# COMPUTER SYSTEMS AND ORGANIZATION

## C compilation

Daniel G. Graham Ph.D

UNIVERSITY of VIRGINIA | ENGINEERING

# Contents

1. Types in C
2. Pointers  (Review)
3. Swap Example (Review)
4. Pointers and Arrays
5. Strings the begin

# TYPES IN C

| type | size (bytes) |
|---|---|
| char | 1 |
| short | 2 |
| int | 4 |
| long | 8 |
| float | 4 |
| double | 8 |

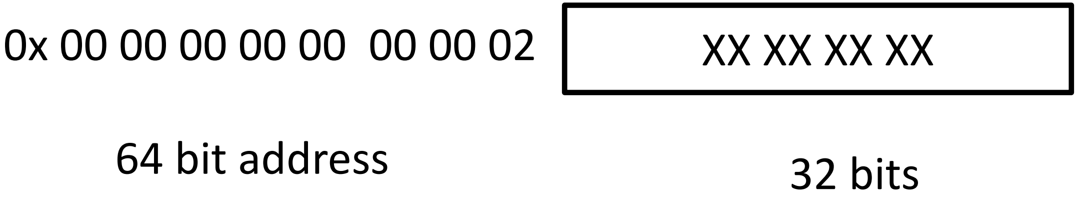
```
int x = 3;
int number_of_bytes = sizeof(x)

char letter = ‘A’
int number_of_bytes = sizeof(letter)
```

# PRINTF

| Specifier | Argument | Type Example(s) |
|---|---|---|
| %s | char * | Hello, World! |
| %p | any pointer | 0x4005d4 |
| %d | int/short/char | 42 |
| %u | unsigned int/short/char | 42 |
| %x | unsigned int/short/char | 2a |
| %ld | long | 42 |
| %f | double/float | 42.000000 |
| %e | double/float | 4.200000e-19 |
| %% | (no argument) | % |

