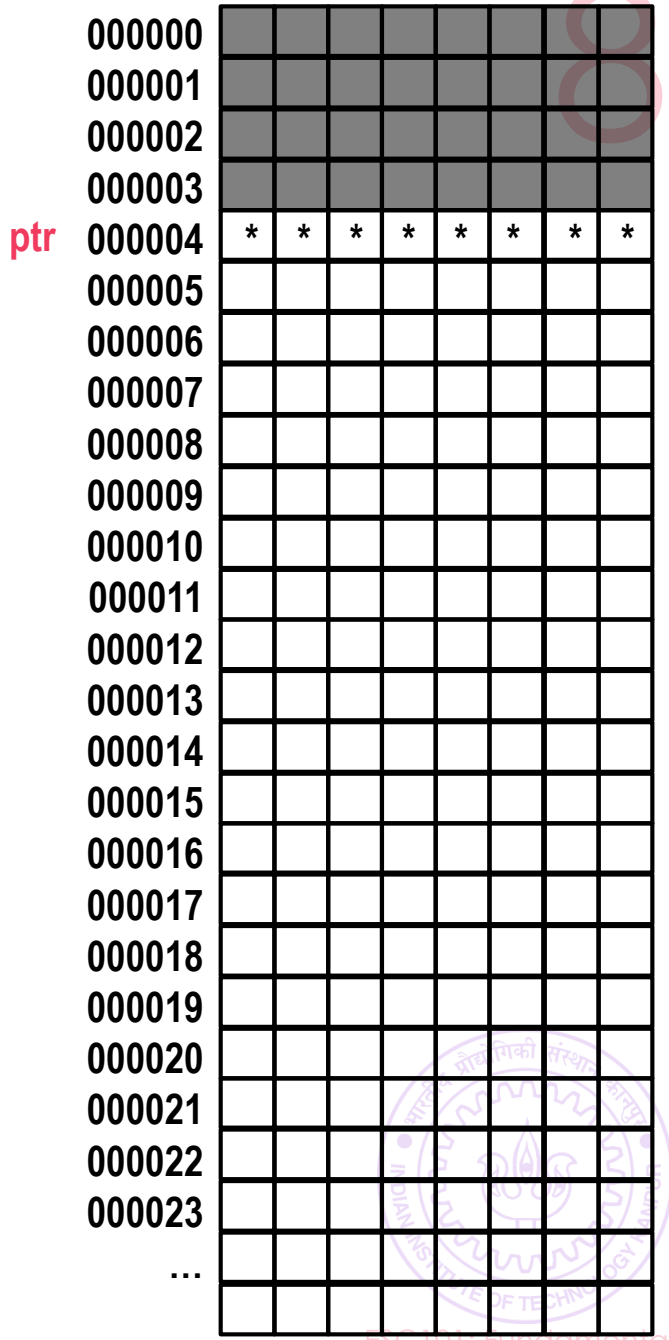
int *ptr; // may contain a junk address now



free

Used to deallocate memory allocated using malloc/calloc/realloc

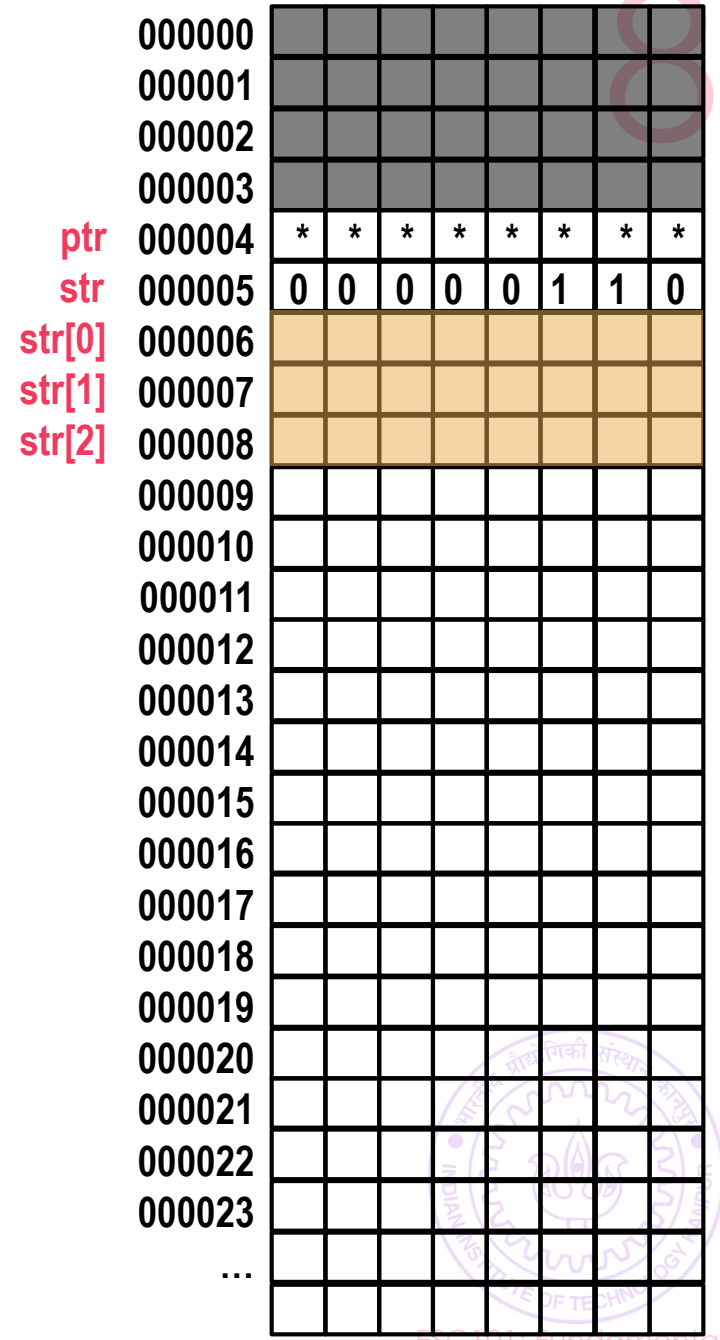
```
int *ptr; // may contain a junk address now
char str[3];
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

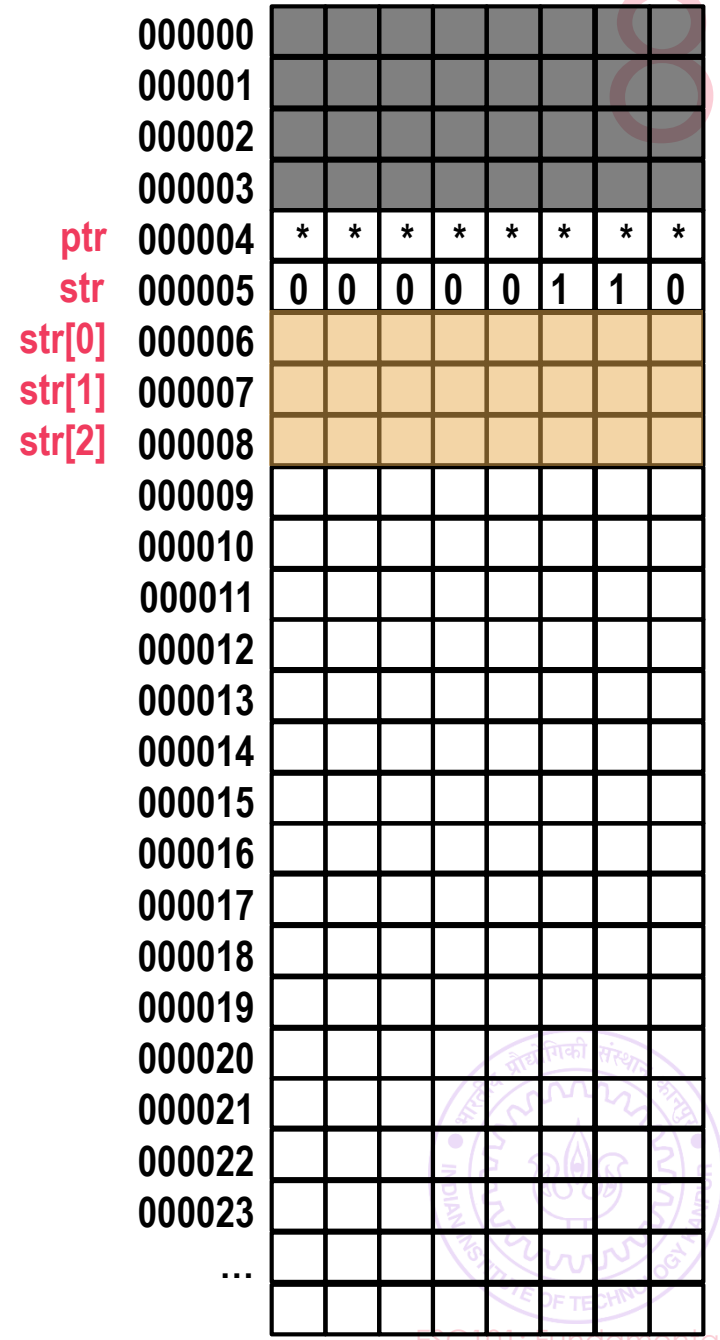
```
int *ptr; // may contain a junk address now
char str[3];
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

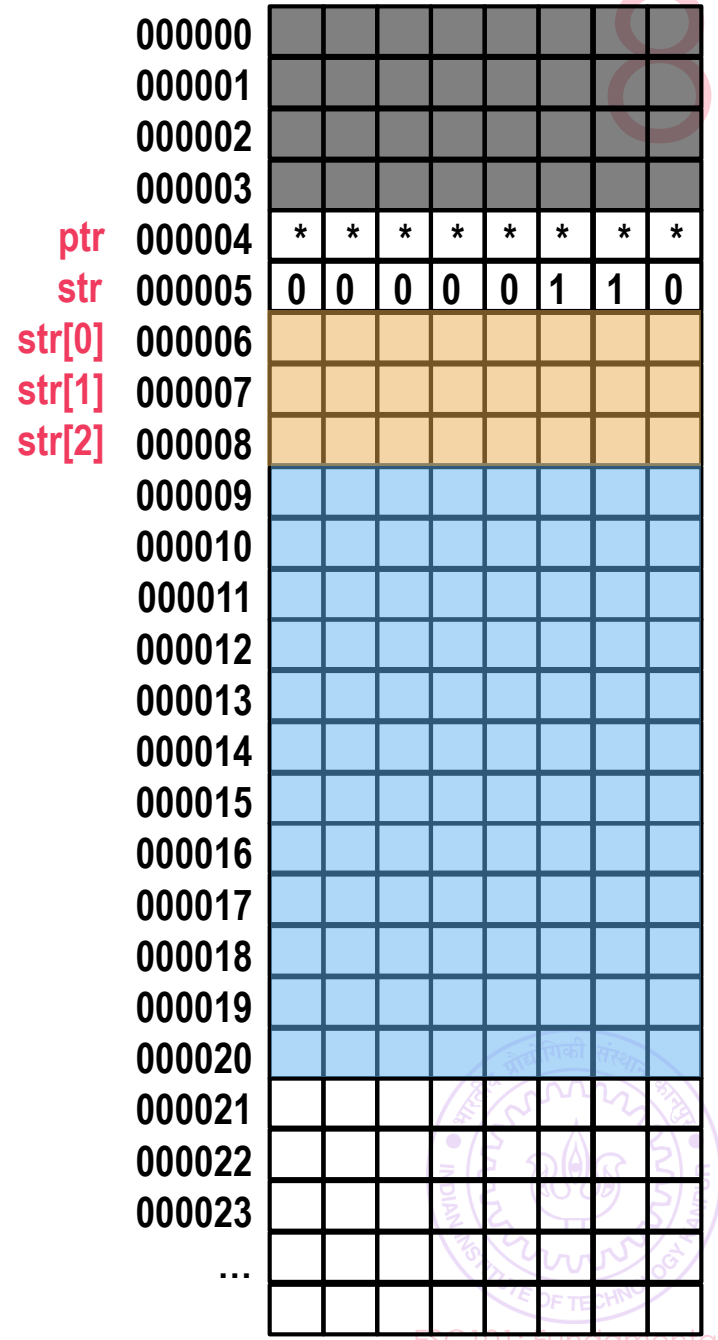
```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

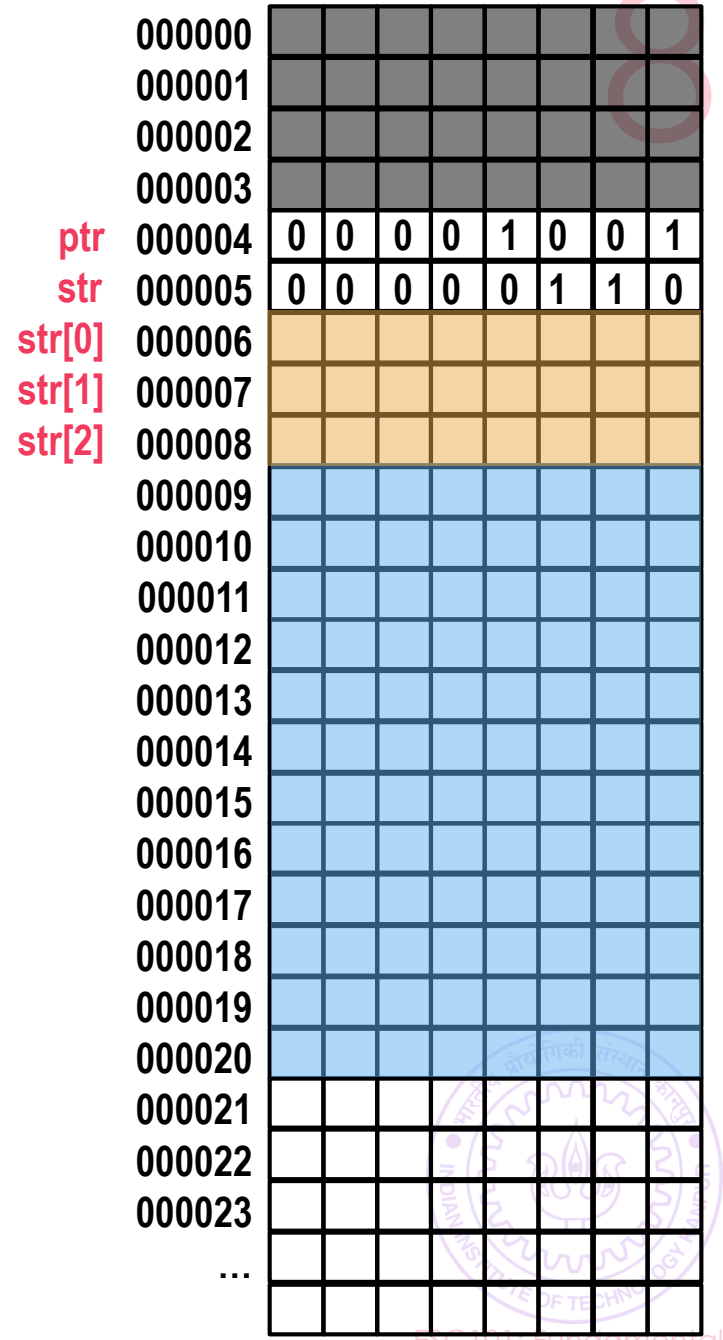
```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

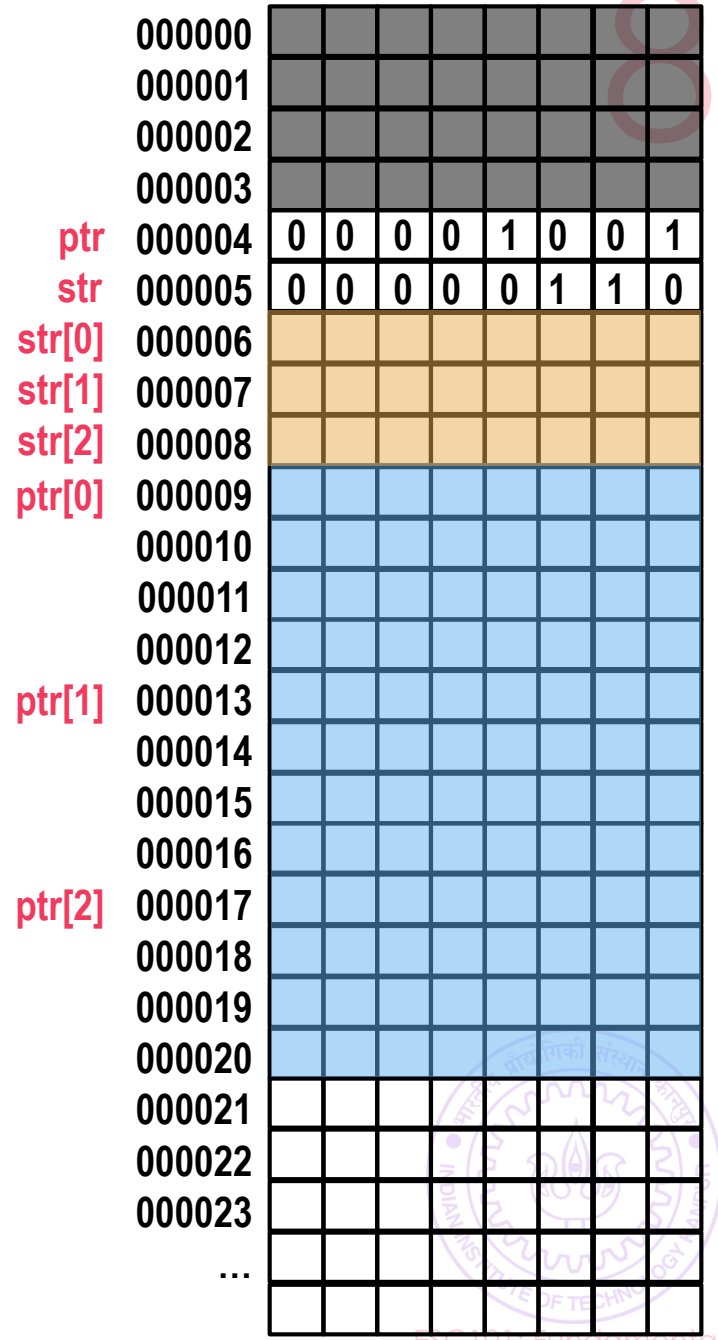
```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

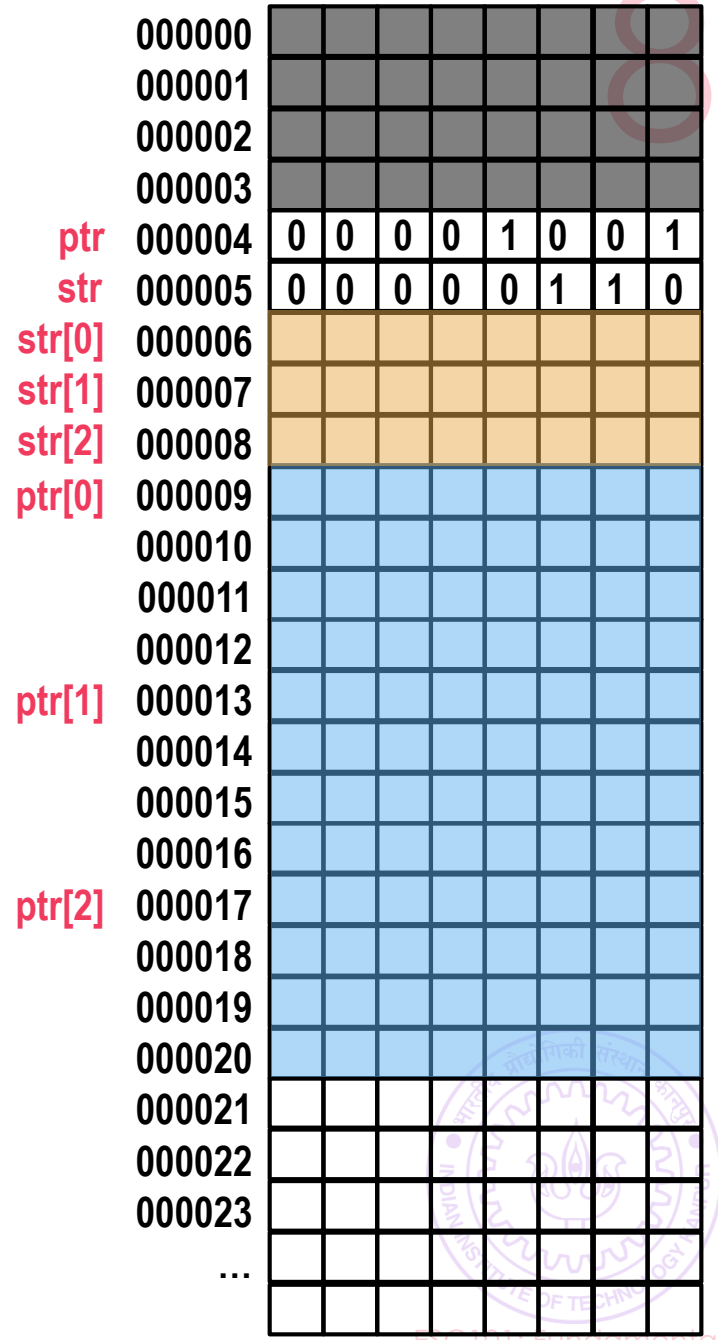
```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

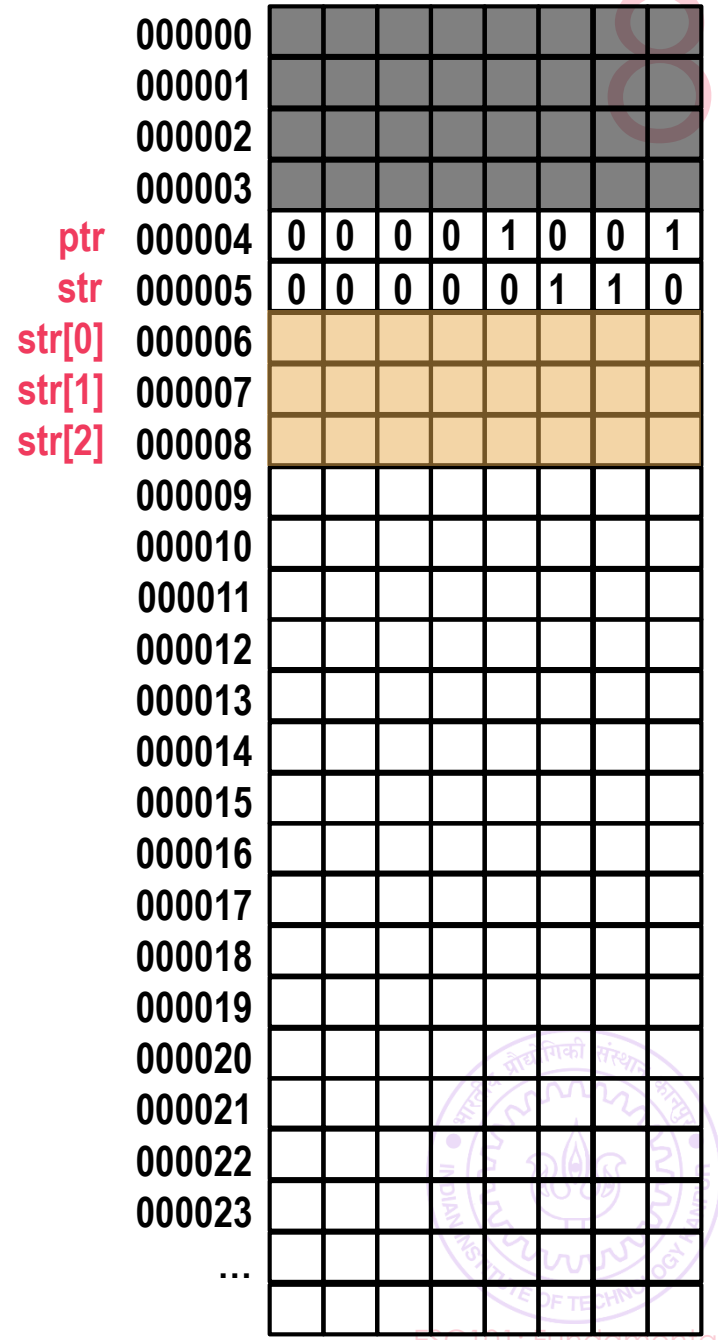
```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

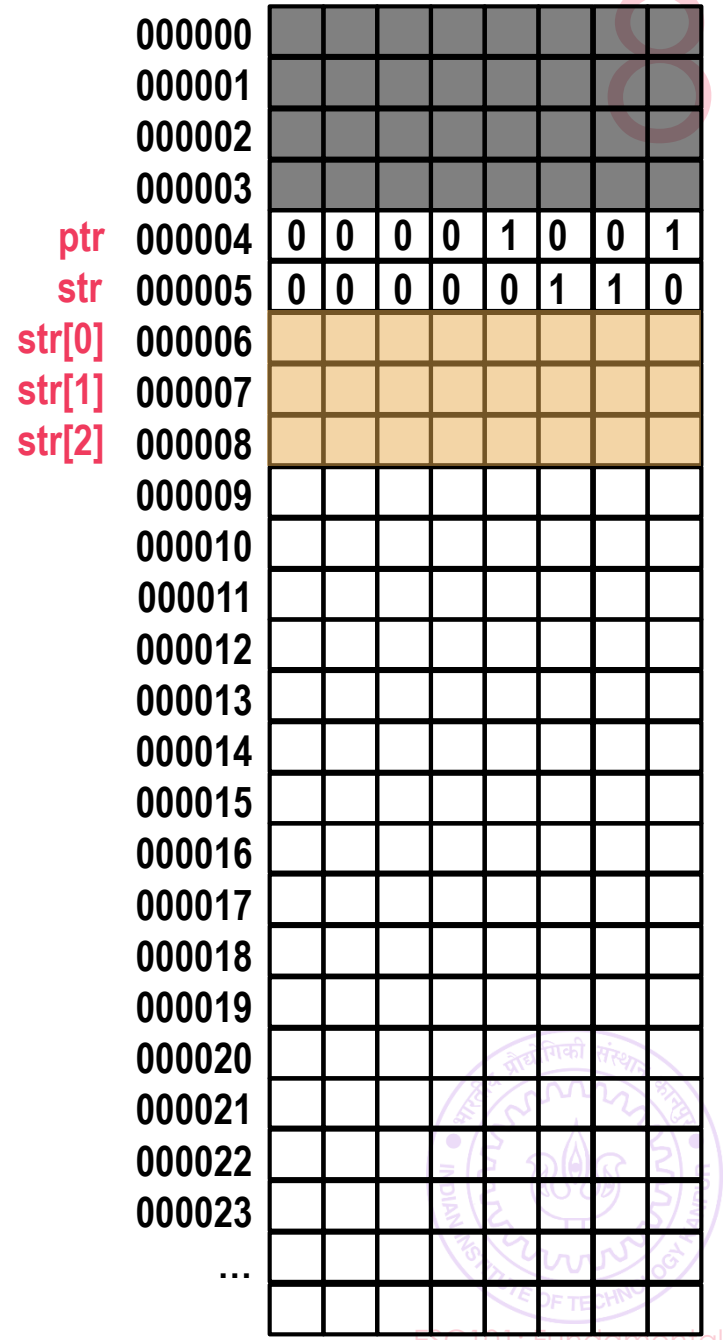
```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
```



free

Used to deallocate memory allocated using malloc/calloc/realloc

```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
printf("%d", ptr[1]);
```

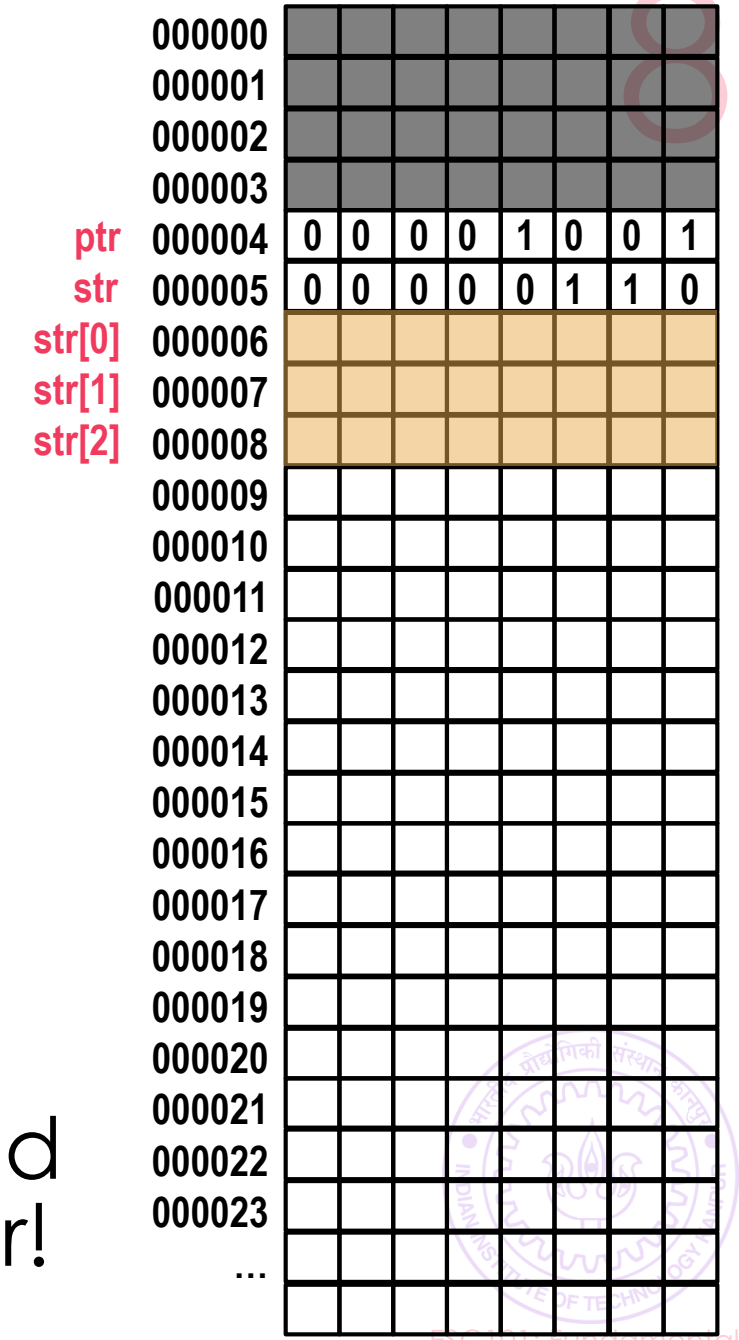


free

Used to deallocate memory allocated using malloc/calloc/realloc

```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
printf("%d", ptr[1]);
```

Don't use freed memory or free memory twice or free non-malloc/calloc/realloc-ed arrays – will cause segfault or runtime error!



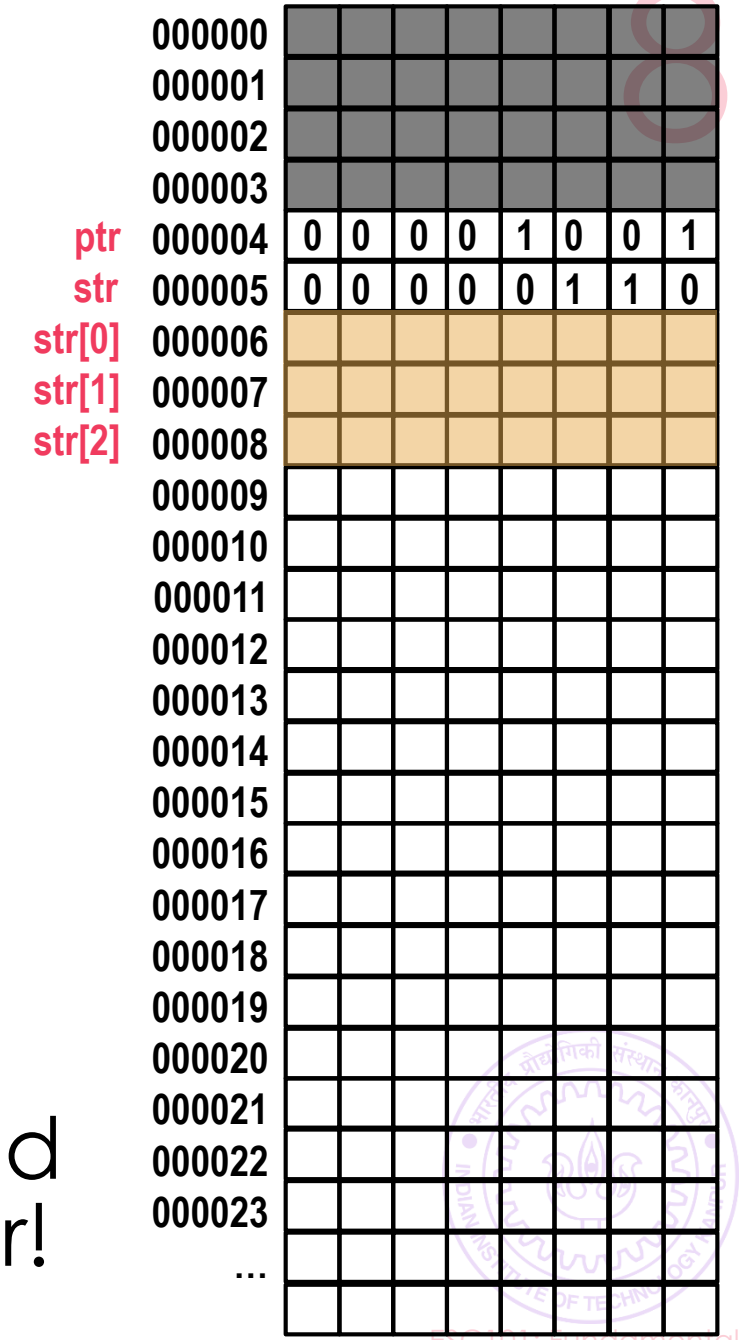
free

Used to deallocate memory allocated using malloc/calloc/realloc

```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
printf("%d", ptr[1]);
```

Don't use freed memory or free memory twice or free non-malloc/calloc/realloc-ed arrays – will cause segfault or runtime error!

