EEE243 – Applied Computer Programming

Pointers

ROYAL MILITARY COLLEGE OF CANADA
ELECTRICAL & COMPUTER
ENGINEERING

GÉNIE ÉLECTRIQUE
ET GÉNIE INFORMATIQUE
COLLÈGE MILITAIRE ROYAL DU CANADA

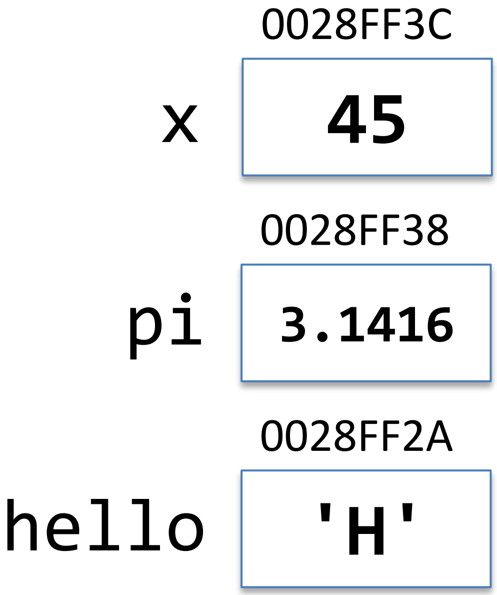
RMC CMR

Outline

1. Addresses
2. The address operator: &
3. Pointers
4. Indirection operator: *
5. Initialization of pointers
6. Working with pointers and addresses
7. Pointers and functions
8. Pointers and arrays
9. Pointer Types
10. Pointer Arithmetic

Addresses

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    return EXIT_SUCCESS;
}
```



0028FF3C

x 45

0028FF38

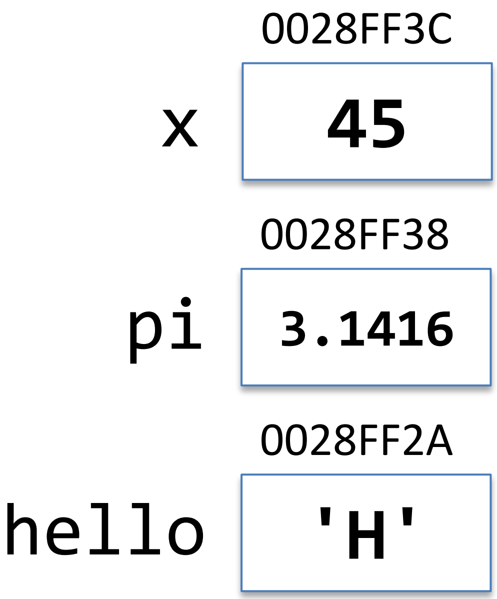
pi 3.1416

0028FF2A

hello 'H'

Addresses

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    return EXIT_SUCCESS;
}
```

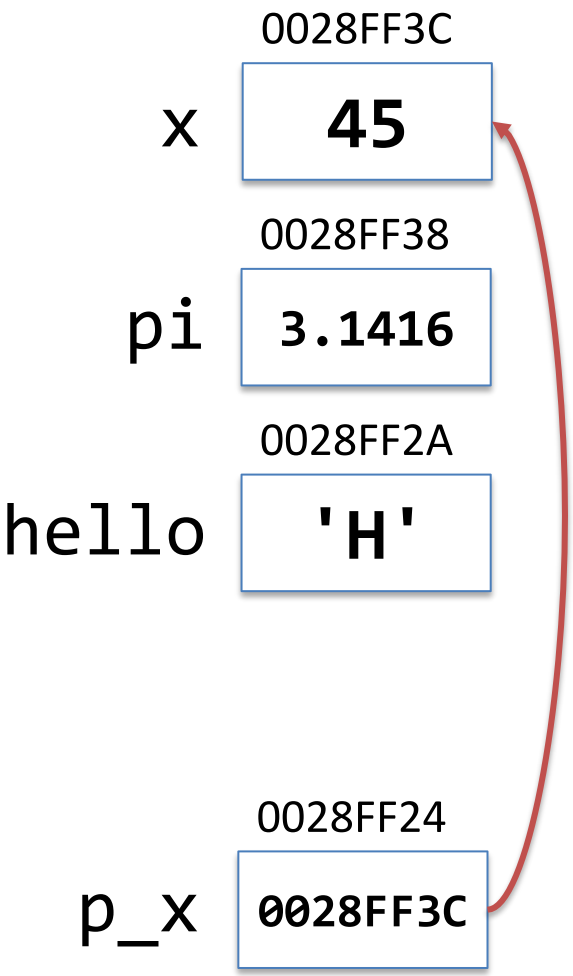


0028FF3C
x 45
0028FF38
pi 3.1416
0028FF2A
hello 'H'

identifier (a symbol)

Addresses

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    return EXIT_SUCCESS;
}
```



0028FF3C

x 45

0028FF38

pi 3.1416

0028FF2A

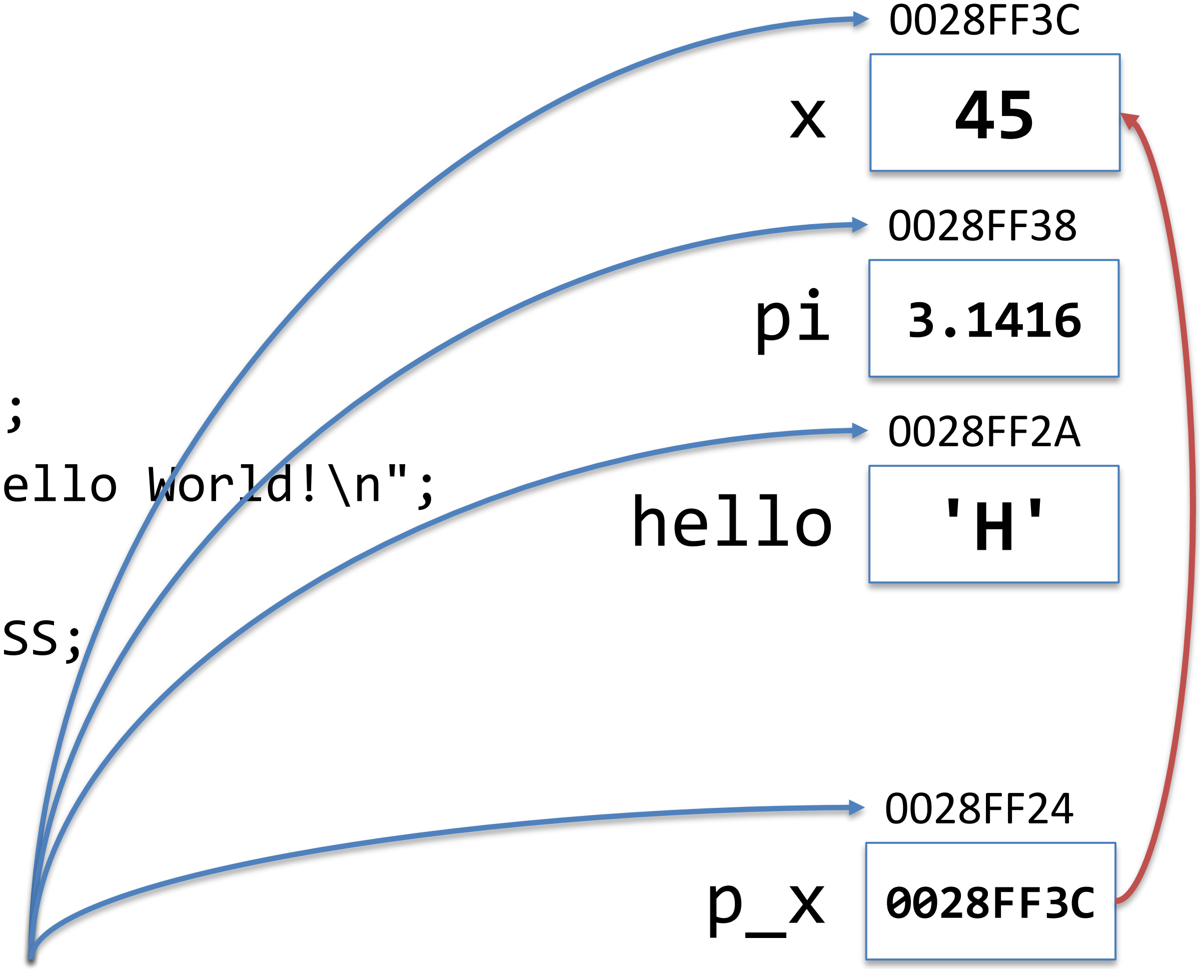
hello 'H'

0028FF24

p_x 0028FF3C

Addresses

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    return EXIT_SUCCESS;
}
```



0028FF3C

x 45

0028FF38

pi 3.1416

0028FF2A

hello 'H'