# THIS DECLARES A VARIABLE

```
int variable;
```



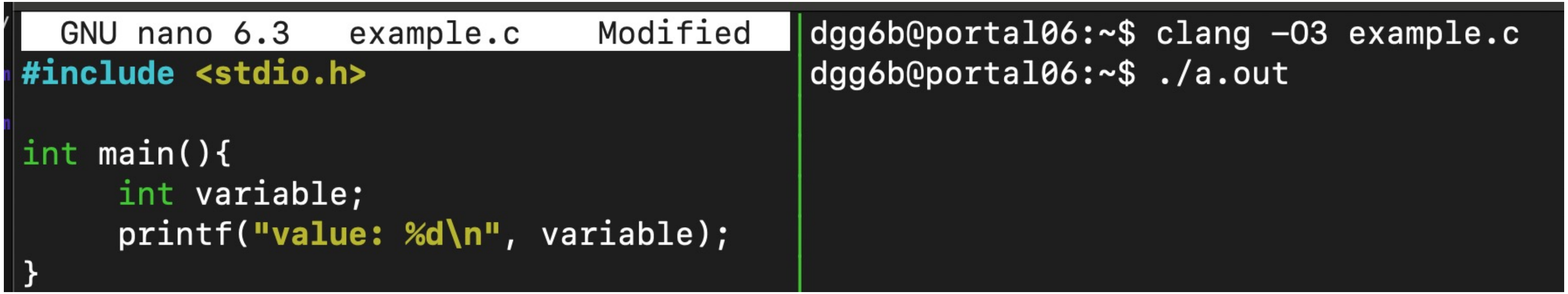
0x 00 00 00 00 00  00 00 02

XX XX XX XX

64 bit address

32 bits

# WHAT GET’S PRINTED?

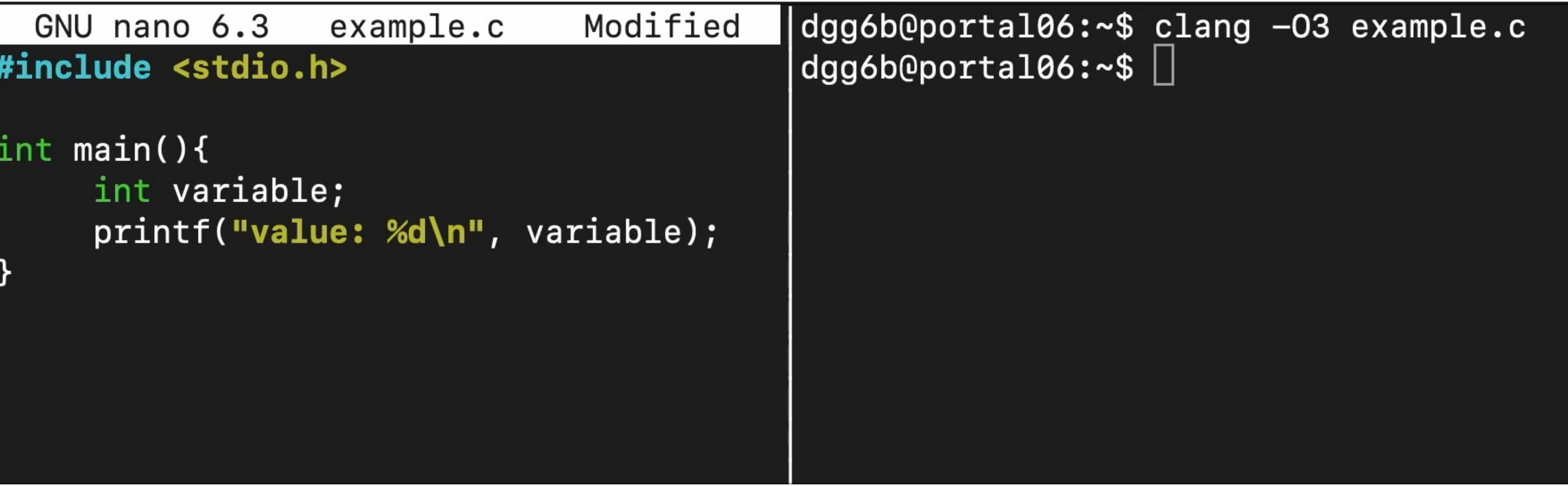


```
  GNU nano 6.3     example.c      Modified
#include <stdio.h>

int main(){
    int variable;
    printf("value: %d\n", variable);
}
```

```
dgg6b@portal06:~$ clang -O3 example.c
dgg6b@portal06:~$ ./a.out
```

Is it the same every time we run the program?
What if we didn’t optimize the program?

# WHAT GET’S PRINTED?



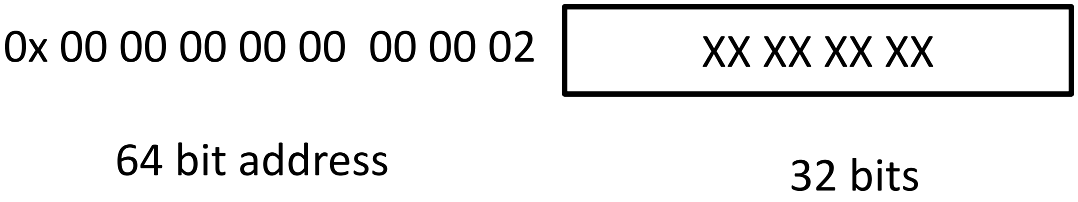
```
 GNU nano 6.3    example.c     Modified
#include <stdio.h>

int main(){
     int variable;
     printf("value: %d\n", variable);
}
```

```
dgg6b@portal06:~$ clang -O3 example.c
dgg6b@portal06:~$ 
```