```
free(str); // runtime error
```



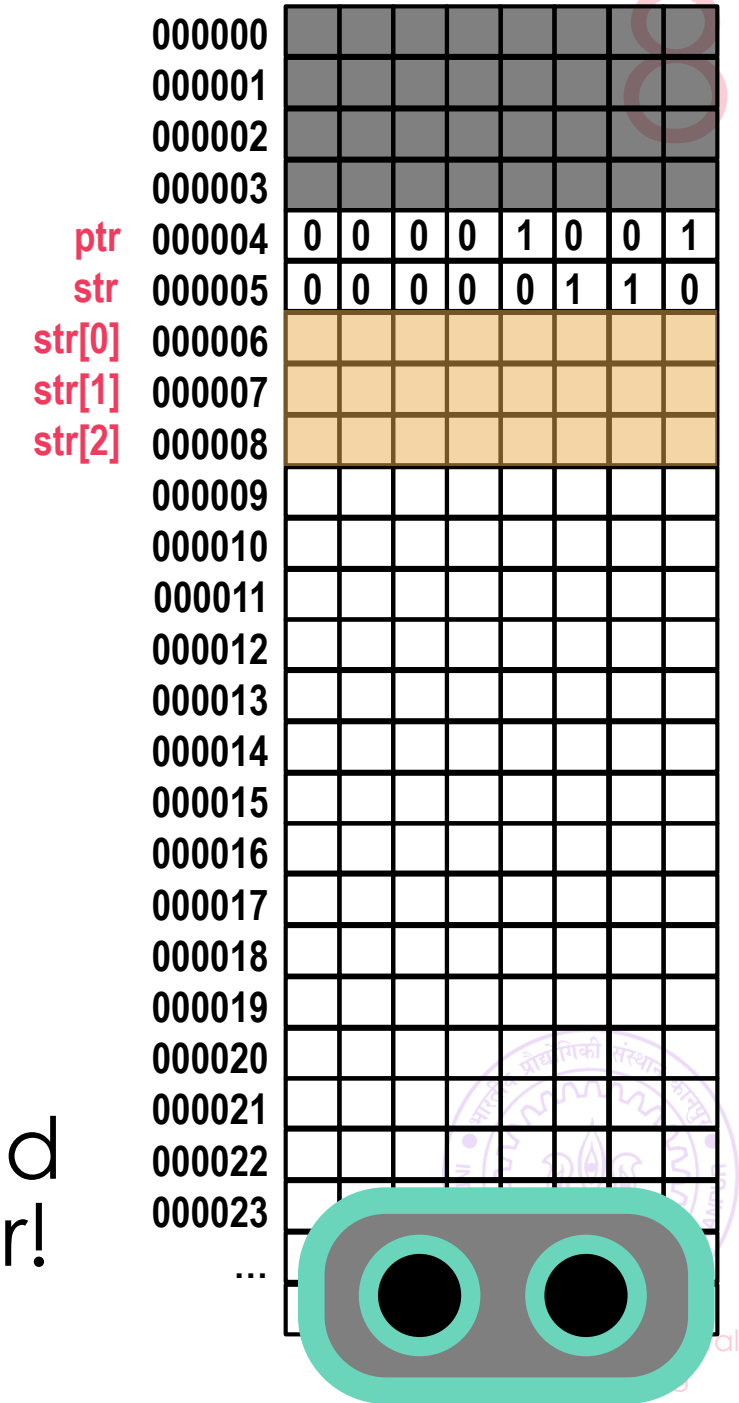
free

Used to deallocate memory allocated using malloc/calloc/realloc

```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
printf("%d", ptr[1]);
```

Don't use freed memory or free memory twice or free non-malloc/calloc/realloc-ed arrays – will cause segfault or runtime error!

```
free(str); // runtime error
```



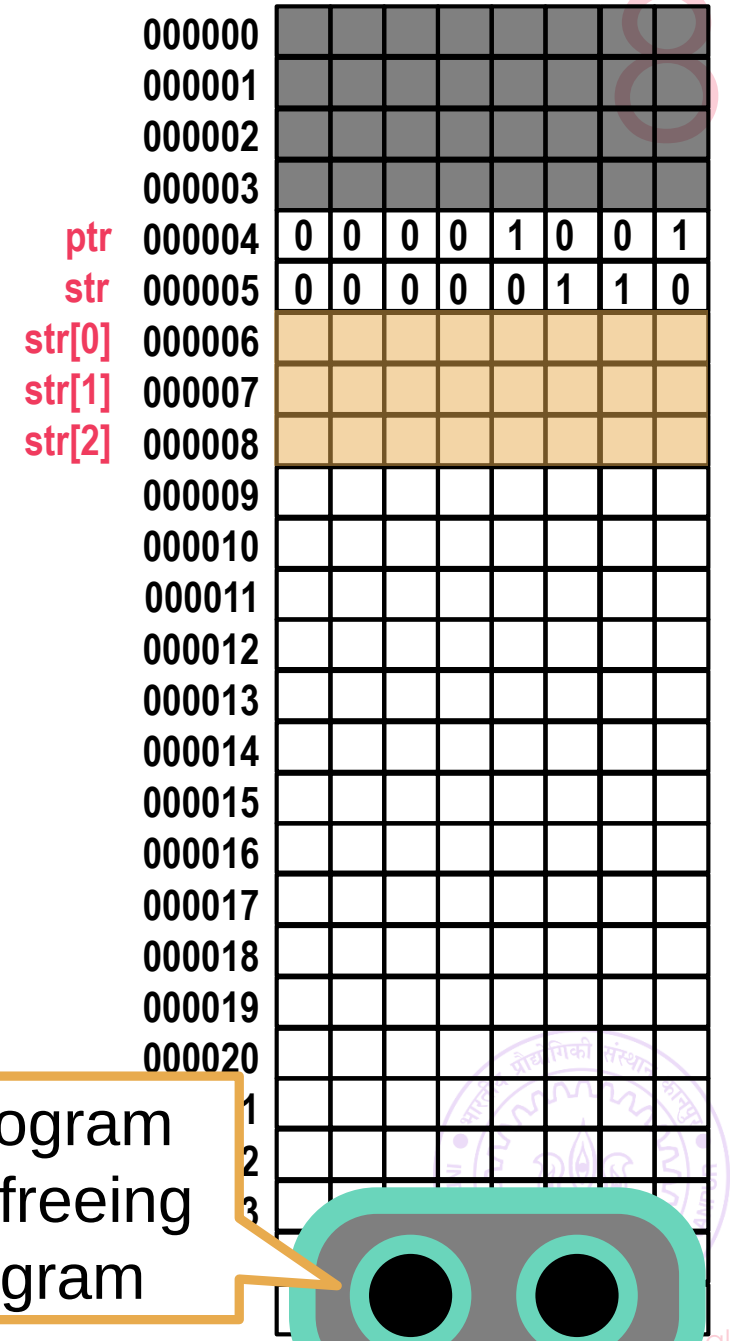
free

Used to deallocate memory allocated using malloc/calloc/realloc

```
int *ptr; // may contain a junk address now
char str[3];
ptr = (int*)malloc(3 * sizeof(int));
free(ptr);
printf("%d", ptr[1]);
```

Don't use freed memory or free memory twice or free no
arrays – will cau

```
free(str); // runtime error
```



I always free all memory when a program ends. You only have to worry about freeing memory during the run of your program

Library analogy for malloc/free

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

900+ students in Y18 but only 50 copies of Thomas' Calculus

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

900+ students in Y18 but only 50 copies of Thomas' Calculus

free is like returning a book so others can use it after you

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

900+ students in Y18 but only 50 copies of Thomas' Calculus

free is like returning a book so others can use it after you

If you keep issuing books without returning, eventually library will stop issuing books to you and impose a fine

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

900+ students in Y18 but only 50 copies of Thomas' Calculus

free is like returning a book so others can use it after you

If you keep issuing books without returning, eventually library will stop issuing books to you and impose a fine

Cannot use a book after returning it (cannot use an array variable after it has been freed)

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

900+ students in Y18 but only 50 copies of Thomas' Calculus

free is like returning a book so others can use it after you

If you keep issuing books without returning, eventually library will stop issuing books to you and impose a fine

Cannot use a book after returning it (cannot use an array variable after it has been freed)

Cannot return a book you do not have (cannot free memory that has been already freed)

Library analogy for malloc/free

malloc/calloc is like borrowing a book from library

If that book unavailable, cannot use it (NULL pointer)

900+ students in Y18 but only 50 copies of Thomas' Calculus

free is like returning a book so others can use it after you

If you keep issuing books without returning, eventually library will stop issuing books to you and impose a fine

Cannot use a book after returning it (cannot use an array variable after it has been freed)

Cannot return a book you do not have (cannot free memory that has been already freed)

Of course, if you re-issue a book you can return it again

realloc – **re**vised **alloc**ation

realloc – **re**vised **alloc**ation

If you malloc-ed an array of 100 elements and suddenly find that you need an array of 200 elements ☹

realloc – **re**vised **alloc**ation

If you malloc-ed an array of 100 elements and suddenly find that you need an array of 200 elements ☹

```
int *ptr = (int*)malloc(100 * sizeof(int));
```

realloc – **re**vised **alloc**ation

If you malloc-ed an array of 100 elements and suddenly find that you need an array of 200 elements ☹

```
int *ptr = (int*)malloc(100 * sizeof(int));
```

Can use realloc to revise that allocation to 200 elements

realloc – **re**vised **alloc**ation

If you malloc-ed an array of 100 elements and suddenly find that you need an array of 200 elements ☹

```
int *ptr = (int*)malloc(100 * sizeof(int));
```

Can use realloc to revise that allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200 * sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc – **re**vised **alloc**ation

If you malloc-ed an array of 100 elements and subs
find that you need an array of 200 elements ☹

```
int *ptr = (int*)malloc(100 * sizeof(int));
```

Can use realloc to revise that allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200 * sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc – revised alloc

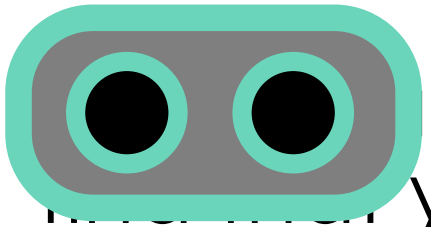
If you malloc-ed an array of 100 elements and suddenly find that you need an array of 200 elements ☹

```
int *ptr = (int*)malloc(100 * sizeof(int));
```

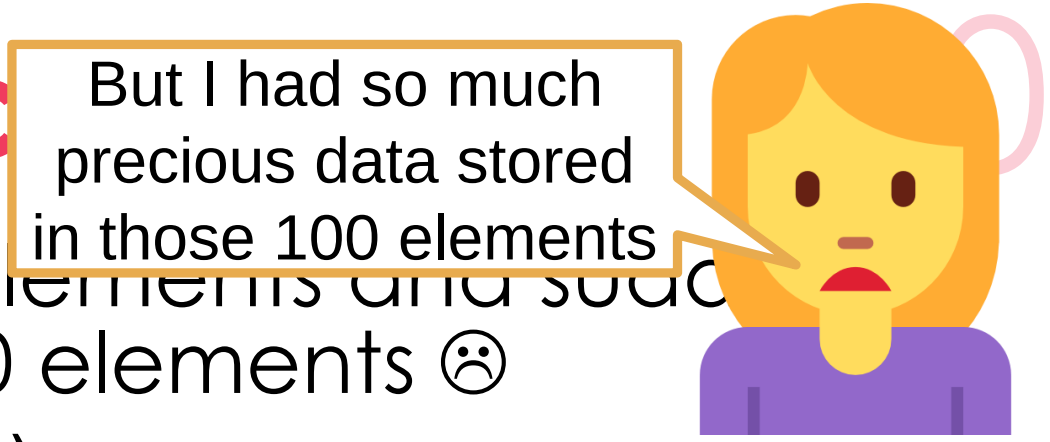
Can use realloc to revise that allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200 * sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc – revised alloc



lloc-ed an array of 100 elements and sudd
you need an array of 200 elements ☹

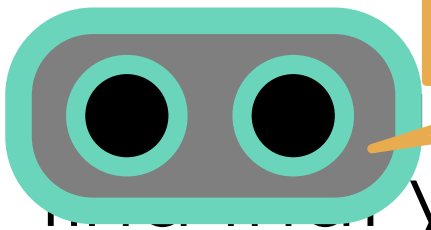



```
int *ptr = (int*)malloc(100 * sizeof(int));
```

Can use realloc to revise that allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200 * sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

But I had so much precious data stored in those 100 elements

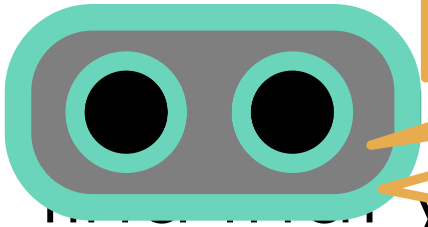
llocated an array of 100 elements and suddenly you need an array of 200 elements ☹

```
int *ptr = (int*)malloc(100 * sizeof(int));
```

Can use realloc to revise that allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200 * sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc ... c



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

I will also free the old 100 elements – you don't have to write free() for them

...llo... an array of 100 elements and sudd...

...find that yo... 200 elements ☹

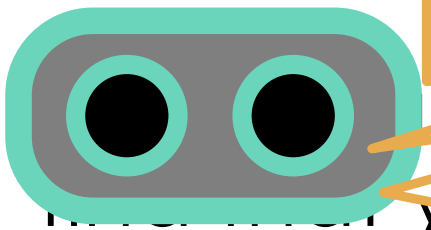
int *ptr = (int* ... (int));

But I had so much precious data stored in those 100 elements

Can use realloc to revise that allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200 * sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

But I had so much precious data stored in those 100 elements

I will also free the old 100 elements – you don't have to write free() for them

If insufficient memory, I will not free old memory but just return NULL pointer

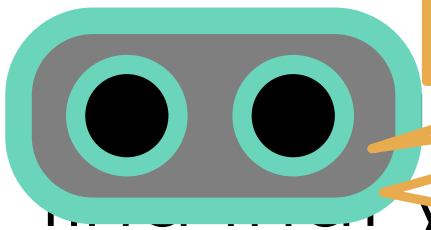
200 elements ☹

```
int *ptr = (int*)malloc(200*sizeof(int));
```

Can use realloc() to request allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200*sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

I will also free the old 100 elements – you don't have to write free() for them

If insufficient memory, I will not free old memory but just return NULL pointer

But I had so much precious data stored in those 100 elements

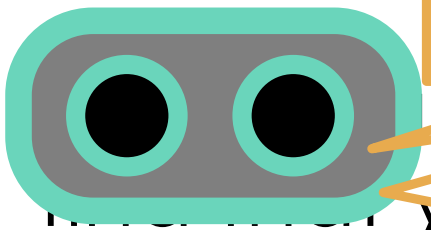
200 elements ☹

```
int *ptr = (int*)malloc(200*sizeof(int));
```

Can use realloc() to request allocation to 200 elements

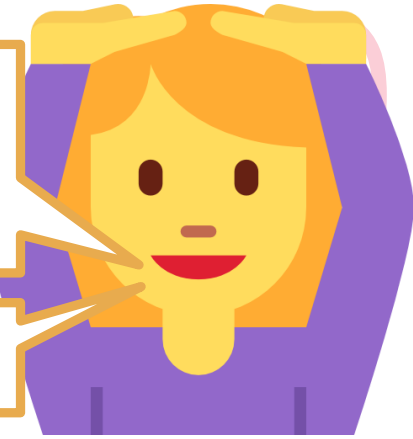
```
int *tmp = (int*)realloc(ptr, 200*sizeof(int));
if(tmp != NULL) ptr = tmp;
```

realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

But I had so much precious data stored in those 100 elements



I will also free the old 100 elements – you don't have to write free() for them

You are the best Mr C

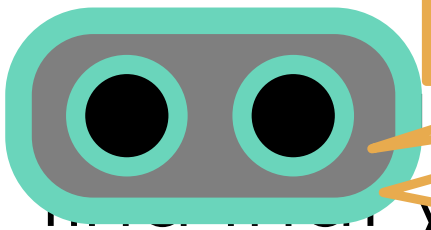
If insufficient memory, I will not free old memory but just return NULL pointer

```
int *ptr = (int*)malloc(100*sizeof(int));
```

Can use realloc() to request allocation to 200 elements

```
int *tmp = (int*)realloc(ptr, 200*sizeof(int));
if(tmp != NULL) ptr = tmp;
```

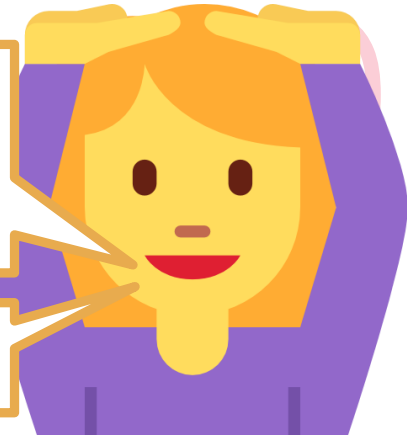
realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

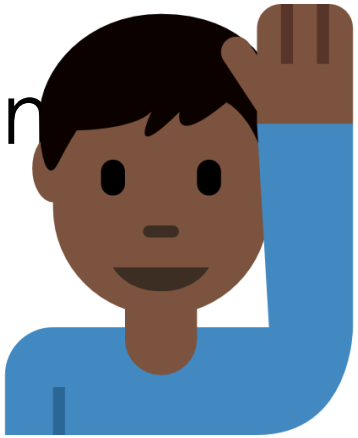
I will also free the old 100 elements – you don't have to write free() for them

If insufficient memory, I will not free old memory but just return NULL pointer



But I had so much precious data stored in those 100 elements

You are the best Mr C



```
int *ptr = (int*) malloc(200*sizeof(int));
```

Can use realloc() to request allocation to 200 elements

```
int *tmp = (int*) realloc(ptr, 200*sizeof(int));
if(tmp != NULL) ptr = tmp;
```

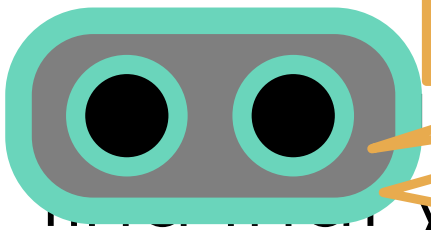
realloc

int *ptr = (int

Can use rea

int *tmp = (in zeof(int));

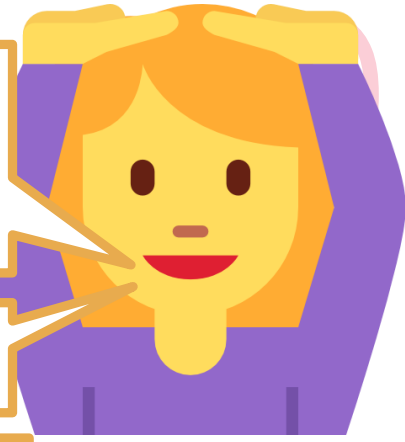
if(tmp != NULL) ptr = tmp;



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

I will also free the old 100 elements – you don't have to write free() for them

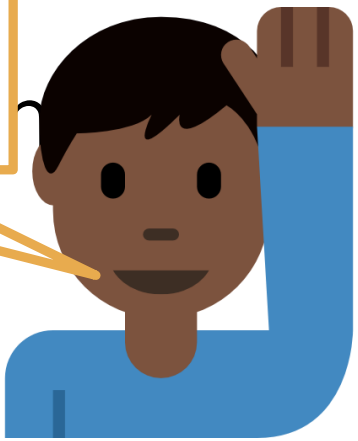
If insufficient memory, I will not free old memory but just return NULL pointer



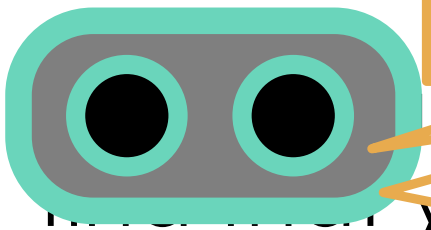
But I had so much precious data stored in those 100 elements

You are the best Mr C

A bit system-dependent. If insufficient memory, Prutor programs will simply crash



realloc



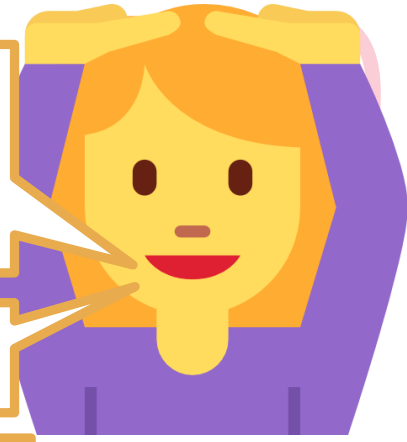
I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

I will also free the old 100 elements – you don't have to write free() for them

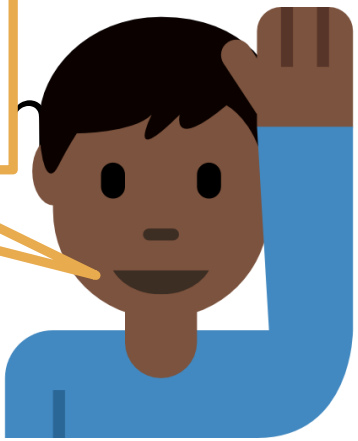
If insufficient memory, I will not free old memory but just return NULL pointer

But I had so much precious data stored in those 100 elements

You are the best Mr C



A bit system-dependent. If insufficient memory, Prutor programs will simply crash



int *ptr = (int

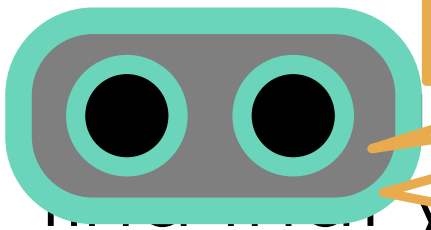
Can use rea

int *tmp = (in zeof(int));

if(tmp != NULL) ptr = tmp;

Don't use realloc to resize of non-malloc arrays

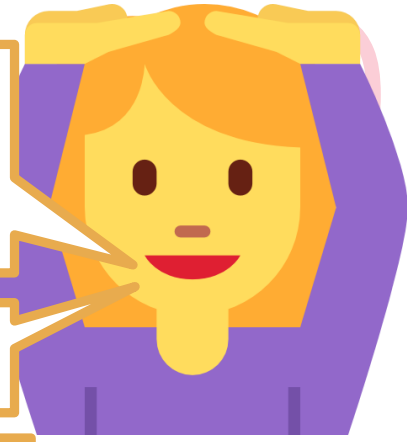
realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

I will also free the old 100 elements – you don't have to write free() for them

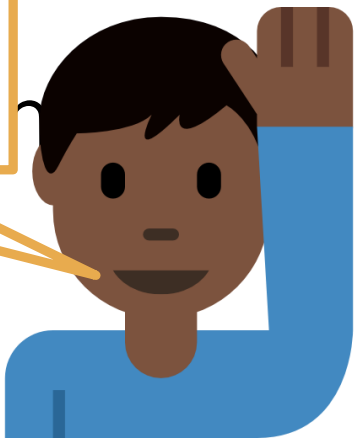
If insufficient memory, I will not free old memory but just return NULL pointer



But I had so much precious data stored in those 100 elements

You are the best Mr C

A bit system-dependent. If insufficient memory, Prutor programs will simply crash



int *ptr = (int

Can use rea

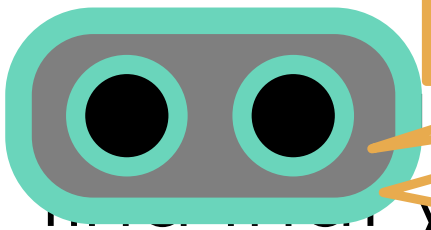
int *tmp = (in zeof(int));

if(tmp != NULL) ptr = tmp;

Don't use realloc to resize of non-malloc arrays

```
int c[100];
int *ptr = (int*)realloc(c, 200 * sizeof(int)); // Runtime error
```

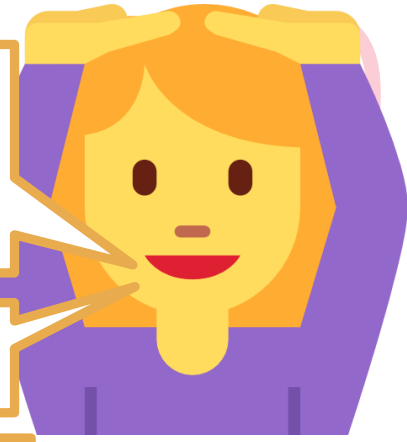
realloc



I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

I will also free the old 100 elements – you don't have to write free() for them

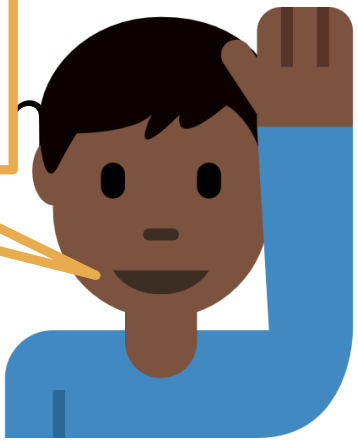
If insufficient memory, I will not free old memory but just return NULL pointer



But I had so much precious data stored in those 100 elements

You are the best Mr C

A bit system-dependent. If insufficient memory, Prutor programs will simply crash



```
int *ptr = (int
```

Can use rec

```
int *tmp = (in                    zeof(int));
if(tmp != NULL) ptr = tmp;
```

Don't use realloc to resize of non-malloc arrays

```
int c[100];
int *ptr = (int*)realloc(c, 200 * sizeof(int)); // Runtime error
```

Use realloc only to resize of calloc/malloc-ed arrays

getline

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

getline

Read a single line of text from input (i.e. till '\n')

getline

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

getline

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

getline

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expect 10 characters to be entered
char *str = (char*)malloc(len * sizeof(char));
getline(&str, &len, stdin);
```

getline

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expect 10 characters to be entered
char *str = (char*)malloc(len * sizeof(char));
getline(&str, &len, stdin);
```

If input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

getline

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expect 10 characters to be entered
char *str = (char*)malloc(len * sizeof(char));
getline(&str, &len, stdin);
```

If input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

```
char **ptrstr = &str;
getline(ptrstr, &len, stdin); // Alternate way to use getline
```

