Pointers and Memory Allocation

ESC101: Fundamentals of Computing

Nisheeth

Pointers

The pointer itself takes 8 bytes in memory

A pointer refers to an address in memory (2^^{64} – 1 possible addresses)

Syntax for declaration: type *ptr; // ptr is pointer to a variable with data type “type” (examples: int *ptr, char *ptr)

Can declare pointer and regular variables on same line

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

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

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

Dereferencing a pointer gives the value stored at that address

Dereferencing is done using * operator

If ptr is a pointer to a variable i then *ptr means i

For arrays, the name itself is the pointer and points to first element

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	? :	i
Assignment, Compound assignment	=, +=, -=, *=, /=, %=	Right

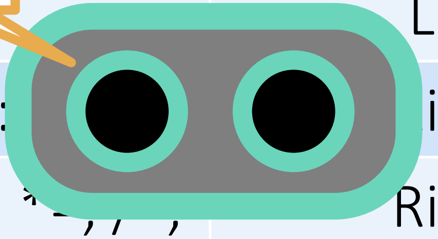
HIGH PRECEDENCE

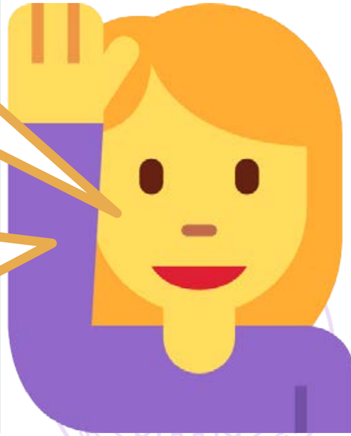
LOW PRECEDENCE

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

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

30





Pointer Arithmetic

Can take a pointer variable add and subtract integers

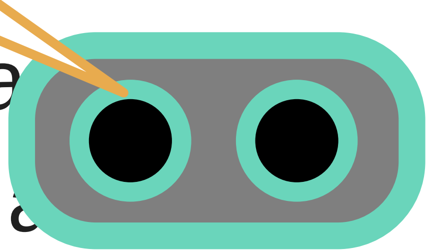
Result depends on the type of the pointer

- Pointers to int advance by 4 upon adding 1 or doing ++
- Pointers to int go back by 4 upon subtracting 1 or doing --
- Pointers to char advance by 1 upon adding 1 or doing ++
- Pointers to char go back by 1 upon subtracting 1 or doing --
- Pointers to double advance by 8 upon adding 1 or doing ++
- Pointers to double go back by 8 upon subtracting 1 or doing --

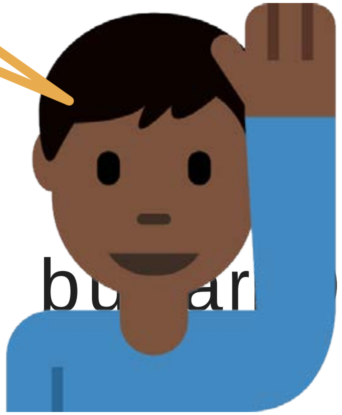
Note: Can’t increment/decrement an array pointer (more on this later)

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 a

```
int arr[10]
int a, b, *ptr = arr;
```

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

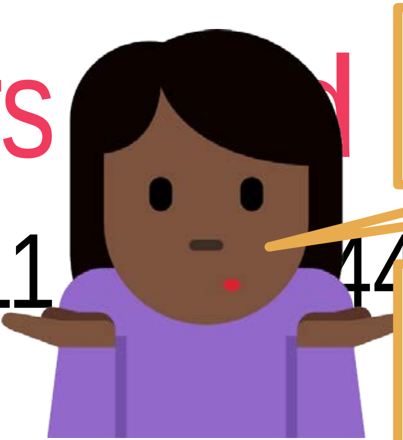
Pointer arithmetic often used to traverse (go back and forth in) arrays and calculate offsets

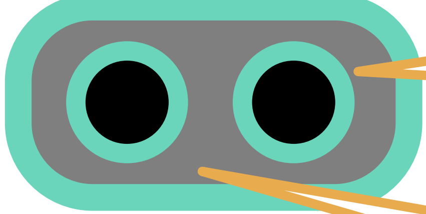
arr[2] and *(arr+2) both give value of the 3rd element in arr

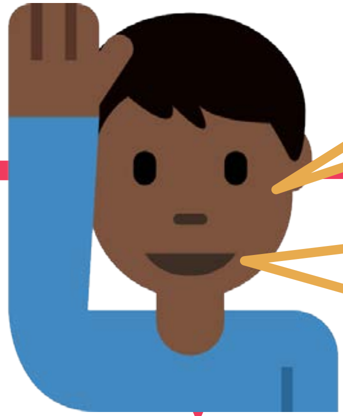
Warning: arr++ will give error, ptr++ will move pointer to arr[1]

Pointers d

```
int a[6] = {11                44,55,66};        ptr += 2;
int *ptr = a;                                     = 2;
                                                  "%d",
```