Try not use uninitialized variables

# THIS DECLARES A VARIABLE

```
int variable;
```



0x 00 00 00 00 00  00 00 02

XX XX XX XX

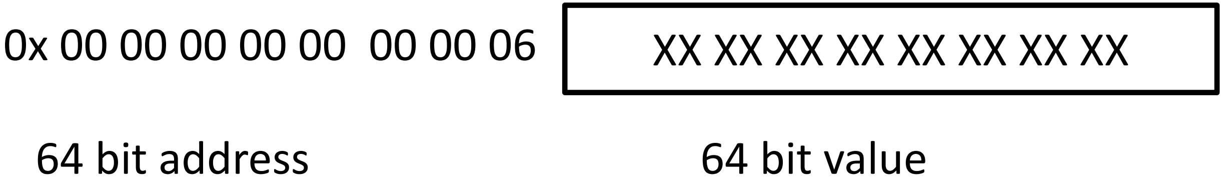
64 bit address

32 bits

# THIS DECLARES A POINTER

```
int *pointer;
```

Be careful with uninitialized pointers



0x 00 00 00 00 00  00 00 06

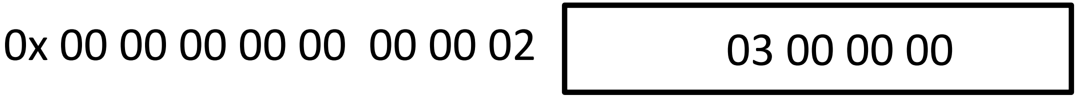
XX XX XX XX XX XX XX XX

64 bit address

64 bit value

# THIS INITIALIZES A VARIABLE

```
int variable = 3;
```

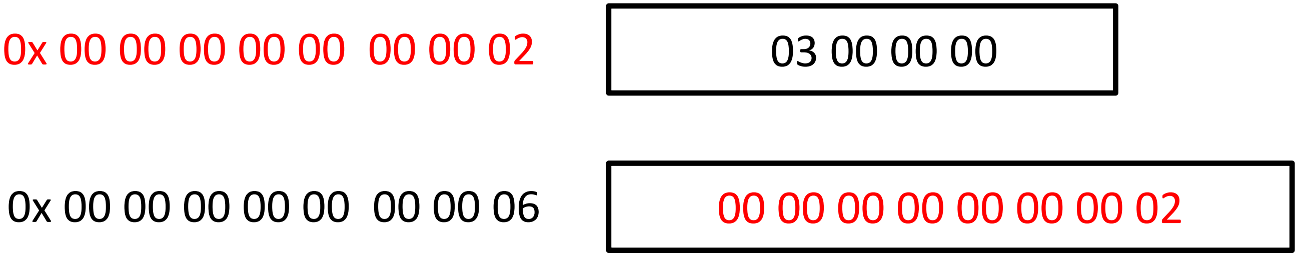


0x 00 00 00 00 00  00 00 02

03 00 00 00

# THIS INITIALIZES A POINTER

```
int *pointer = &variable;
```

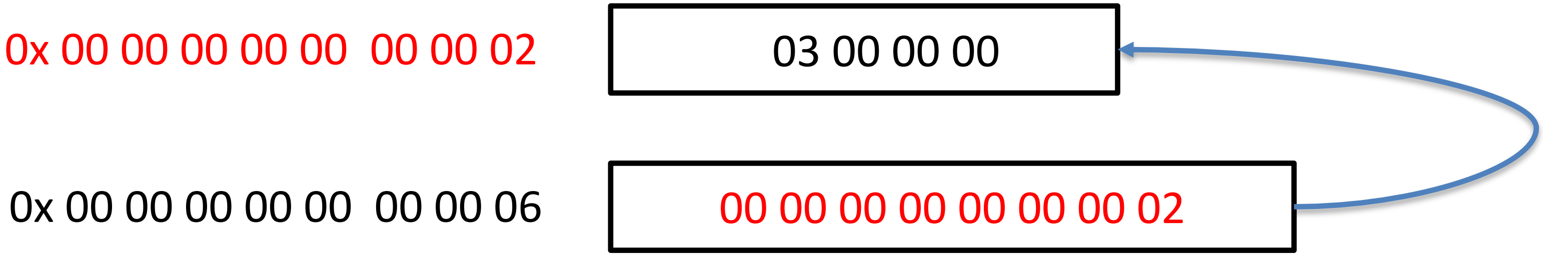


0x 00 00 00 00 00  00 00 02	03 00 00 00

0x 00 00 00 00 00  00 00 06	00 00 00 00 00 00 00 02

# THIS INITIALIZES A POINTER

```
int *pointer = &variable;
```



0x 00 00 00 00 00  00 00 02

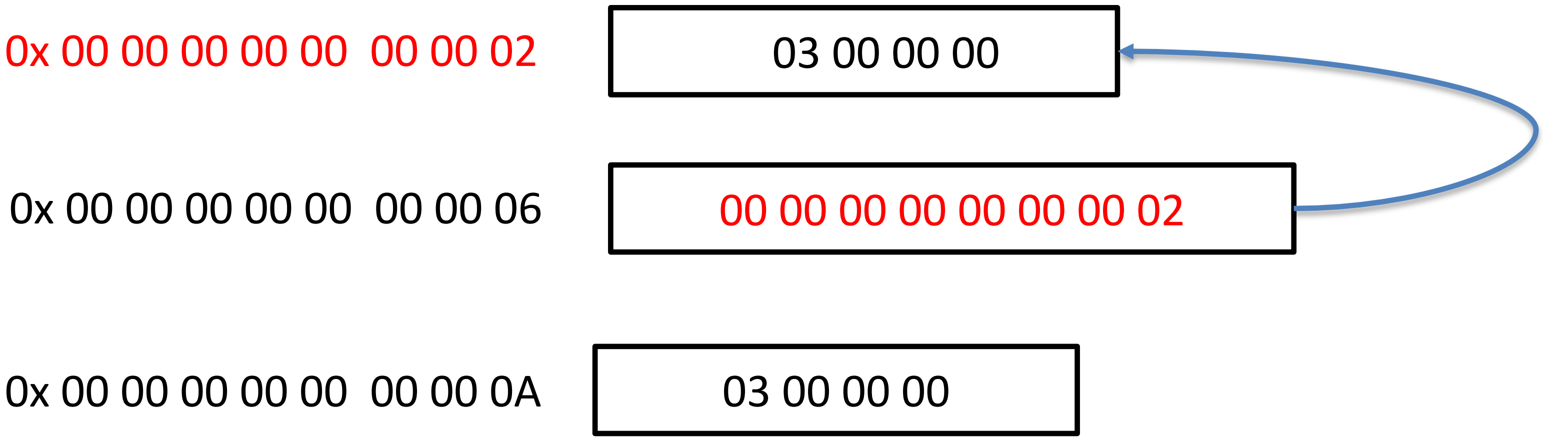