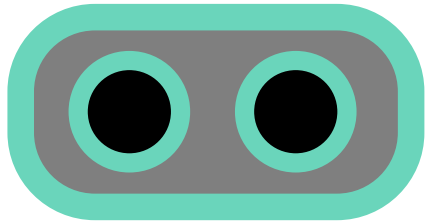
getline

Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expect 10 characters to be entered
char *str = (char*)malloc(len * sizeof(char));
getline(&str, &len, stdin);
```



If input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

```
char **ptrstr = &str;
getline(ptrstr, &len, stdin); // Alternate way to use getline
```

getline

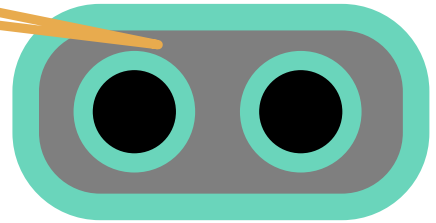
Read a single line of text from input (i.e. till '\n')

Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expe                    ered
char *str = (char*)malloc(len * sizeof(char));
getline(&str, &len, stdin);
```

Pointer to a pointer simply stores the address of a pointer variable



If input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

```
char **ptrstr = &str;
getline(ptrstr, &len, stdin); // Alternate way to use getline
```

getline

Read a single line of text from input (i.e. till '\n')

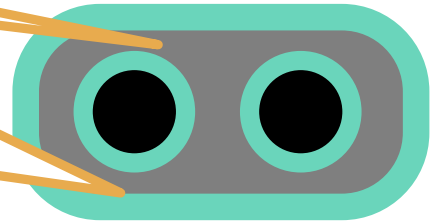
Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expe                    ered
char *str =
getline(&s
```

Pointer to a pointer simply stores the address of a pointer variable

printf("%ld",*ptrstr) will print address of first char in str
printf("%c",**ptrstr) will print the first char in str
printf("%s",*ptrstr) will print entire string str



If input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

```
char **ptrstr = &str;
getline(ptrstr, &len, stdin); // Alternate way to use getline
```

getline

Read a single line of text from input (i.e. till '\n')

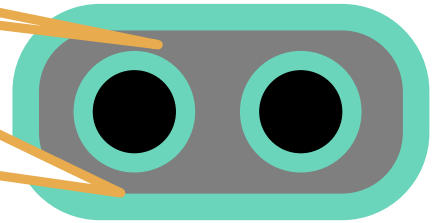
Uses realloc-like methods to expand array size

Needs a malloc-ed array for this reason

```
int len = 11; // I only expe                    ered
char *str =
getline(&s
```

Pointer to a pointer simply stores the address of a pointer variable

printf("%ld",*ptrstr) will print address of first char in str
printf("%c",**ptrstr) will print the first char in str
printf("%s",*ptrstr) will print entire string str



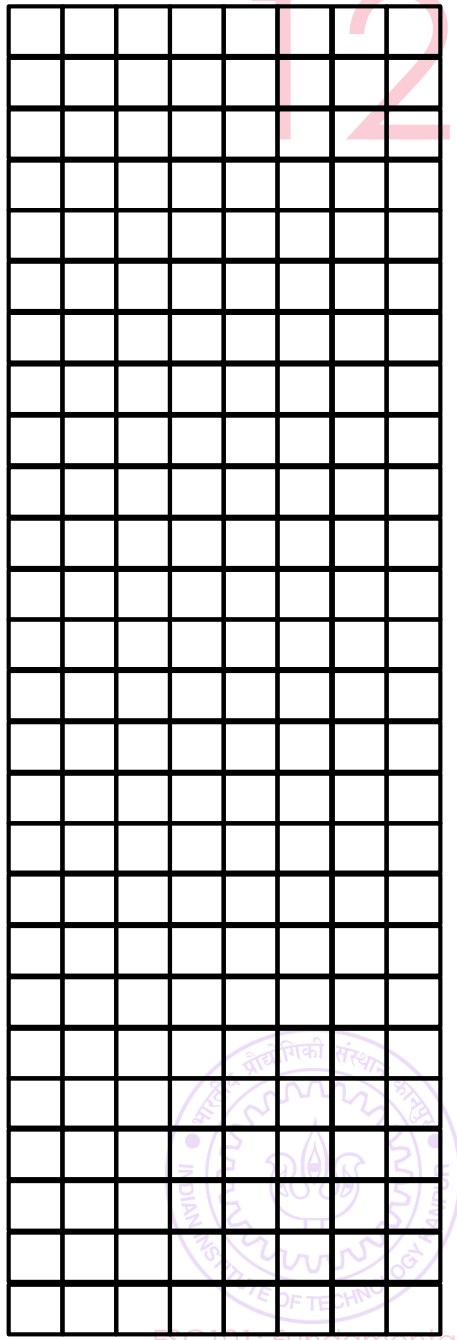
If input doesn't fit inside original array, str will contain pointer to expanded array, len will be length of new array

```
char **ptrstr = &str;
getline(ptrstr, &len, sto
```

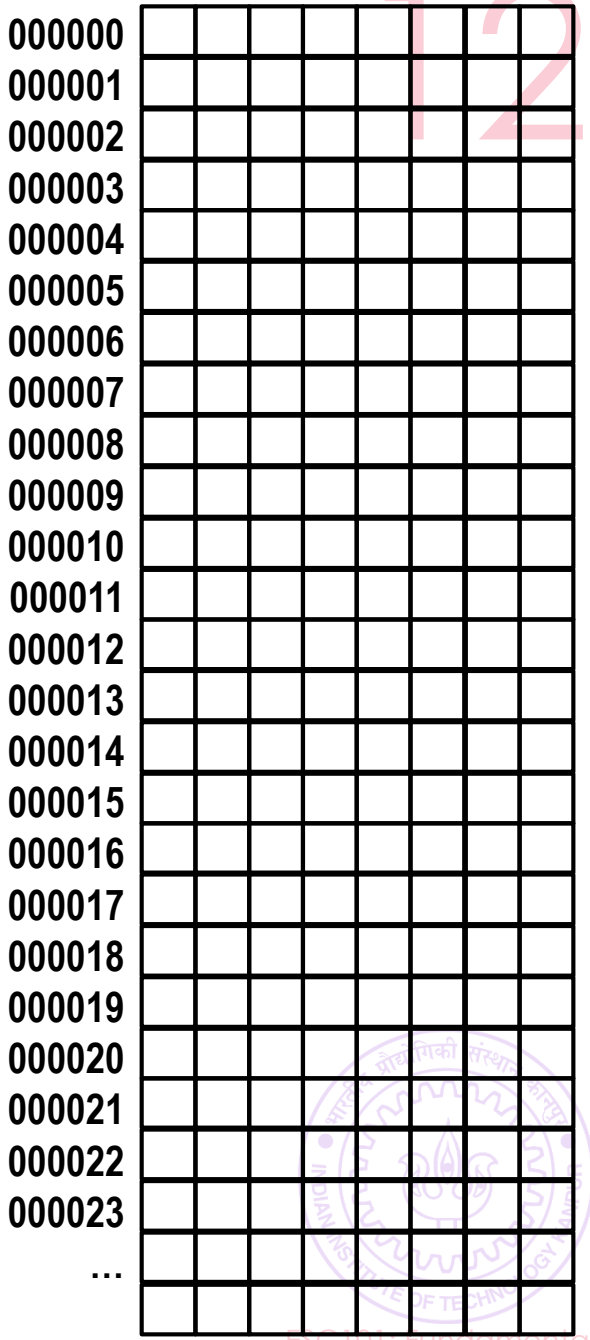
WARNING: len may be larger than length of input + 1
Get actual length of input using strlen() from string.h

Array of pointers?

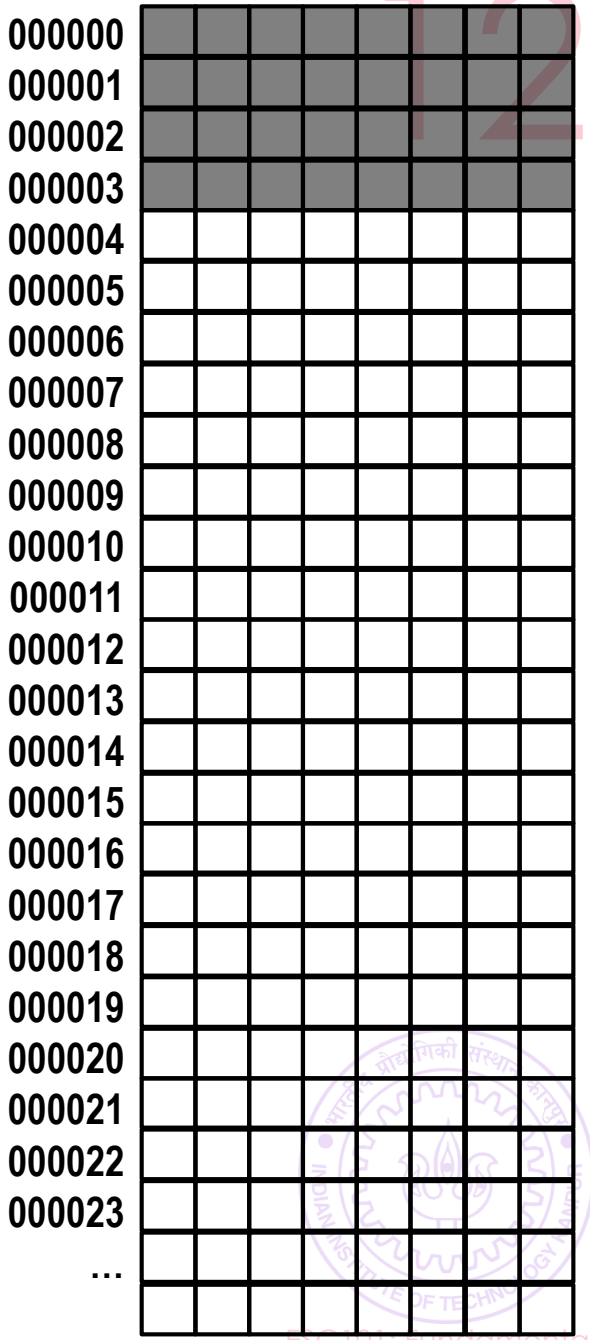
Array of pointers?



Array of pointers?

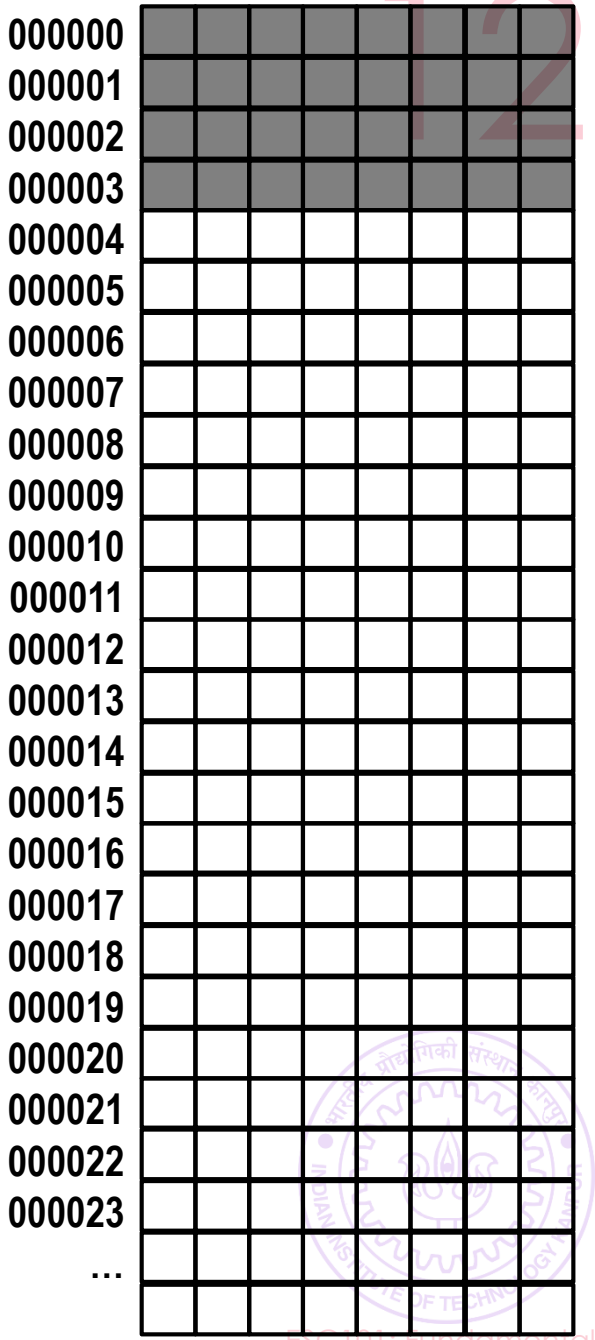


Array of pointers?



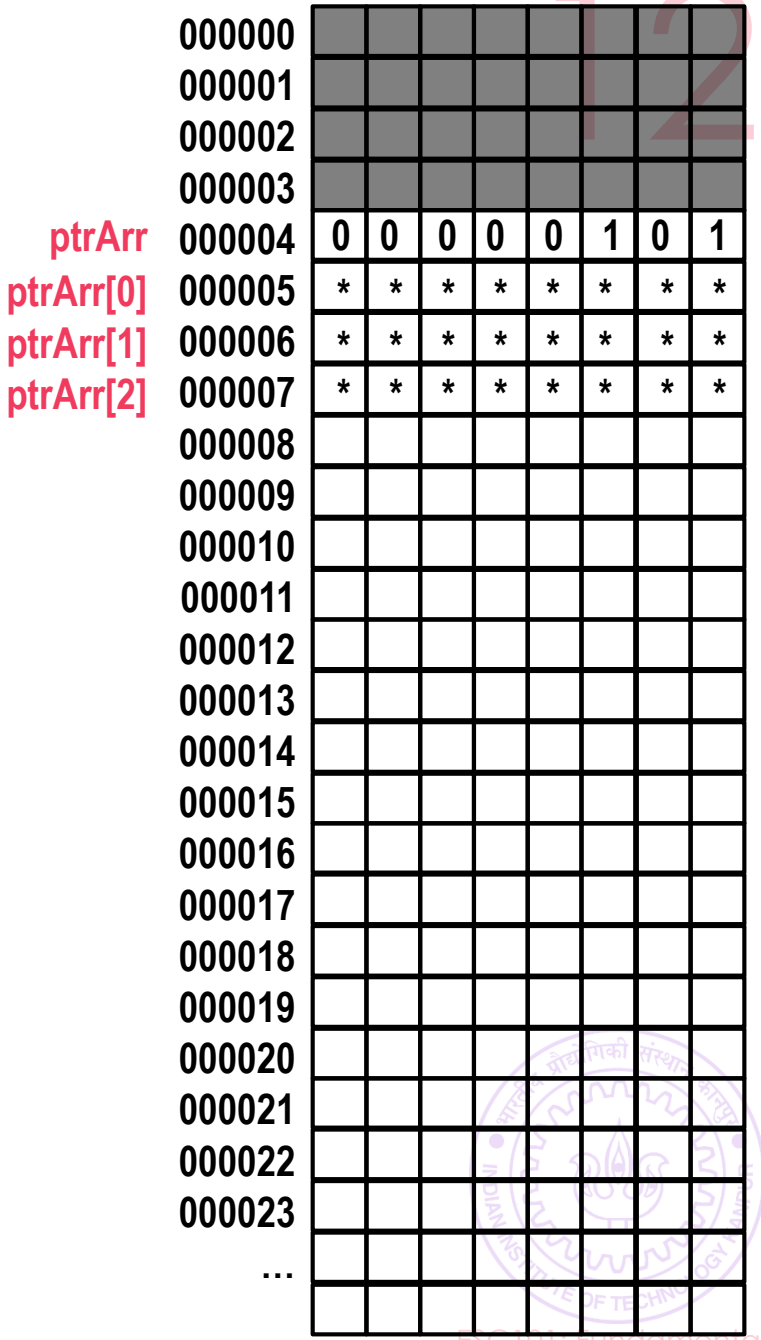
Array of pointers?

char *ptrArr[3];



Array of pointers?

char *ptrArr[3];



Array of pointers?

char *ptrArr[3];

	000000								
	000001								
	000002								
	000003								
ptrArr	000004	0	0	0	0	0	1	0	1
ptrArr[0]	000005	*	*	*	*	*	*	*	*
ptrArr[1]	000006	*	*	*	*	*	*	*	*
ptrArr[2]	000007	*	*	*	*	*	*	*	*
	000008								
	000009								
	000010								
	000011								
	000012								
	000013								
	000014								
	000015								
	000016								
	000017								
	000018								
	000019								
	000020								
	000021								
	000022								
	000023								
	...								

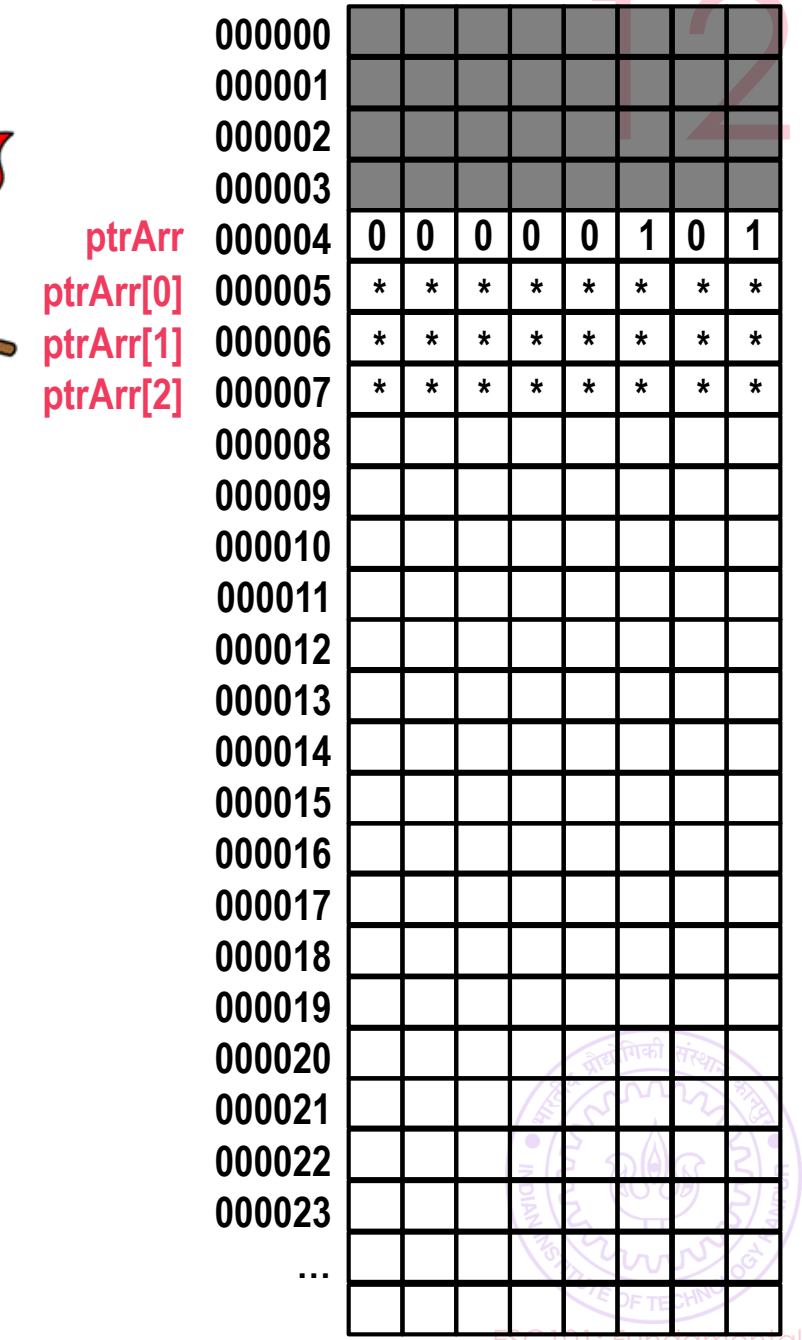
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
```

	000000								
	000001								
	000002								
	000003								
ptrArr	000004	0	0	0	0	0	1	0	1
ptrArr[0]	000005	*	*	*	*	*	*	*	*
ptrArr[1]	000006	*	*	*	*	*	*	*	*
ptrArr[2]	000007	*	*	*	*	*	*	*	*
	000008								
	000009								
	000010								
	000011								
	000012								
	000013								
	000014								
	000015								
	000016								
	000017								
	000018								
	000019								
	000020								
	000021								
	000022								
	000023								
	...								

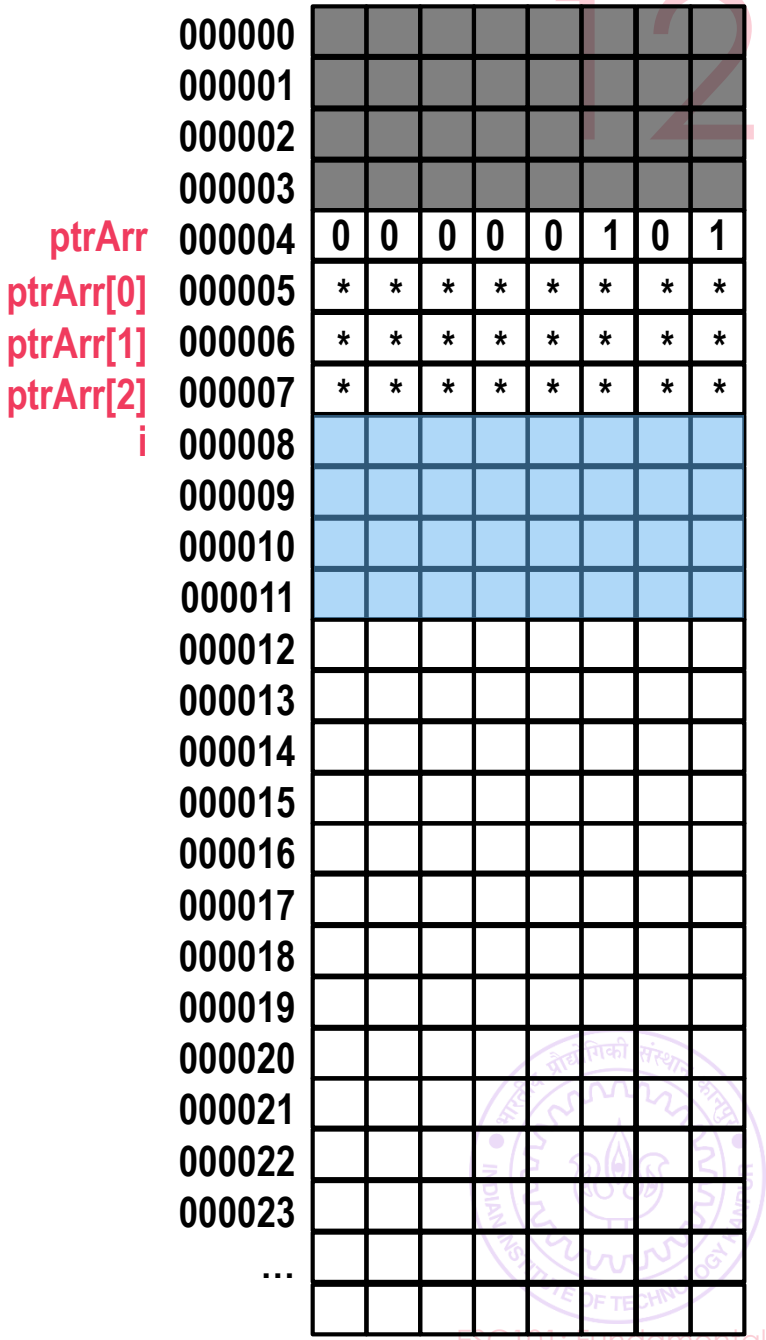
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
```



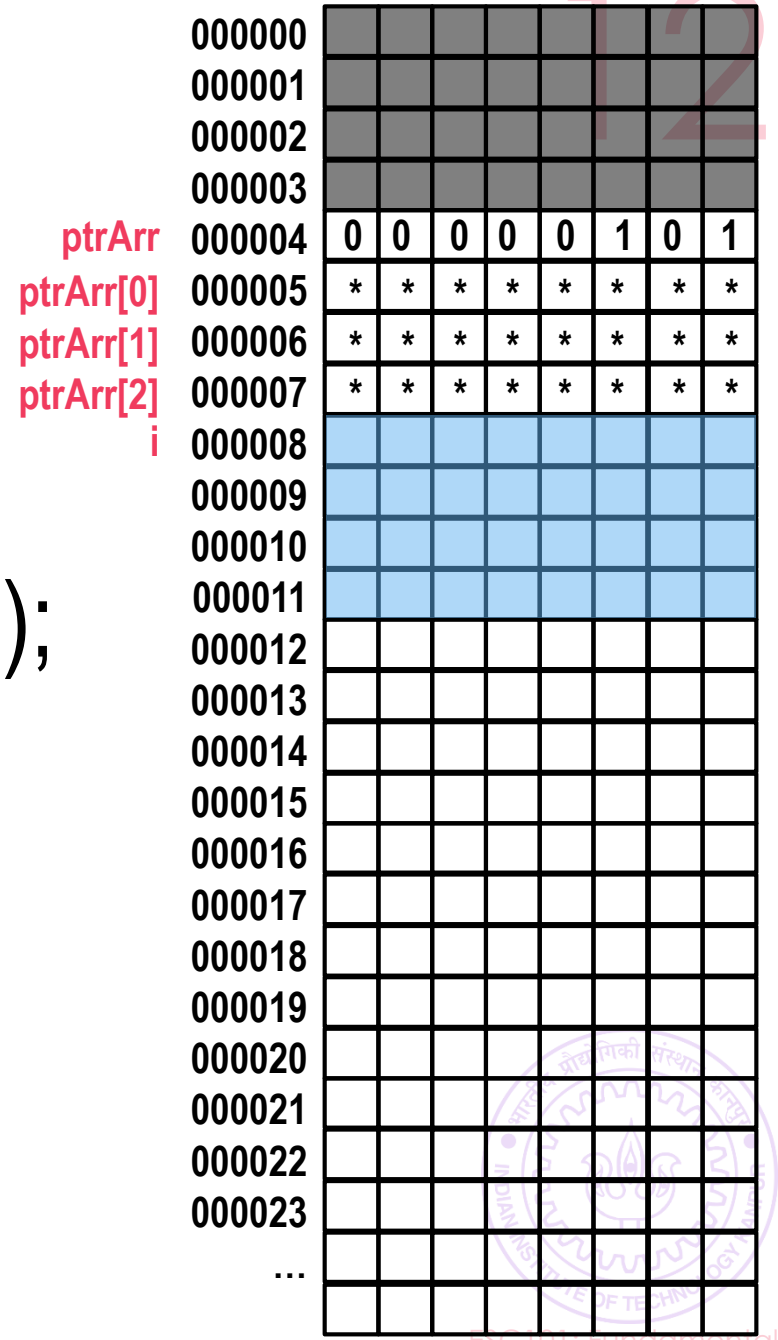
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
```



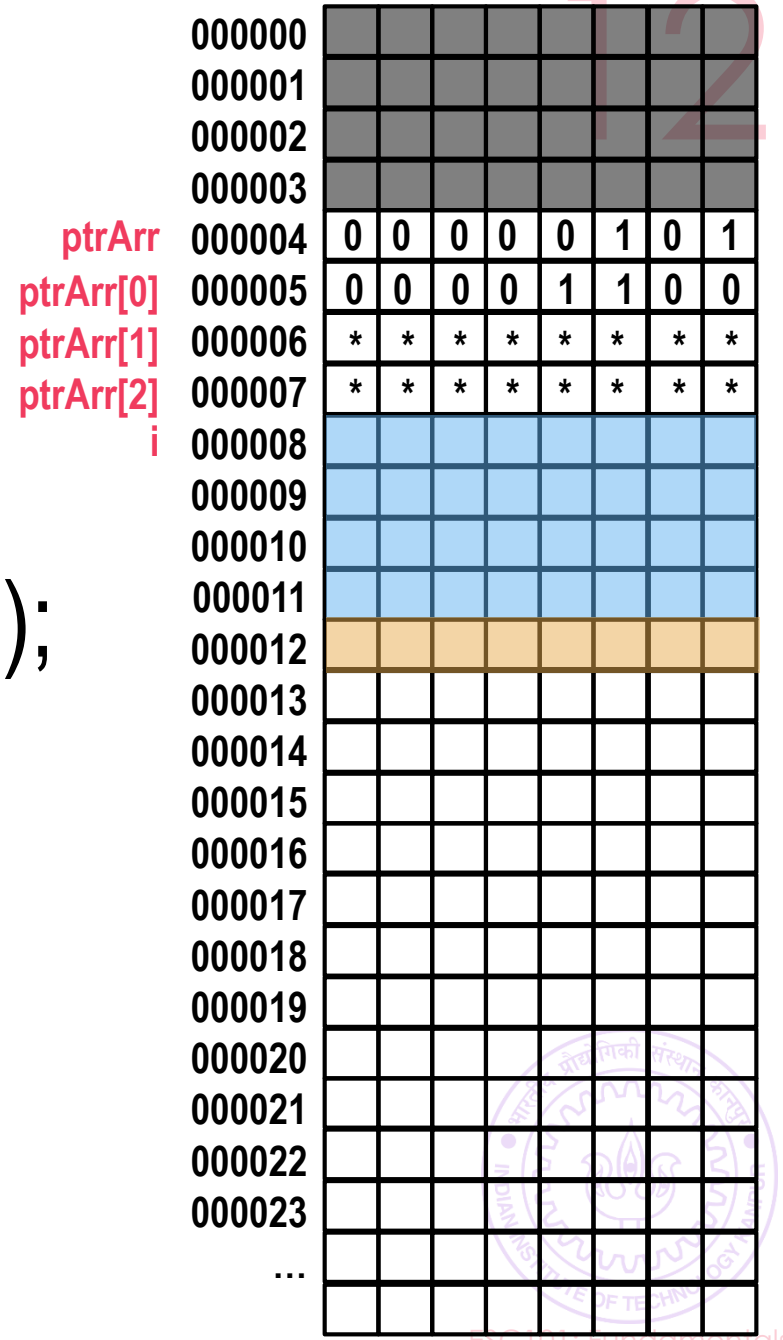
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```



