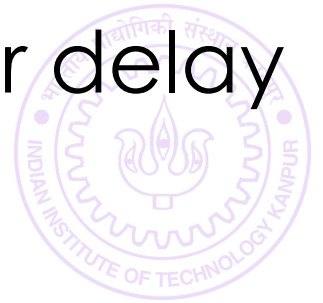
On-demand Service from Mr C

ESC101: Fundamentals of Computing

Purushottam Kar

Announcements

- October 2 (Tuesday) is a holiday – no lecture, no lab
- We will have 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.
- Will release remaining grades this week – sorry for delay



Pointers - Recap

Pointers - Recap

Pointers are special variables that store addresses

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

```
int a, b, *x, *y;
x = &a, y = &b;
```

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

```
int a, b, *x, *y;
x = &a, y = &b;
```

Pointer arithmetic: depends on the type of the pointer

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

```
int a, b, *x, *y;
x = &a, y = &b;
```

Pointer arithmetic: depends on the type of the pointer

- Pointers to int advance by 4 upon ++, decrease by 4 upon --

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

```
int a, b, *x, *y;
x = &a, y = &b;
```

Pointer arithmetic: depends on the type of the pointer

- Pointers to int advance by 4 upon ++, decrease by 4 upon --
- Pointers to char advance by 1 upon ++, decrease by 1 upon --

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

```
int a, b, *x, *y;
x = &a, y = &b;
```

Pointer arithmetic: depends on the type of the pointer

- Pointers to int advance by 4 upon ++, decrease by 4 upon --
- Pointers to char advance by 1 upon ++, decrease by 1 upon --
- Pointers to double advance by 8 upon ++, decrease by 8 upon –

Pointers - Recap

Pointers are special variables that store addresses

Some functions like scanf, string functions accept pointers

Can declare pointer and regular variables on same line

```
int a, b, *x, *y;
x = &a, y = &b;
```

Pointer arithmetic: depends on the type of the pointer

- Pointers to int advance by 4 upon ++, decrease by 4 upon --
- Pointers to char advance by 1 upon ++, decrease by 1 upon --
- Pointers to double advance by 8 upon ++, decrease by 8 upon –

Can typecast pointers too – will see a cute example

Pointers and Arrays

Pointers and Arrays

Array names are pointers to first element of the array

Pointers and Arrays

Array names are pointers to first element of the array

Warning: consecutive addresses only assured in arrays

Pointers and Arrays

Array names are pointers to first element of the array

Warning: consecutive addresses only assured in arrays

```
int c[10]
int a, b, ptr = c;
```

Pointers and Arrays

Array names are pointers to first element of the array

Warning: consecutive addresses only assured in arrays

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart) but c[0], c[1] will always be 4 bytes apart (int takes 4 bytes)

Pointers and Arrays

Array names are pointers to first element of the array

Warning: consecutive addresses only assured in arrays

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart) but c[0], c[1] will always be 4 bytes apart (int takes 4 bytes)

Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

Pointers and Arrays

Array names are pointers to first element of the array

Warning: consecutive addresses only assured in arrays

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart) but c[0], c[1] will always be 4 bytes apart (int takes 4 bytes)

Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

`c[2]` and `*(c+2)` both give value of the 3rd element in c

Pointers and Arrays

Array names are pointers to first element of the array

Warning: consecutive addresses only assured in arrays

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart) but c[0], c[1] will always be 4 bytes apart (int takes 4 bytes)

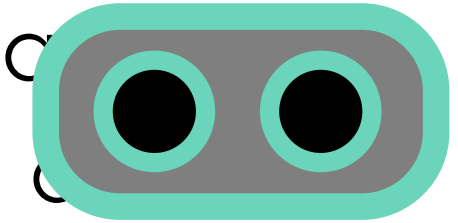
Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

`c[2]` and `*(c+2)` both give value of the 3rd element in c

Warning: c++ will give error, ptr++ will move pointer to c[1]

Pointers and Arrays

Array names are pointers to first element of the a



Warning: consecutive addresses only assured in a

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart) but c[0], c[1] will always be 4 bytes apart (int takes 4 bytes)

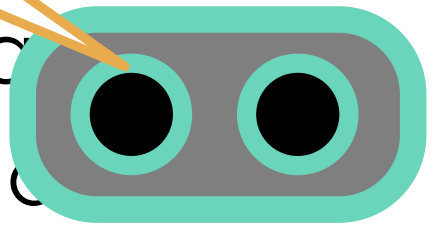
Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

`c[2]` and `*(c+2)` both give value of the 3rd element in c

Warning: c++ will give error, ptr++ will move pointer to c[1]

Pointers and Ar

The array name will always point to the first element of the array. Cannot change that!



Array names are pointers to first element of the a

Warning: consecutive addresses only assured in a

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart) but c[0], c[1] will always be 4 bytes apart (int takes 4 bytes)

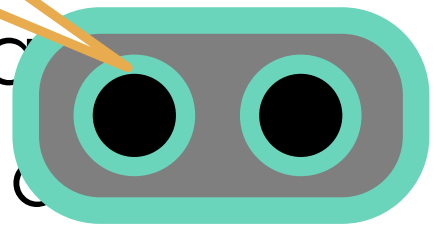
Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

`c[2]` and `*(c+2)` both give value of the 3rd element in c

Warning: c++ will give error, ptr++ will move pointer to c[1]

Pointers and Ar

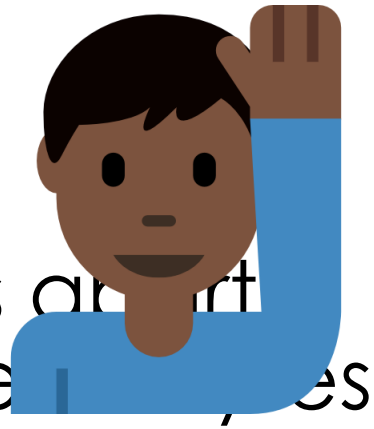
The array name will always point to the first element of the array. Cannot change that!



Array names are pointers to first element of the a

Warning: consecutive addresses only assured in a

```
int c[10]
int a, b, ptr = c;
```



a, b need not be placed side-by-side (i.e. 4 bytes apart but c[0], c[1] will always be 4 bytes apart (int take es)

Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

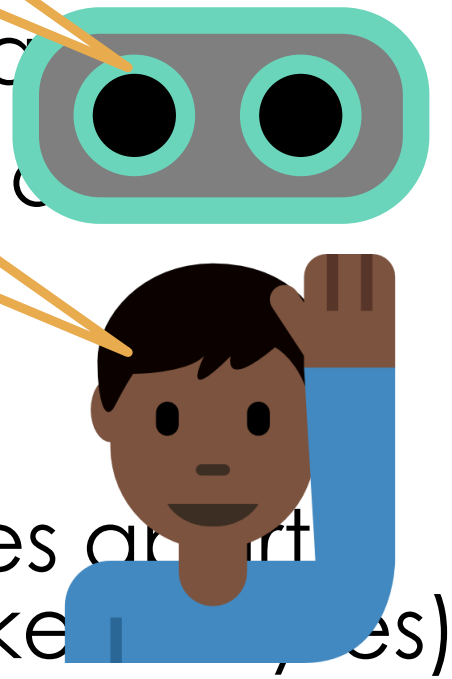
`c[2]` and `*(c+2)` both give value of the 3rd element in c

Warning: c++ will give error, ptr++ will move pointer to c[1]

Pointers and Ar

The array name will always point to the first element of the array. Cannot change that!

To do fancy pointer arithmetic, we should create a fresh pointer variable e.g. ptr



Array names are pointers to first element of the a

Warning: consecutive a in c

```
int c[10]
int a, b, ptr = c;
```

a, b need not be placed side-by-side (i.e. 4 bytes apart but c[0], c[1] will always be 4 bytes apart (int take es)

Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

`c[2]` and `*(c+2)` both give value of the 3rd element in c

Warning: c++ will give error, ptr++ will move pointer to c[1]