But the address difference is 31-23 = 8

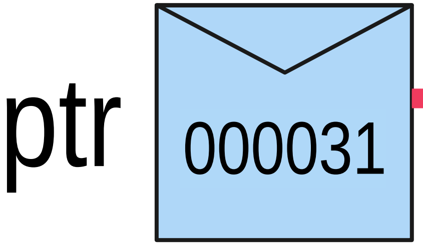
2

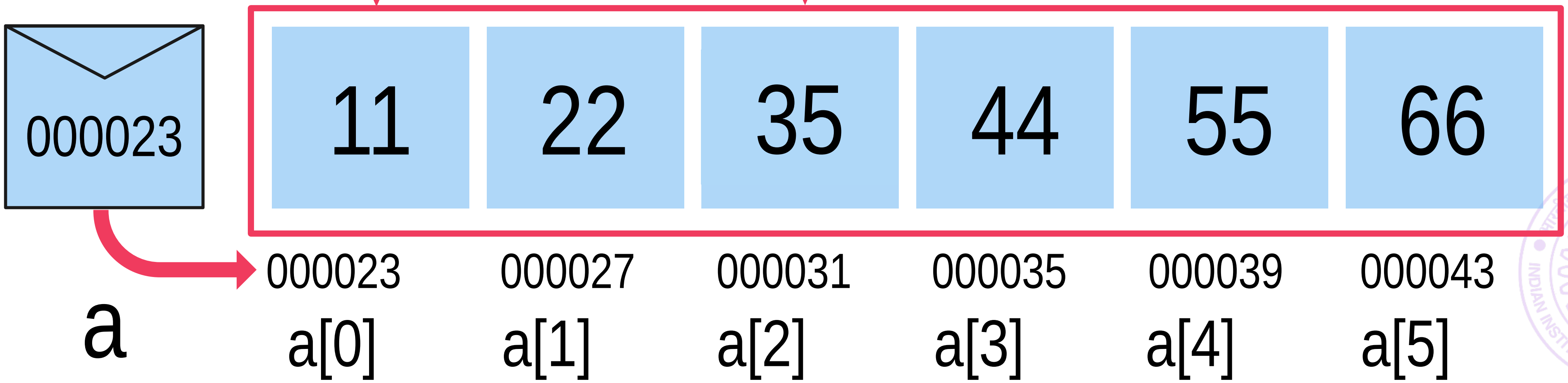
Yes, but since this is int type, I treat 4 bytes as a unit

Mr C also disallows subtraction of pointers of different types

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

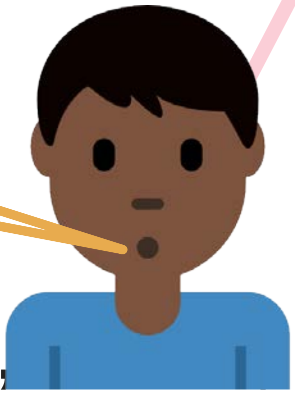
If we really want to subtract a char* from int*, do a typecast!





Pointers and Strings

```
char mind[] = "blown";
```



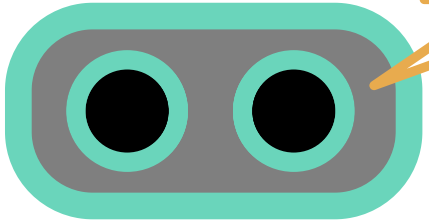
Pointers are invaluable in managing strings

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);
```



```
Hello World
ello World
```

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 – **memory 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 – contiguous **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

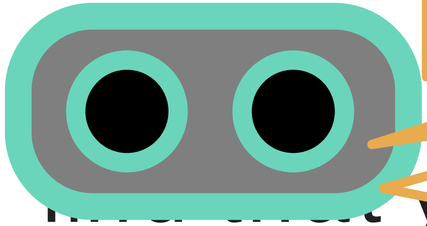
- length of array (number of elements in the array)
- number of bytes per element

Sends back a NULL pointer if insufficient memory – careful!

Need to typecast the pointer returned by calloc too!

See example in accompanying code

realloc c

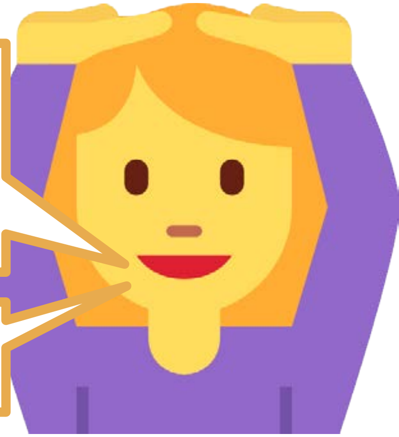


I realize that. That is why I will copy those 100 elements to the new array of 200 elements ☺

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

But what if I had precious data stored in those 100 elements

You are the best Mr C



llo an array of 100 elements and suc

find that yo 200

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;
```

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