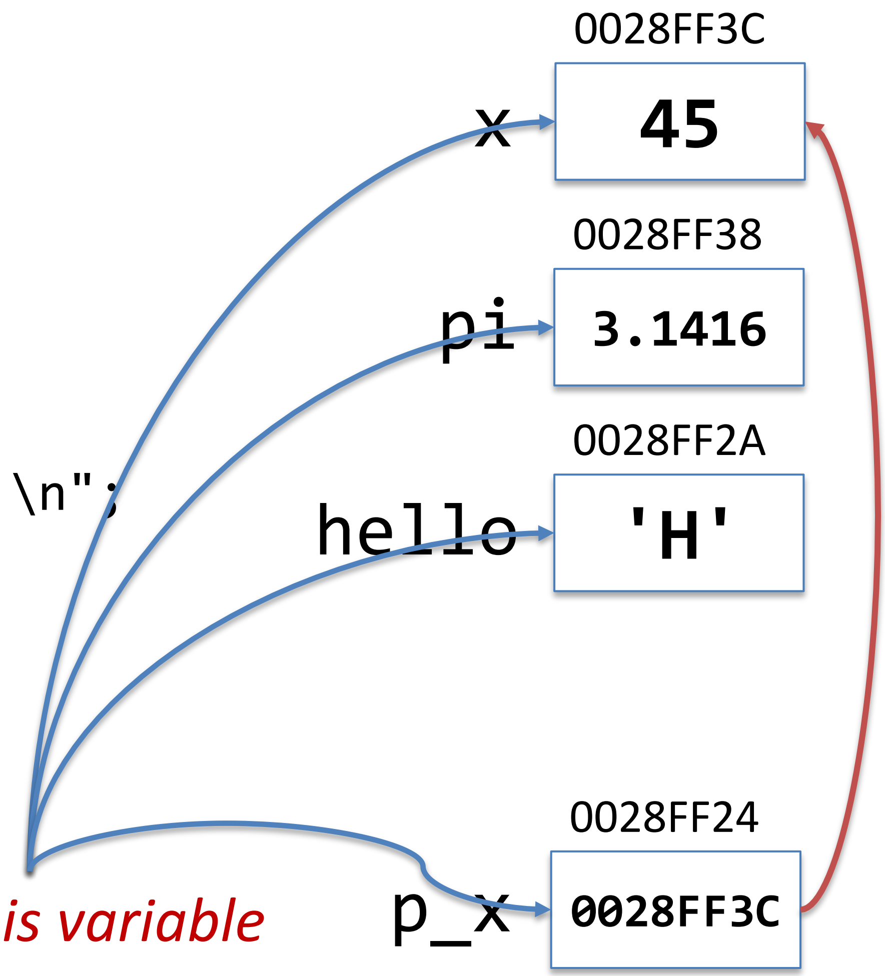
0028FF24

p_x 0028FF3C

Addresses are constants (for the current execution)

Addresses

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    return EXIT_SUCCESS;
}
```



0028FF3C

x

45

0028FF38

pi

3.1416

0028FF2A

hello

'H'

0028FF24

p_x

0028FF3C

Content is variable

Address operator (&)

- So where does my variable live?

  - The address operator (&) gives the location in memory

  - If `my_variable` in my program is a `char` then `&my_variable` is the address of that `char`

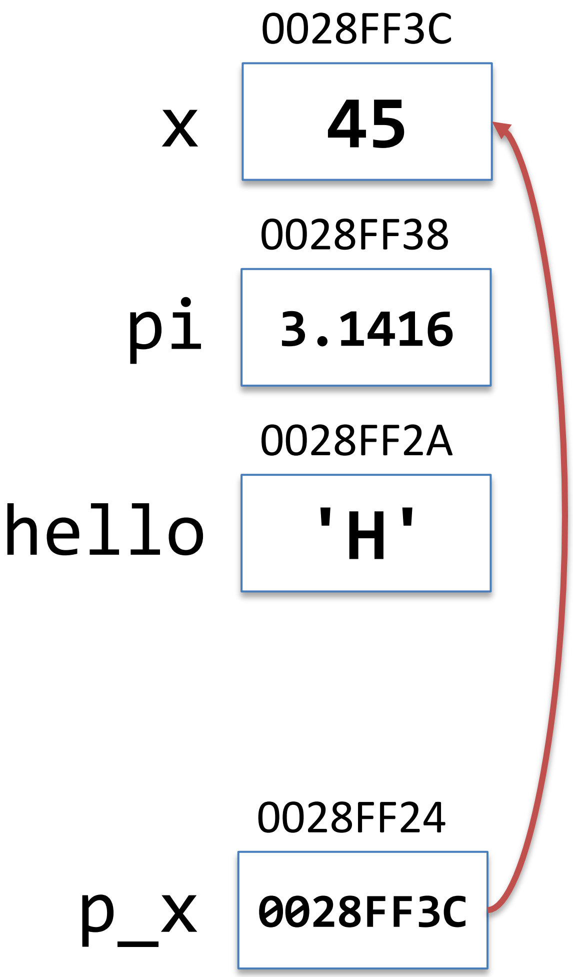
Pointer variables

- We can store the result of the & operator in a pointer variable
- A pointer is declared as follows:

```
int *p_to_int;    //pointer to an int

char *p_to_char;  //pointer to a char
```

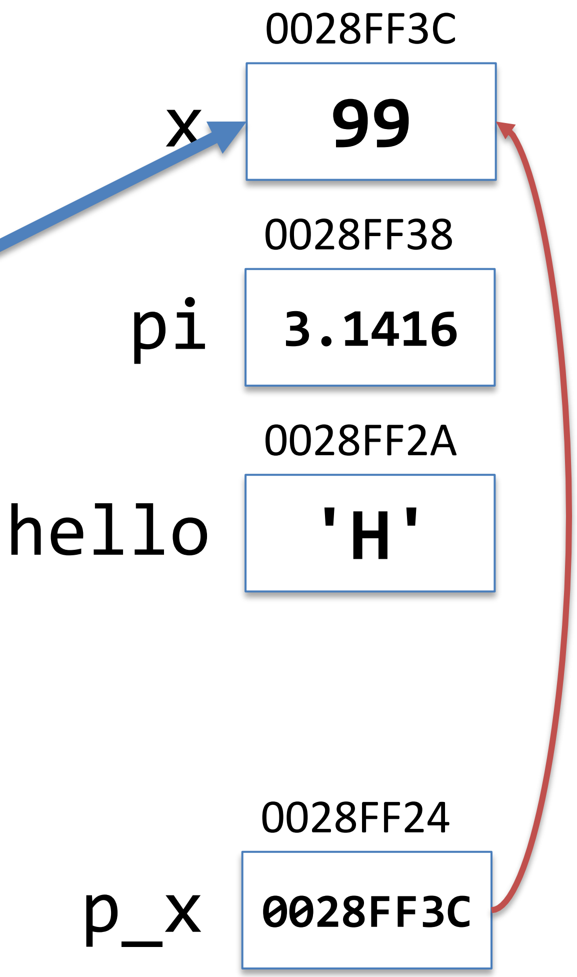
The indirection operator (*)

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    *p_x = 99;
    return EXIT_SUCCESS;
}
```



The indirection operator (*)

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    *p_x = 99;
    return EXIT_SUCCESS;
}
```



0028FF3C

x 99

0028FF38

pi 3.1416

0028FF2A

hello 'H'

0028FF24

p_x 0028FF3C

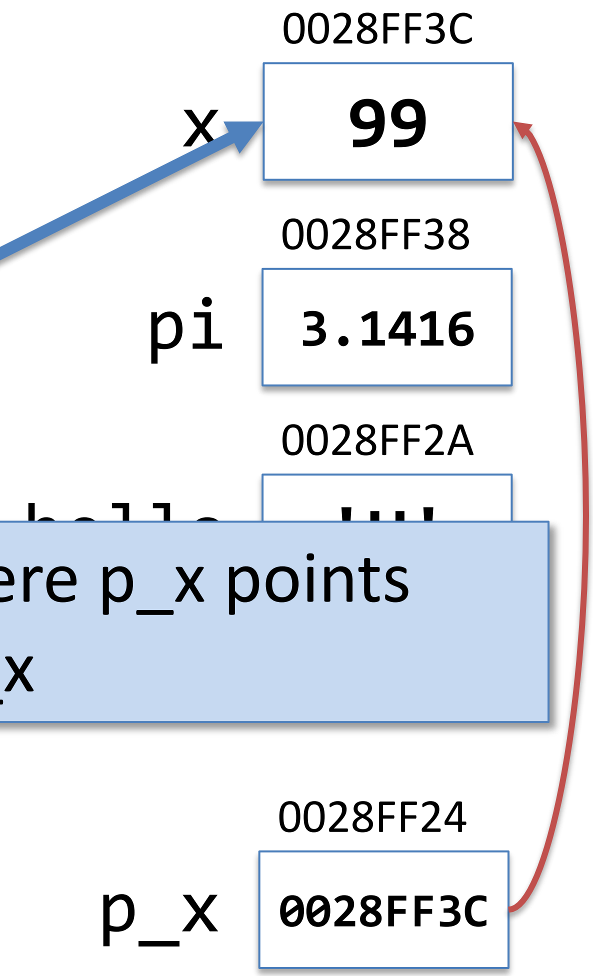
The indirection operator (*)

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

int main(void) {
    int x = 45;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
```

```
    return EXIT_SUCCESS;
}
```