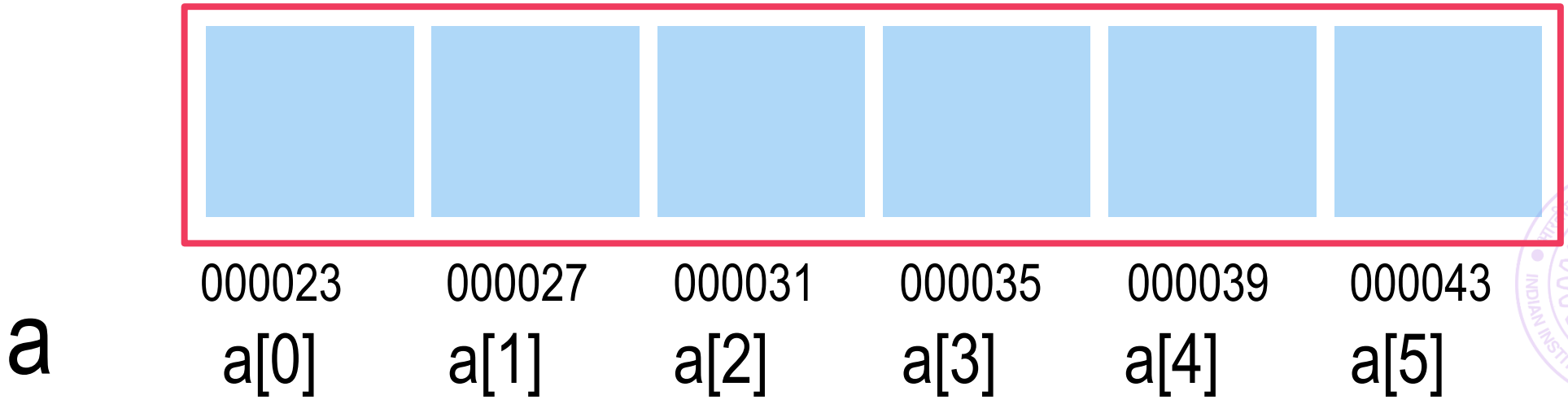
Pointers and Arrays

Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
```

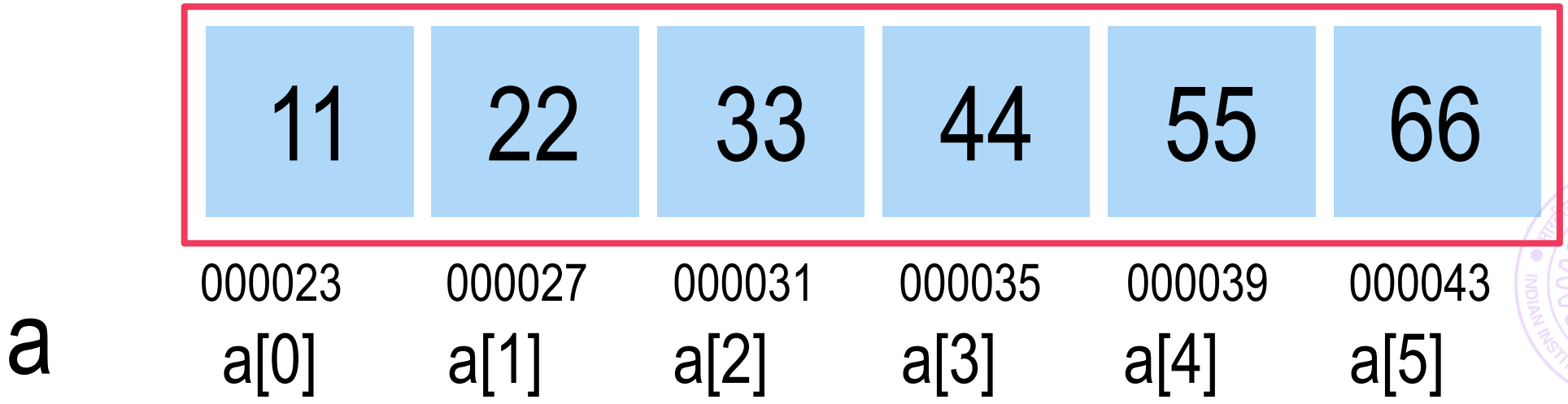
Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
```



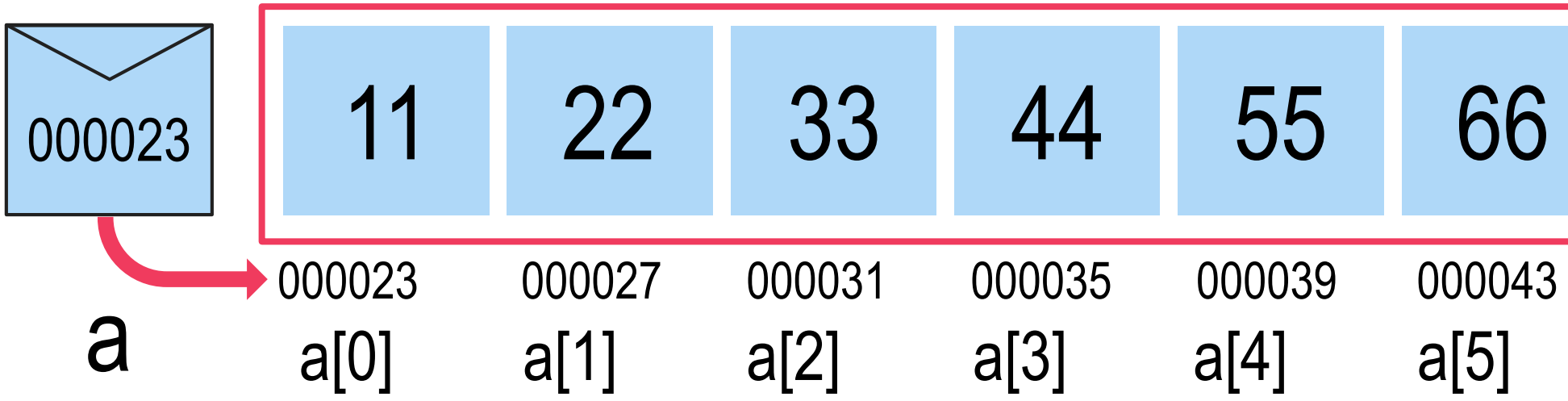
Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
```



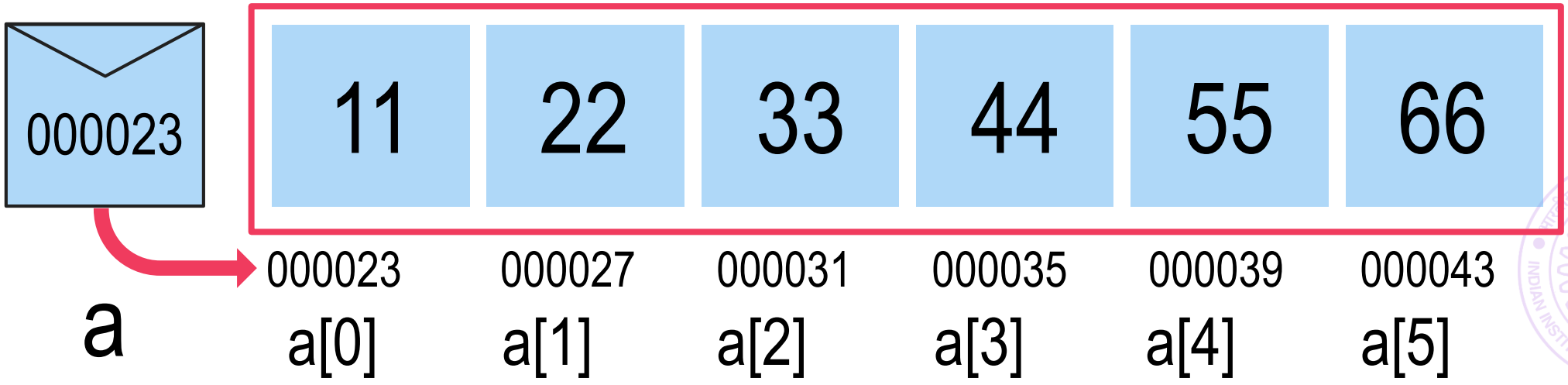
Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
```



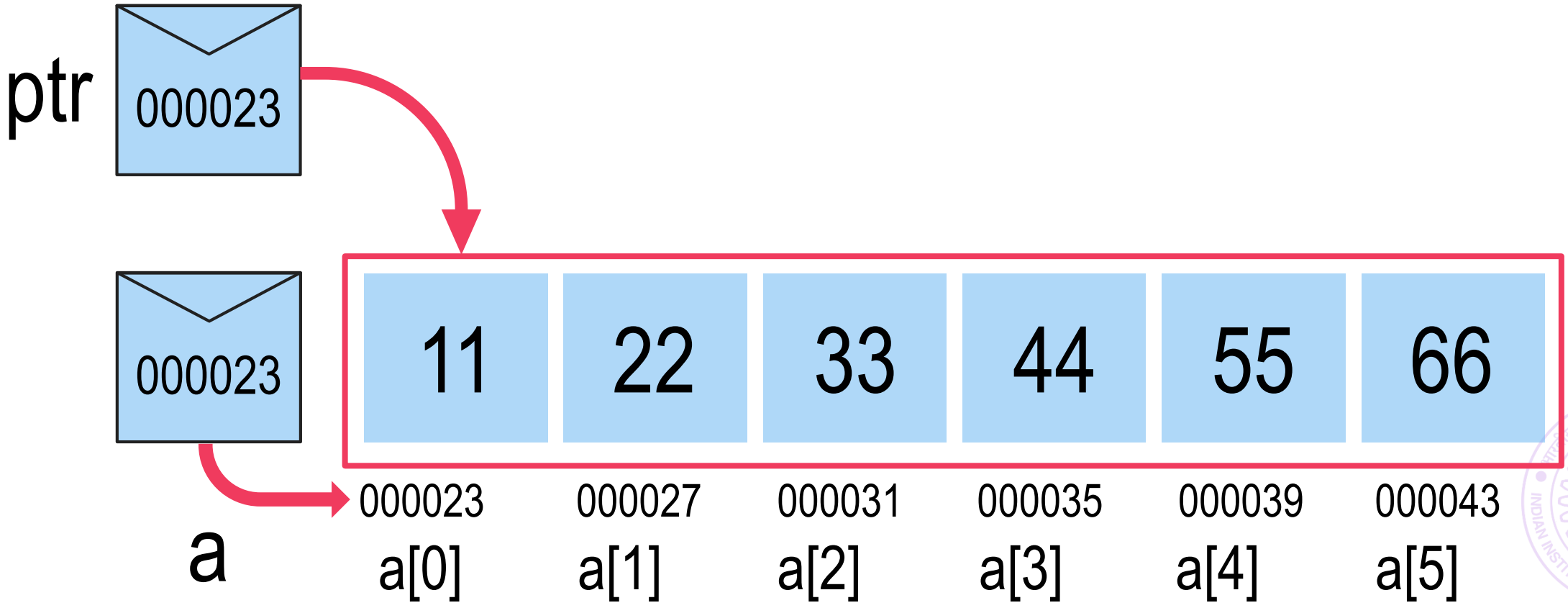
Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```