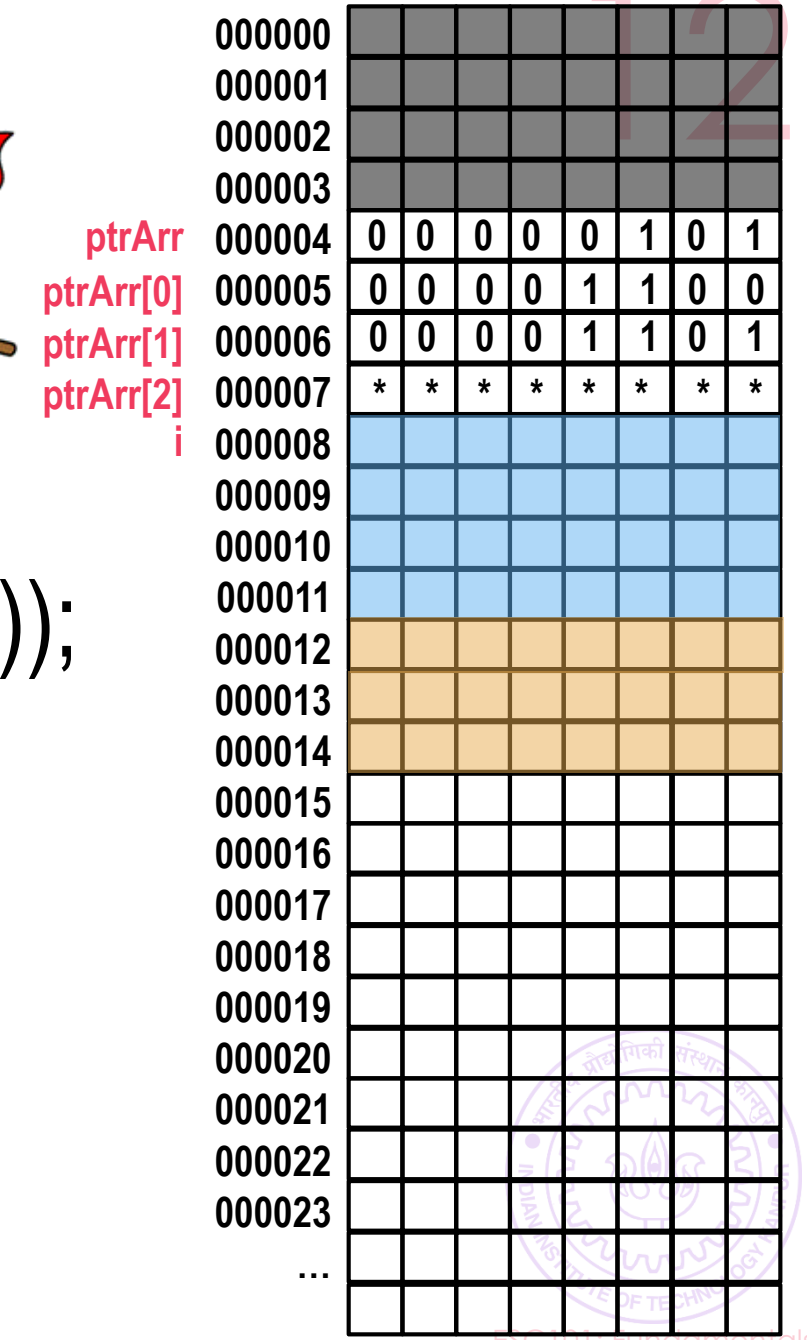
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```



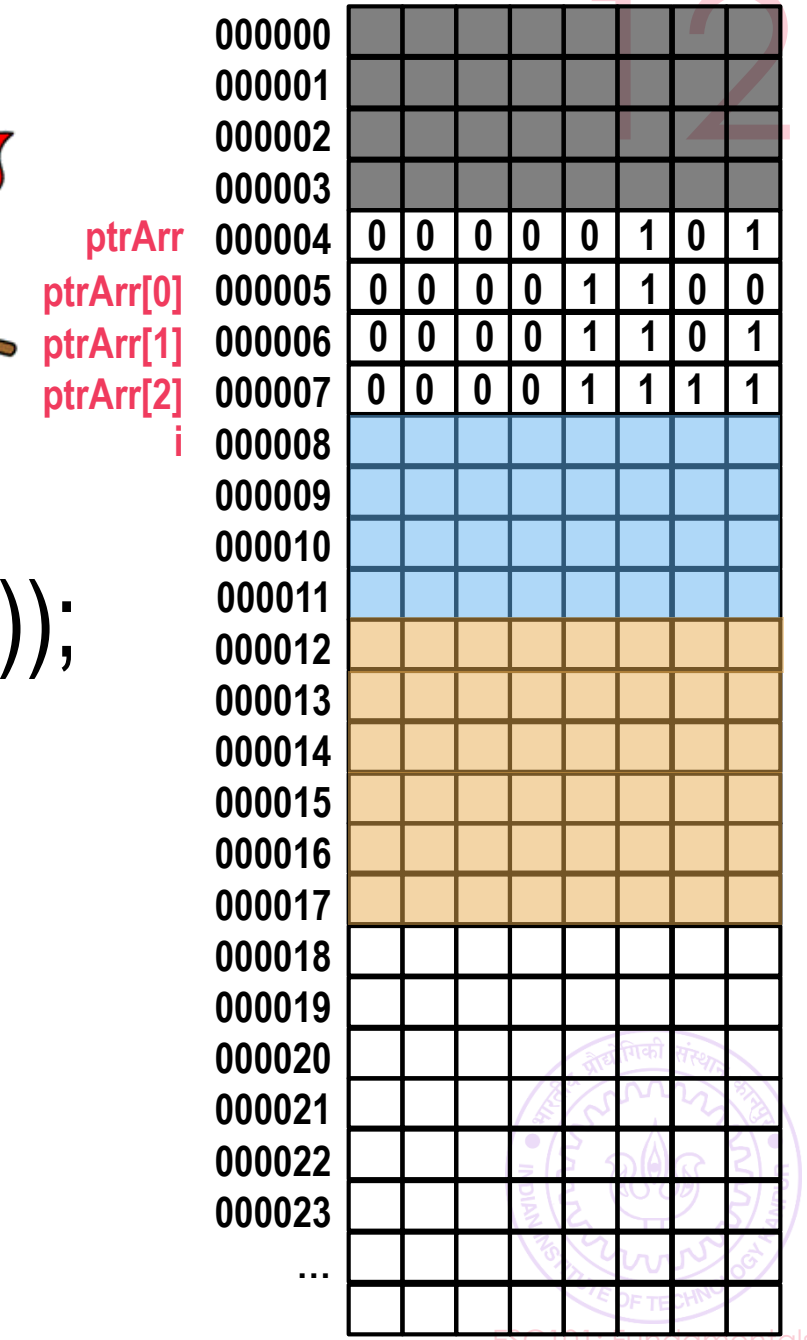
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```



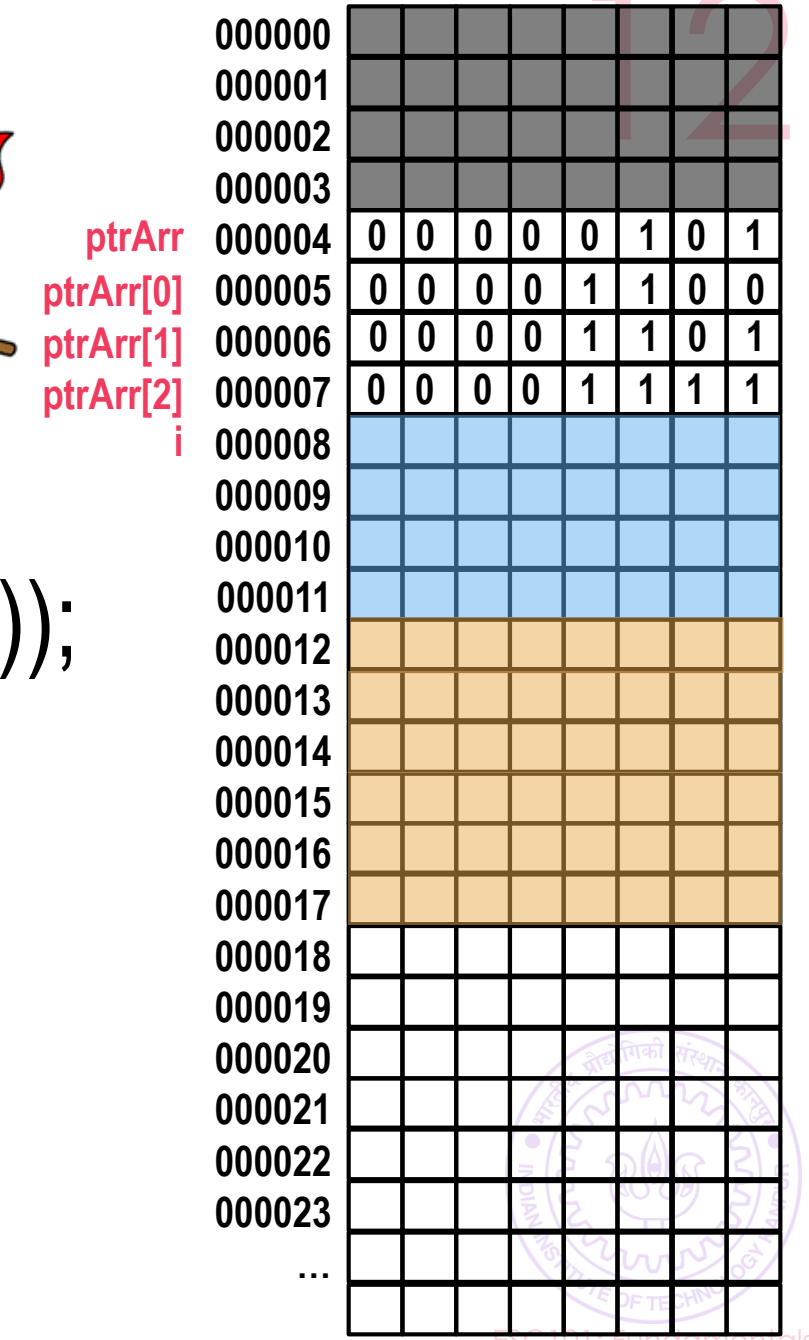
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```



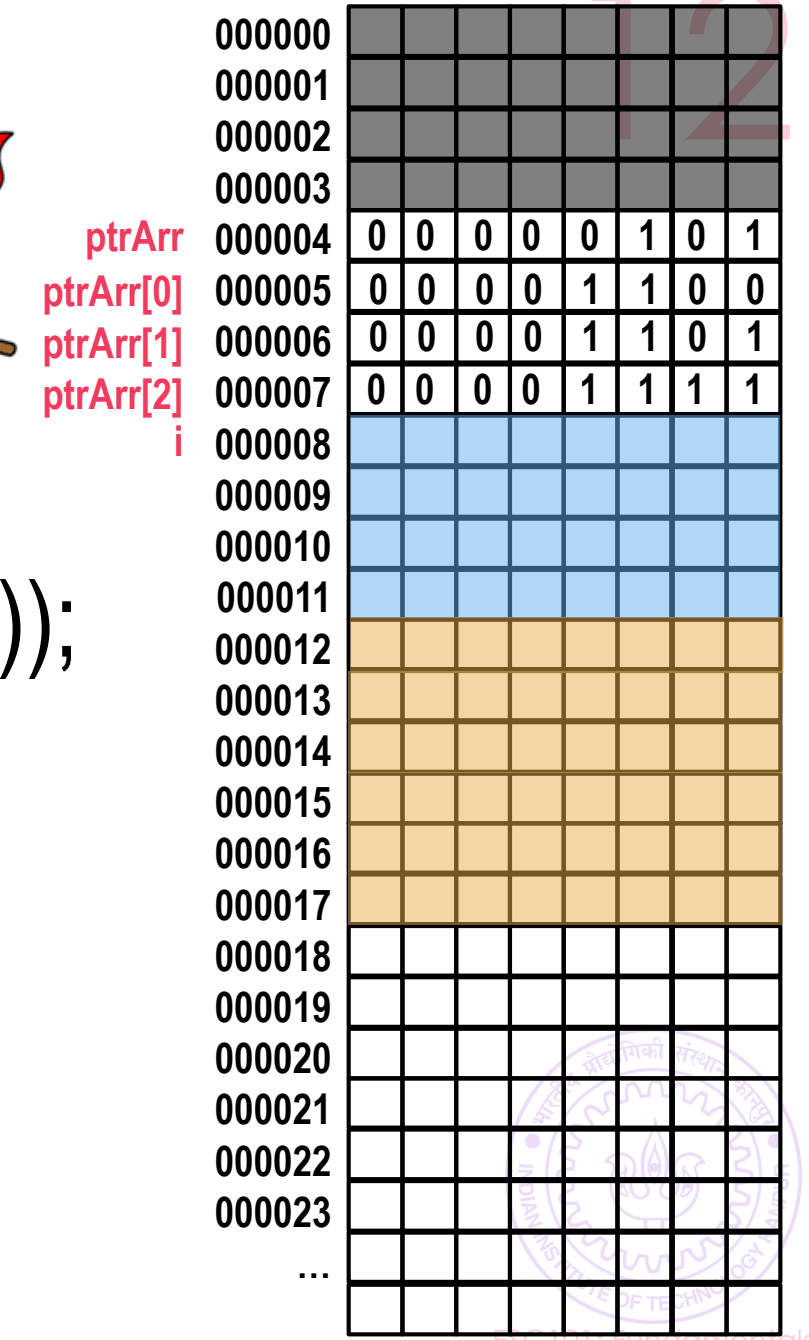
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
```



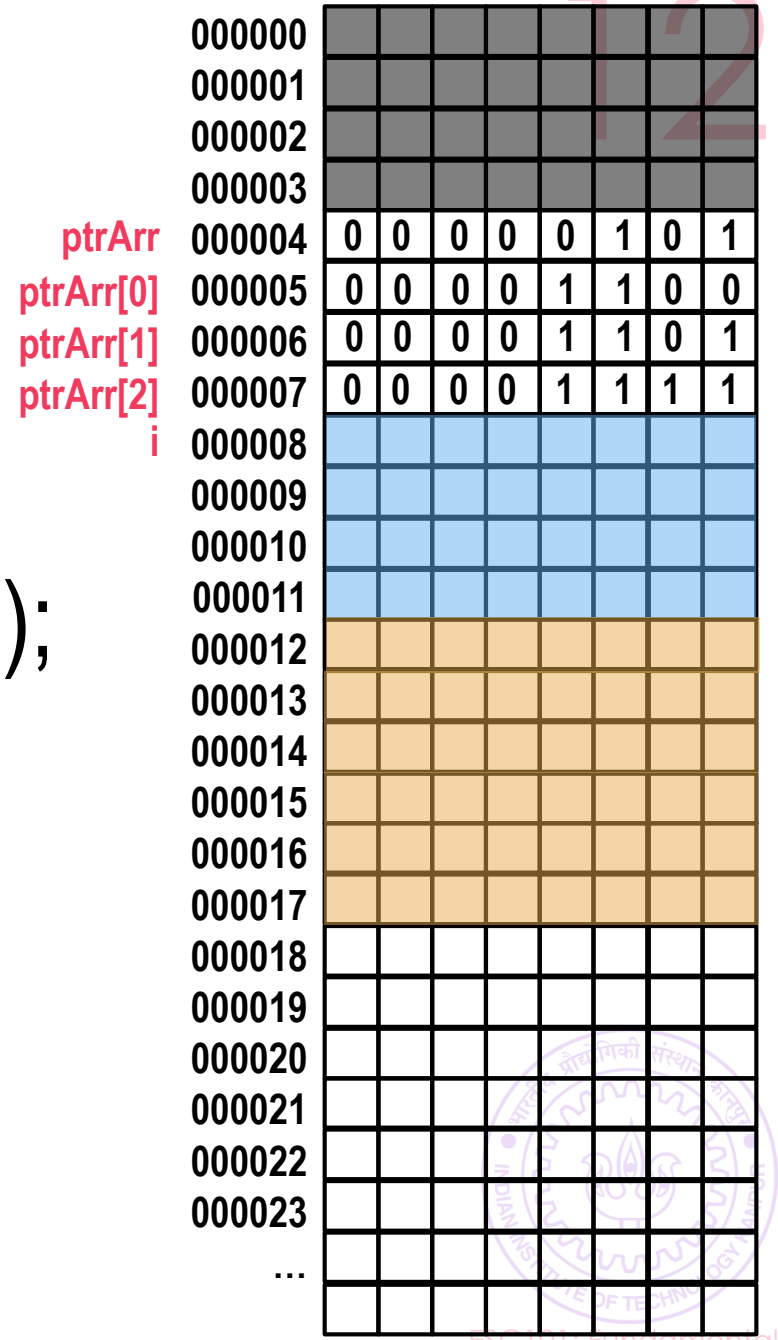
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
```



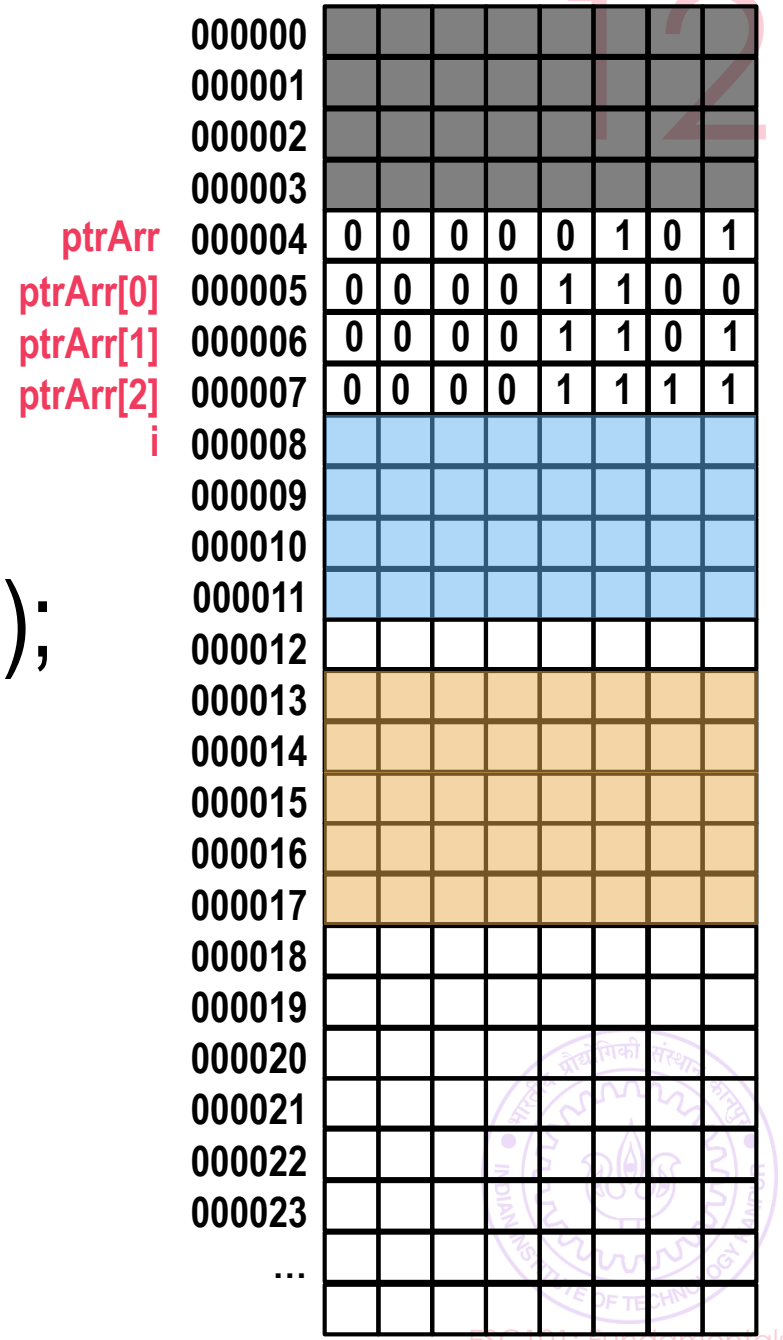
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
```



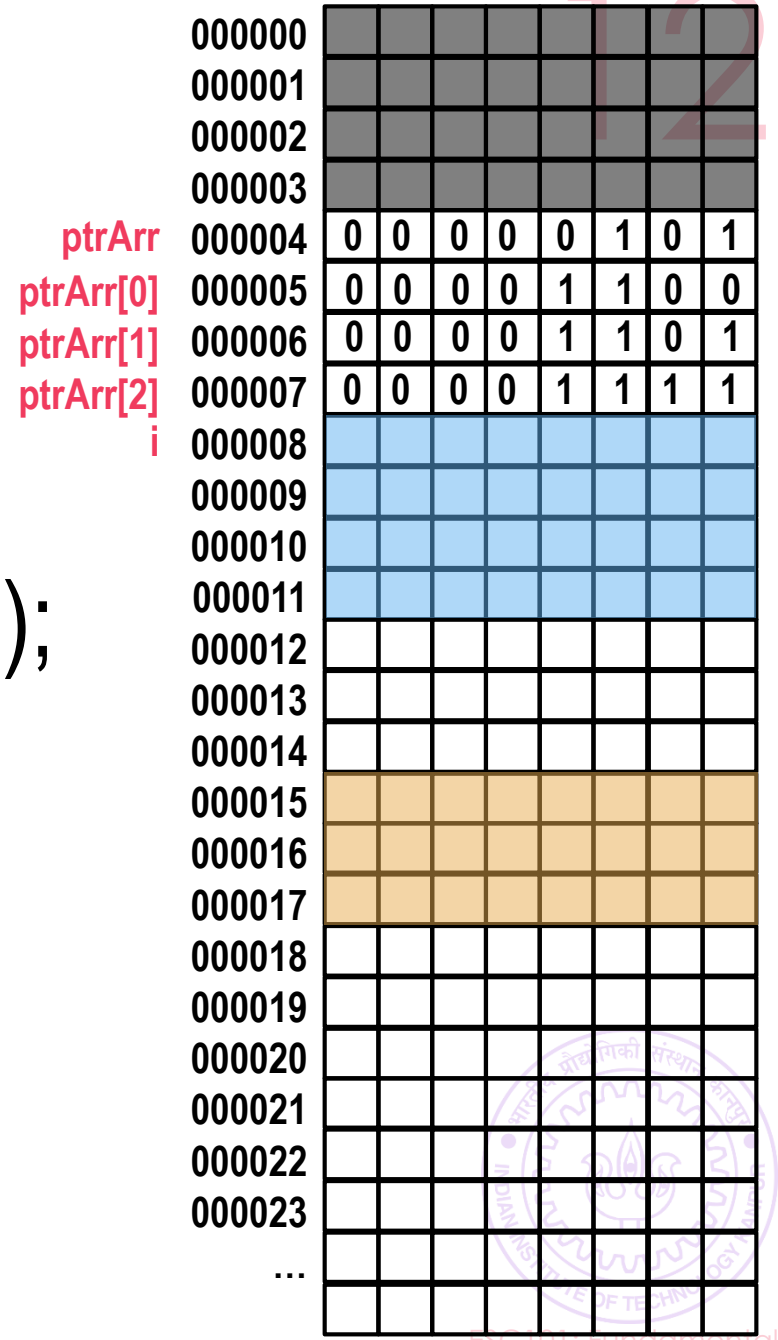
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
```



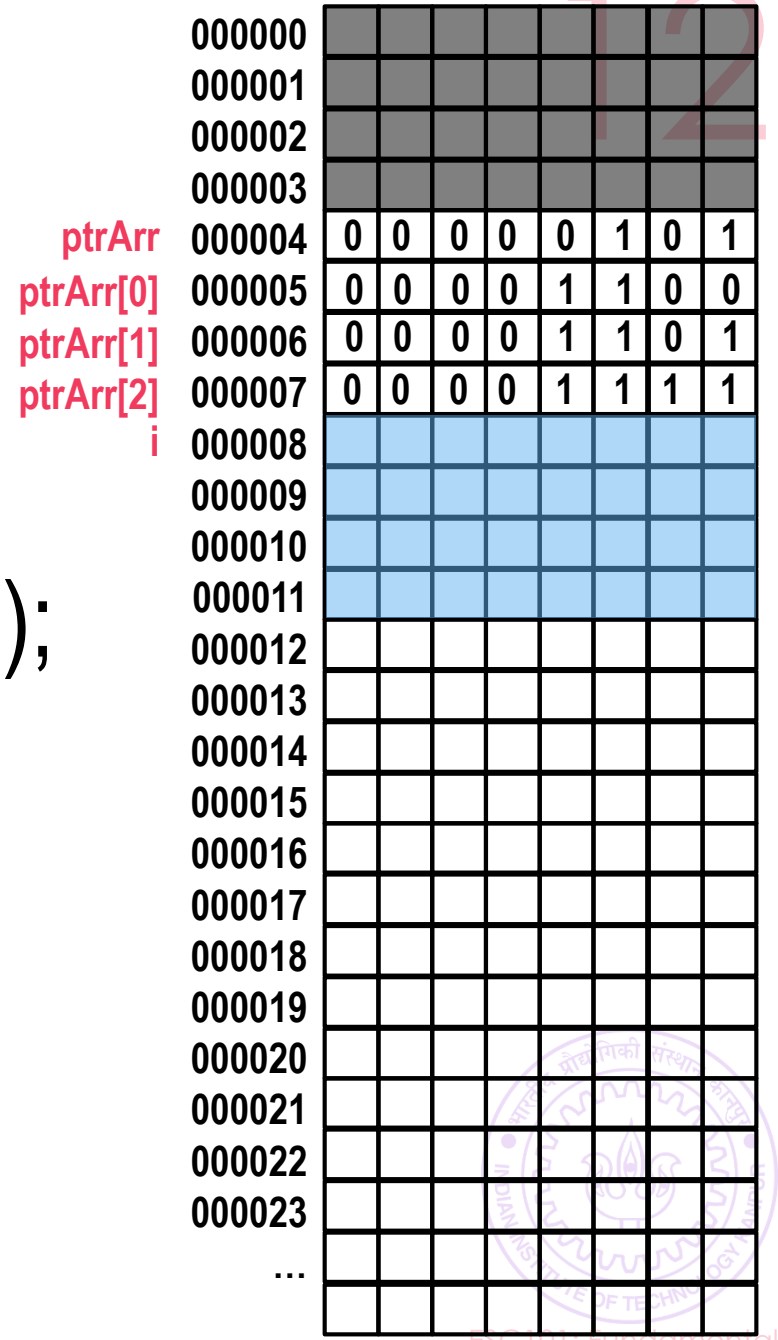
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
```



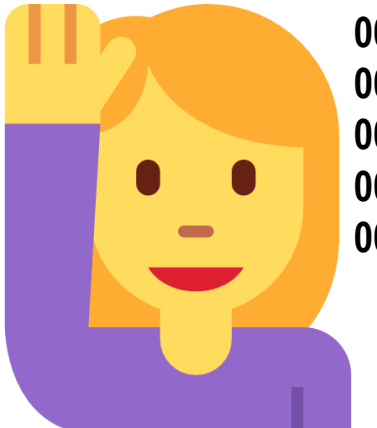
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

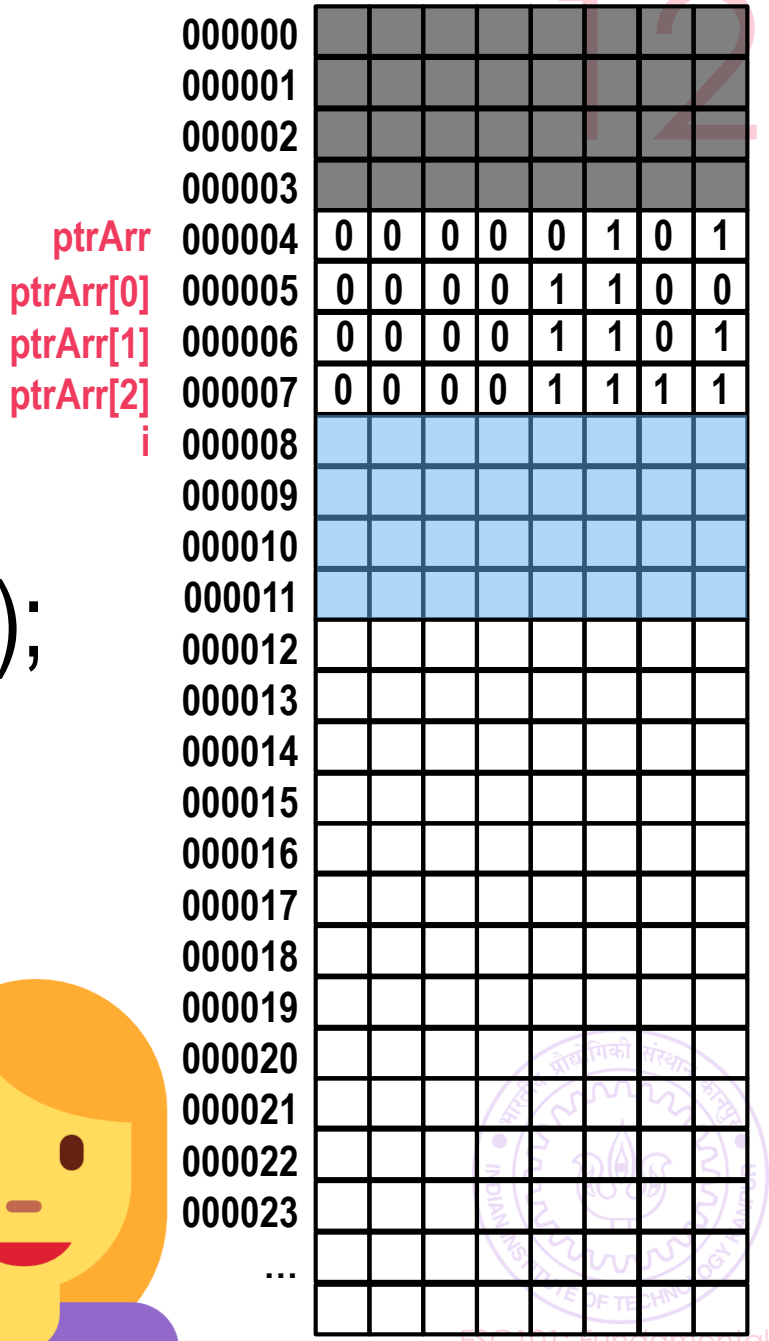
```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
```



All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s", ptrArr[2]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
```