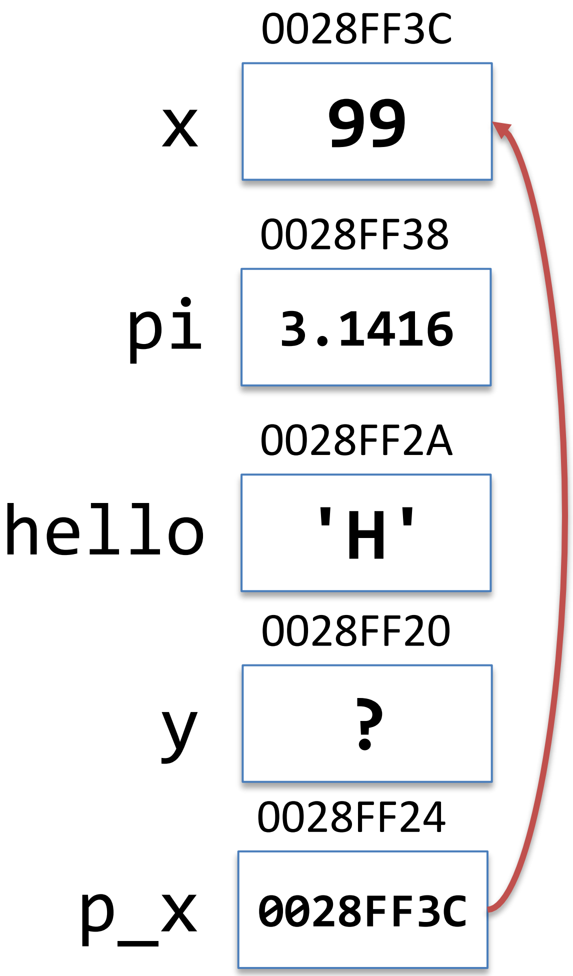
*p_x = 99;   Changes the content where p_x points

p_x = 99;   Changes the content of p_x



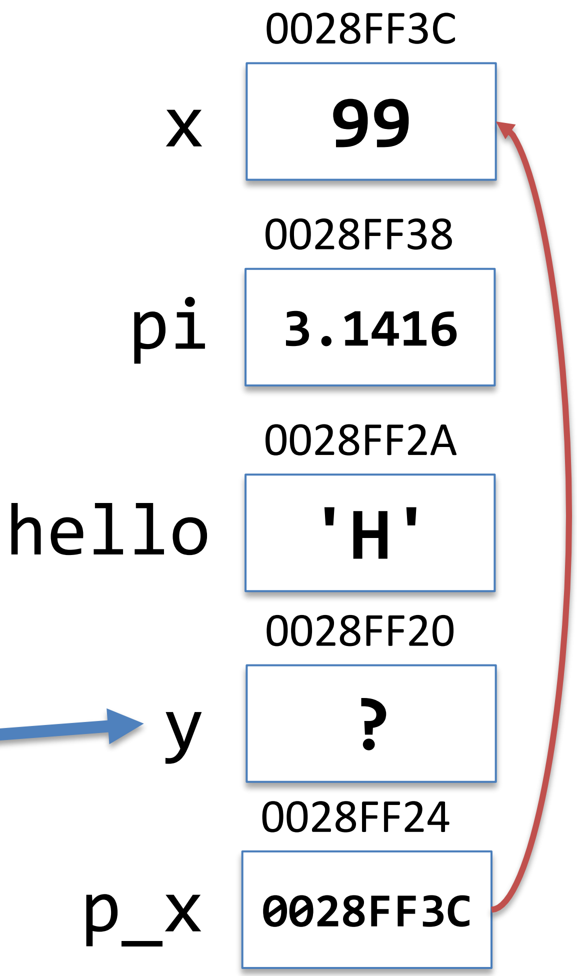
The indirection operator (*)

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

int main(void) {
    int x = 45;
    int y;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    *p_x = 99;
    return EXIT_SUCCESS;
}
```



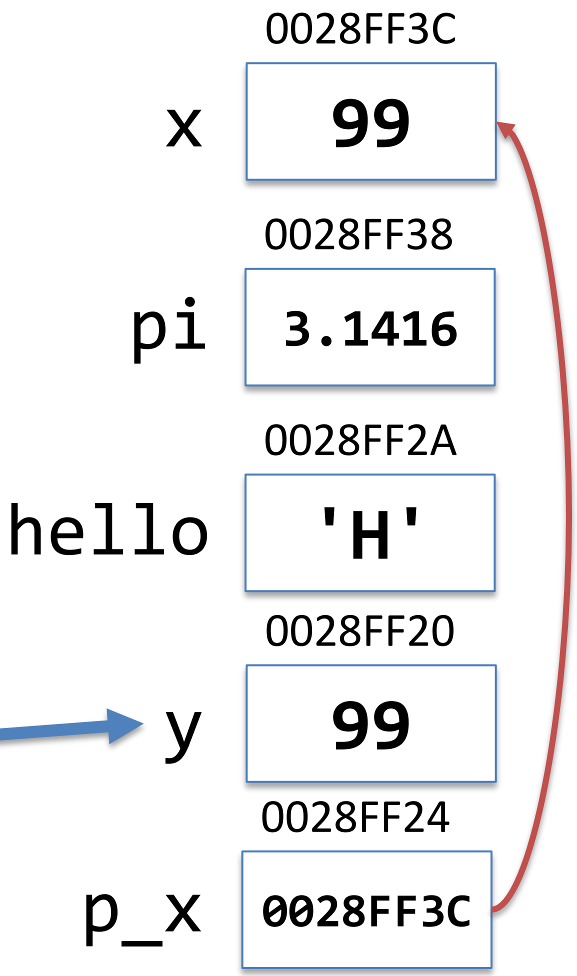
The indirection operator (*)

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

int main(void) {
    int x = 45;
    int y;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    *p_x = 99;
    y = *p_x;
    return EXIT_SUCCESS;
}
```



The indirection operator (*)

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

int main(void) {
    int x = 45;
    int y;
    float pi = 3.1416;
    char hello[] = "Hello World!\n";
    int *p_x = &x;
    *p_x = 99;
    y = *p_x;
    return EXIT_SUCCESS;
}
```



0028FF3C
x 99
0028FF38
pi 3.1416
0028FF2A
hello 'H'
0028FF20
y 99
0028FF24
p_x 0028FF3C

Initializing Pointers

- A pointer, like any other variable in C, is not automatically initialized
  - It contains garbage upon declaration
- You should initialize all your pointers explicitly
  - This is good practice for all variables …
  - … it is critical for pointers!
- You can initialize a pointer using a real address:

```
int a;
int *p;
p = &a;
int *p_x = NULL;
```

Pointers - Advantages

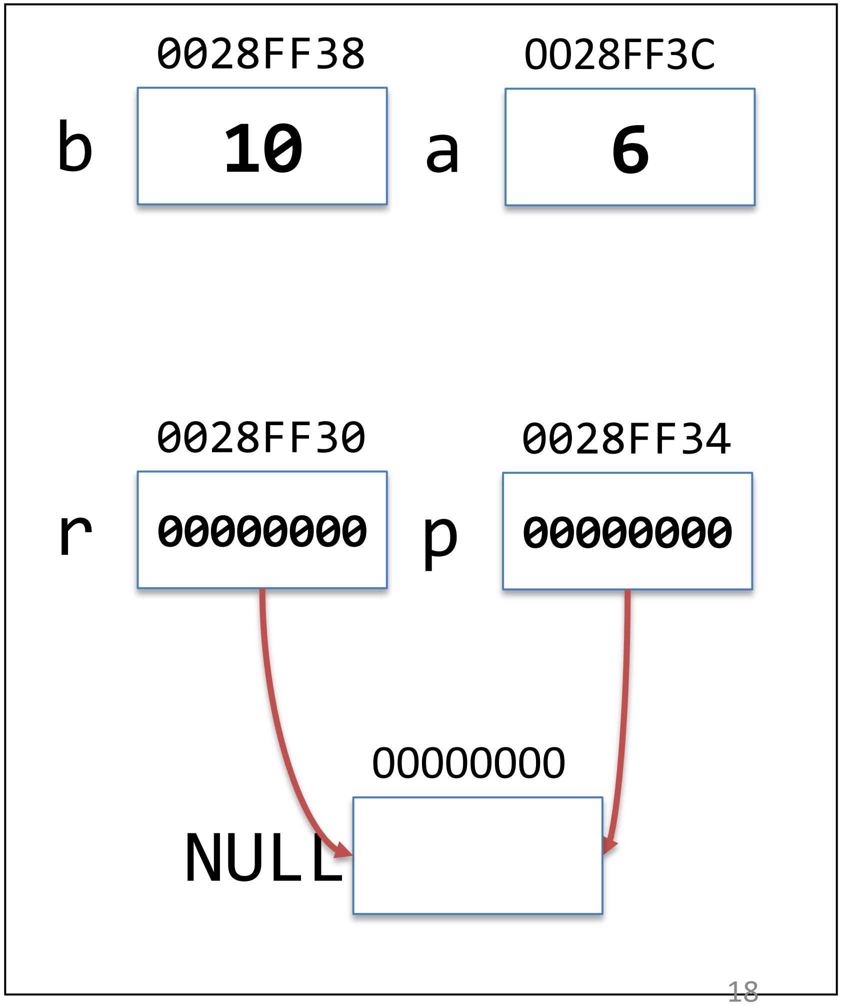
- Pointers allow us to pass the address of variables as parameters to functions
- They are at the basis of dynamic memory allocation in C
  - Allow us to grow and shrink data structures if we do not know the size of data we will encounter upon variable declaration
  - Effective use of memory – Excellent for small microcontrollers
- Pointers allow for efficient manipulation of data in arrays

Working with pointers and addresses

Symbolic