Pointers and Arrays

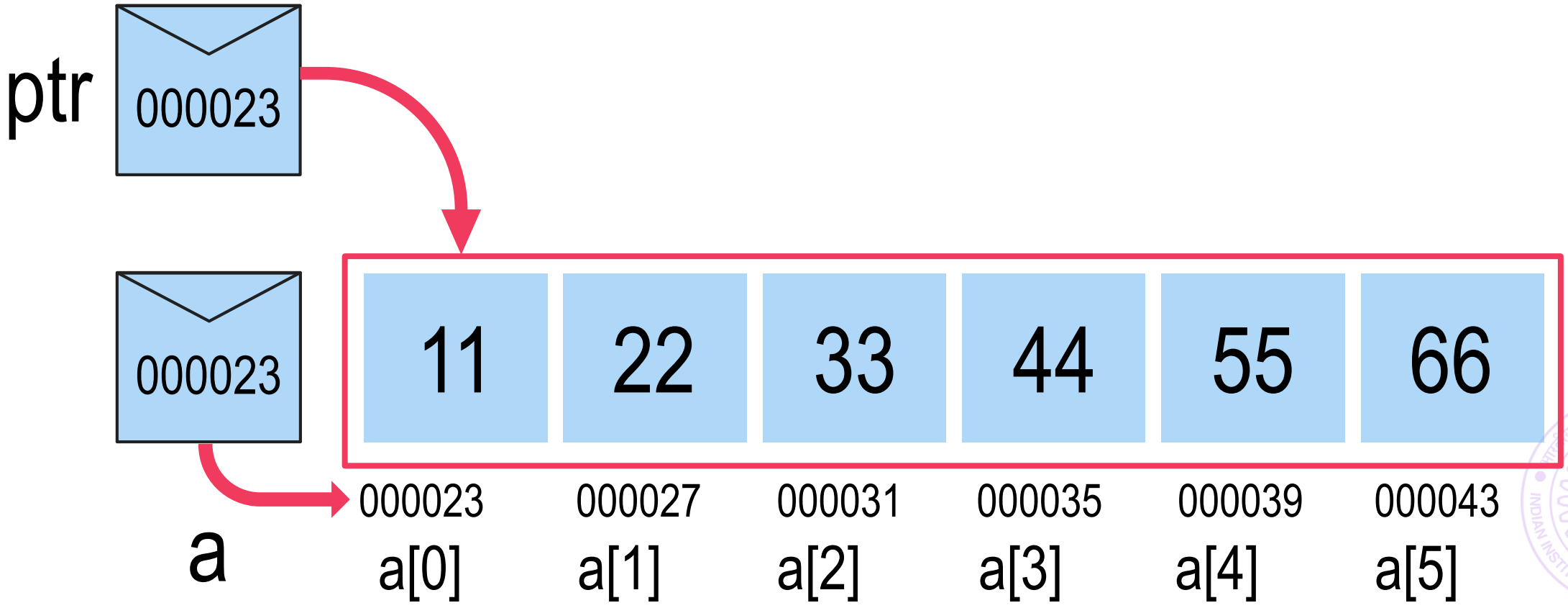

```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```



Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

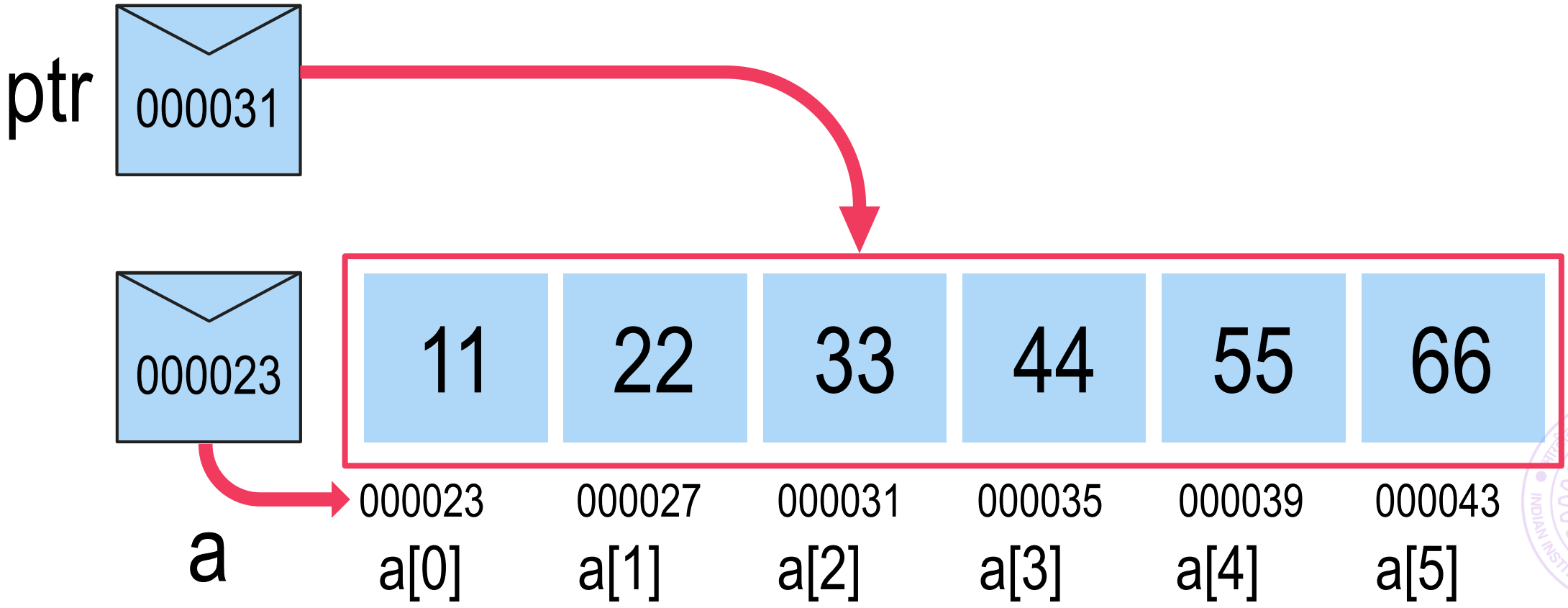
```
ptr += 2;
```



Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

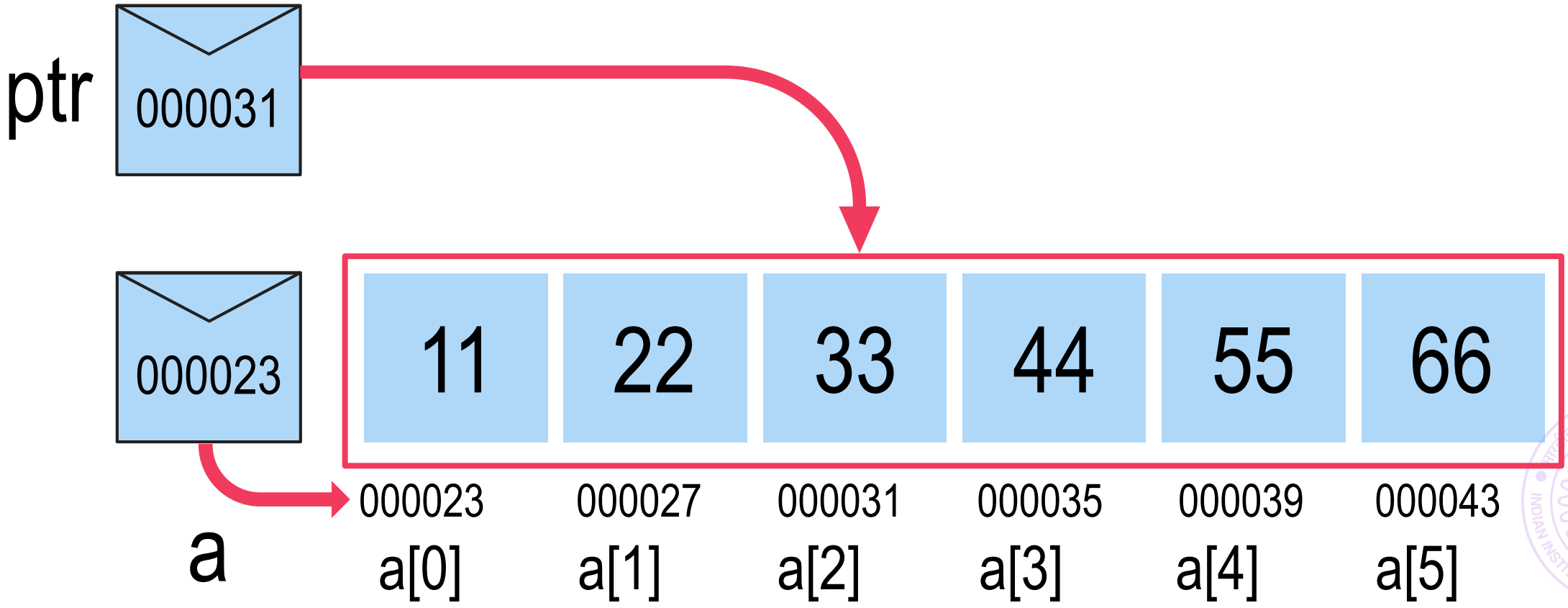
```
ptr += 2;
```



Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