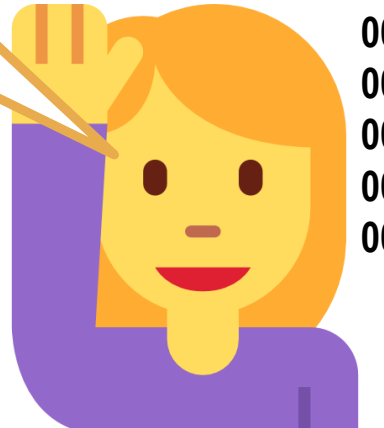




All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%s", ptrArr[2]);
printf("%s",
for(i = 0; i <
        free(ptrArr[i]);
```

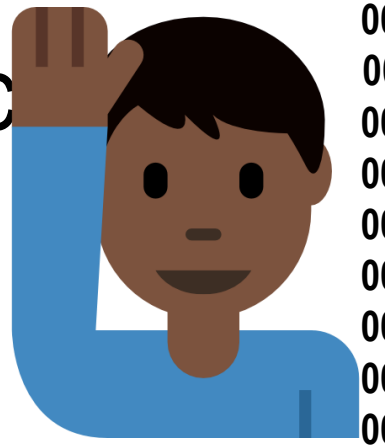
Very useful in programs where we have to create a known number of arrays of unknown length

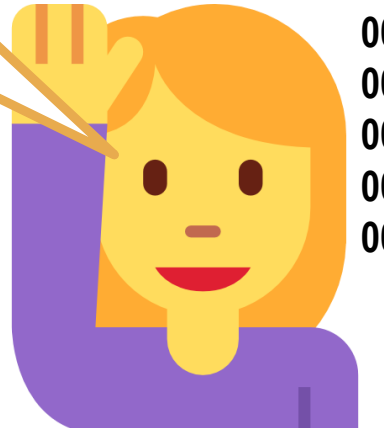


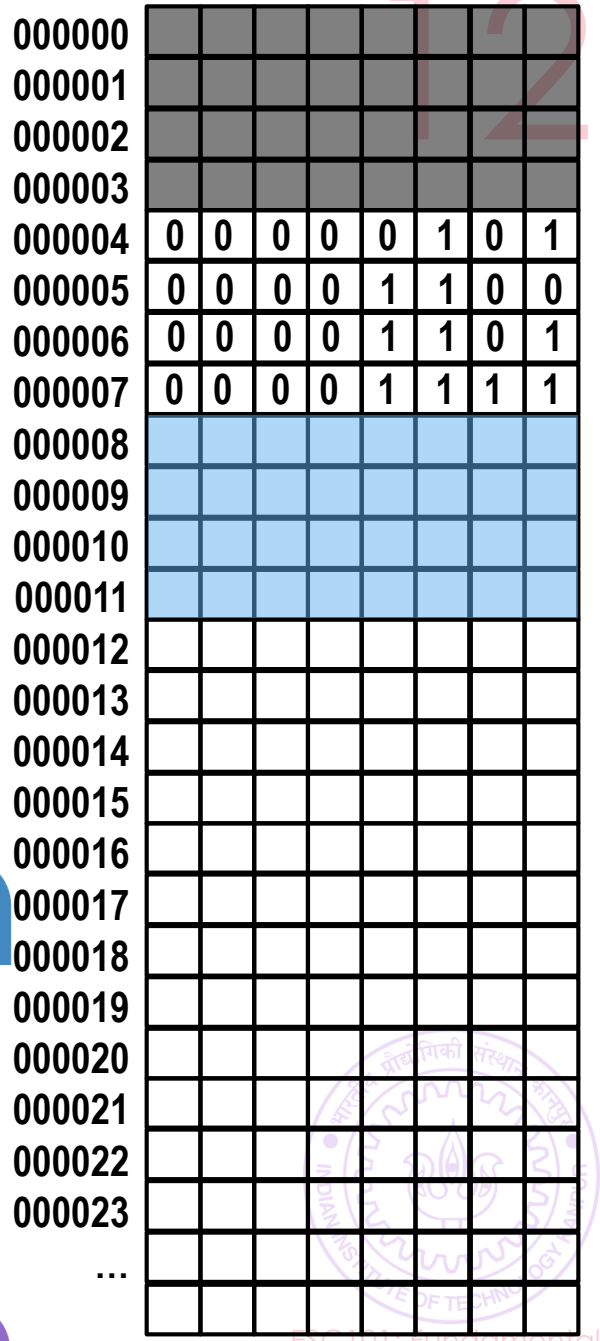
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(c
scanf("%s", ptrArr[2]);
printf("%s",
for(i = 0; i <
        free(ptrArr[i]);
```

Very useful in programs where we have to create a known number of arrays of unknown length







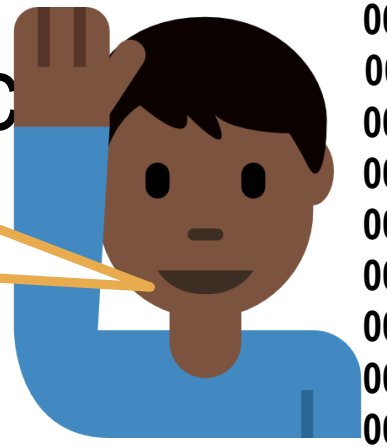
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

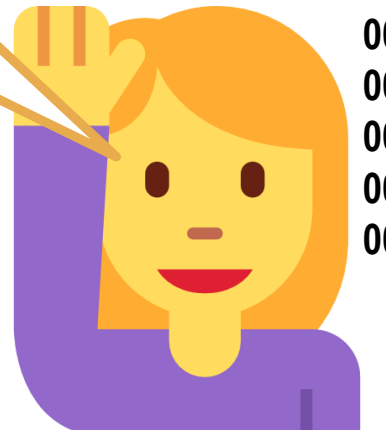
```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(c
printf("%s",
for(i = 0; i <
    free(ptrArr[i]);
```

Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```

Very useful in programs where we have to create a known number of arrays of unknown length





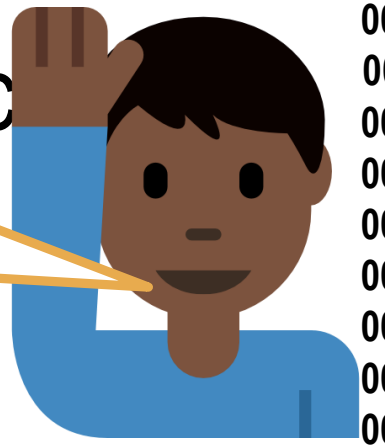
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

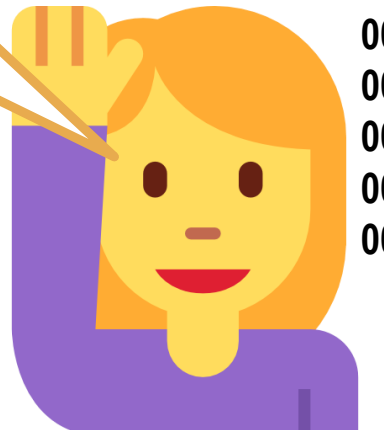
```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(c
printf("%s",
for(i = 0; i <
    free(ptrArr[i]);
```

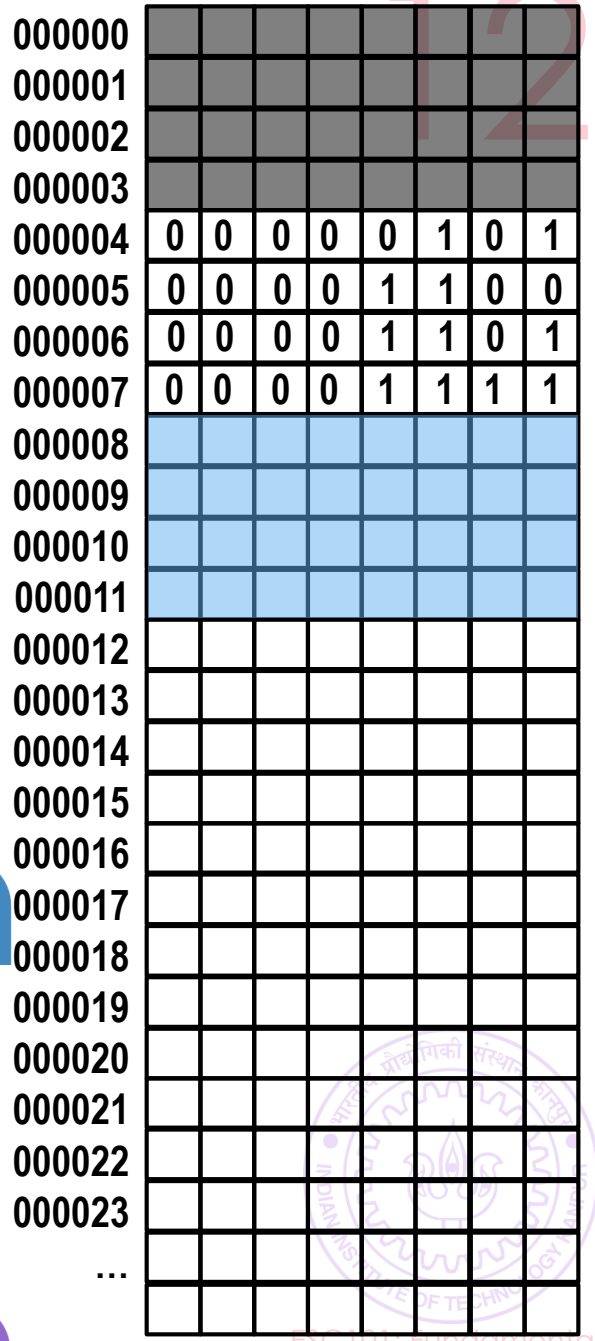
Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```

Very useful in programs where we have to create an unknown number of arrays of unknown length







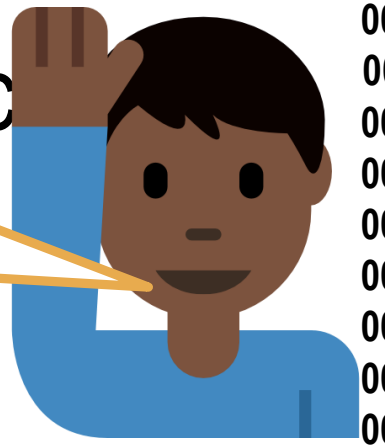
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(c
printf("%s",
for(i = 0; i <
    free(pt
```

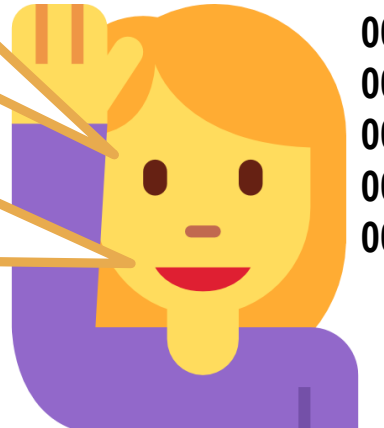
	000000							
	000001							
	000002							
	000003							
ptrArr	000004	0	0	0	0	0	1	0	1
ptrArr[0]	000005	0	0	0	0	1	1	0	0
ptrArr[1]	000006	0	0	0	0	1	1	0	1
ptrArr[2]	000007	0	0	0	0	1	1	1	1
i	000008								
	000009								
	000010								
	000011								
	000012								
	000013								
	000014								
	000015								
	000016								
	000017								
	000018								
	000019								
	000020								
	000021								
	000022								
	000023								
	...								

Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```



Very useful in programs where we have to create an unknown number of arrays of unknown length

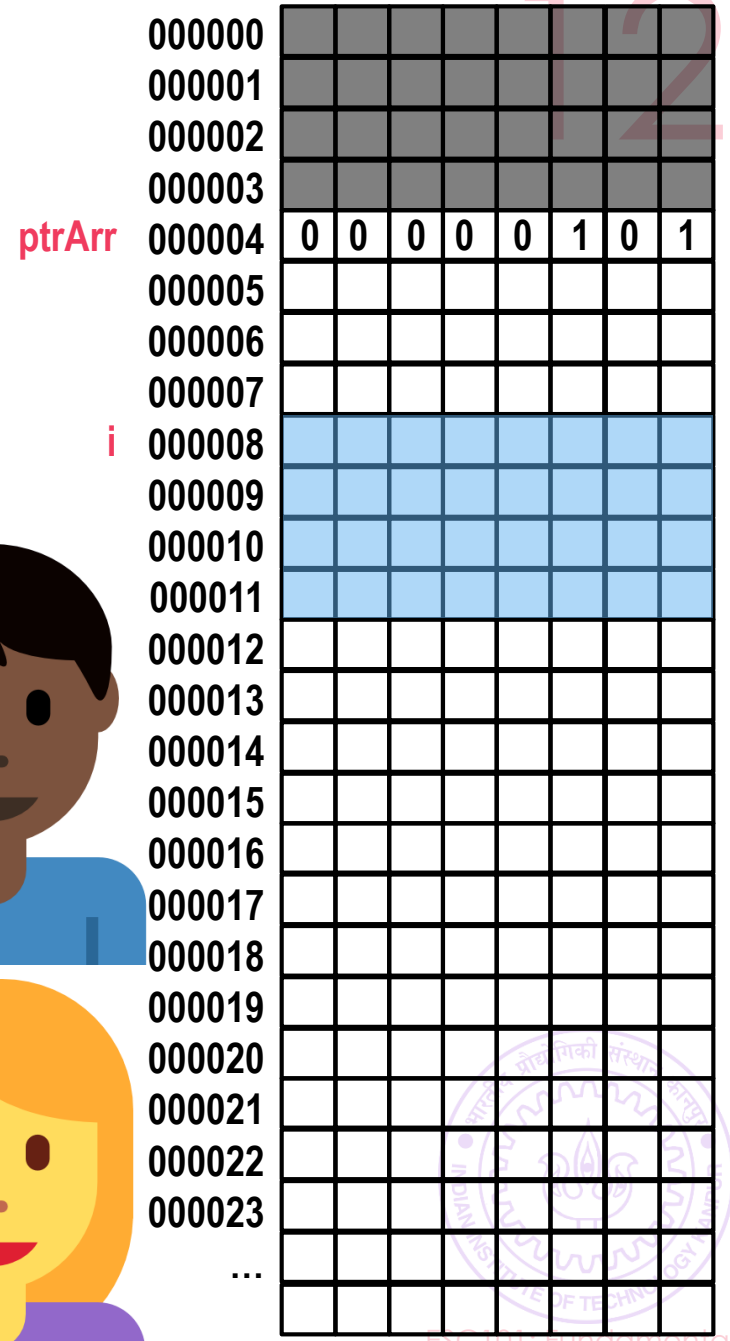


However, in this case we should also free pointer array by writing

```
free(ptrArr);
```

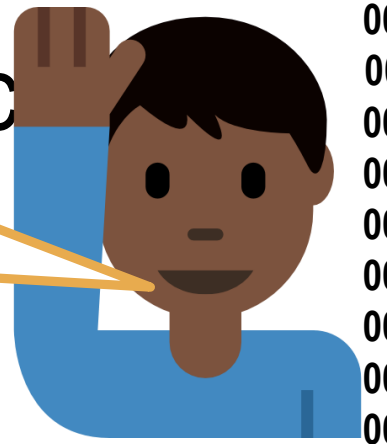
All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(c
printf("%s",
for(i = 0; i <
    free(pt
```

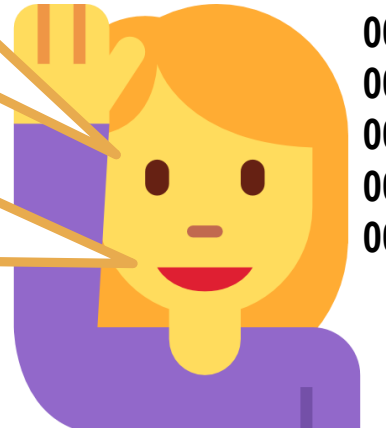


Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```



Very useful in programs where we have to create an unknown number of arrays of unknown length

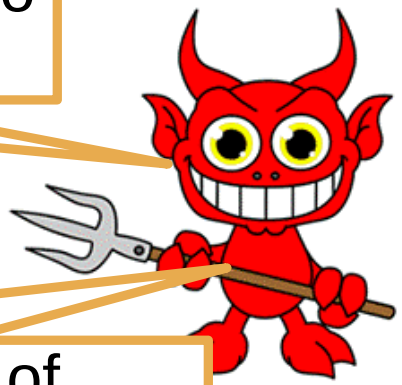


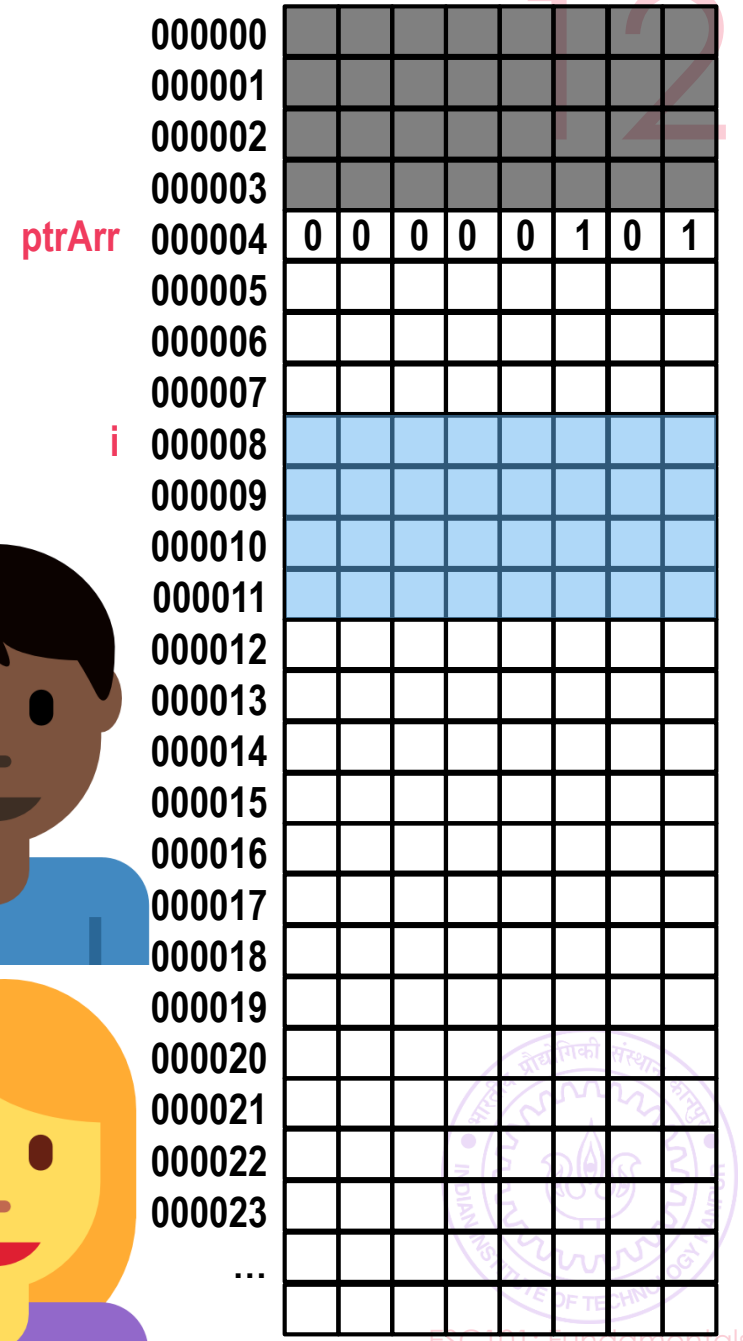
However, in this case we should also free pointer array by writing

```
free(ptrArr);
```

All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately

```
char *ptrArr[3];
int i;
```



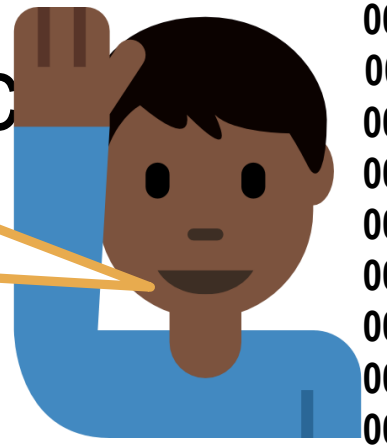


Exercise: Write a program to create an array of strings containing all substrings of string entered by user. The string entered by user of unknown length

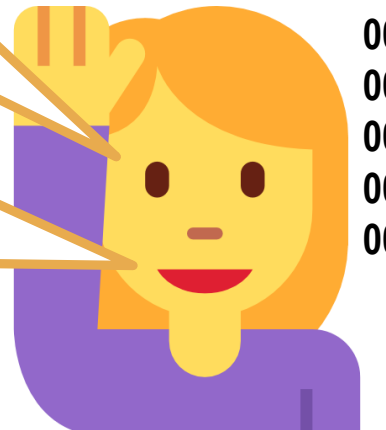
```
printf("%s",
for(i = 0; i <
    free(pt
```

Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```



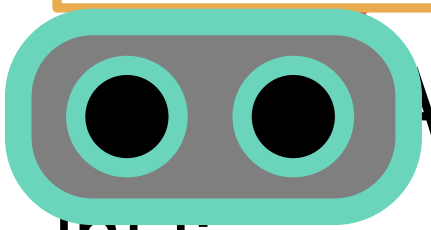
Very useful in programs where we have to create an unknown number of arrays of unknown length

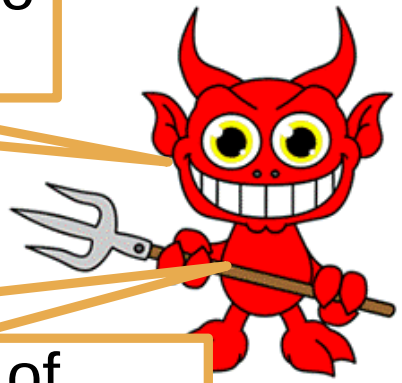


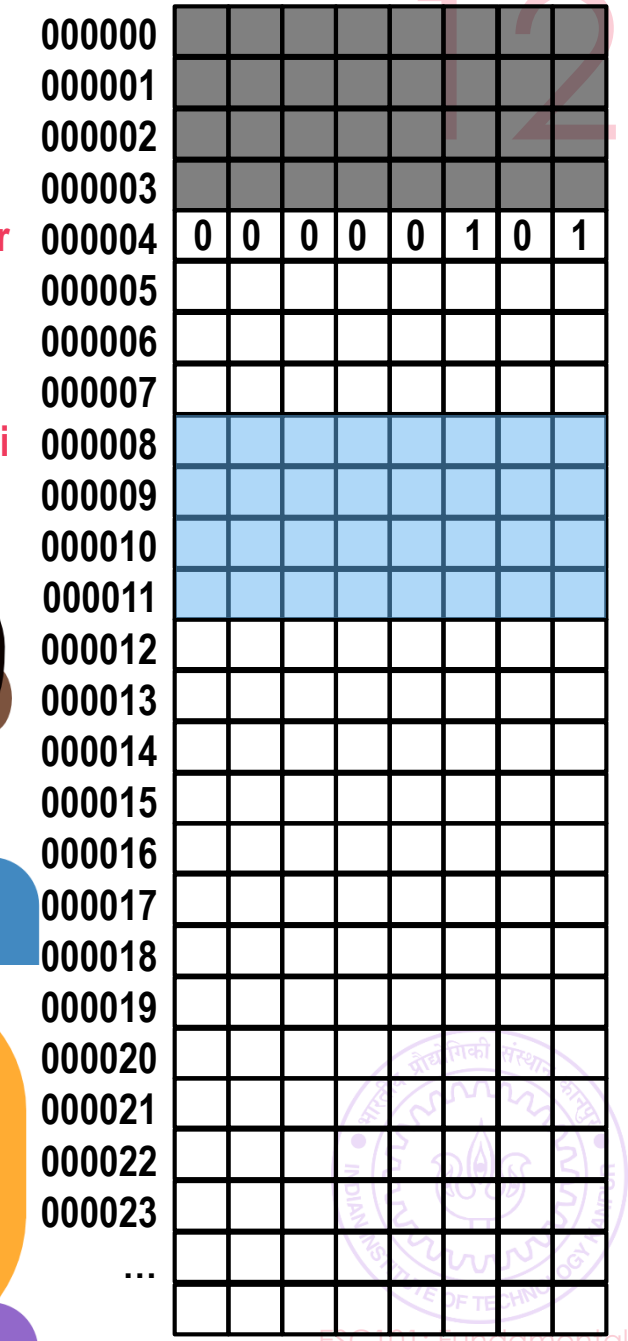
However, in this case we should also free pointer array by writing

```
free(ptrArr);
```

All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately



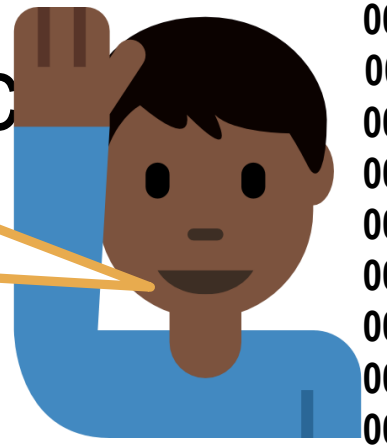




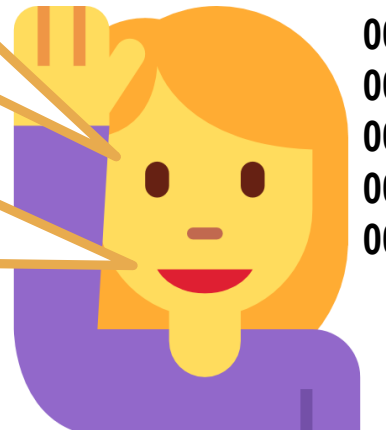
Exercise: Write a program to create an array of strings containing all substrings of string entered by user. The string entered by user of unknown length

Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```



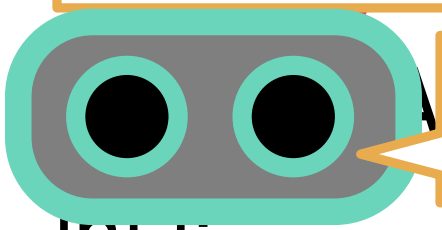
Very useful in programs where we have to create an unknown number of arrays of unknown length



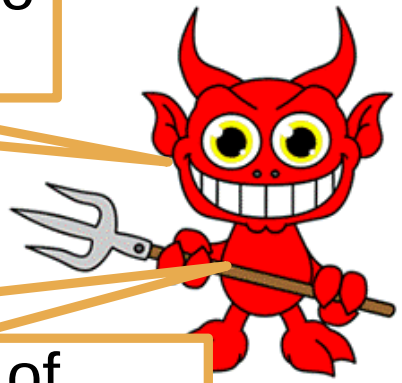
However, in this case we should also free pointer array by writing

```
free(ptrArr);
```

All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately



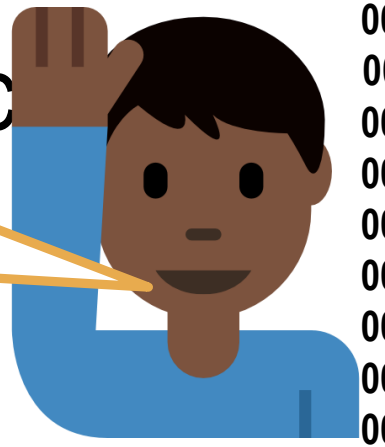
Can we make animations a bit easier to remember?