```
int a = 6;
int b = 10;
int *p = NULL;
int *r = NULL;
p = &a;
r = p; //points at same
      //variable
b = *r;
p = &b;
*p = 8;
```

Memory

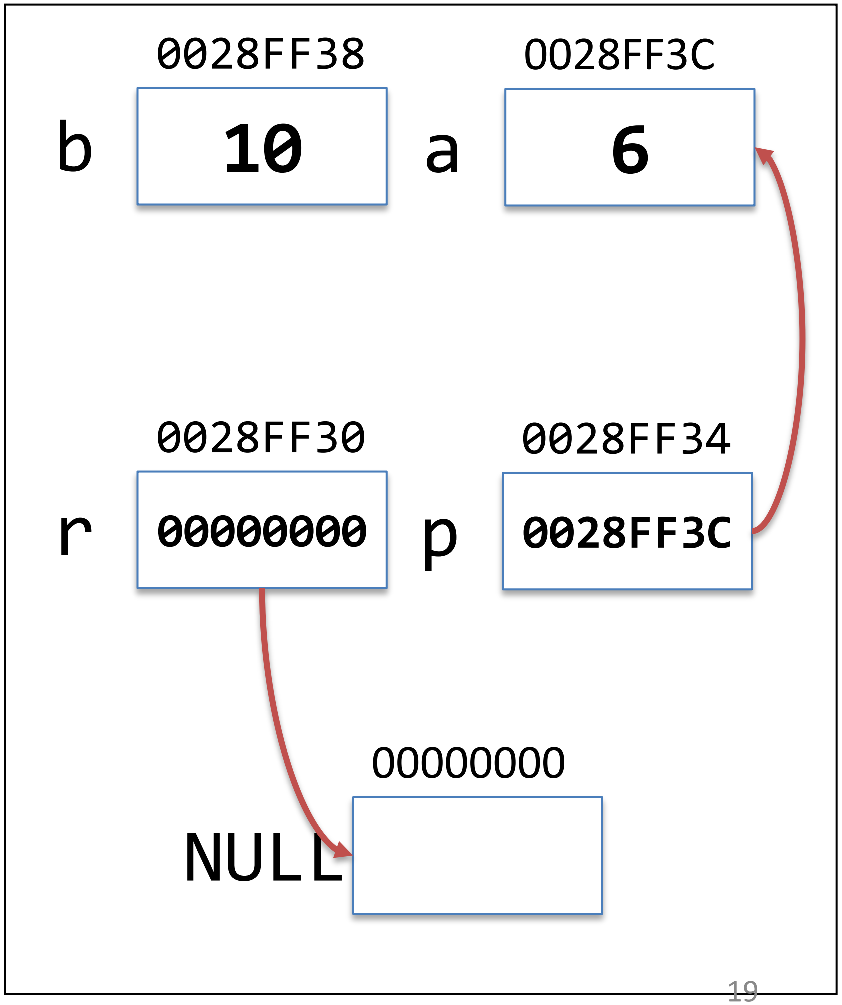


Working with pointers and addresses

Symbolic

```
int a = 6;
int b = 10;
int *p = NULL;
int *r = NULL;
p = &a;
r = p; //points at same
      //variable
b = *r;
p = &b;
*p = 8;
```

Memory

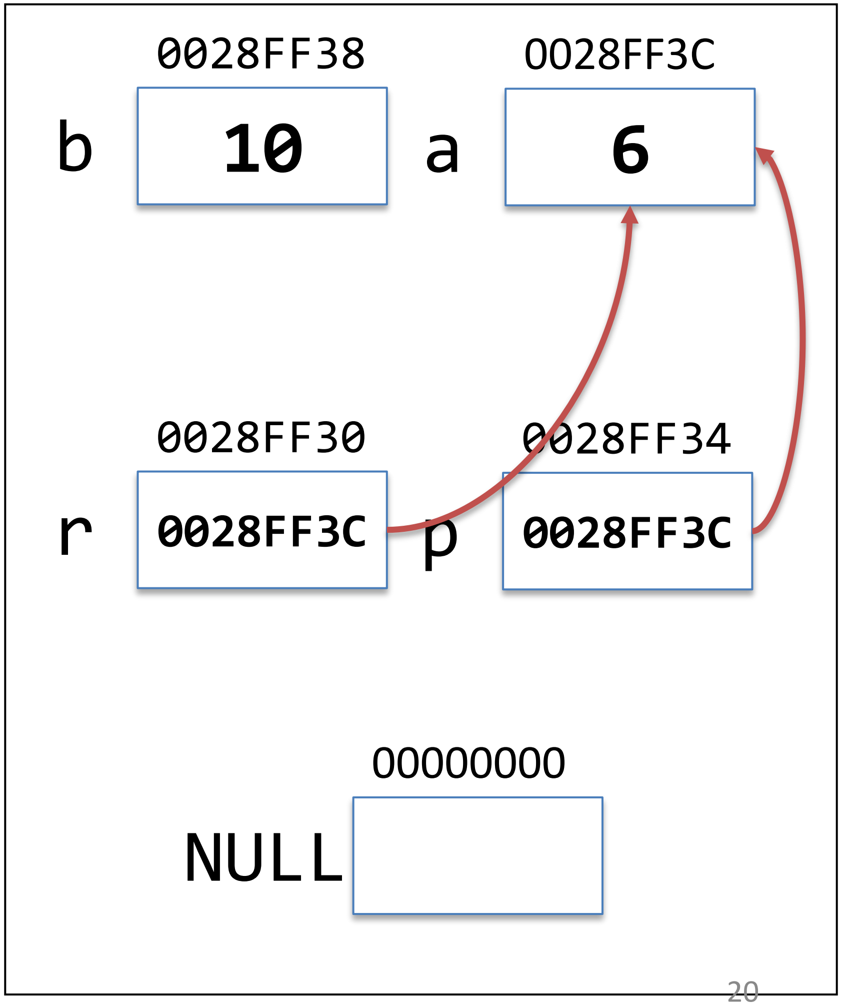


Working with pointers and addresses

Symbolic

```
int a = 6;
int b = 10;
int *p = NULL;
int *r = NULL;
p = &a;
r = p; //points at same
      //variable
b = *r;
p = &b;
*p = 8;
```

Memory

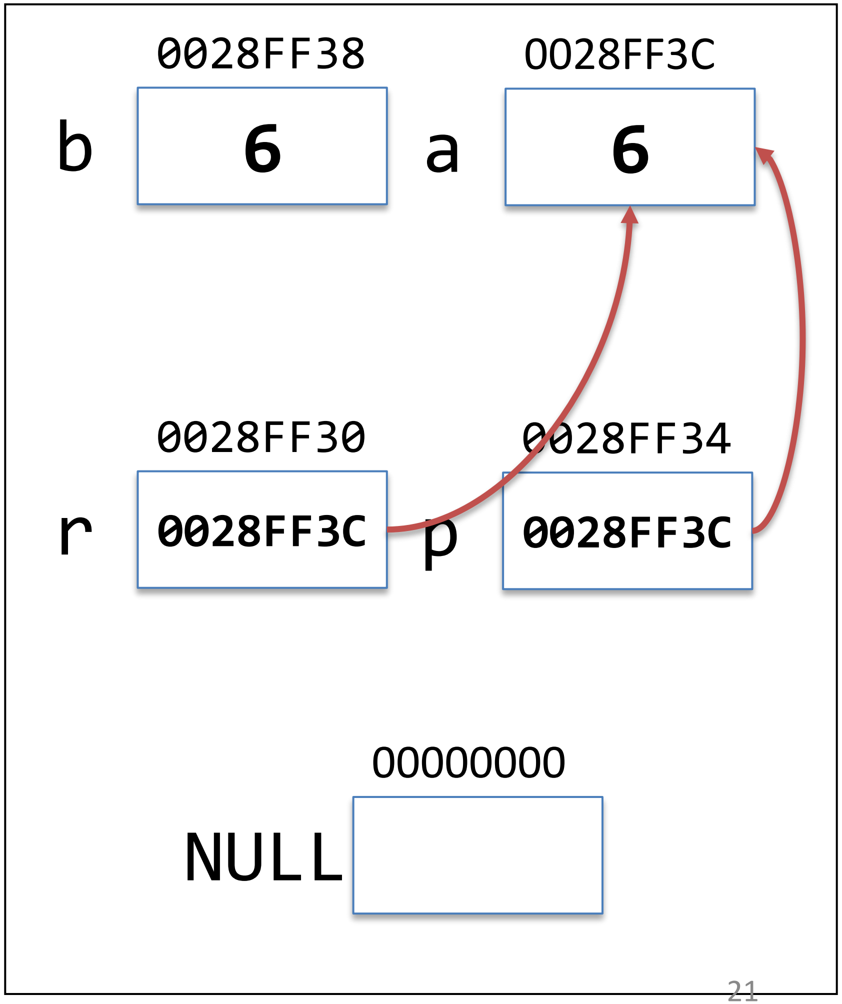


Working with pointers and addresses

Symbolic