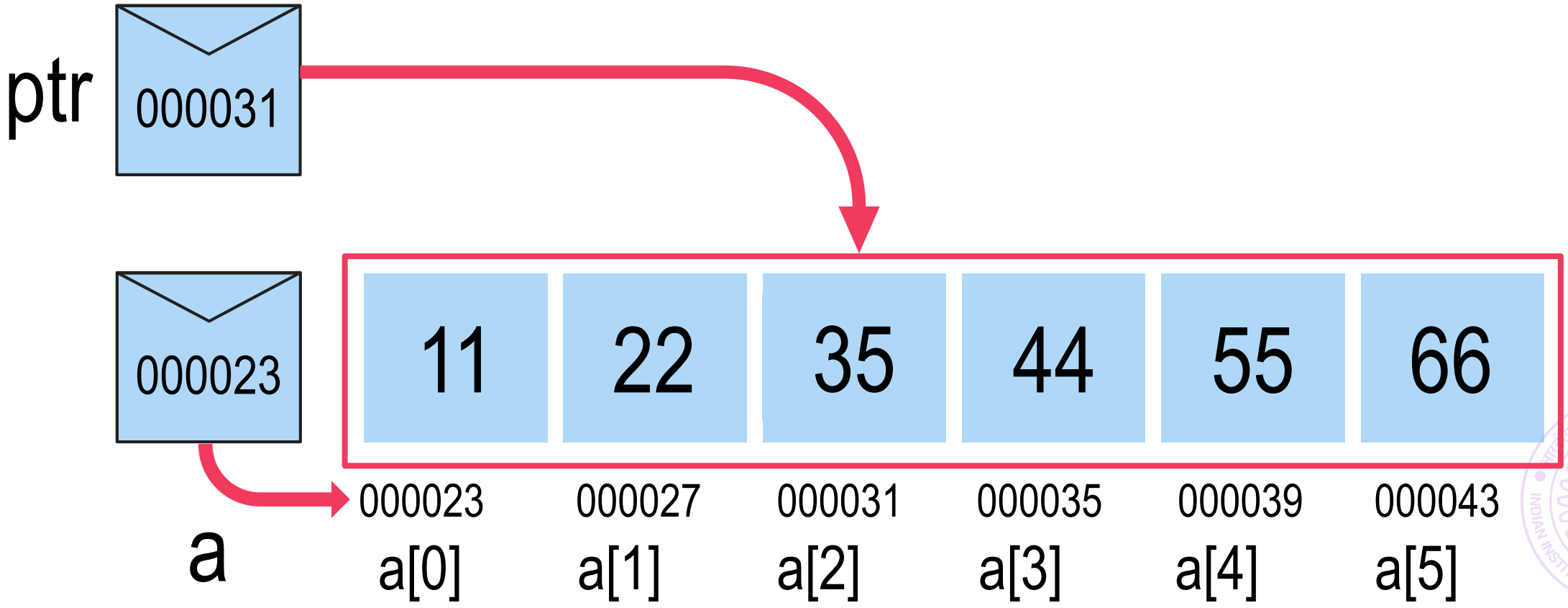
```
ptr += 2;
*ptr += 2;
```



Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

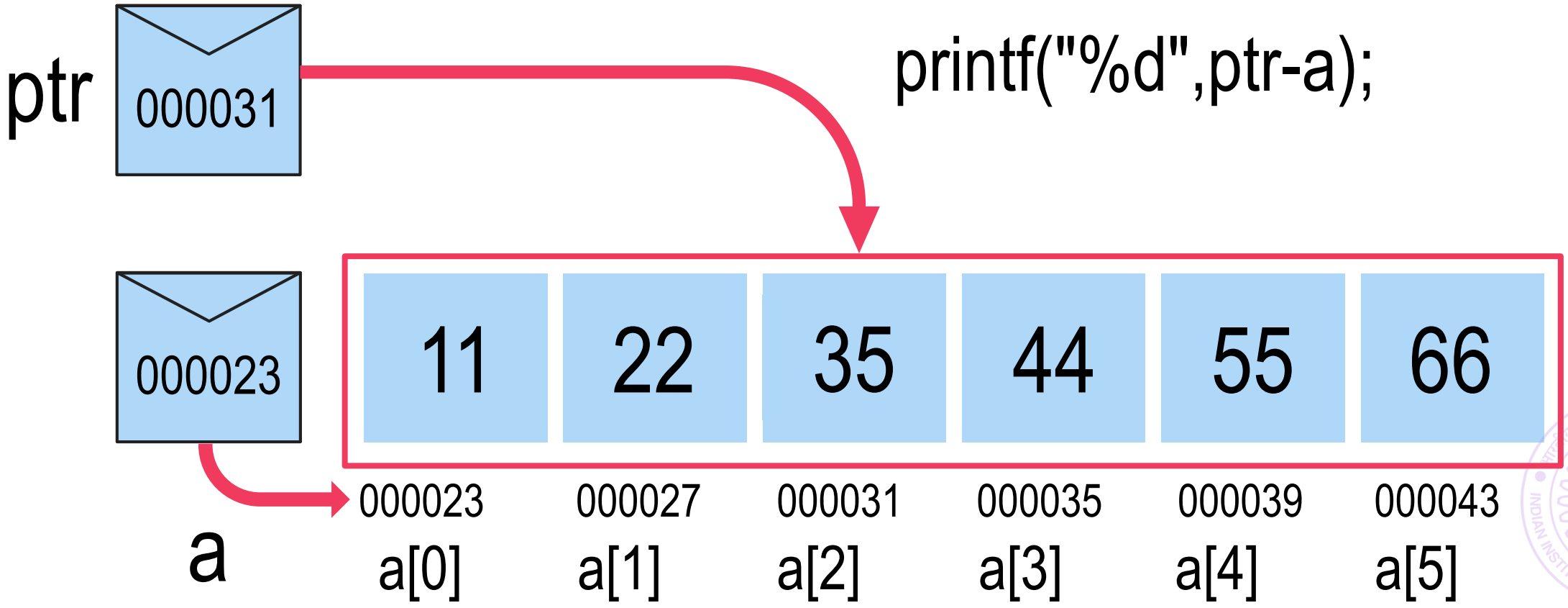
```
ptr += 2;
*ptr += 2;
```



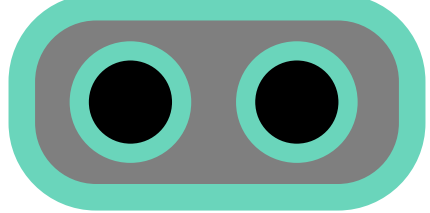
Pointers and Arrays


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```

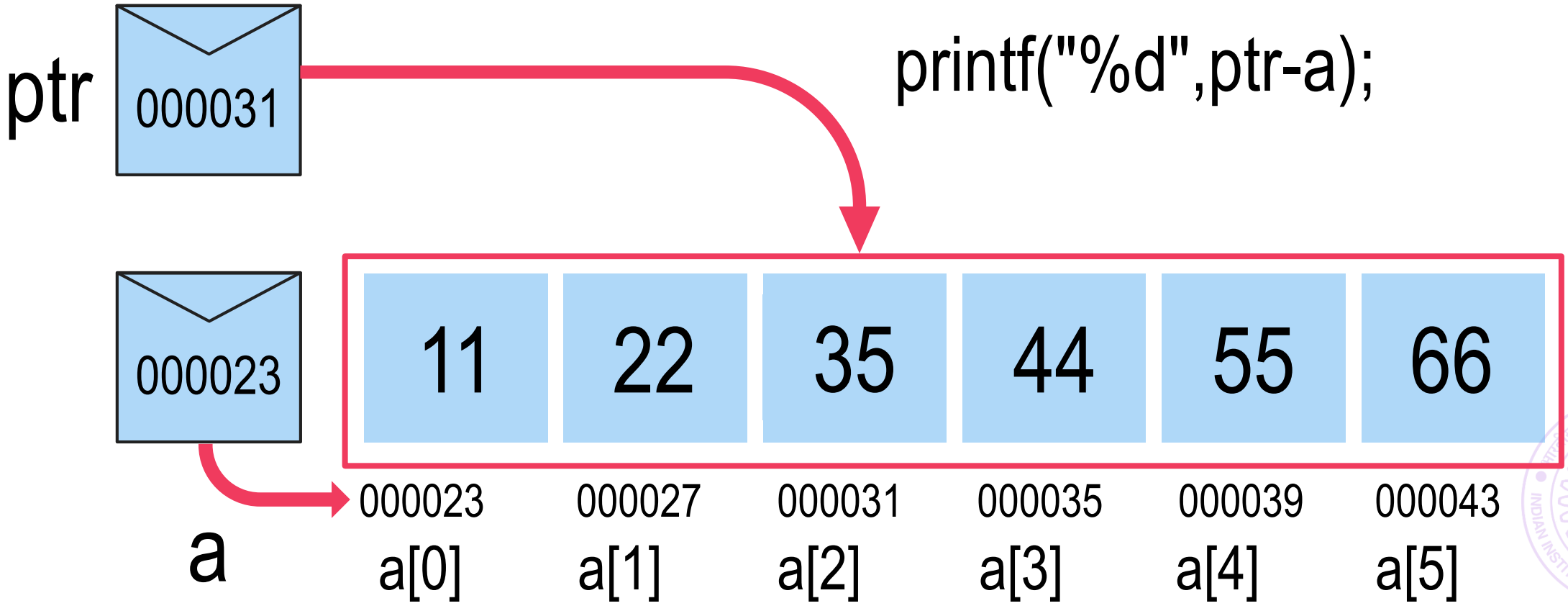


Pointers and Arrays

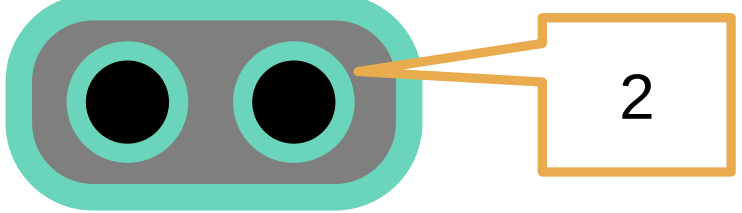