03 00 00 00

0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 02

# DEREFERENCE VALUE (USE)

```
int variable2 = *pointer;
```



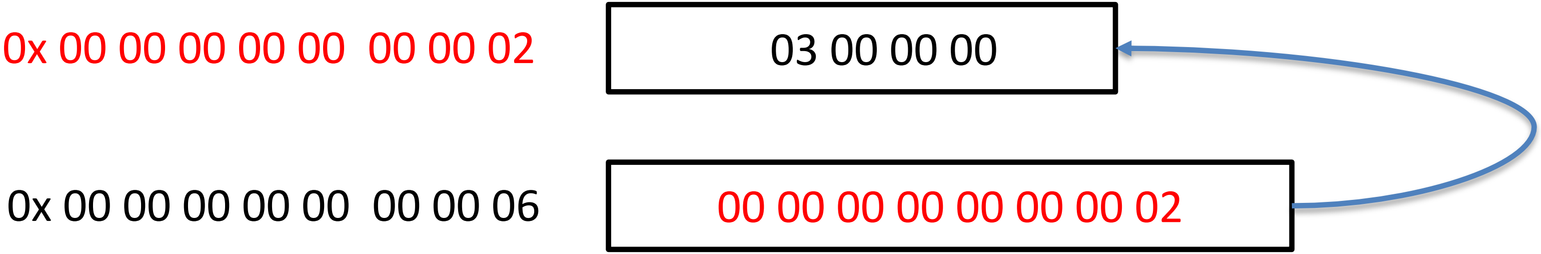
0x 00 00 00 00 00  00 00 02    03 00 00 00

0x 00 00 00 00 00  00 00 06    00 00 00 00 00 00 00 02

0x 00 00 00 00 00  00 00 0A    03 00 00 00

# ASSIGNMENT POINTER

int *pointer = &variable;



0x 00 00 00 00 00  00 00 02

03 00 00 00

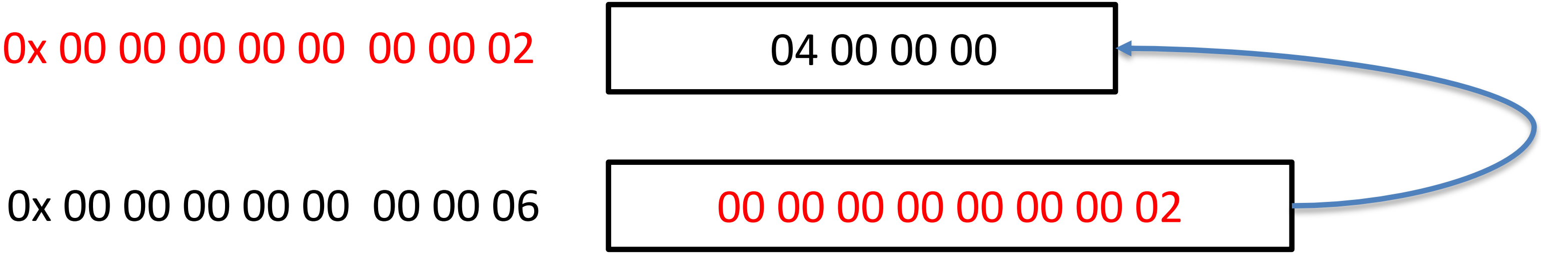
0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 02

*pointer = 4;

# ASSIGNMENT POINTER

```
int *pointer = &variable;
```



0x 00 00 00 00 00  00 00 02

04 00 00 00

0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 02

```
*pointer = 4;
```

# IF YOU MISS EVERYTHING FROM THE LECTURE JUST LISTEN TO THESE FOUR RULES

# POINTER RULES RULE 1

```
int *p;
```

If we have:

    type
    *
    variable_name

Then it is a declaration.