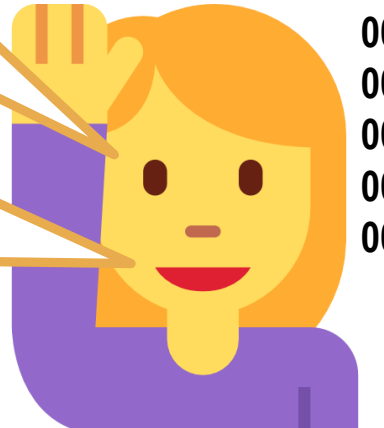
Exercise: Write a program to create an array of strings containing all substrings of string entered by user. The string entered by user of unknown length

Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```

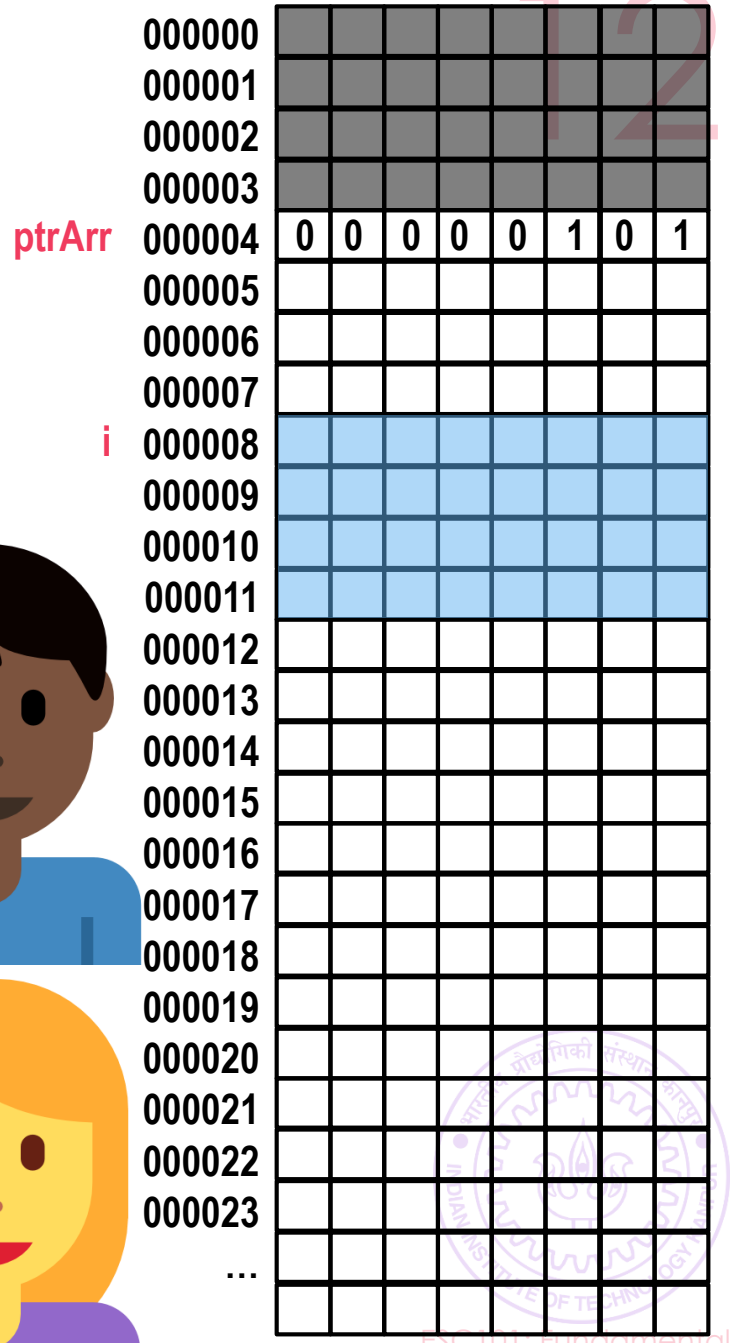


Very useful in programs where we have to create an unknown number of arrays of unknown length

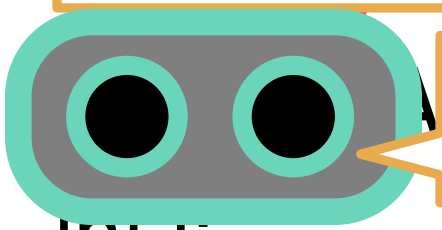


However, in this case we should also free pointer array by writing

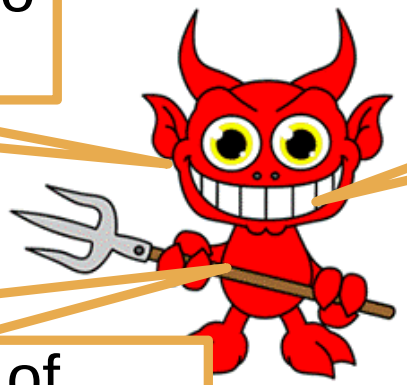
```
free(ptrArr);
```



All elements of ptrArr array will take 8 bytes to store – sorry for not drawing accurately



Can we make animations a bit easier to remember?

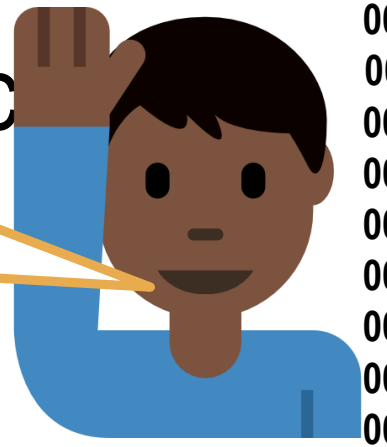


Anything for you, Mr C!

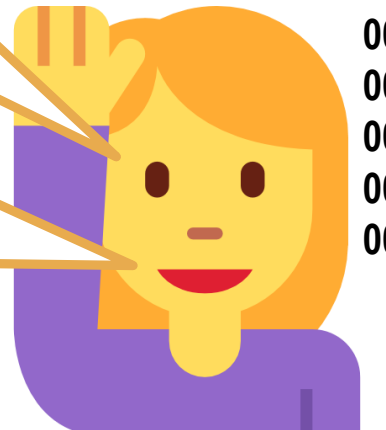
Exercise: Write a program to create an array of strings containing all substrings of string entered by user. The string entered by user of unknown length

Alternate way to declare array of arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
```

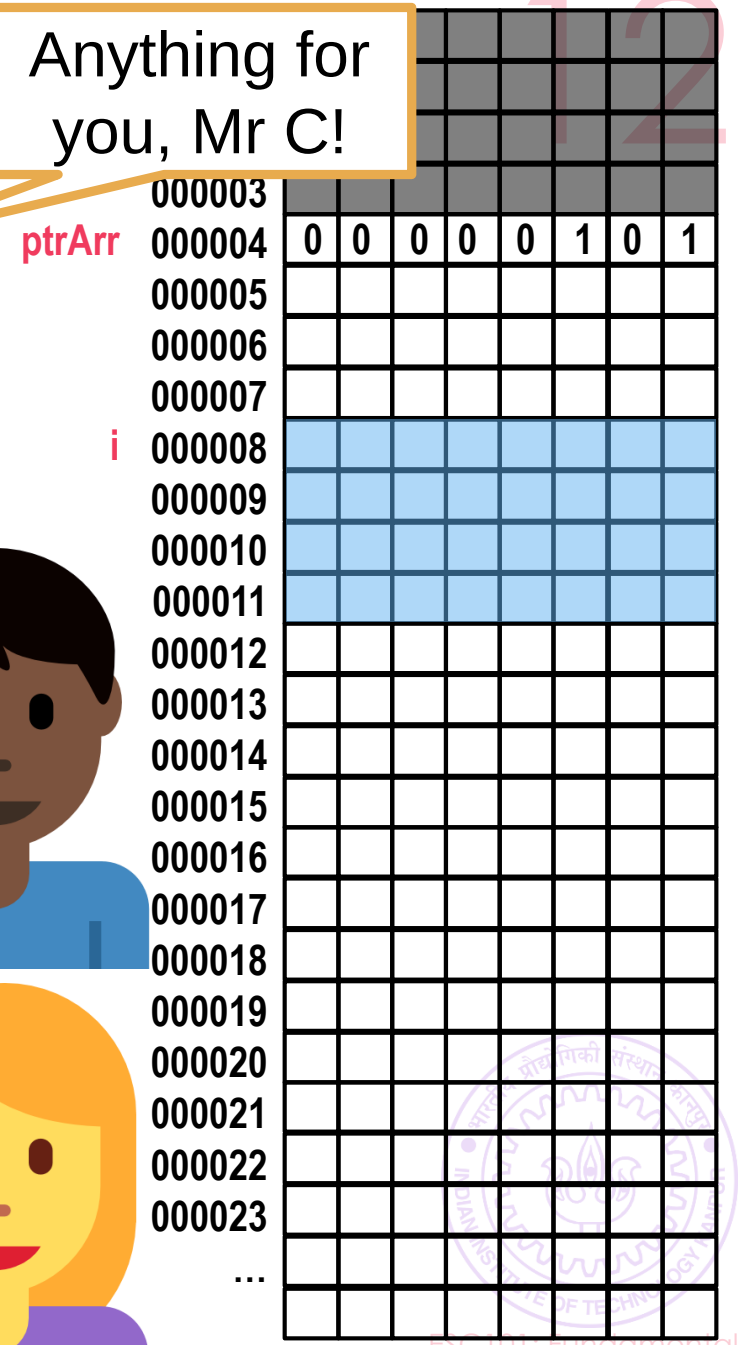


Very useful in programs where we have to create an unknown number of arrays of unknown length



However, in this case we should also free pointer array by writing

```
free(ptrArr);
```



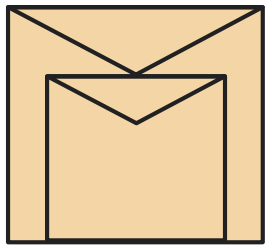
Array of Pointers ➔ Arrays of Arrays

Array of Pointers ➔ Arrays of Arrays


```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```

Array of Pointers ➔ Arrays of Arrays

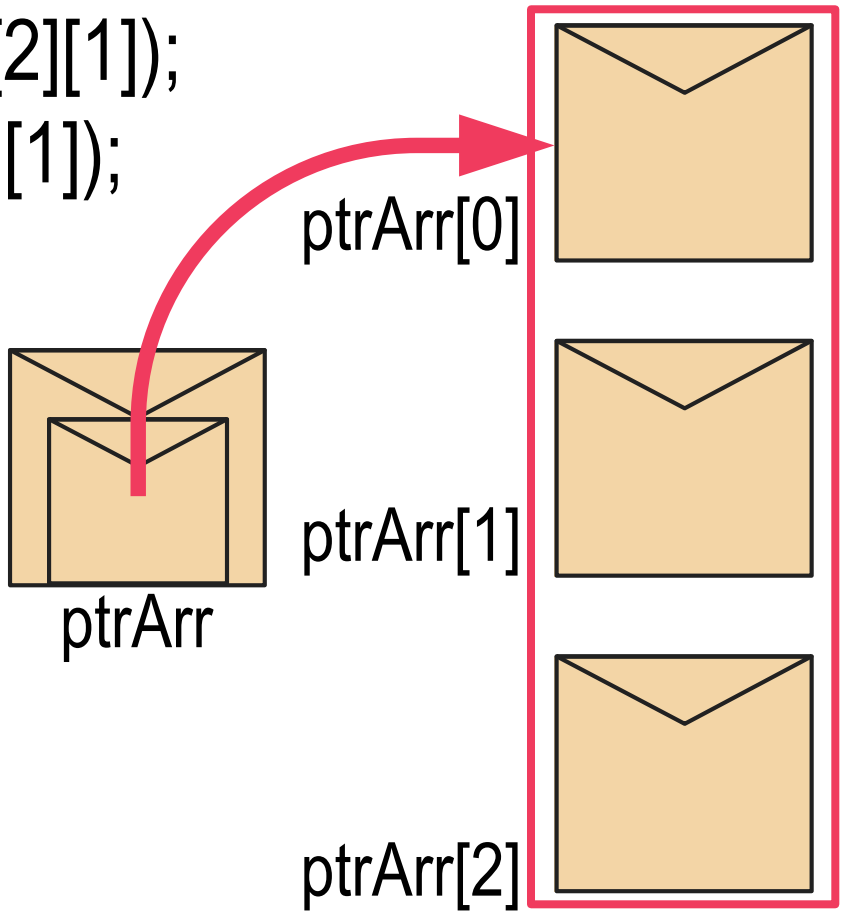

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```



ptrArr

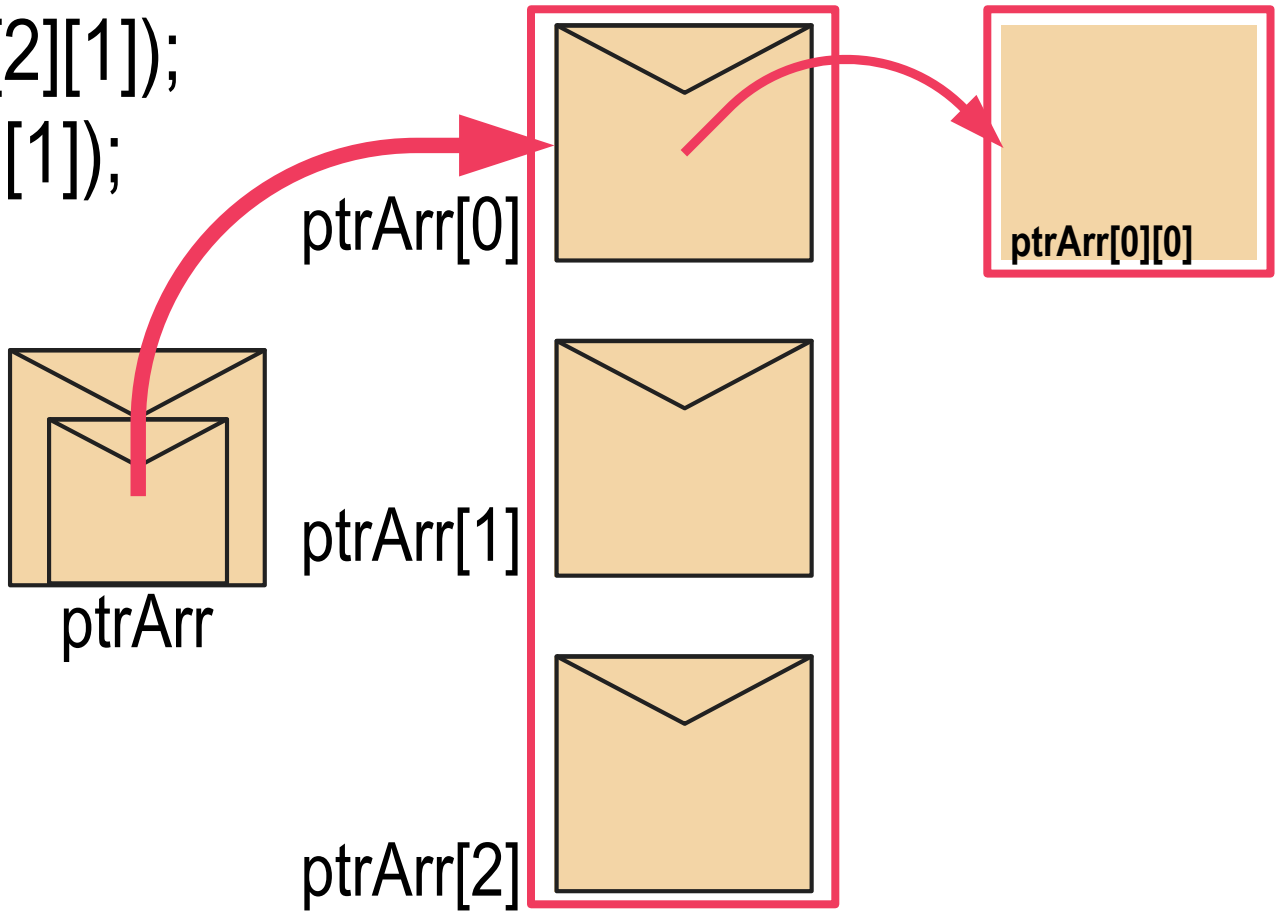
Array of Pointers ➔ Arrays of Arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```



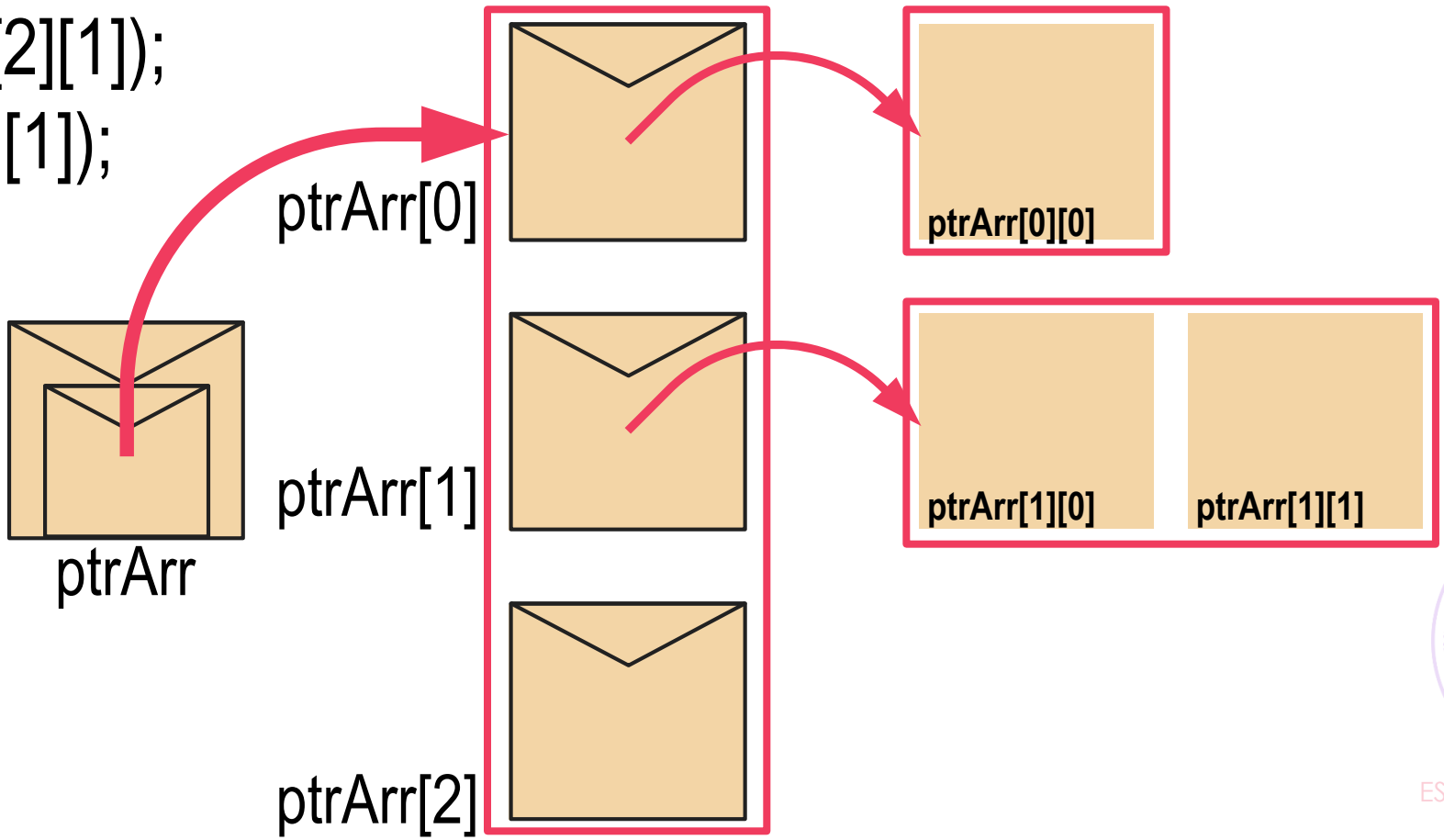
Array of Pointers ➔ Arrays of Arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```



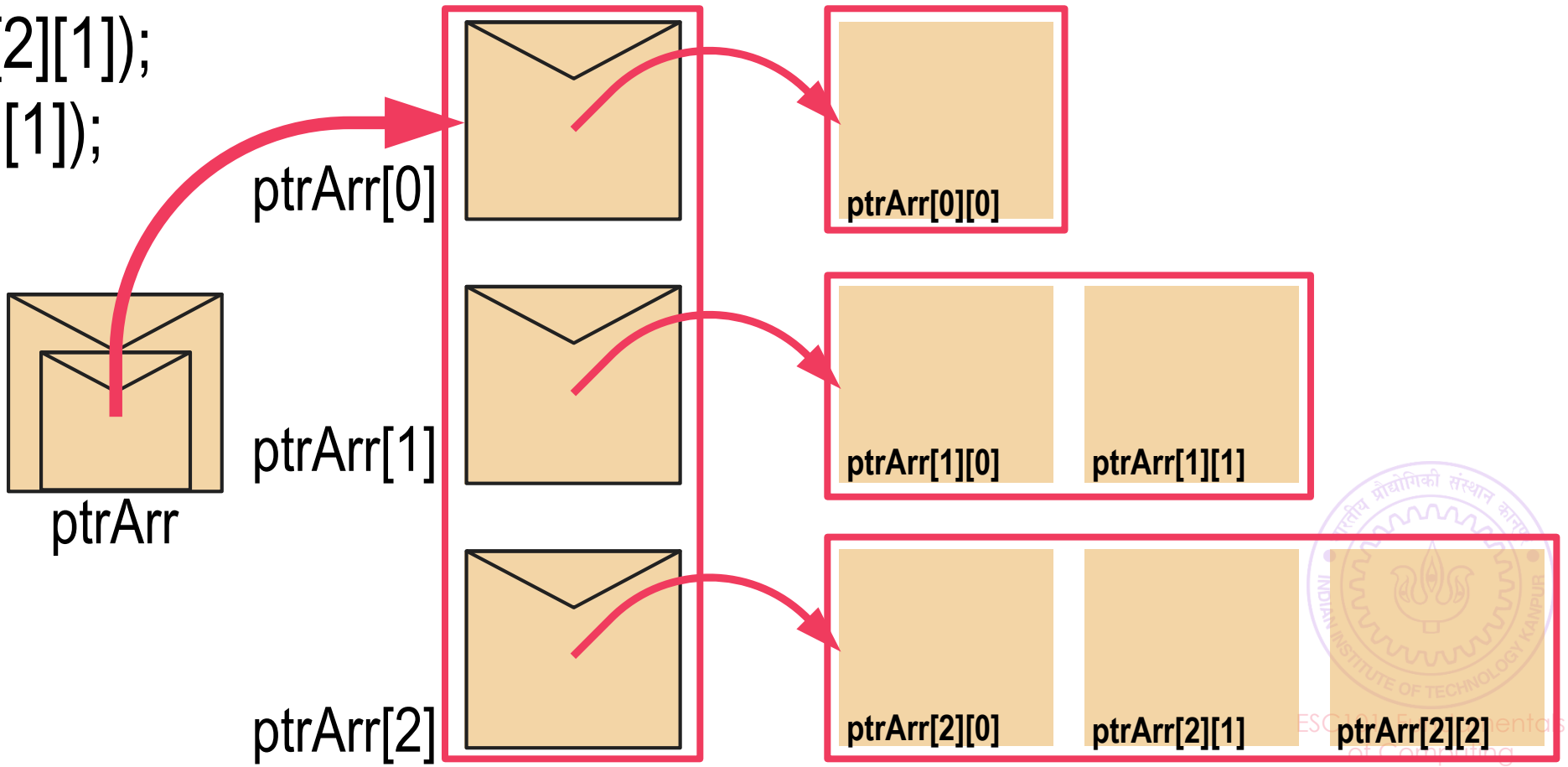
Array of Pointers ➔ Arrays of Arrays


```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```



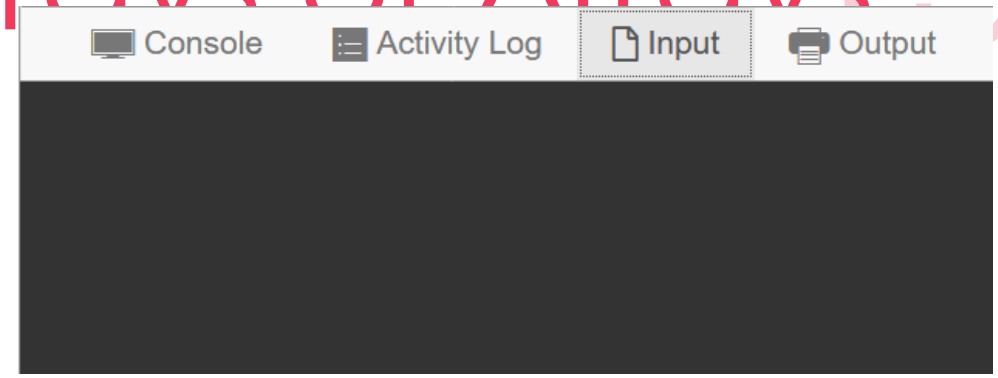
Array of Pointers ➔ Arrays of Arrays

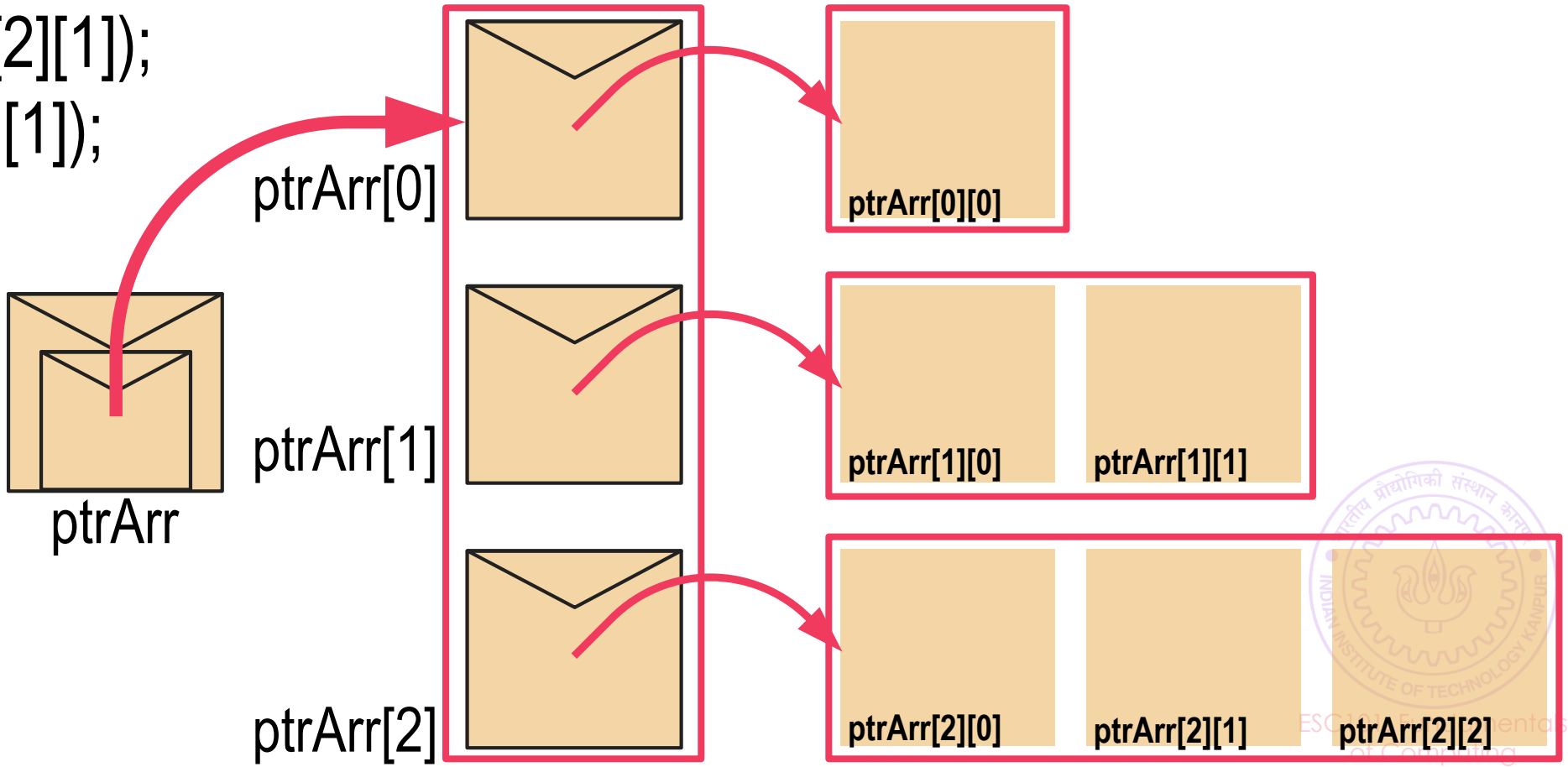
```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```



Array of Pointers ➔ Arrays of Arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```



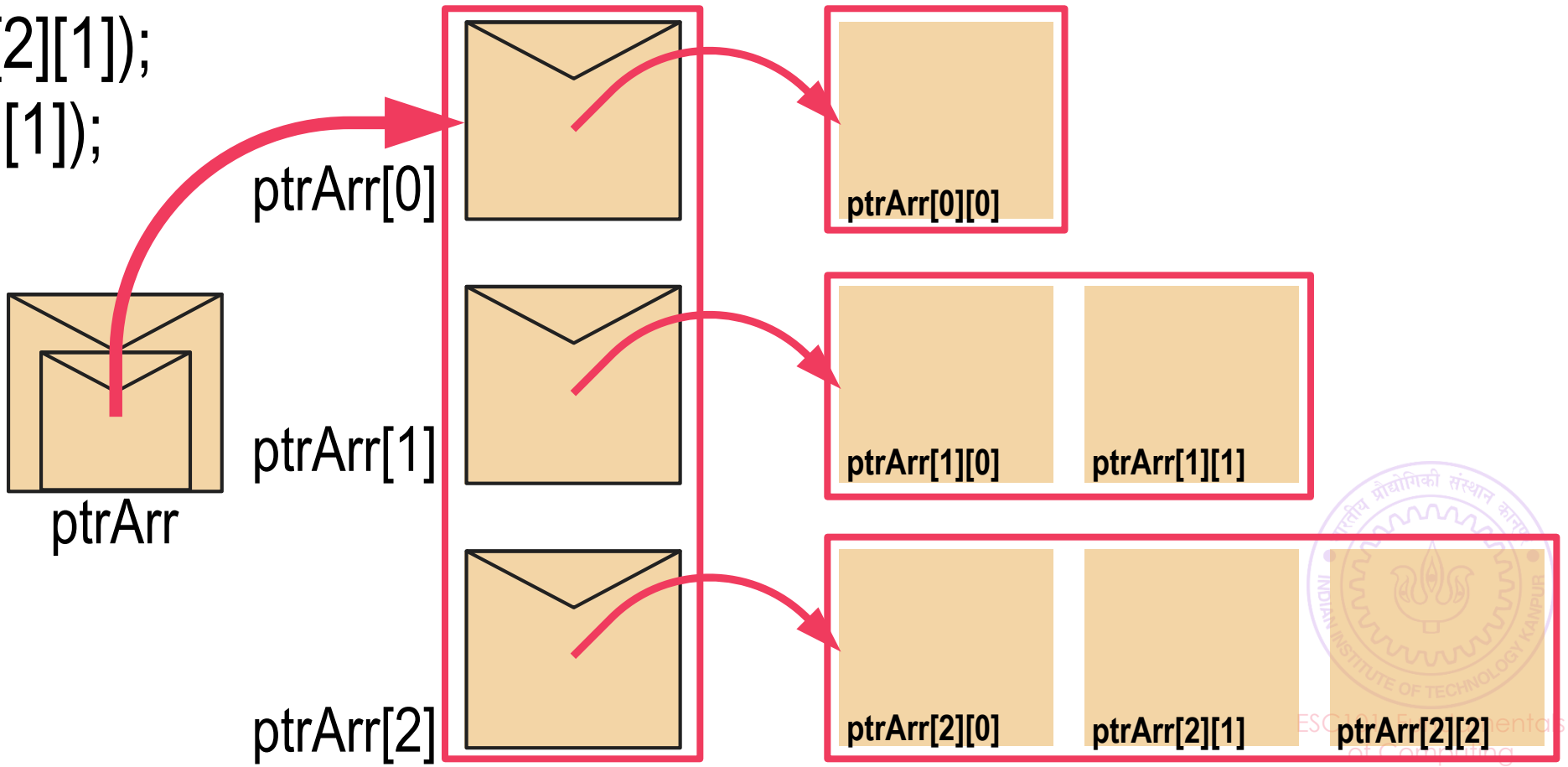


Array of Pointers ➔ Arrays of Arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```