```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```

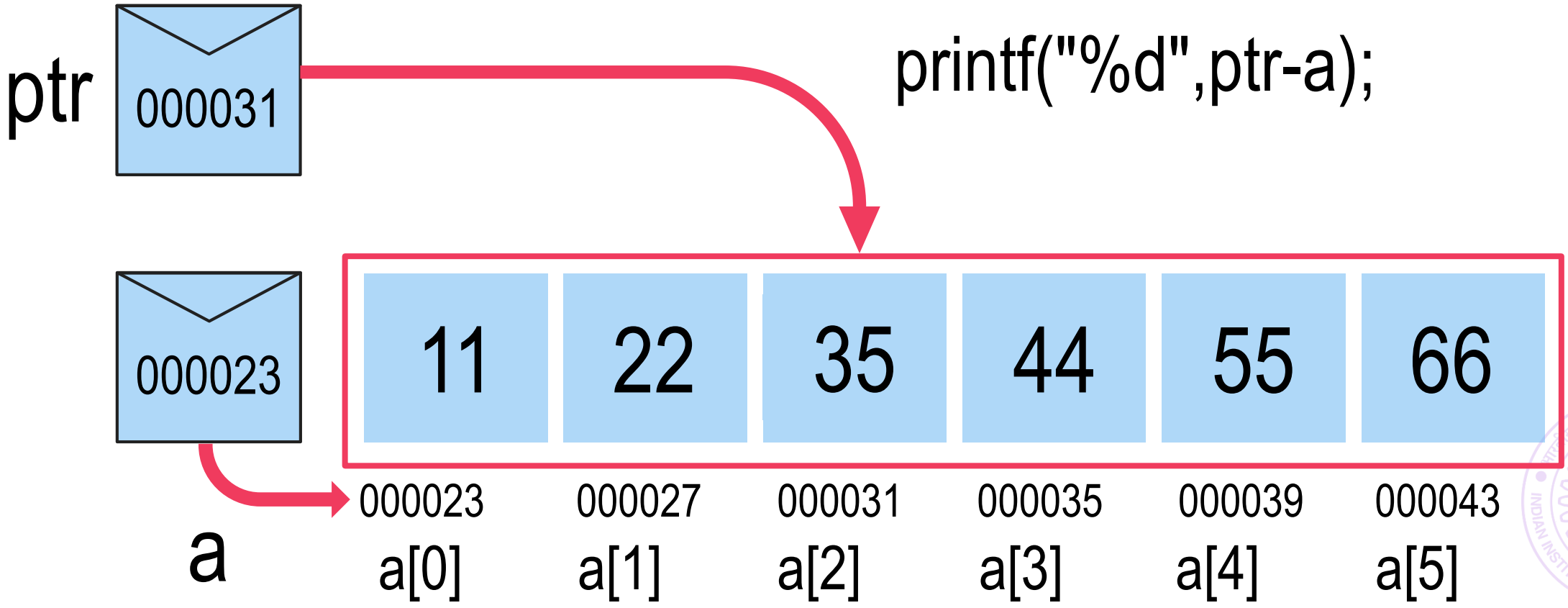


Pointers and Arrays

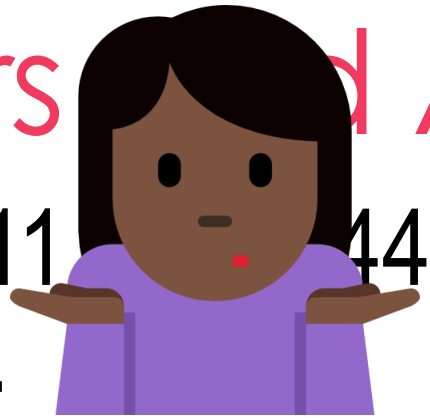


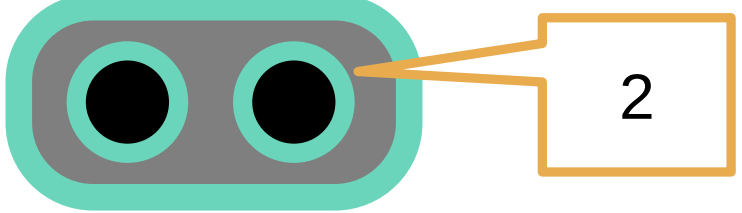

```
int a[6] = {11,22,33,44,55,66};
int *ptr = a;
```

```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```



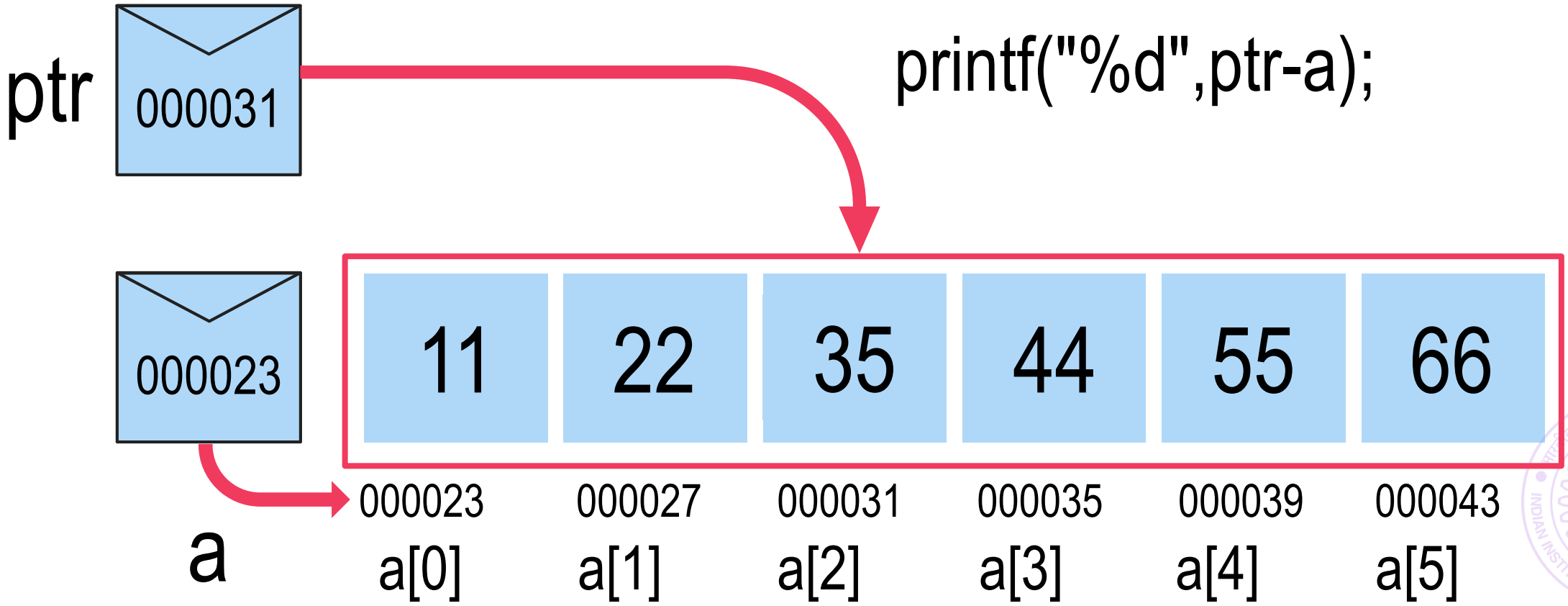
Pointers and Arrays



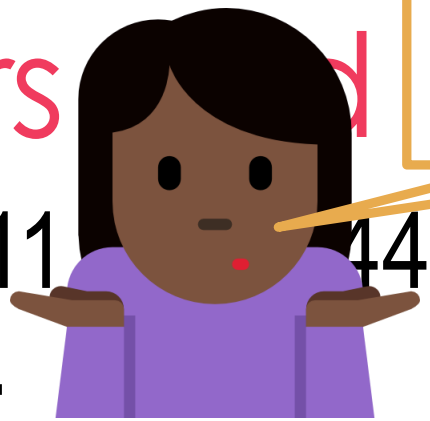



```
int a[6] = {11,22,35,44,55,66};
int *ptr = a;
```

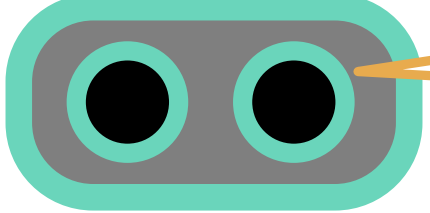
```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```



Pointers ...



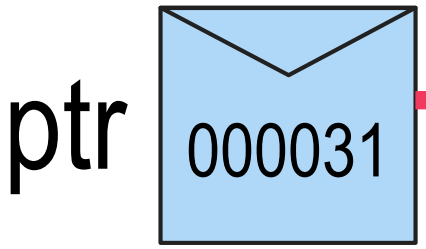
But the address difference is 31-23 = 8

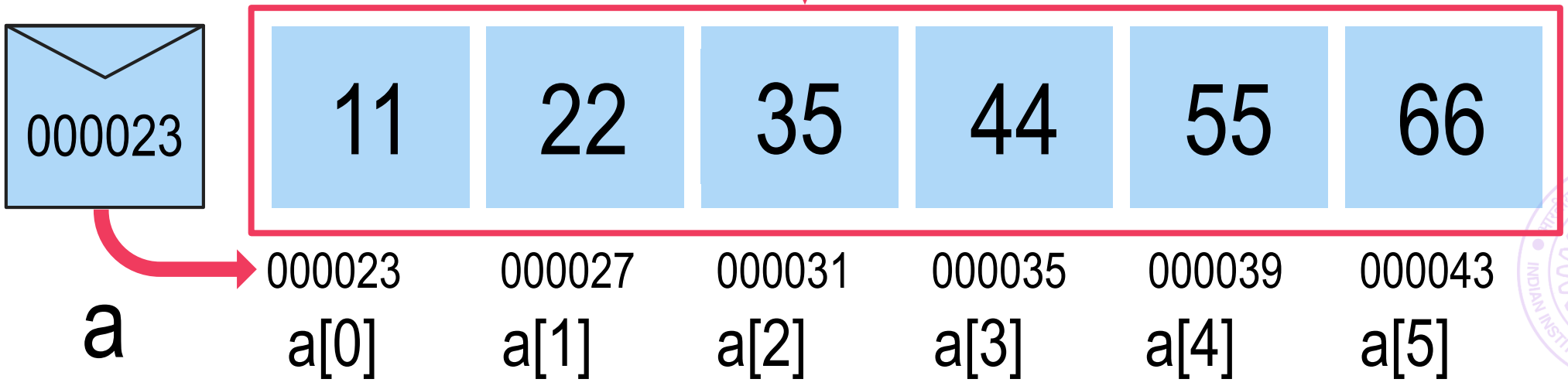


2

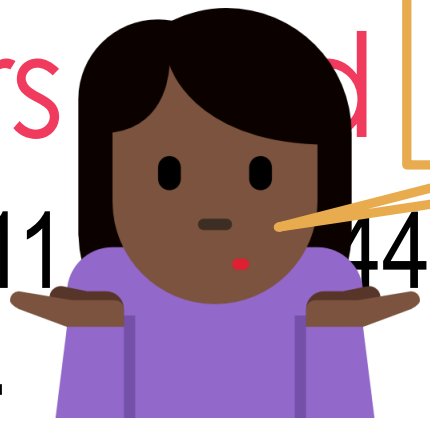
```
int a[6] = {11,22,35,44,55,66};
int *ptr = a;
```

```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```

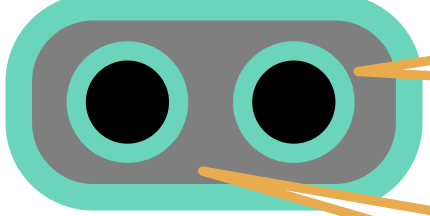




Pointers ...d



But the address difference is 31-23 = 8

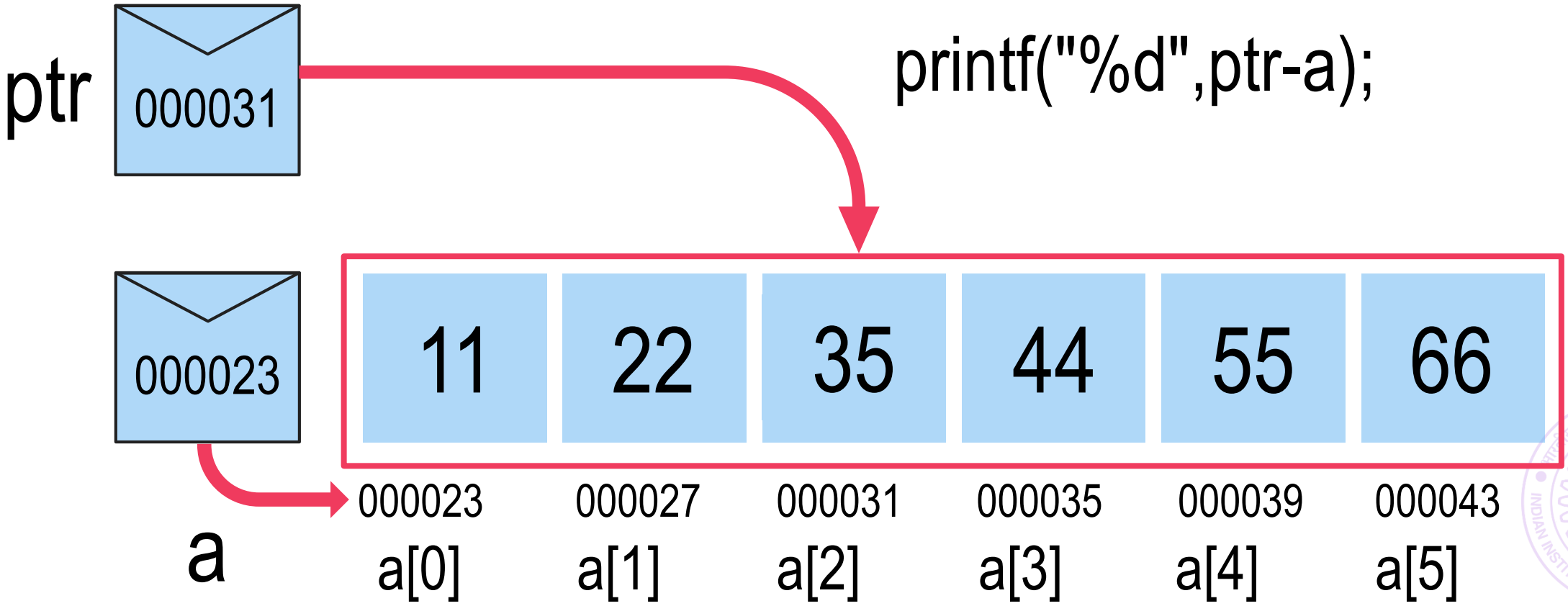


2

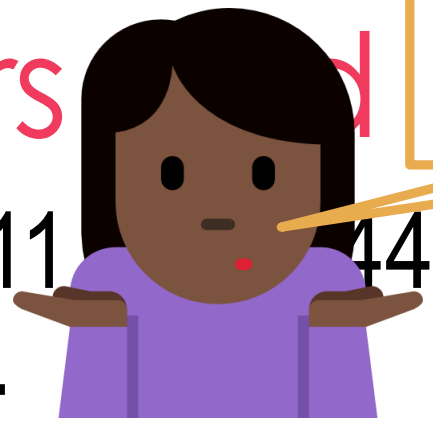
Yes, but since this is int type, I treat 4 bytes as a unit