```
int a = 6;
int b = 10;
int *p = NULL;
int *r = NULL;
p = &a;
r = p; //points at same
      //variable
b = *r;
p = &b;
*p = 8;
```

Memory

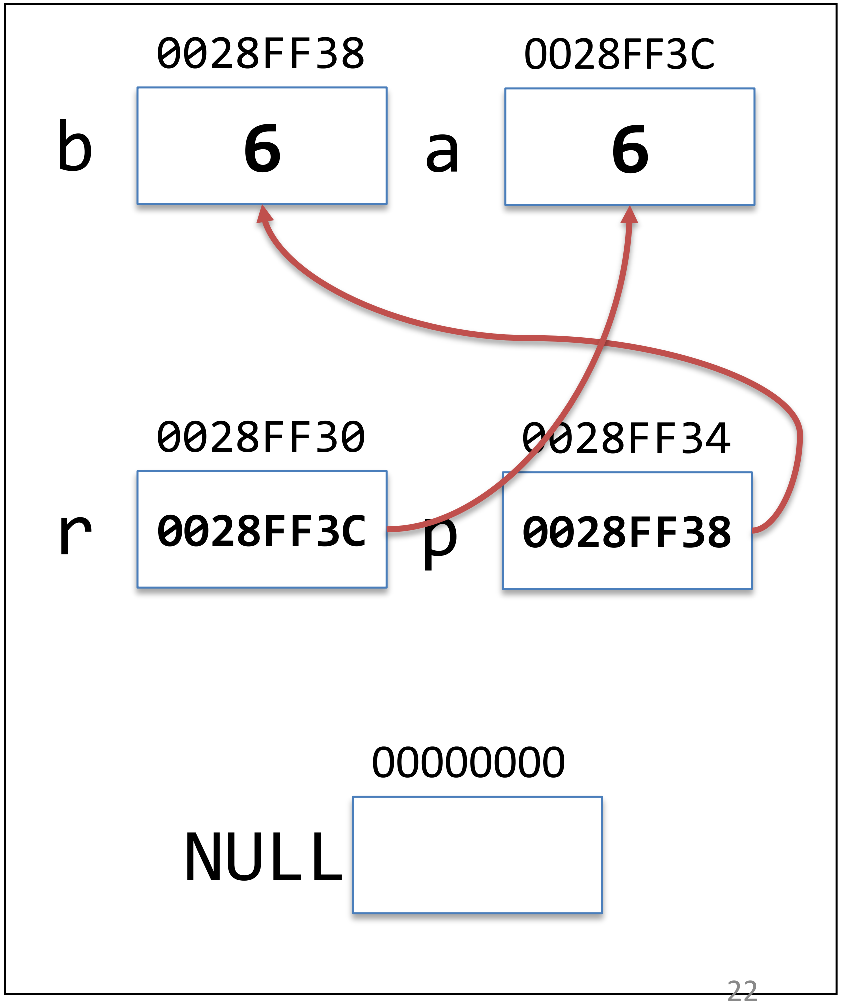


Working with pointers and addresses

Symbolic

```
int a = 6;
int b = 10;
int *p = NULL;
int *r = NULL;
p = &a;
r = p; //points at same
      //variable
b = *r;
p = &b;
*p = 8;
```

Memory

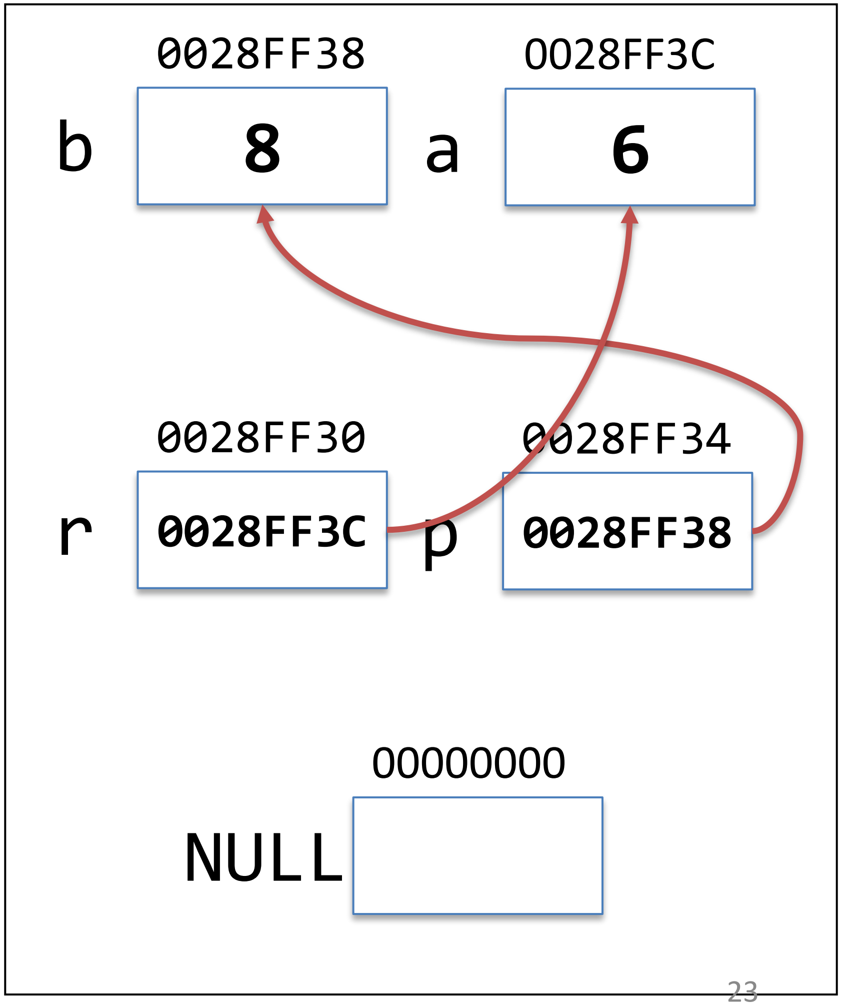


Working with pointers and addresses

Symbolic

```
int a = 6;
int b = 10;
int *p = NULL;
int *r = NULL;
p = &a;
r = p; //points at same
      //variable
b = *r;
p = &b;
*p = 8;
```

Memory



Pointers - flexibility

- Given these declarations and initialization…

```
int a = 0;
int *p = &a;
```

- … all of the following statements are equivalent

```
a = a + 1;
a++;
*p = *p + 1;
(*p)++;                  //note the ()
a = *p + 1;
```

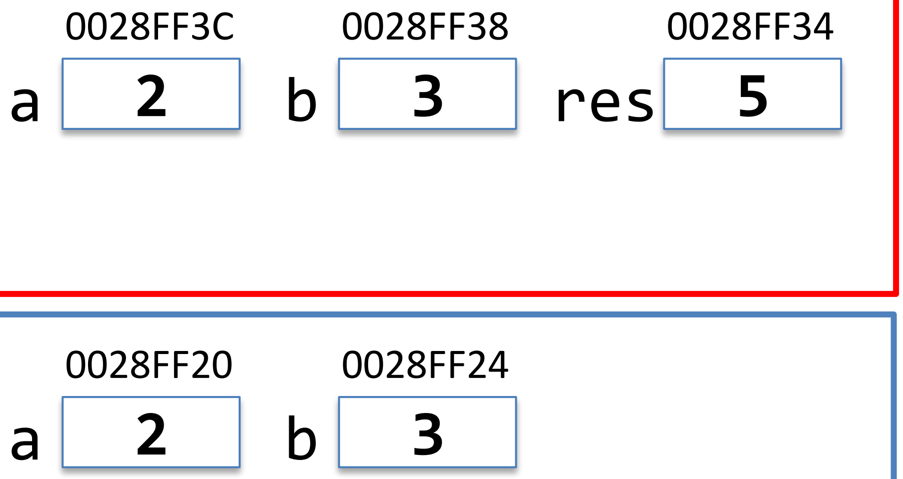
Why?

So, we can write `(*p)++` instead of `a++`, but is that the only use for pointers?

Pointers Meet Functions

Passing parameters by value

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

int add(int a, int b);

int main(void) {
    int a = 2;
    int b = 3;
    int res = add(a, b);
    return EXIT_SUCCESS;
}

int add(int a, int b) {
    return a + b;
}
```