Console | Activity Log | Input | Output

X

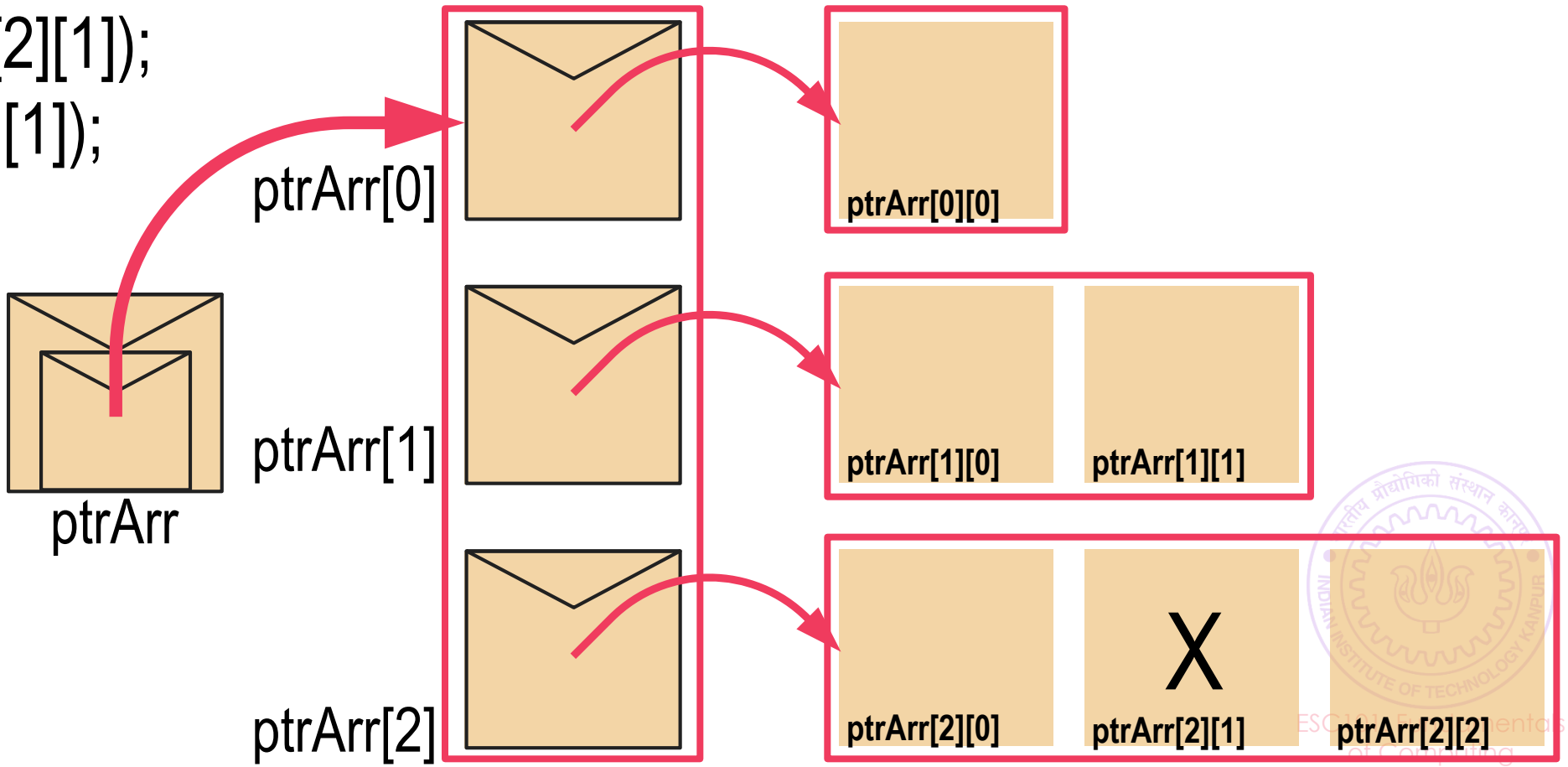


Array of Pointers ➔ Arrays of Arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```

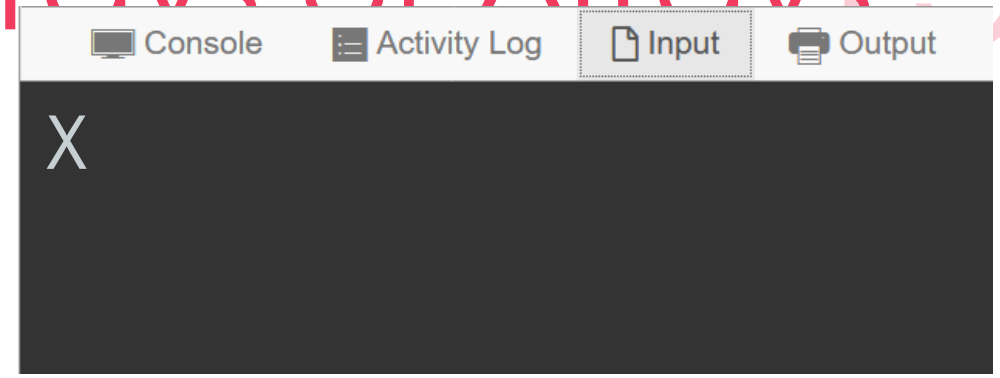
Console | Activity Log | Input | Output

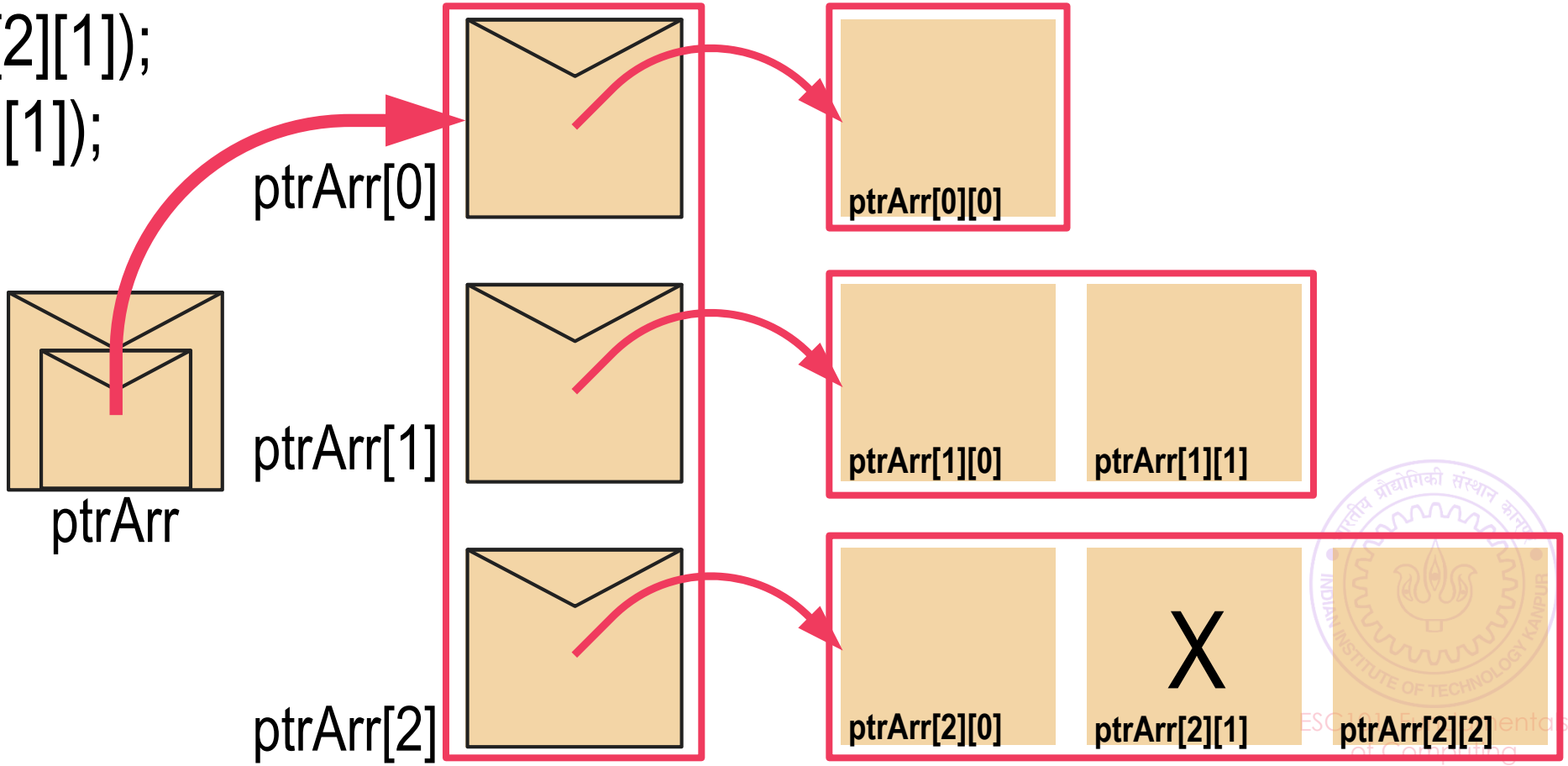
X

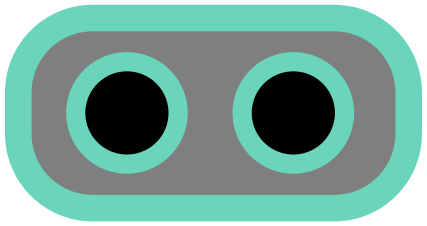


Array of Pointers ➔ Arrays of Arrays

```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```





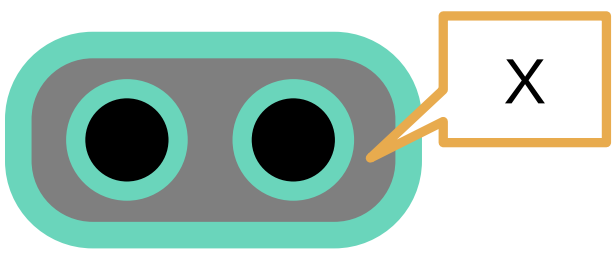


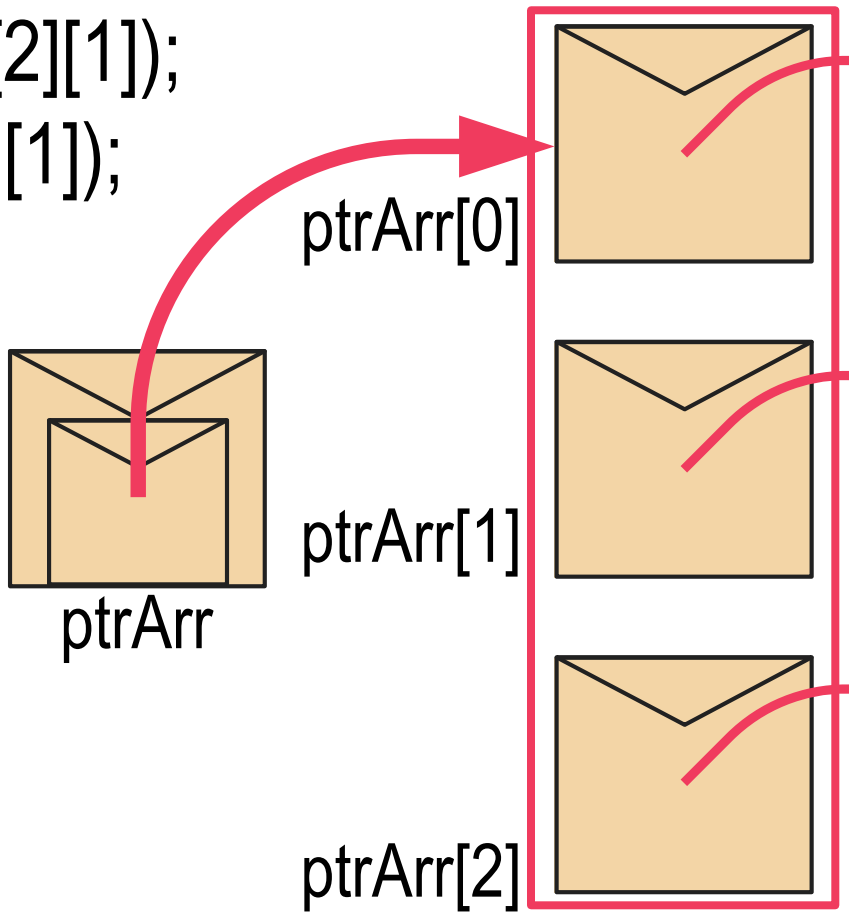
Array of Pointers ➔ Arrays of Arrays


```
char **ptrArr = (char**)malloc(3*sizeof(char*));
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
scanf("%c", &ptrArr[2][1]);
printf("%c", ptrArr[2][1]);
for(i = 0; i < 3; i++)
        free(ptrArr[i]);
free(ptrArr);
```

Console | Activity Log | Input | Output

X





Array of Pointers/Arrays

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

ptrArr[0], ptrArr[1], ptrArr[2] are all arrays of chars

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

ptrArr[0], ptrArr[1], ptrArr[2] are all arrays of chars

How to access individual elements of these arrays?

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

ptrArr[0], ptrArr[1], ptrArr[2] are all arrays of chars

How to access individual elements of these arrays?

- Two ways to access index 2 element of str: str[2], *(str+2)

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

ptrArr[0], ptrArr[1], ptrArr[2] are all arrays of chars

How to access individual elements of these arrays?

Two ways to access index 2 element of str: str[2], *(str+2)

Apply exact same rule ☺: ptrArr[2][2], *(ptrArr[2]+2) both give index 2 element of the array ptrArr[2]

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

ptrArr[0], ptrArr[1], ptrArr[2] are all arrays of chars

How to access individual elements of these arrays?

Two ways to access index 2 element of str: str[2], *(str+2)

Apply exact same rule ☺: ptrArr[2][2], *(ptrArr[2]+2) both give index 2 element of the array ptrArr[2]

Note that ptrArr[1] does not have 3 elements so ptrArr[1][2] may cause segfault!

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

- In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

- ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

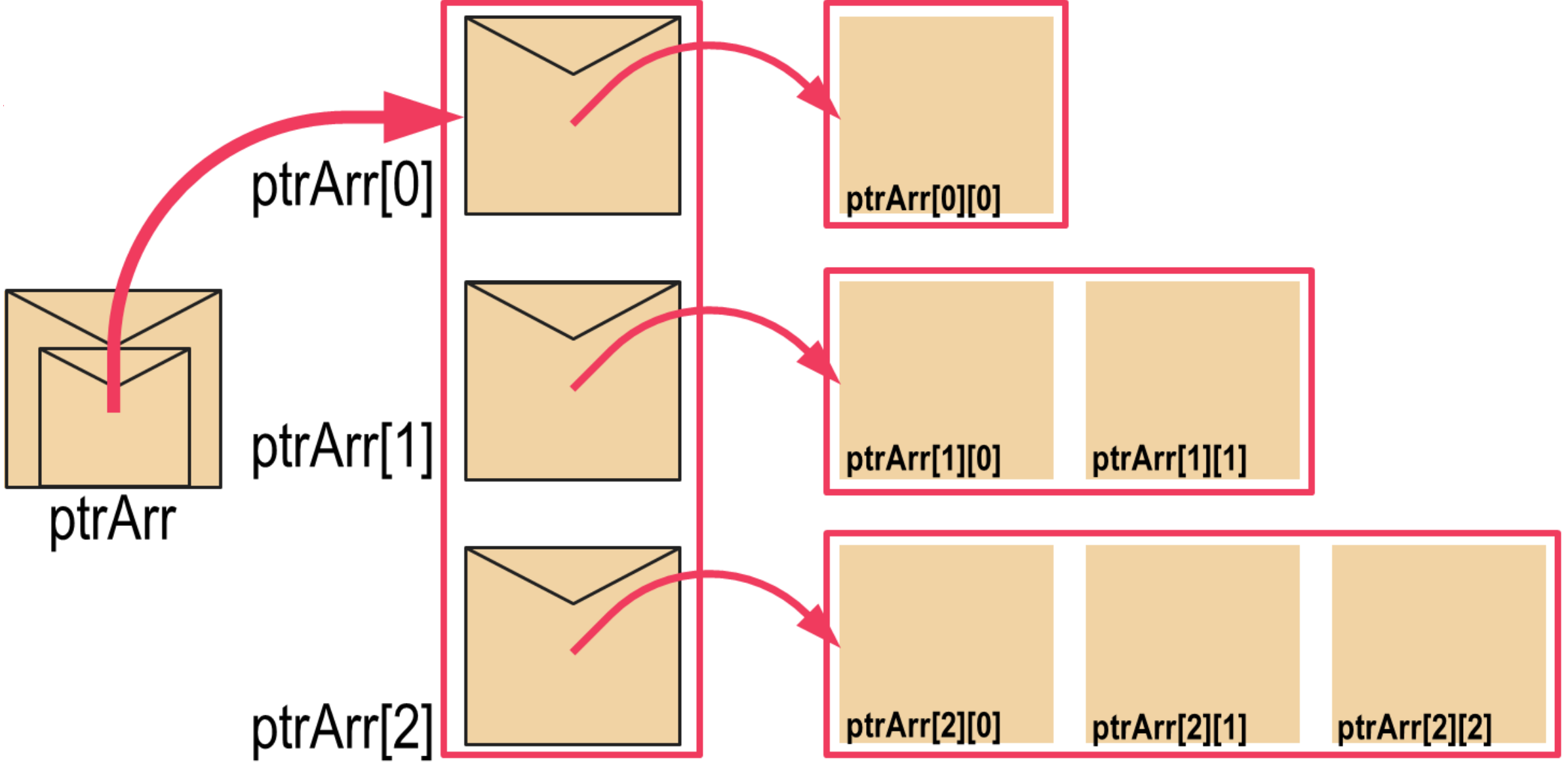
- In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

- ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

We can access index 2 of the third array in many ways

ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]



We can access index 2 of the third array in many ways
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/Arrays

Rest assured, the same rules apply as do with pointers

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

- In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

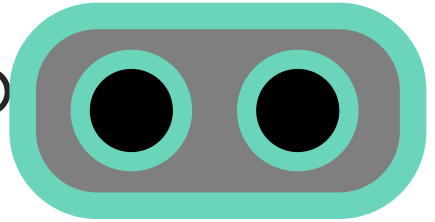
- ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

We can access index 2 of the third array in many ways

ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/Arrays

Rest assured, the same rules apply as do with po



```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

- In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

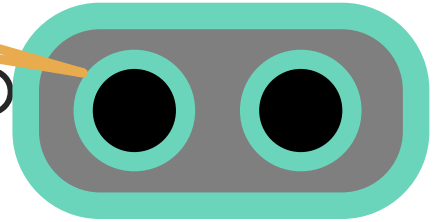
str + 2 gives address of str[2]

- ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

We can access index 2 of the third array in many ways
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!



Rest assured, the same rules apply as do with po

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access tricks ☺

Remember that str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

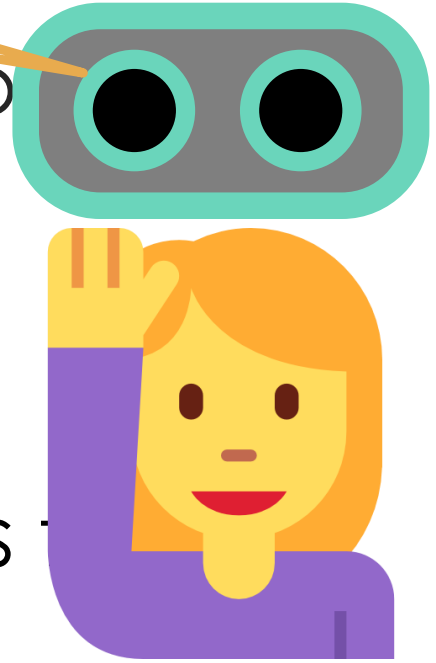
str + 2 gives address of str[2]

ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

We can access index 2 of the third array in many ways
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!



Rest assured, the same rules apply as do with po

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access

Remember that str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

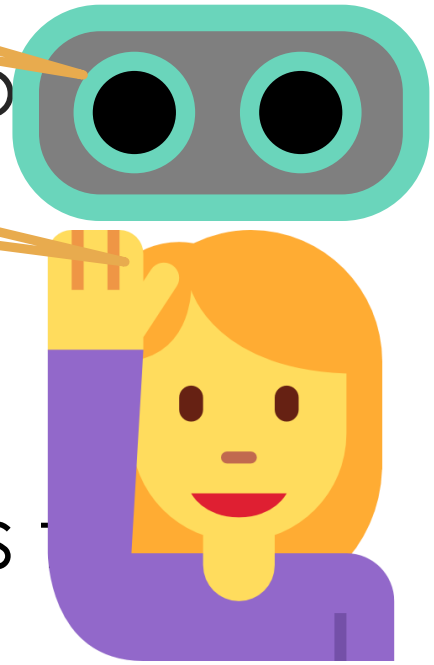
ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

We can access index 2 of the third array in many ways
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]



Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access

Remember that str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

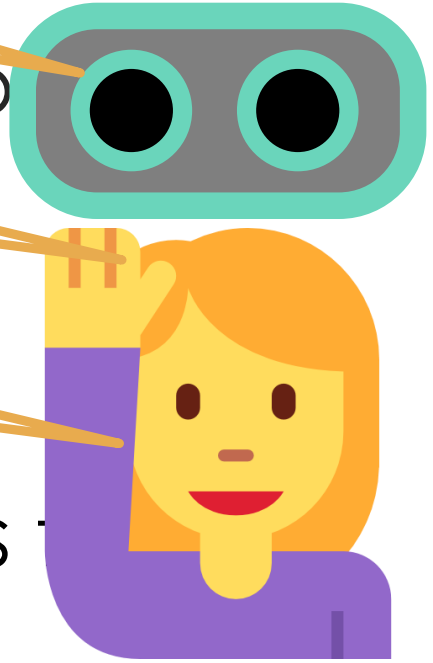
We can access index 2 of the third array in many ways
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]

I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]



Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access

Remember that str is a pointer to str[0]

- In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

- ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – same rules!

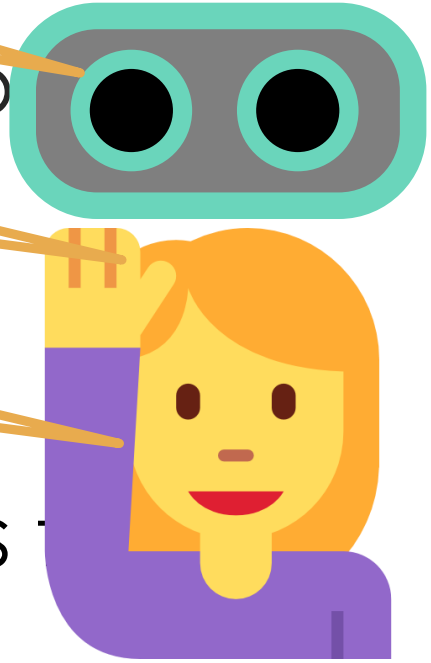
We can access index 2 of the third array in many ways
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrArr+2))[2]

Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]

I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]



Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show-off your skills by cool array access

Remember that str is a pointer to str[0]

- In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives address of str[2]

- ptrArr + 2 also gives address of ptrArr[2] (pointers take 8 bytes) – s s!

We can access index 2 of the third array in many w

ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrA)[2]

Array of Pointers/

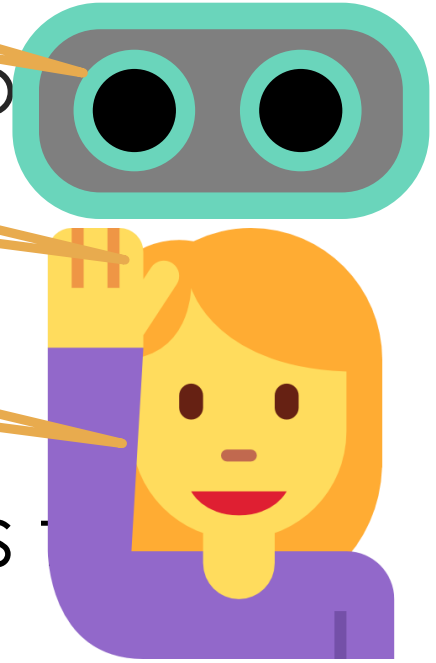
Don't write ptrArr++ illegal!
Even str++ illegal!

Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]

I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]



You can show-off your skills by cool array access

Remember that str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives

ptrArr + 2 also

Just one potentially confusing notation in C
int *ptr[5]; is an array of 5 pointers to int but
int (*ptr)[5] is a single pointer to an array of 5 ints ☹

We can access index 2 of the third array in many w

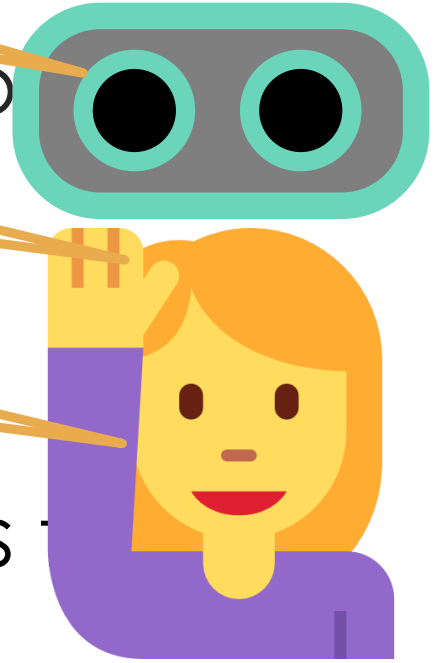
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrA

Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]

I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]



Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show your skills by cool array access

Remember, str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives

ptrArr + 2 also g

Just one potentially confusing notation in C
int *ptr[5]; is an array of 5 pointers to int but
int (*ptr)[5] is a single pointer to an array of 5 ints ☹

We can access index 2 of the third array in many w

ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrA

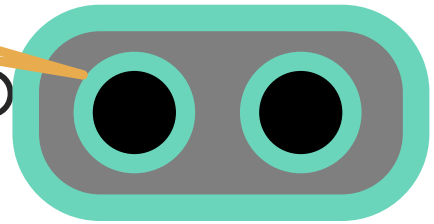
Array of Pointers/

Don't write ptrArr++ illegal!
Even str++ illegal!

Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]



I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]

You can show

Don't worry. I won't ask exam questions on int (*ptr)[5];

access

Remember, str is a pointer to str[0]

In the same way, ptrArr is also a pointer to ptrArr[0] (which is an array)

str + 2 gives

ptrArr + 2 also

Just one potentially confusing notation in C
int *ptr[5]; is an array of 5 pointers to int but
int (*ptr)[5] is a single pointer to an array of 5 ints ☹

We can access index 2 of the third array in many w

ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrA

Array of Pointers/

Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show

access

Remember

In the same way,

an array)

str + 2 gives

ptrArr + 2 also

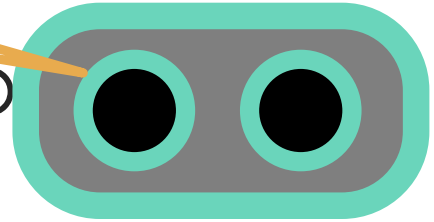
s!

We can access index 2 of the third array in many w

ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrArr + 2) + 2),(*(ptrA

Don't write ptrArr++ illegal!
Even str++ illegal!

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]



I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]

Don't worry. I won't ask exam questions on int (*ptr)[5];

I will ask questions on pointers to pointers, array of pointers, etc though

Just one potentially confusing notation in C
int *ptr[5]; is an array of 5 pointers to int but
int (*ptr)[5] is a single pointer to an array of 5 ints ☹

Array of Pointers/

Rest assured, the same

```
char *ptrArr[3], str[3];
for(i = 0; i < 3; i++)
    ptrArr[i] = (char*)malloc((i+1)*sizeof(char));
```

You can show

access

Remember

In the same way,

an array)

str + 2 gives

ptrArr + 2 also

s!

We can access index 2 of the th

any w

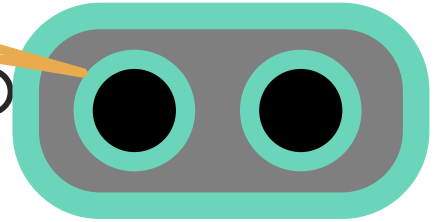
ptrArr[2][2],*(ptrArr[2] + 2),*(*(ptrA

),(*(ptrA

)[2]

Don't write ptrArr++ illegal!
Even str++ illegal!