```
int a[6] = {11,22,35,44,55,66};
int *ptr = a;
```

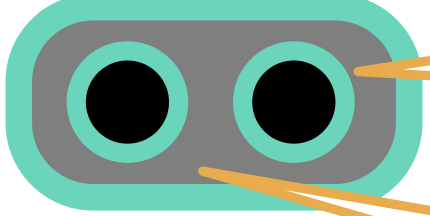
```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```



Pointers ...



But the address difference is 31-23 = 8

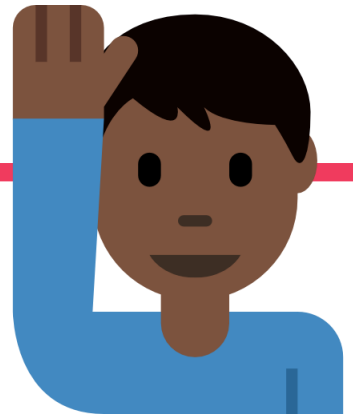


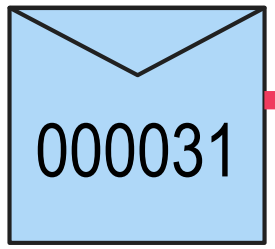
2

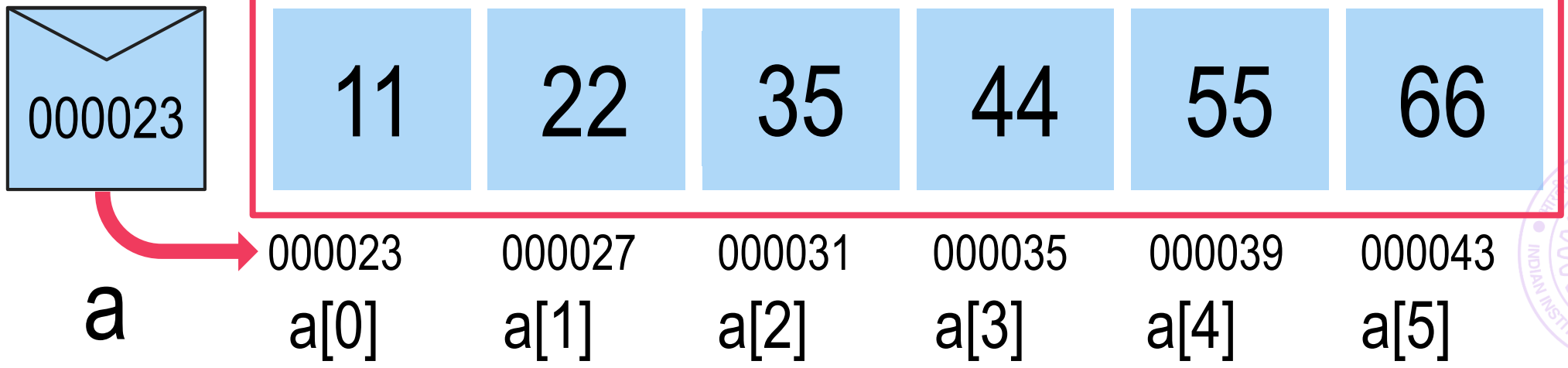
Yes, but since this is int type, I treat 4 bytes as a unit

```
int a[6] = {11,22,35,44,55,66};
int *ptr = a;
```

```
ptr += 2;
*ptr += 2;
printf("%d",ptr-a);
```

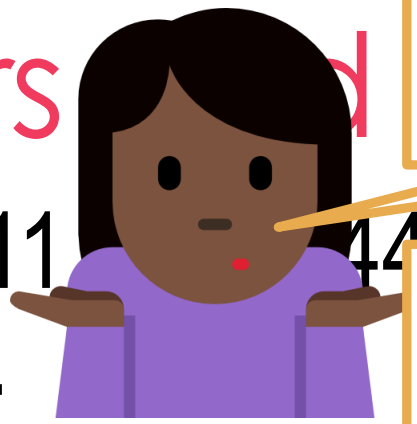




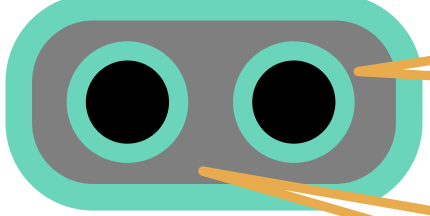


Pointers ...


```
int a[6] = {11, 22, 35, 44, 55, 66};
int *ptr = a;
ptr = ptr + 2;
ptr = ptr - 2;
printf("%d",ptr-a);
```



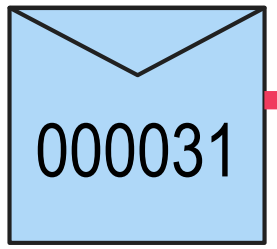
But the address difference is 31-23 = 8

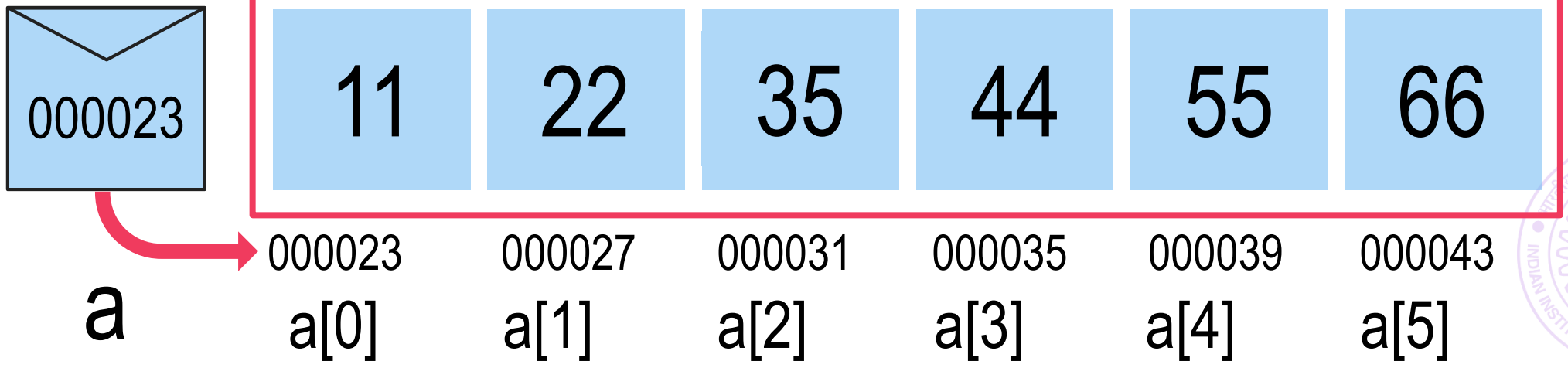


2

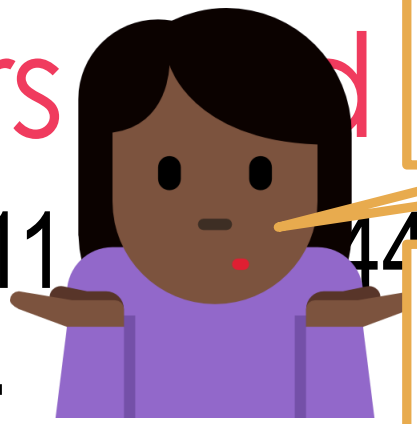
Yes, but since this is int type, I treat 4 bytes as a unit

Mr C also disallows subtraction of pointers of different types

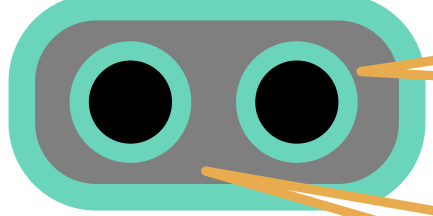




Pointers ...