0028FF3C 0028FF38 0028FF34

a 2 b 3 res 5

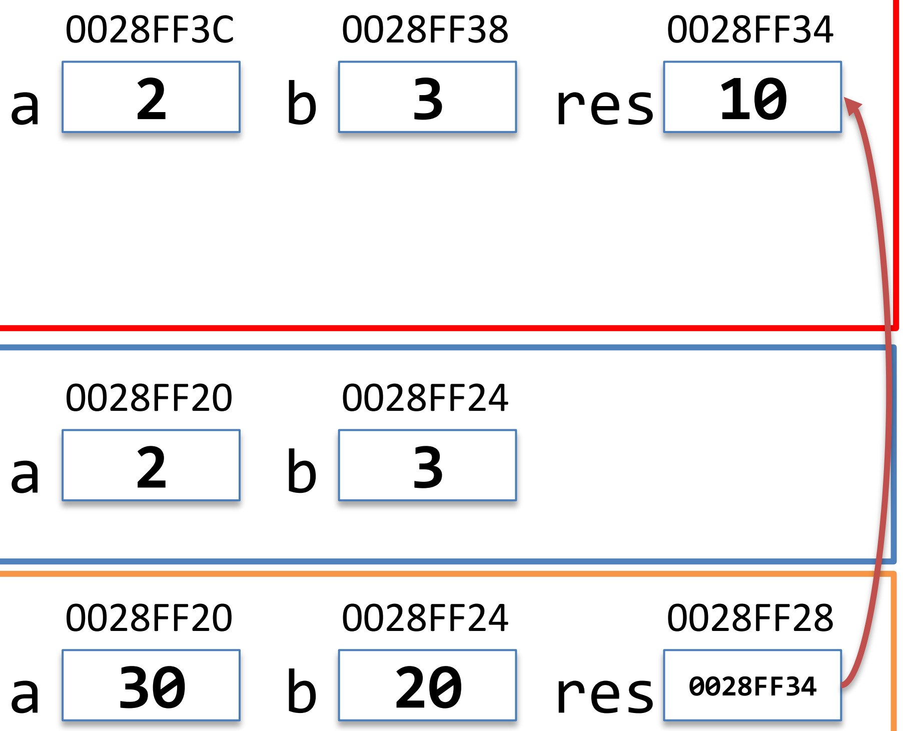
0028FF20 0028FF24

a 2 b 3

Pointers Meet Functions

Passing parameters by reference

```
…

int main(void) {
    int a = 2;
    int b = 3;
    int res = add(a, b);
    sub(30, 20, &res);
    return EXIT_SUCCESS;
}

int add(int a, int b) {
    return a + b;
}

void sub(int a, int b, int *res) {
    *res = a - b;
}
```



0028FF3C 0028FF38 0028FF34

a 2 b 3 res 10

0028FF20 0028FF24

a 2 b 3

0028FF20 0028FF24 0028FF28

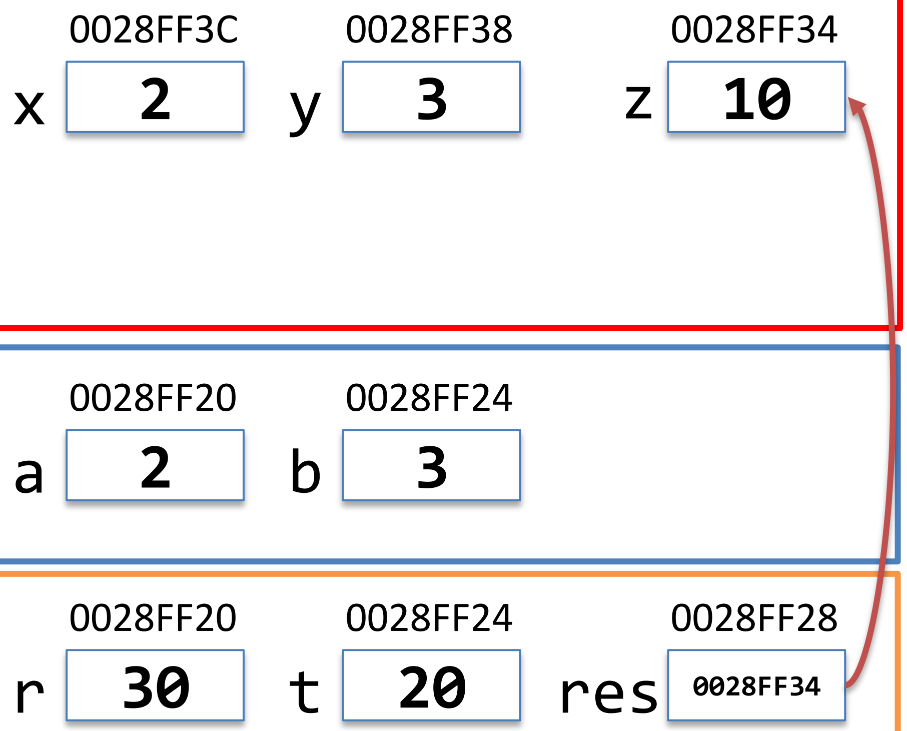
a 30 b 20 res 0028FF34

Pointers Meet Functions

Passing parameters by reference

```
…

int main(void) {
    int x = 2;
    int y = 3;
    int z = add(x, y);
    sub(30, 20, &z);
    return EXIT_SUCCESS;
}

int add(int a, int b) {
    return a + b;
}

void sub(int r, int t, int *res) {
    *res = r - t;
}
```

Name of the variables in the functions do not have to be the same.



0028FF3C 0028FF38 0028FF34

x 2 y 3 z 10

0028FF20 0028FF24

a 2 b 3

0028FF20 0028FF24 0028FF28

r 30 t 20 res 0028FF34

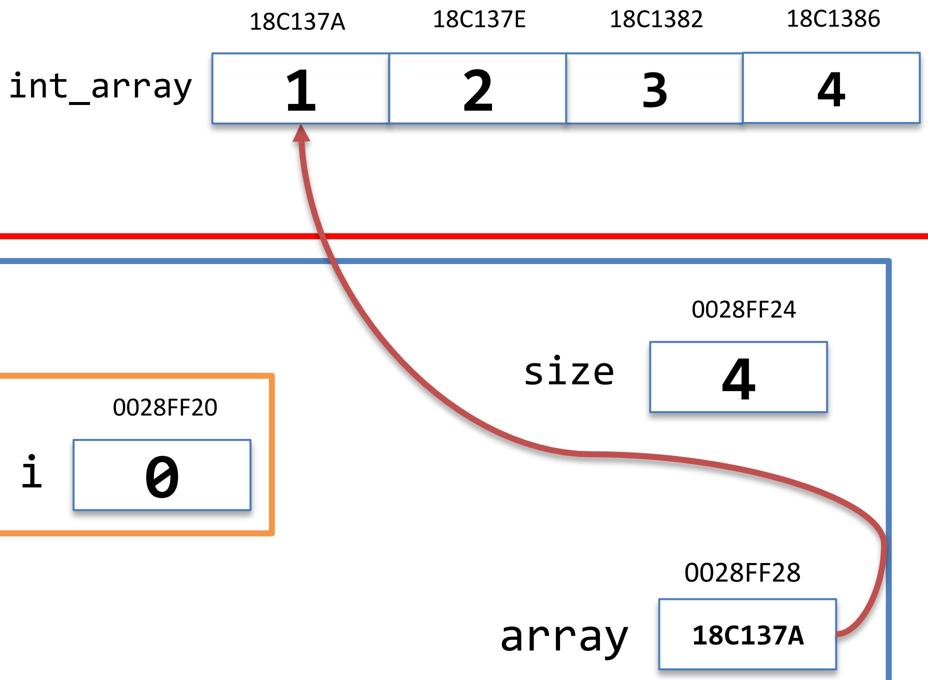
Pointers Meet Functions

- Only pass pointers (parameter by reference) when it is necessary to have access to the memory location denoted by the variable
  - when you want to change the value of the actual parameter
- If there is no need to modify it, pass your parameter by value
  - you will therefore protect the actual memory location from unintended changes

Pointers and Array

In fact, the name of an array is a constant pointer.

```
…

int main() {
    int int_array[4] = { 1, 2, 3, 4 };
    print_array(int_array, 4);
    return EXIT_SUCCESS;
}

void print_array(int *array, int size) {
    printf("[ ");
    for(int i = 0; i < size; i++){
        printf("%d ", array[i]);
    }
    printf("]");
}
```



int_array

18C137A	18C137E	18C1382	18C1386
1	2	3	4

i — 0028FF20: 0

size — 0028FF24: 4

array — 0028FF28: 18C137A

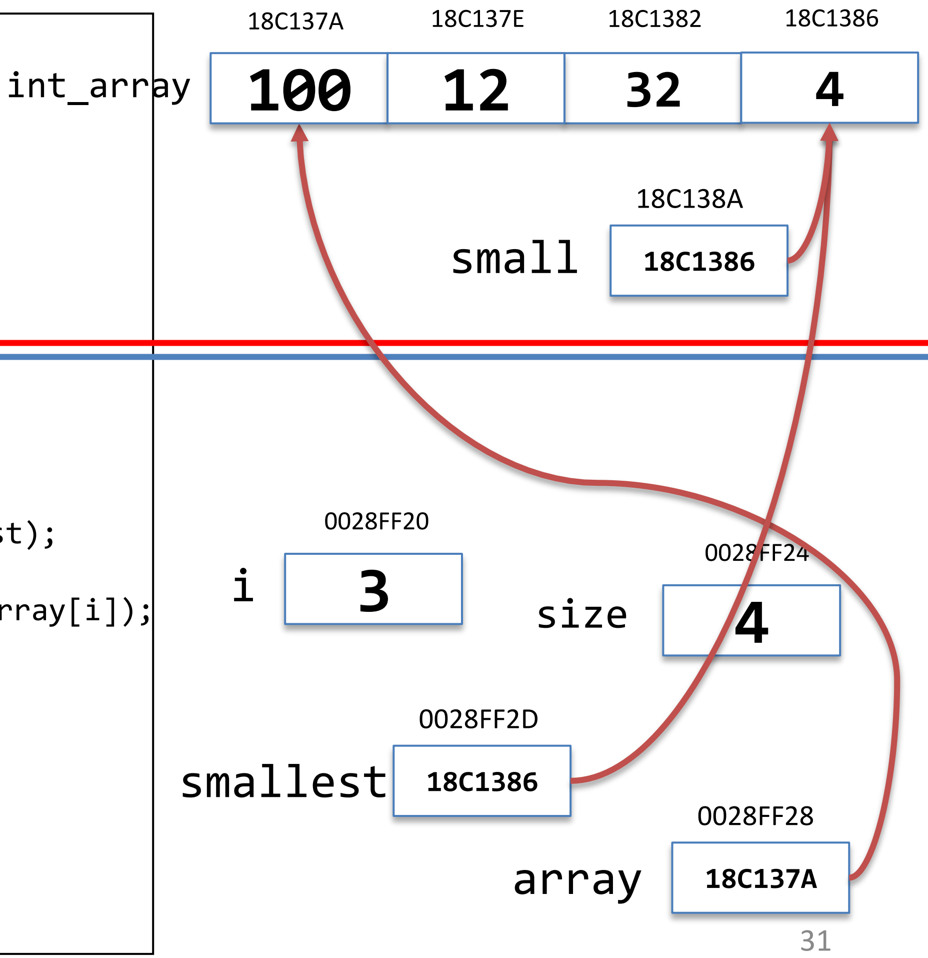
Pointers and Array

We cannot return an array from a function, we can return a pointer

Note that indexes can be used with pointers. It is basically an offset.