However, I can write char* qtr = str;
qtr++; Now qtr points to str[1]



I can also write char** rtr = ptrArr;
rtr++; Now rtr points to ptrArr[1]

Don't worry. I won't ask exam questions on int (*ptr)[5];

I will ask questions on pointers to pointers, array of pointers, etc though

Just one potentially confusing notation in C
int *ptr[5]; is an array of 5 pointers to int but
int (*ptr)[5] is a single pointer to an array of 5 ints ☹

Your avatar befits you ☺

2D Arrays in C

2D Arrays in C


```
int mat[3][5];
```

2D Arrays in C


```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

2D Arrays in C


```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

2D Arrays in C


```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

2D Arrays in C


```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways

```
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]
```

2D Arrays in C


```
int mat[3][5];
```

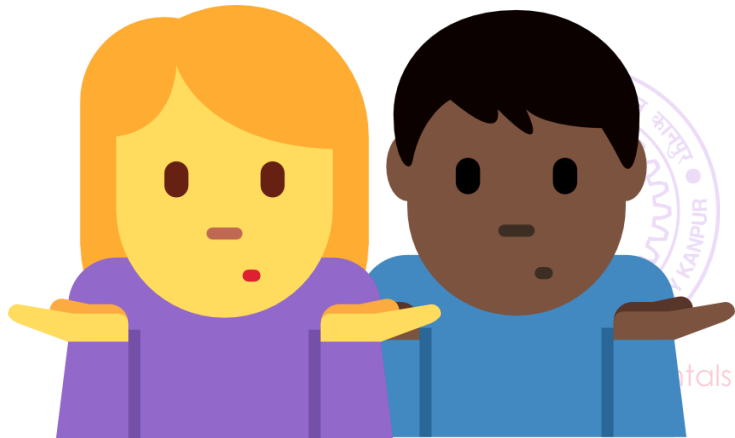
Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways

```
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]
```



2D Arrays in C


```
int mat[3][5];
```

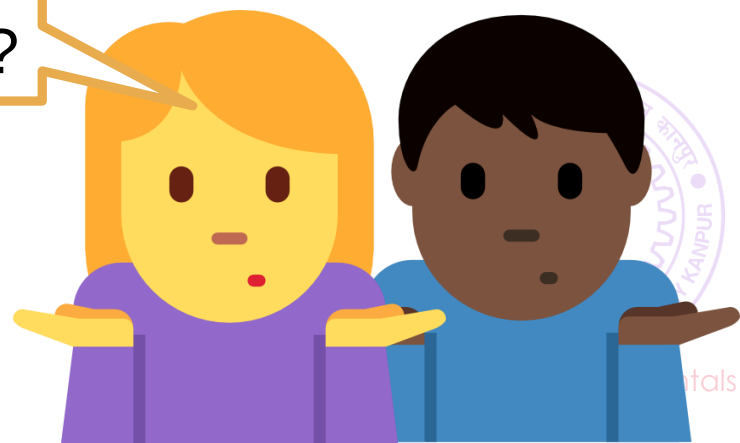
Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

This looks exactly like the way we access an array of arrays – what is the difference?



2D Arrays in C


```
int mat[3][5];
```

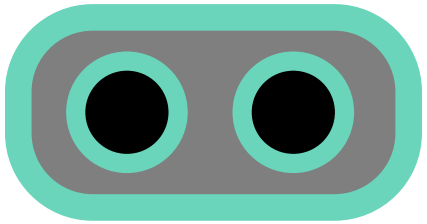
Declares a matrix (2D array) with 3 rows 5 columns

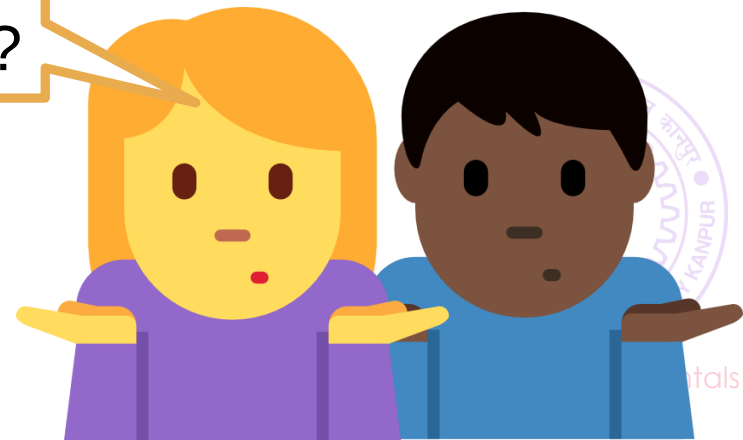
Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]

This looks exactly like the way we access an array of arrays – what is the difference?





2D Arrays in C

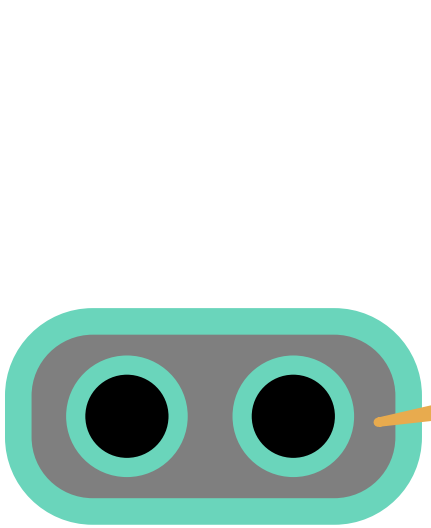

```
int mat[3][5];
```

Declares a matrix (2D array) with 3 rows 5 columns

Rows numbered 0, 1, 2. Columns numbered 0, 1, 2, 3, 4

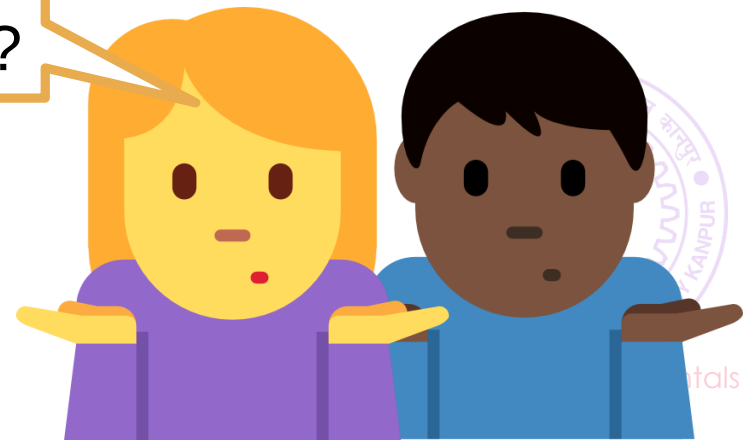
Element at row-index i and column-index j is an int variable

Can access it using several ways
mat[i][j],*(mat[i] + j),*(*(mat + i) + j),(*(mat + i))[j]



Not that much actually – let me show you the differences

This looks exactly like the way we access an array of arrays – what is the difference?



2D arrays vs Array of arrays

2D ARRAYS

ARRAY OF ARRAYS

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

ARRAY OF ARRAYS

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

2D arrays vs Array of arrays

2D ARRAYS

- Number of elements in each row is the same
- All elements of 2D array are located contiguously in memory
- Easier to initialize

ARRAY OF ARRAYS

- Different arrays can have different number of elements – more flexibility
- Elements of a single array are contiguous but different arrays could be located far off in memory

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

Have to be initialized element by element

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

Very convenient ☺

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

Elements of a single array are contiguous but different arrays could be located far off in memory

Have to be initialized element by element

2D arrays vs Array of arrays

2D ARRAYS

Number of elements in each row is the same

All elements of 2D array are located contiguously in memory

Easier to initialize

```
int mat[3][5] = { {1,2}, {3},
{4,5,6},{7,8,9,10,11},{-1,2,3,4}};
```

Very convenient ☺

ARRAY OF ARRAYS

Different arrays can have different number of elements – more flexibility

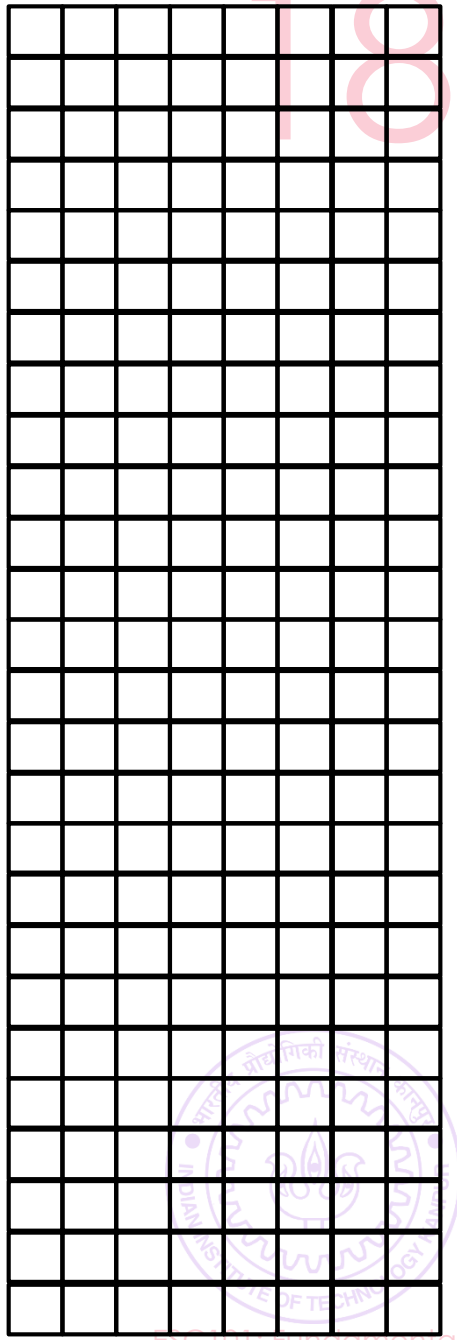
Elements of a single array are contiguous but different arrays could be located far off in memory

Have to be initialized element by element

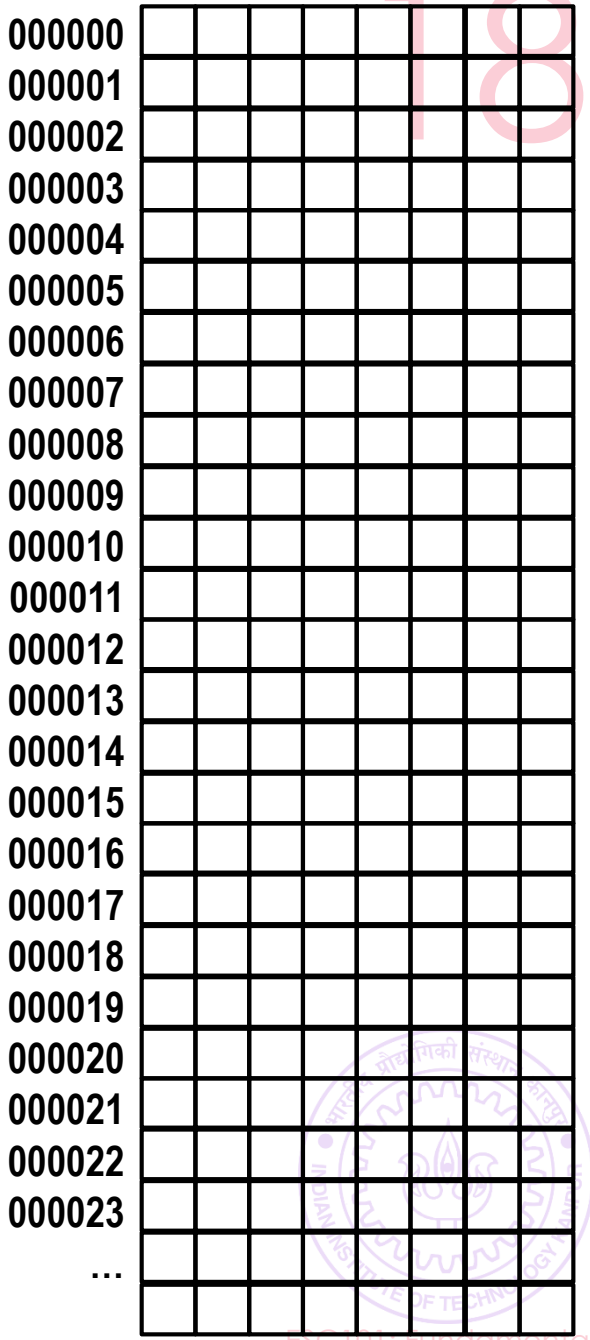
More power, responsibility

Memory layout of 2D arrays

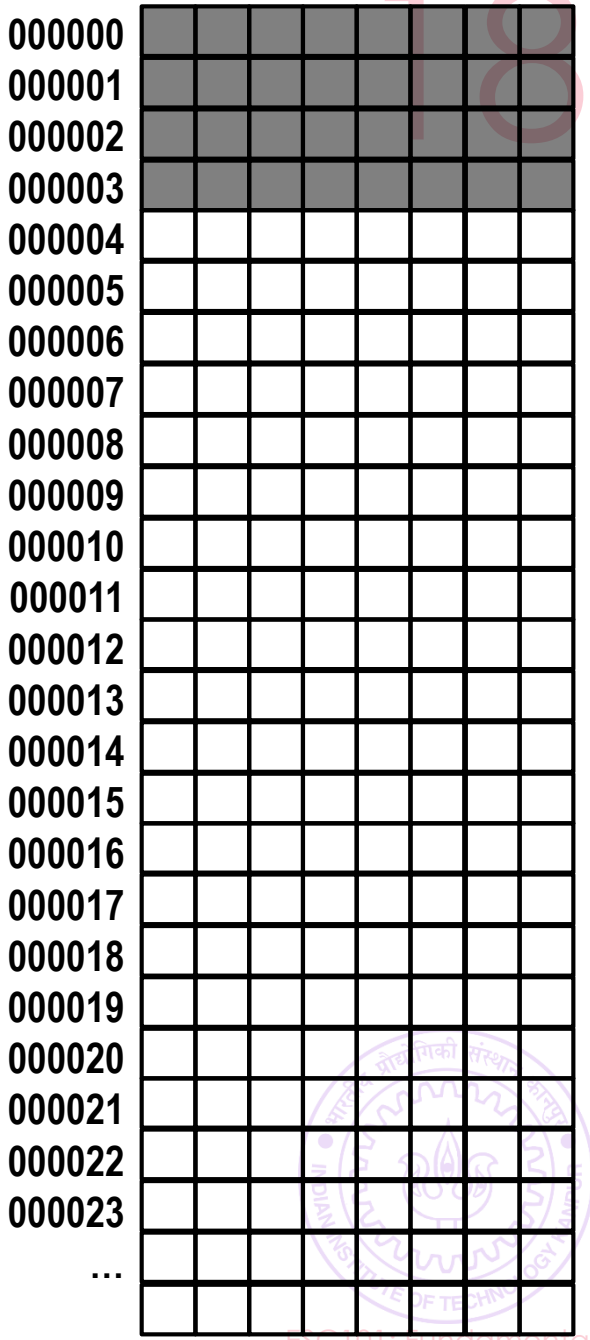
Memory layout of 2D arrays



Memory layout of 2D arrays

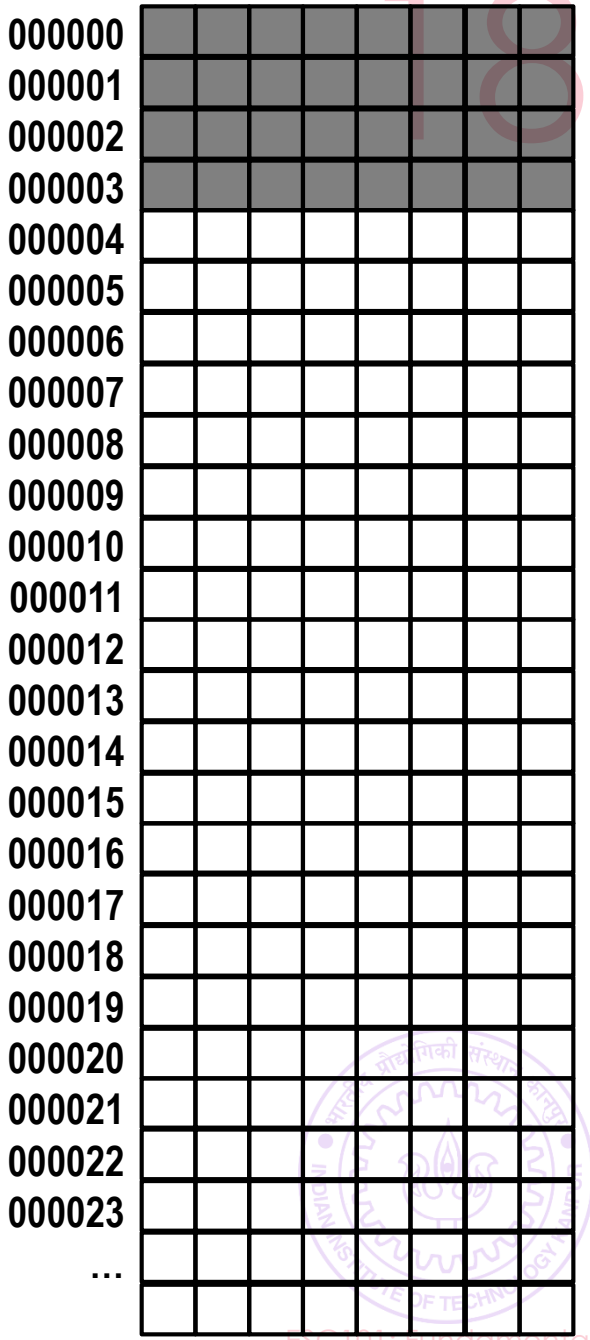


Memory layout of 2D arrays



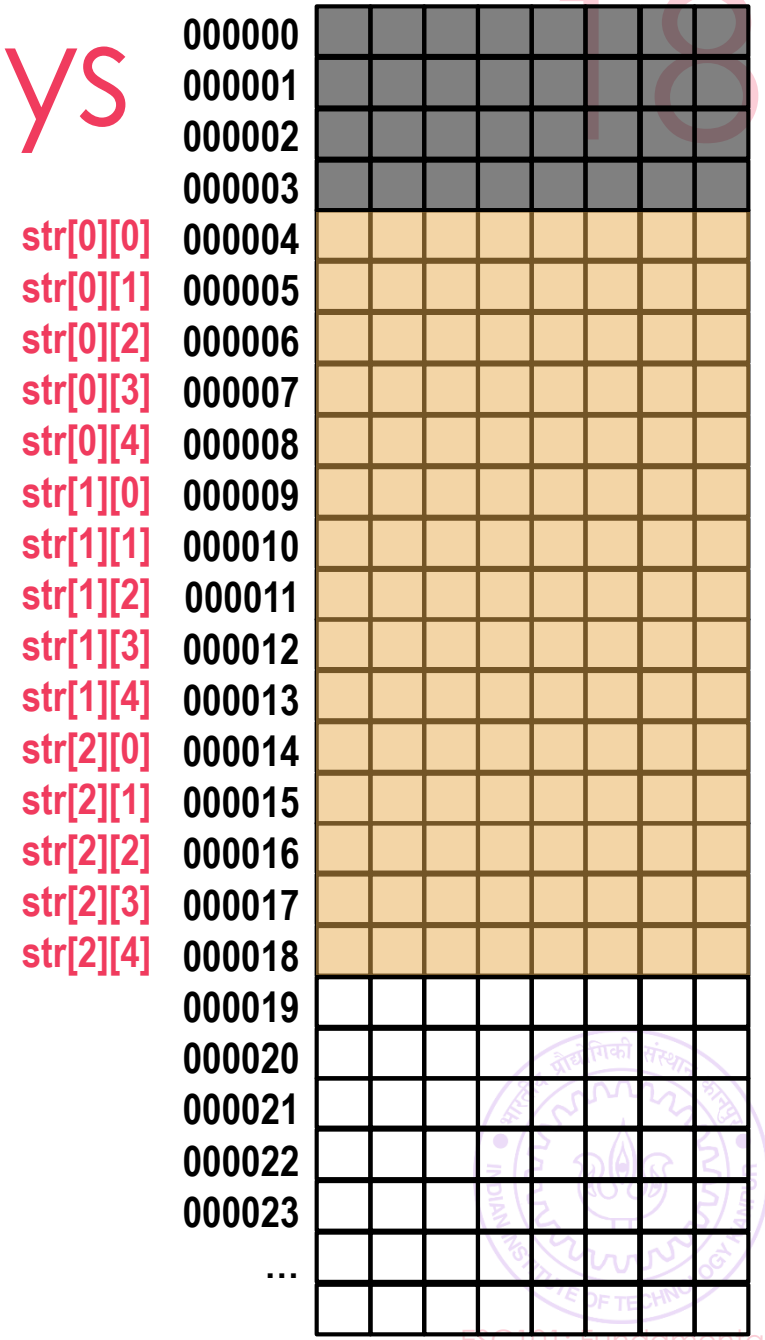
Memory layout of 2D arrays

char str[3][5];



Memory layout of 2D arrays

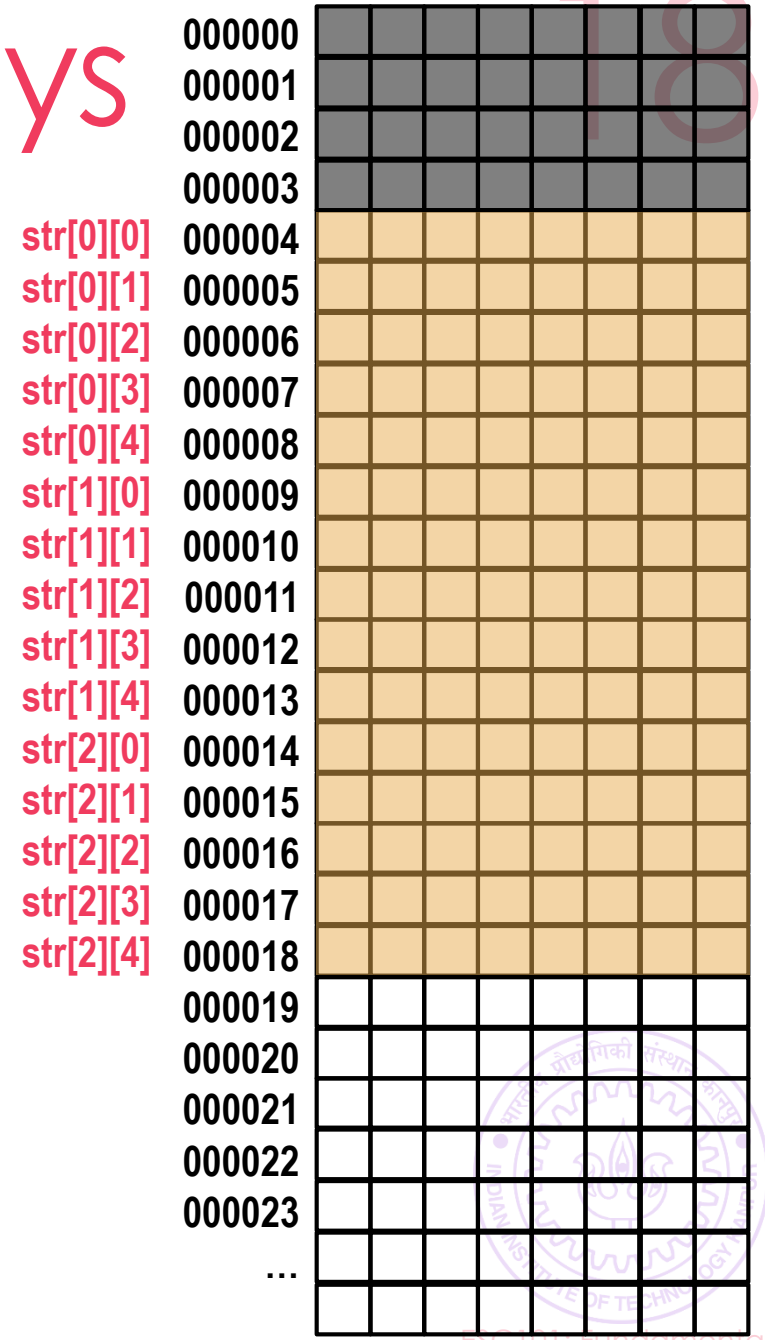
char str[3][5];



Memory layout of 2D arrays

char str[3][5];

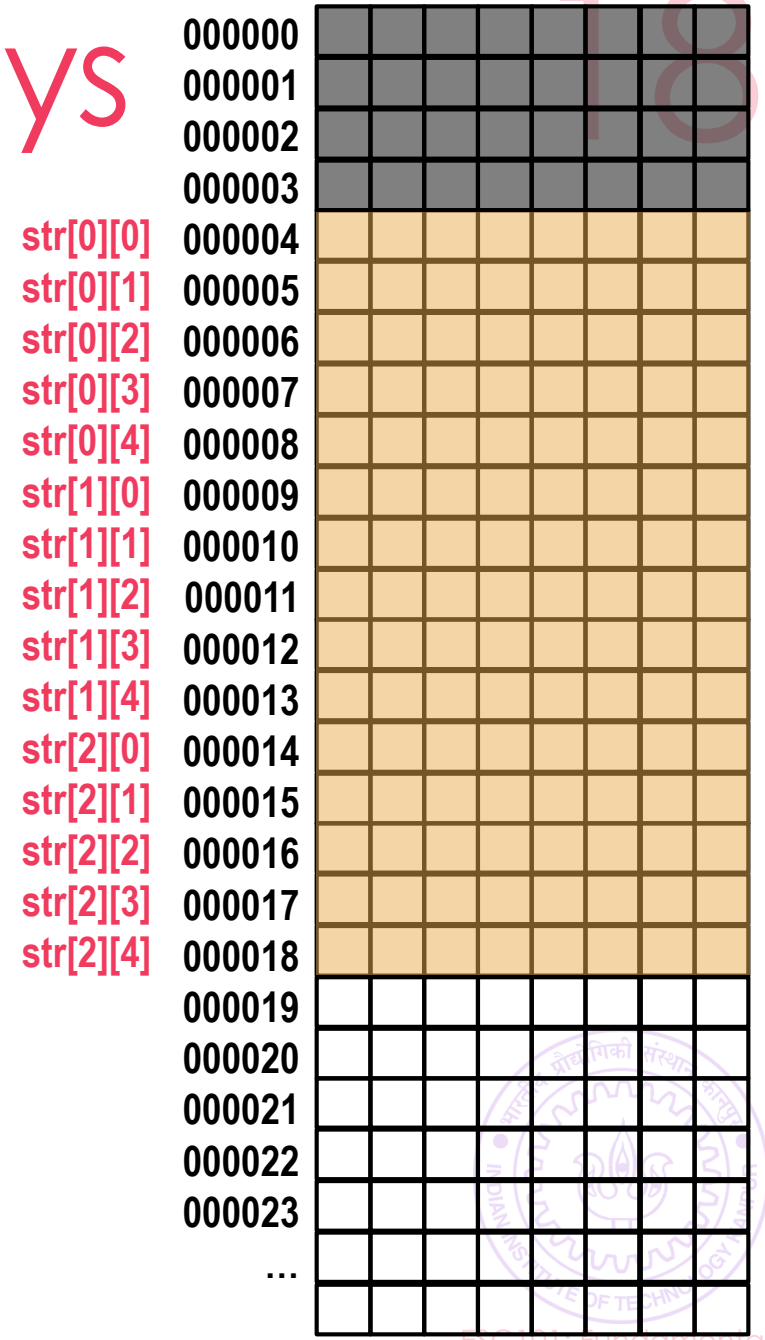
Location of the str pointer not shown



Memory layout of 2D arrays

char str[3][5];

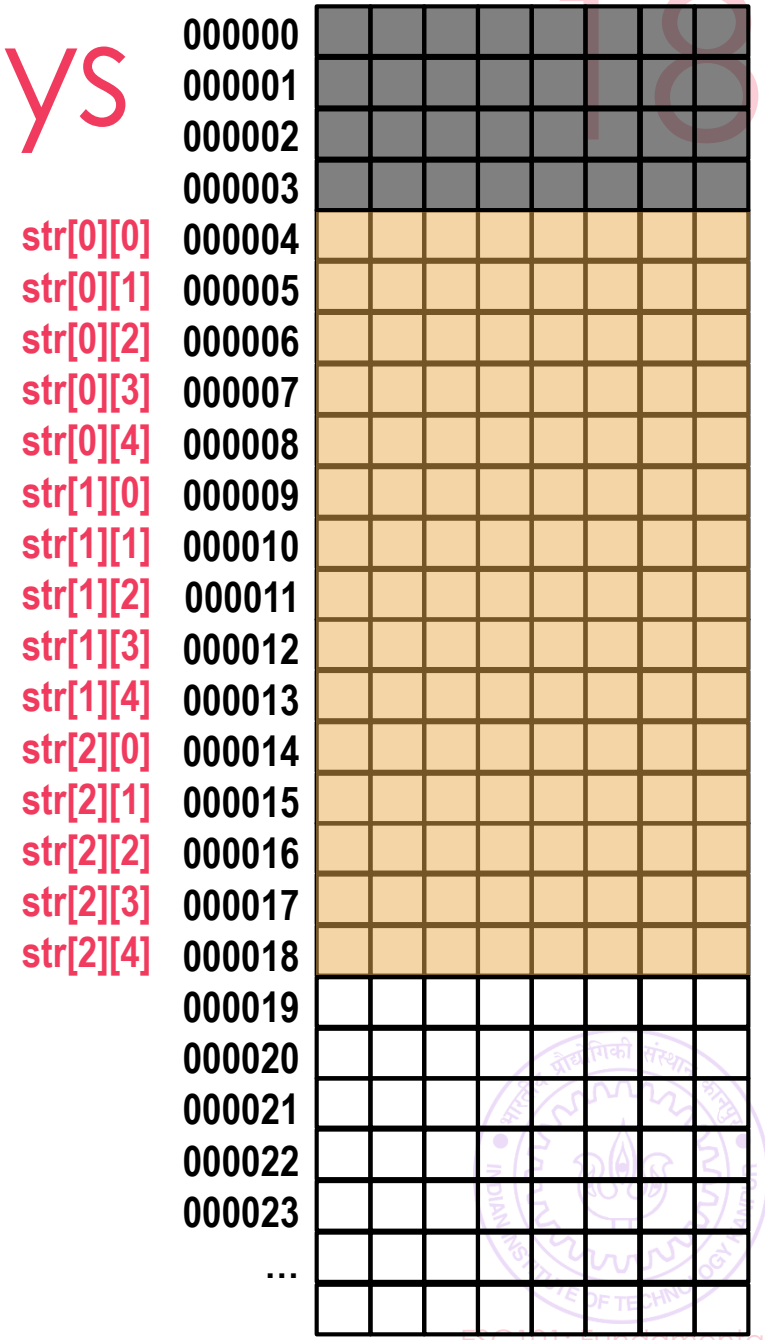
- Location of the str pointer not shown
- First all elements of row 0 stored in continuous sequence



Memory layout of 2D arrays

char str[3][5];

- Location of the str pointer not shown
- First all elements of row 0 stored in continuous sequence
- Then without breaking sequence, all elements of row 1 stored and so on



Memory layout of 2D arrays


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

- Location of the str pointer not shown
- First all elements of row 0 stored in continuous sequence
- Then without breaking sequence, all elements of row 1 stored and so on

```
char* ptr = *str; // ptr points to str[0][0]
ptr += 5; // ptr now points to str[1][0]
ptr += 5; // ptr now points to str[2][0]
ptr += 3; // ptr now points to str[2][3]
```