But the address difference is 31-23 = 8



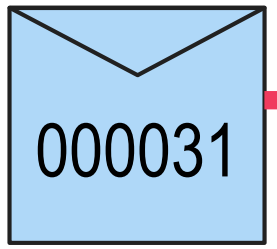
2

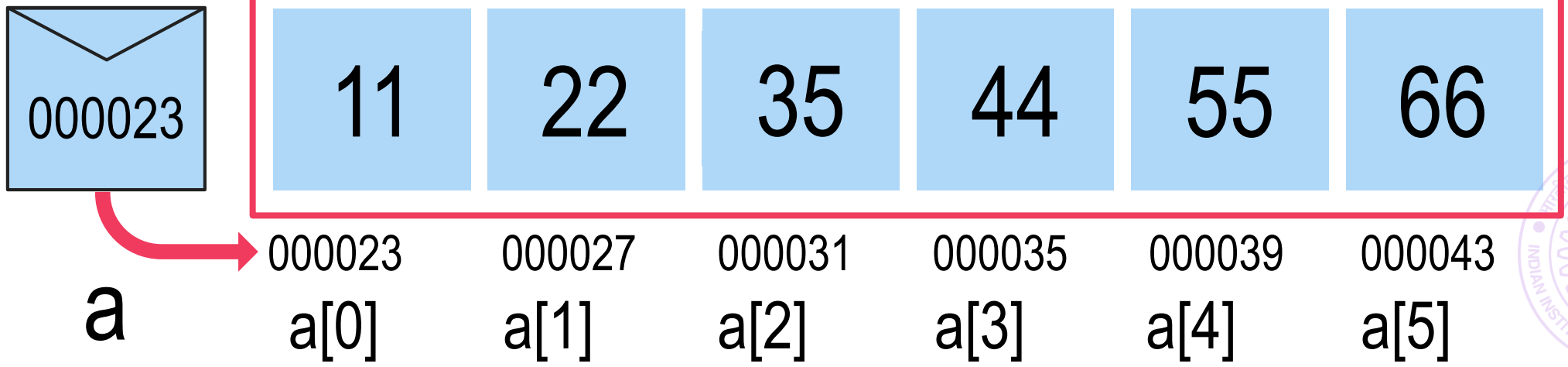
Yes, but since this is int type, I treat 4 bytes as a unit

```
int a[6] = {11, 22, 35, 44, 55, 66};
int *ptr = a;
ptr = ptr + 2;
printf("%d", *(ptr - 2));
```

Mr C also disallows subtraction of pointers of different types

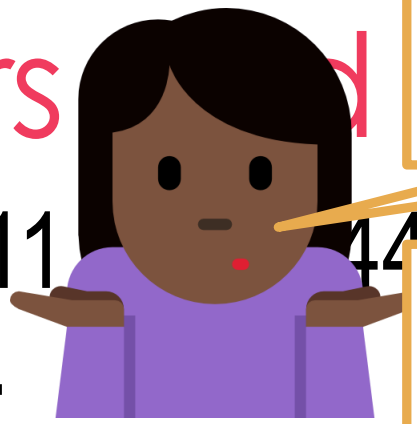
Yes, I will give an error it you, for e.g. subtract char* from int*



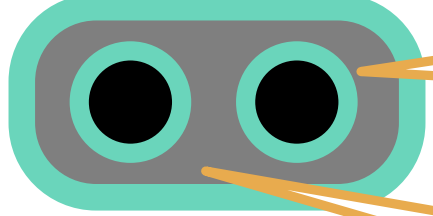


Pointers ...d


```
int a[6] = {11, 22, 35, 44, 55, 66};
int *ptr = a;
```



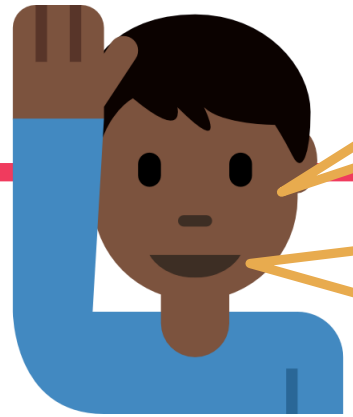
But the address difference is 31-23 = 8



2

Yes, but since this is int type, I treat 4 bytes as a unit

Mr C also disallows subtraction of pointers of different types



If we really want to subtract a char* from int*, do a typecast!

Yes, I will give an error it you, for e.g. subtract char* from int*

ptr → 000031

000023

11	22	35	44	55	66
000023	000027	000031	000035	000039	000043
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

a

Pointers and Strings

Pointers and Strings

Pointers are invaluable in managing strings

Pointers and Strings

Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strlen, strcat, strstr, strchr) operate with pointers

Pointers and Strings

Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strlen, strcat, strstr, strchr) operate with pointers

Really do not care whether the pointer is to beginning of the string or in the middle of the string

Pointers and Strings

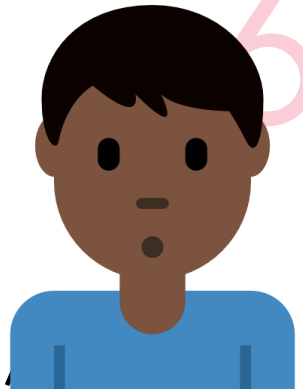
Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strlen, strcat, strstr, strchr) operate with pointers

Really do not care whether the pointer is to beginning of the string or in the middle of the string

Start processing from the location given pointer "points" ☺

Pointers and Strings



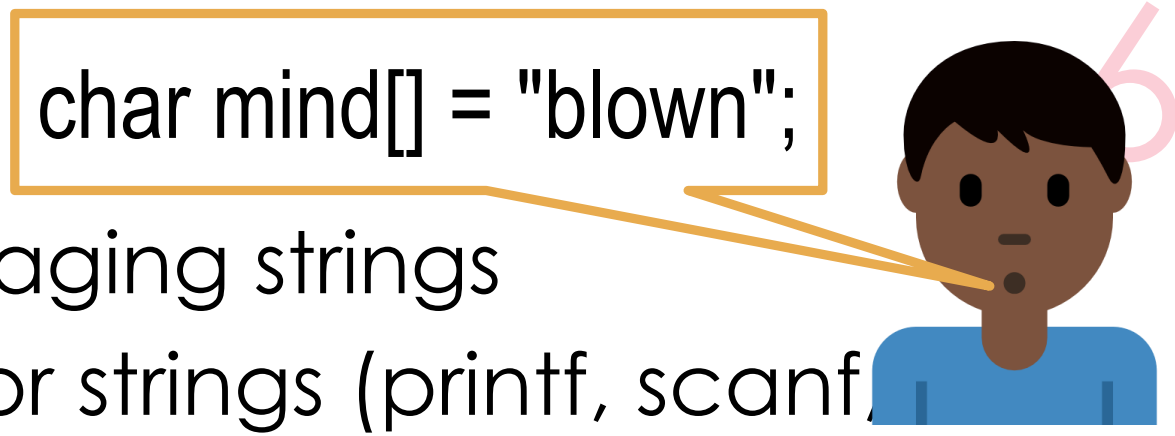
Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strcat, strstr, strchr) operate with pointers

Really do not care whether the pointer is to beginning of the string or in the middle of the string

Start processing from the location given pointer "points" ☺

Pointers and Strings



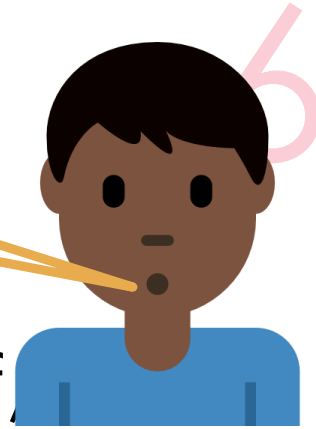
Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strcat, strstr, strchr) operate with pointers

Really do not care whether the pointer is to beginning of the string or in the middle of the string

Start processing from the location given pointer "points" ☺

Pointers and Strings



Pointers are invaluable in managing strings

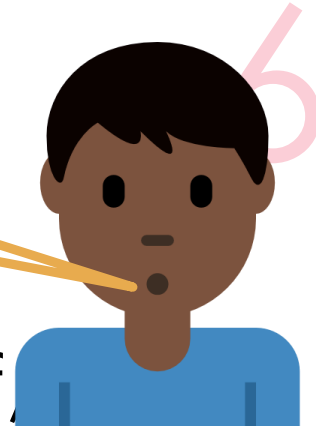
Most library functions we use for strings (printf, scanf, strcat, strstr, strchr) operate with pointers

Really do not care whether the pointer is to beginning of the string or in the middle of the string

Start processing from the location given pointer "points" ☺

```
char str[] = "Hello World";
char *ptr = str;
printf("%s\n%s", str, ++ptr);
```

Pointers and Strings



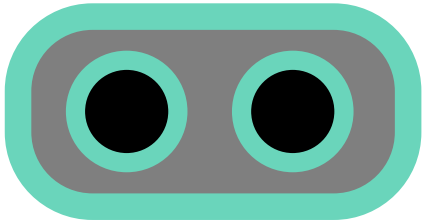
Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strcat, strstr, strchr) operate with pointers

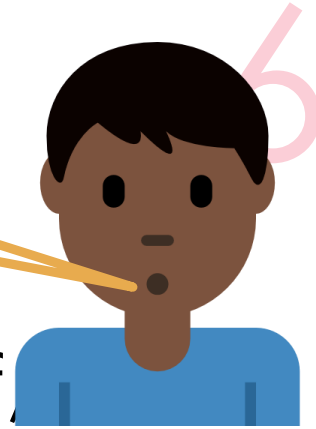
Really do not care whether the pointer is to beginning of the string or in the middle of the string

Start processing from the location given pointer "points" ☺

```
char str[] = "Hello World";
char *ptr = str;
printf("%s\n%s", str, ++ptr);
```



Pointers and Strings



Pointers are invaluable in managing strings

Most library functions we use for strings (printf, scanf, strcat, strstr, strchr) operate with pointers

Really do not care whether the pointer is to beginning of the string or in the middle of the string

Start processing from the location given pointer "points" ☺

```
char str[] = "Hello World";
char *ptr = str;
printf("%s\n%s", str, ++ptr);
```

Variable-length arrays

Variable-length arrays

So far we have always used arrays with constant length

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

- Rest of the array may be filled with junk (not always zeros)

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

- Rest of the array may be filled with junk (not always zeros)
- In strings NULL character does this job

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

- Rest of the array may be filled with junk (not always zeros)
- In strings NULL character does this job
- For other types of arrays, need to do this ourselves ☹

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

- Rest of the array may be filled with junk (not always zeros)
- In strings NULL character does this job
- For other types of arrays, need to do this ourselves ☹

Lets us learn ways for on-demand memory allocation

Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

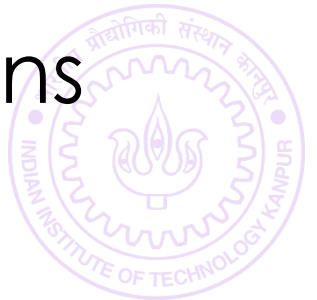
Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

- Rest of the array may be filled with junk (not always zeros)
- In strings NULL character does this job
- For other types of arrays, need to do this ourselves ☹