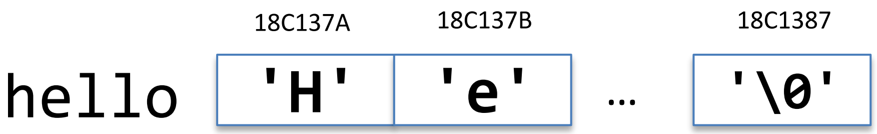
```
…

int main() {
     int int_array[4] = { 100, 12, 32, 4 };
     int *small = find_smallest(int_array, 4);
     printf("The smallest value %d\n", *small);
     return EXIT_SUCCESS;
}
```

```
int *find_smallest(int *array, int size) {
     int *smallest = NULL;
     smallest = &array[0];
     for (int i = 1; i < size; i++) {
          printf("Current smallest is %d\n", *smallest);
          if (*smallest > array[i]) {
               printf("Found a smaller one: %d\n", array[i]);
               smallest = &array[i];
          }
     }
     return smallest;
}
```

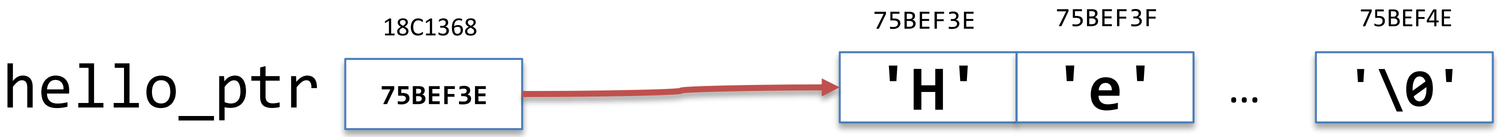


Strings Revisited

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

int main() {
    char hello[] = "Hello World!\n";
    char *hello_ptr = "Hello pointers!\n";
    printf("%s", hello);
    printf("%s", hello_ptr);
    return EXIT_SUCCESS;
}
```

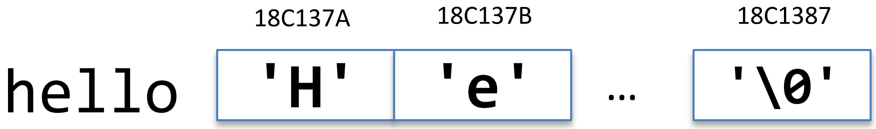


We can specify a size when declaring a string as an array, we cannot do that when declaring the string as a pointer



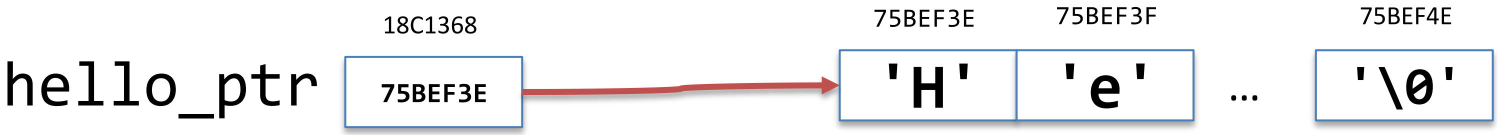
Strings Revisited

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

int main() {
    char hello[] = "Hello World!\n";
    char *hello_ptr = "Hello pointers!\n";
    hello = "Bonjour monde!\n";
    hello_ptr = "Bonjour monde!\n";
    return EXIT_SUCCESS;
}
```



hello

18C137A	18C137B		18C1387
'H'	'e'	…	'\0'



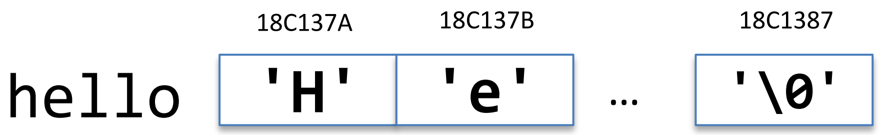
hello_ptr

18C1368
75BEF3E

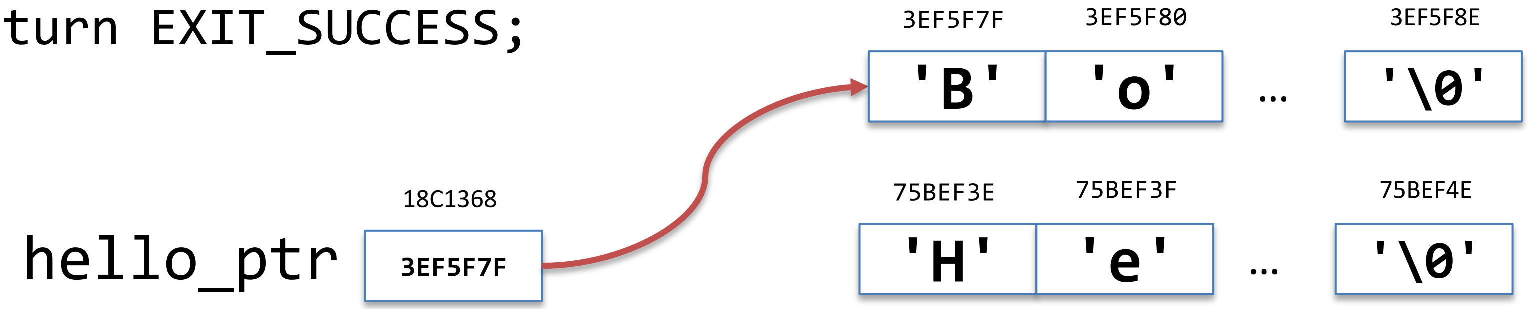
75BEF3E	75BEF3F		75BEF4E
'H'	'e'	…	'\0'

Strings Revisited

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

int main() {
    char hello[] = "Hello World!\n";
    char *hello_ptr = "Hello pointers!\n";
    hello = "Bonjour monde";
    hello_ptr = "Bonjour monde!\n";
    return EXIT_SUCCESS;
}
```



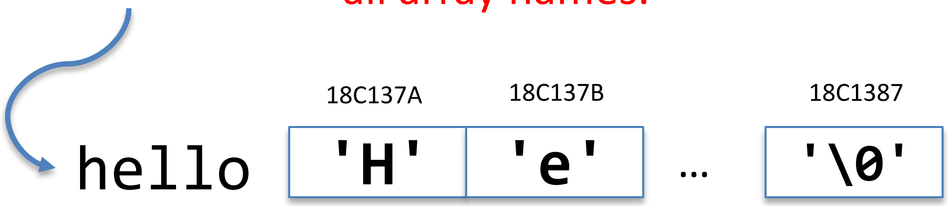
error: array type 'char [14]' is not assignable



Strings Revisited

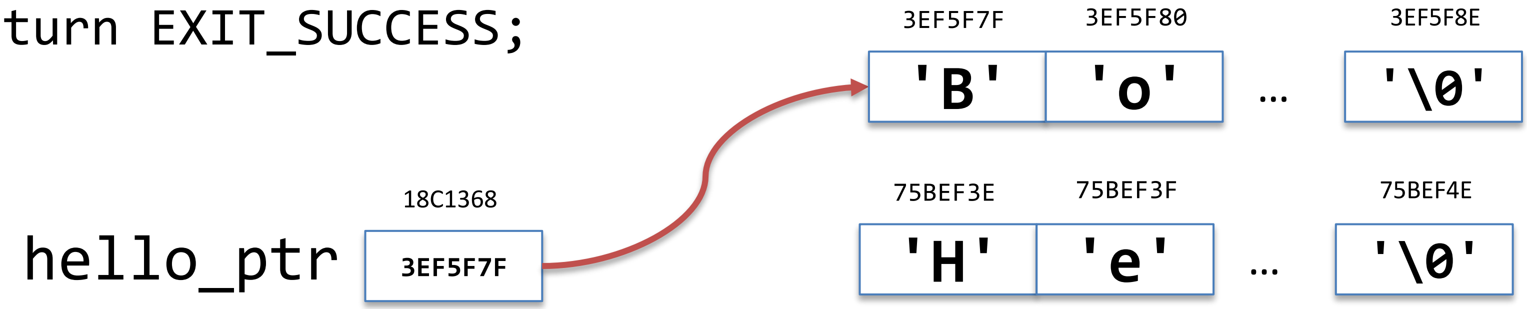
It is constants

Hello is a constant pointer and so are all array names.

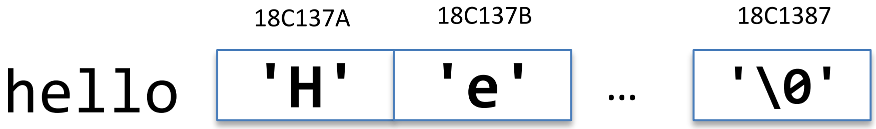