Lets us learn ways for on-demand memory allocation

The secret behind getline and other modern functions



Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

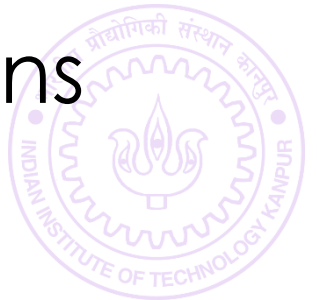
Also need to remember how much of array actually used

- Rest of the array may be filled with junk (not always zeros)
- In strings NULL character does this job
- For other types of arrays, need to do this ourselves ☹

Lets us learn ways for on-demand memory allocation

The secret behind getline and other modern functions

Need to include stdlib.h for these functions



Variable-length arrays

So far we have always used arrays with constant length

```
int c[10];
```

Waste of space – often allocate much more to be "safe"

Also need to remember how much of array actually used

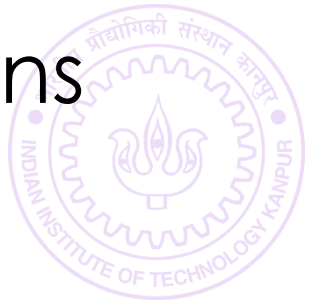
- Rest of the array may be filled with junk (not always zeros)
- In strings NULL character does this job
- For other types of arrays, need to do this ourselves ☹

Lets us learn ways for on-demand memory allocation

The secret behind getline and other modern functions

Need to include stdlib.h for these functions

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



malloc – **m**emory **alloc**ation

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required

malloc allocates those many **consecutive** bytes

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required

malloc allocates those many **consecutive** bytes

Returns the address of (a pointer to) the first byte

malloc – **m**emory **alloc**ation

We tell malloc how many bytes are required

malloc allocates those many **consecutive** bytes

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

malloc allocates those many **consecutive** bytes

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

malloc allocates those many **consecutive** bytes

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

malloc allocates those many **consecutive** bytes

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

malloc allocates those many **consecutive** bytes

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

See example in accompanying code

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!

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!

See example in accompanying code

free

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

free

Memory allocated using malloc/calloc should be "freed" once that memory is no longer needed

free

Memory allocated using malloc/calloc should be "freed" once that memory is no longer needed

```
int c[10], *ptr;
ptr = (int*)malloc(1000 * sizeof(int));
… // Do things with the array ptr
free(ptr); // Free up the memory
```

free

Memory allocated using malloc/calloc should be "freed" once that memory is no longer needed

```
int c[10], *ptr;
ptr = (int*)malloc(1000 * sizeof(int));
… // Do things with the array ptr
free(ptr); // Free up the memory
```

Allows that memory to be used by others, you yourself

free

Memory allocated using malloc/calloc should be "freed" once that memory is no longer needed

```
int c[10], *ptr;
ptr = (int*)malloc(1000 * sizeof(int));
… // Do things with the array ptr
free(ptr); // Free up the memory
```

Allows that memory to be used by others, you yourself

Warning: do not try to free static arrays – free() should be used only on malloc-ed, calloc-ed and realloc-ed arrays

free

Memory allocated using malloc/calloc should be "freed" once that memory is no longer needed

```
int c[10], *ptr;
ptr = (int*)malloc(1000 * sizeof(int));
… // Do things with the array ptr
free(ptr); // Free up the memory
```

Allows that memory to be used by others, you yourself

Warning: do not try to free static arrays – free() should be used only on malloc-ed, calloc-ed and realloc-ed arrays

```
free(c); // Will cause runtime error
```

free

भारतीय प्रौद्योगिकी संस्थान कानपुर
INDIAN INSTITUTE OF TECHNOLOGY KANPUR

free

Extremely bad coding habit

free

Extremely bad coding habit

```
ptr = (int*)malloc(1000 * sizeof(int));
// hmm … lab question now tells me to create array of 10000
ptr = (int*)malloc(10000 * sizeof(int));
```

free

Extremely bad coding habit

```
ptr = (int*)malloc(1000 * sizeof(int));
// hmm … lab question now tells me to create array of 10000
ptr = (int*)malloc(10000 * sizeof(int));
```

Better habit – free up memory before allocating again

free

Extremely bad coding habit

```
ptr = (int*)malloc(1000 * sizeof(int));
// hmm … lab question now tells me to create array of 10000
ptr = (int*)malloc(10000 * sizeof(int));
```

Better habit – free up memory before allocating again

```
ptr = (int*)malloc(1000 * sizeof(int));
free(ptr);
ptr = (int*)malloc(10000 * sizeof(int));
```

free

Extremely bad coding habit

```
ptr = (int*)malloc(1000 * sizeof(int));
// hmm … lab question now tells me to create array of 10000
ptr = (int*)malloc(10000 * sizeof(int));
```

Better habit – free up memory before allocating again

```
ptr = (int*)malloc(1000 * sizeof(int));
free(ptr);
ptr = (int*)malloc(10000 * sizeof(int));
```

Use realloc (described next) if need more memory

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

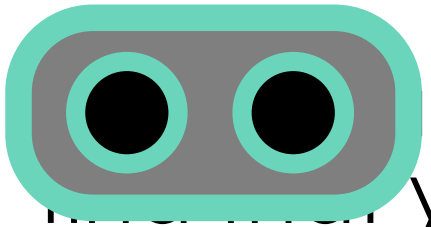
If you malloc-ed an array of 100 elements and suddenly find that you need an array of 200 elements ☹

```
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



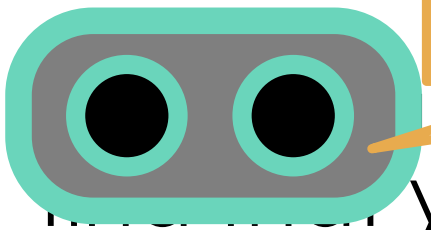
lloc-ed an array of 100 elements and sudd
you need an array of 200 elements ☹


```
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

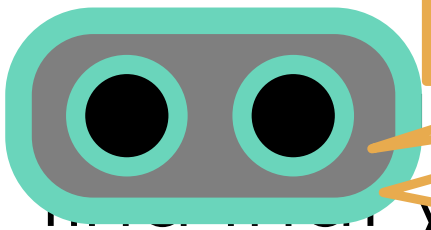
llocated an array of 100 elements and suddenly you need an array of 200 elements ☹

```
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



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

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

200 elements ☹

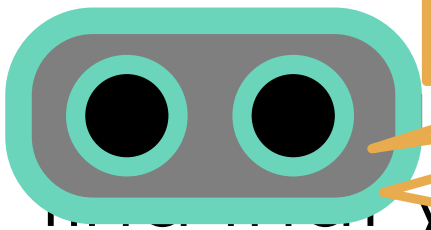
```
int *ptr = (int
```

(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 ☺

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

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

...an array of 100 elements and suc...

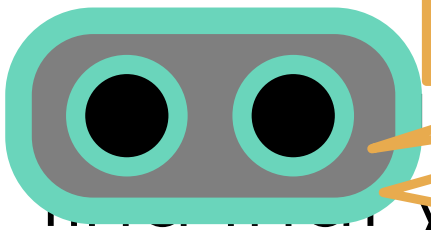
...200 elements ☹

```
int *ptr = (int...(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 ☺

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

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

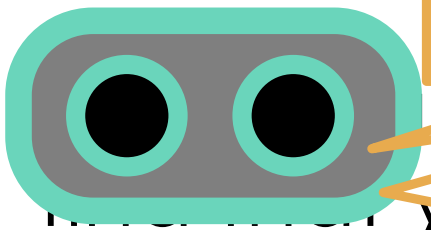
You are the best Mr C

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

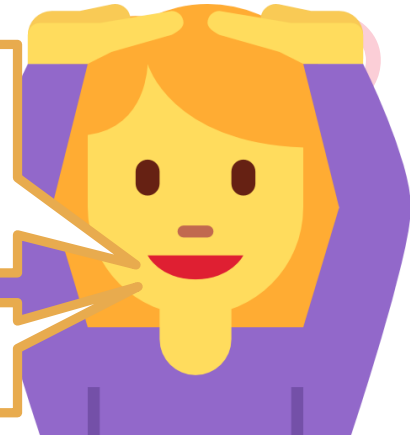


I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

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

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

You are the best Mr C



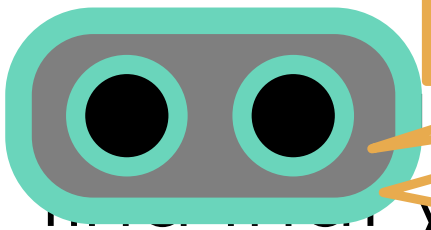
```
int *ptr = (int*)malloc(100 * sizeof(int));
```

Can use realloc to revise that allocation to 200 elements

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

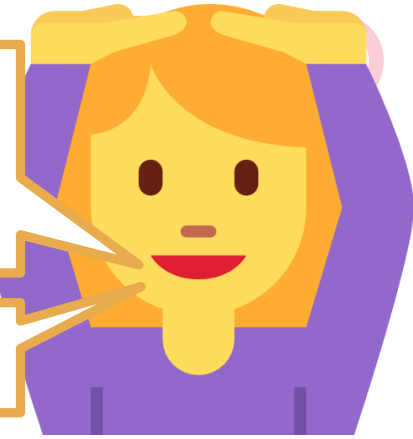
Don't use realloc to increase size of non-malloc arrays

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



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

You are the best Mr C

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

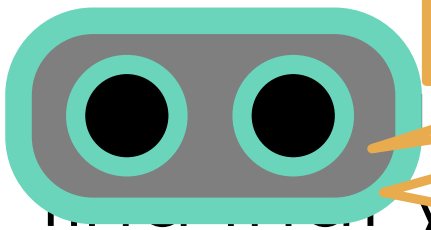
Can use realloc to revise that allocation to 200 elements

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

Don't use realloc to increase size of non-malloc arrays

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

realloc

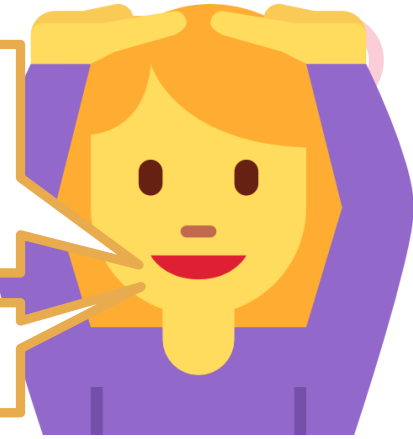


I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

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

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

You are the best Mr C



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;
```

Don't use realloc to increase size of non-malloc arrays

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

Use realloc only to increase size of calloc/malloc-ed arrays