```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char hello[] = "Hello World!\n";
    char *hello_ptr = "Hello pointers!\n";
    hello = "Bonjour monde";
    hello_ptr = "Bonjour monde!\n";
    return EXIT_SUCCESS;
}
```

```
error: array type 'char [14]' is not
assignable
```



Strings Revisited

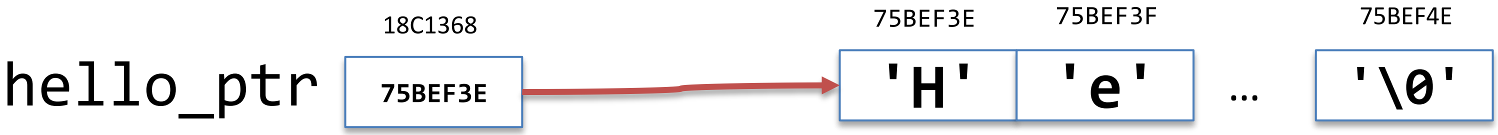


18C137A 18C137B 18C1387

hello 'H' 'e' … '\0'

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

int main() {
    char hello[] = "Hello World!\n";
    char *hello_ptr = "Hello pointers!\n";
    printf("%c\n", hello[1]);
    printf("%c\n", hello_ptr[1]);
    return EXIT_SUCCESS;
}
```

It is possible to use indexes with pointers



18C1368 75BEF3E 75BEF3F 75BEF4E

hello_ptr 75BEF3E

'H' 'e' … '\0'

Pointer types

- Pointers point to a variable of a **specific type**.

- So you cannot mix pointer types in statements:

```
int *p;
char *r;
…
r = p; //compile warning
```

- The only exception to this is the `void` pointer type (more on that later)
- A pointer takes on the attributes of the type it points to, as well as the attributes of a pointer

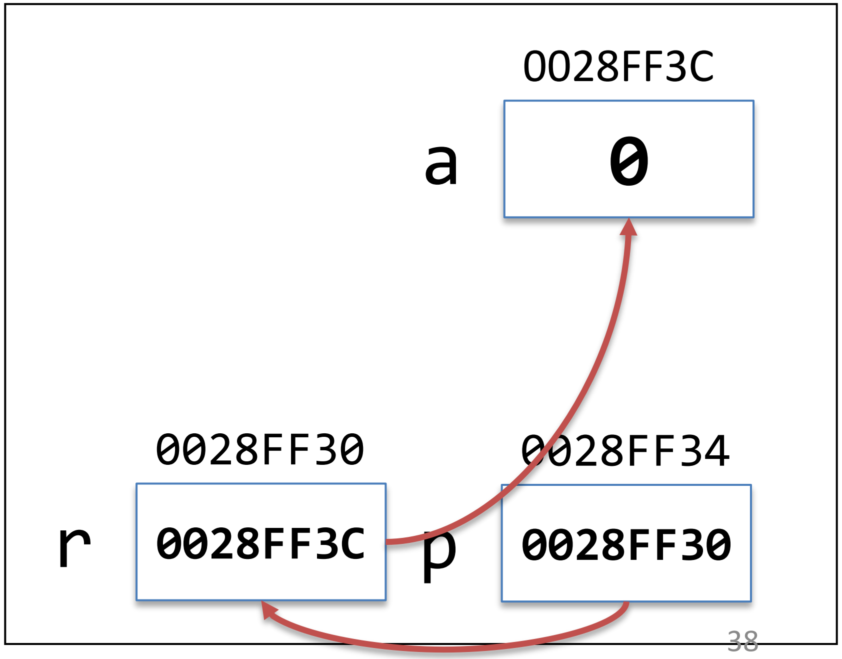
Pointers to pointers

A pointer to a pointer represents two levels of indirection

Symbolic

```
int** p;
int* r;
int a = 0;
r = &a;
p = &r;
**p = 5;
```

Memory



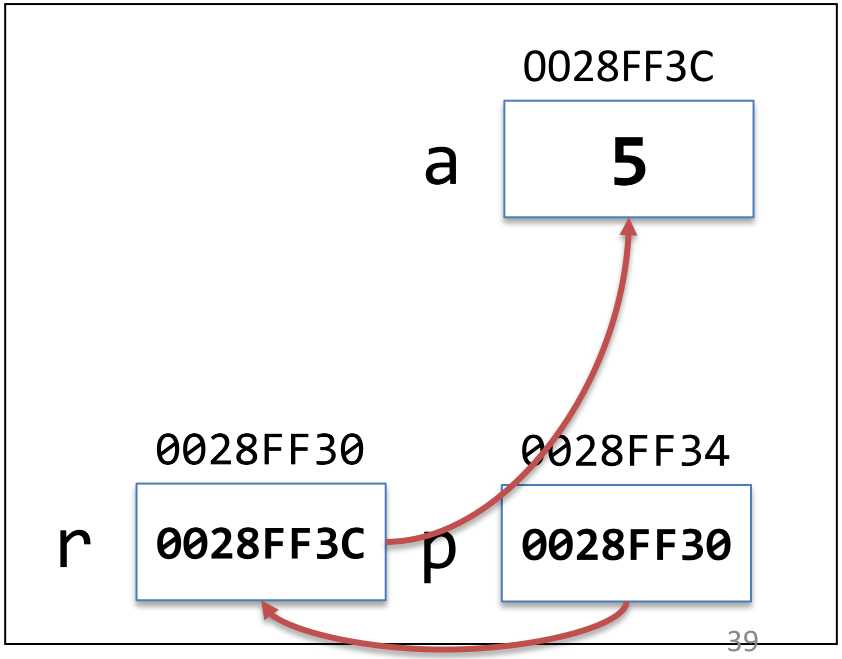
Pointers to pointers

A pointer to a pointer represents two levels of indirection

Symbolic

```
int** p;
int* r;
int a = 0;
r = &a;
p = &r;
**p = 5;
```

Memory

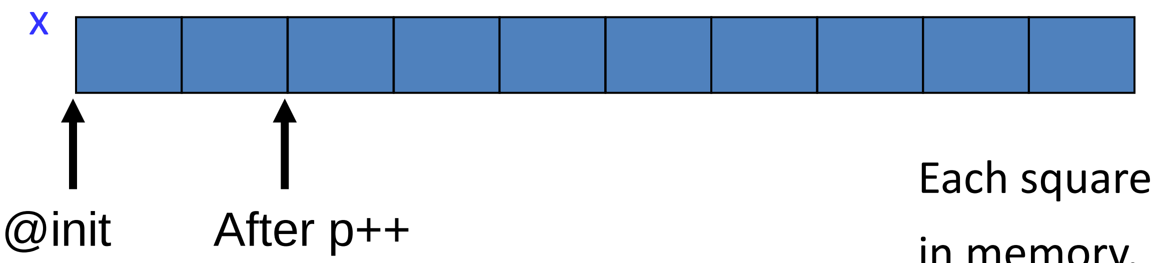


Pointer arithmetic

- Pointers have types because they *point to a type*. This is important when we do pointer arithmetic:

```
int x[5] = {1,2,3,4,5};
int* p = x;
p++;
```

This moves the pointer ahead by two bytes



Each square is a byte in memory.

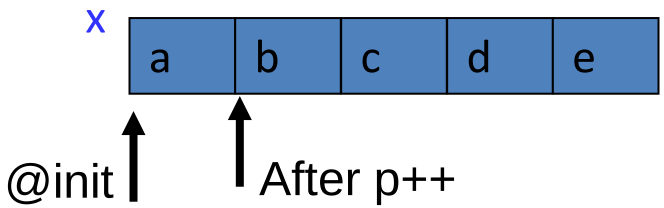
Pointer arithmetic

- Note there is no \0 here
- Dangerous to print or try a string operation on.

And:

```
char x[5] = {'a', 'b', 'c', 'd', 'e'};
char* p = x;
…
p++;
```

This moves the pointer ahead one byte



Each square is a byte in memory.

Pointer arithmetic

Only a few arithmetic operations can be performed on pointers:

```
int *p;
```

- **Postfix:** `p++, p--`
- **Adding an index (an `int`) to a pointer:**

  `p + 5` (advances by 5 positions)
- **Subtracting an index:** `p - 5`
- **Subtracting pointers:** `p1 - p2`
  - Gives the number of ***positions*** between two pointers – useful to calculate offsets in arrays (distance between elements)

Pointer arithmetic

You cannot add, multiply or divide two pointers

- What good could come out of it?
- Pointedly, something that is pointless.

Quiz Time

```
int a = 1;
int b = 2;
int *p = NULL;
int *r = NULL;
…
p = &b;
r = &a;
*r = *p;
```

What are the values of a and b?

Questions?