# POINTER RULES RULE 1

```
int *p;
```

0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 00

Location on the stack

Value at that location

Reserve a memory location on the stack to store an address

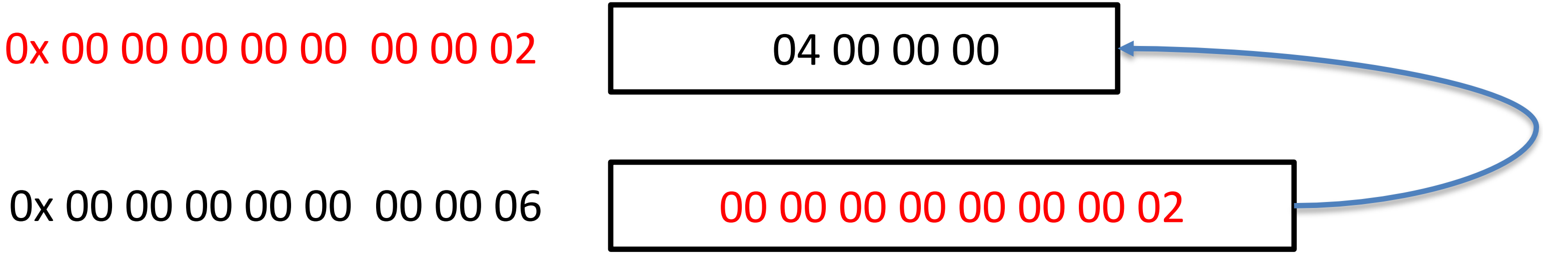
# POINTER RULES RULE 2

*p =

- * and a variable name on the left side of = means:
- **Go to** the address stored in p and **update** the value

# POINTER RULES RULE 2

*p =



0x 00 00 00 00 00  00 00 02 04 00 00 00

0x 00 00 00 00 00  00 00 06 00 00 00 00 00 00 00 02

# POINTER RULES RULE 3

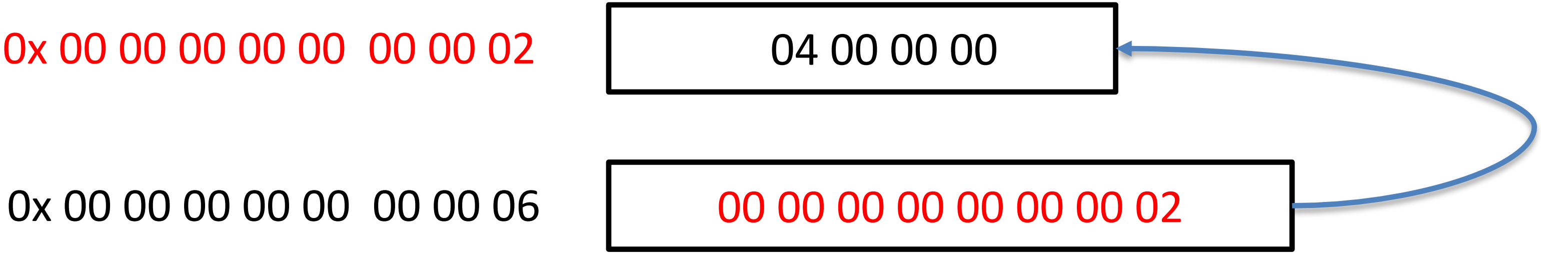
= *p

- * and a variable name on the right side of = or no = means:

- **Go to** the address stored in p and **retrieve** the value

# POINTER RULES RULE 3

= *p

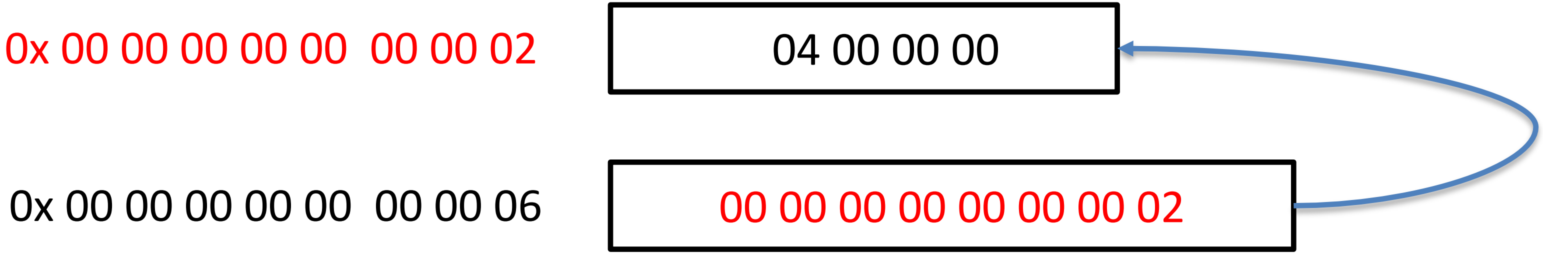


0x 00 00 00 00 00 00 00 02 04 00 00 00

0x 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00 02

# POINTER RULES RULE 3

= 4



# FINAL RULE

= &p

- & and a variable name on the right side of = means:
- **Get the address of variable**

# FINAL RULE

= &p

0x 00 00 00 00 00  00 00 06   00 00 00 00 00 00 00 00

# FINAL RULE

=0x…0006

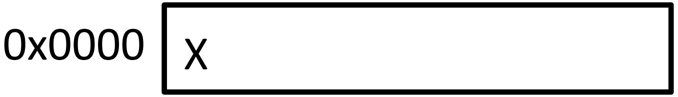
0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 00

# LET’S LOOK AT ANOTHER EXAMPLE

# POINTERS

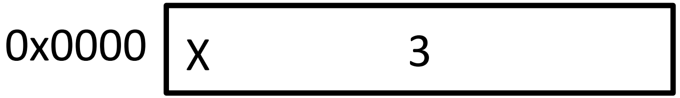
```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
Int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



0x0000 x

# POINTERS

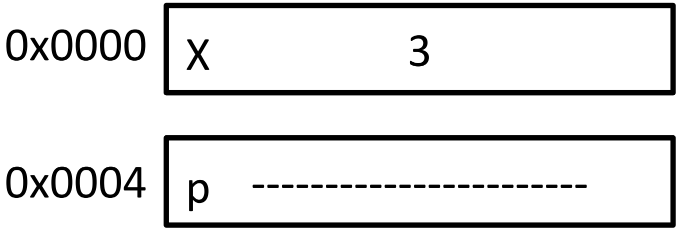
```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
Int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



0x0000 x 3

# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
Int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



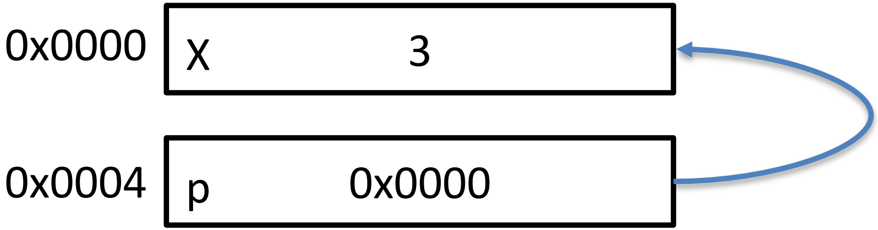
# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
Int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```

| | | |
|---|---|---|
| 0x0000 | X | 3 |
| 0x0004 | p | 0x0000 |

# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
Int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```

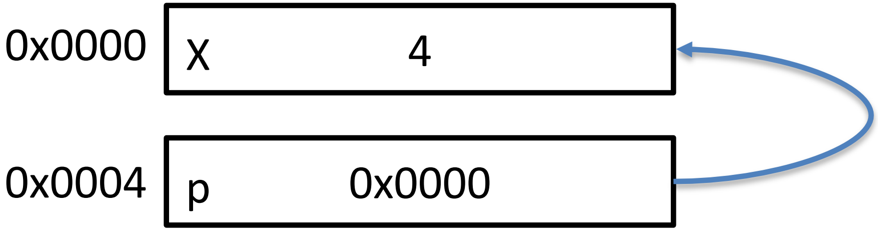


0x0000 X 3

0x0004 p 0x0000

# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
Int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```

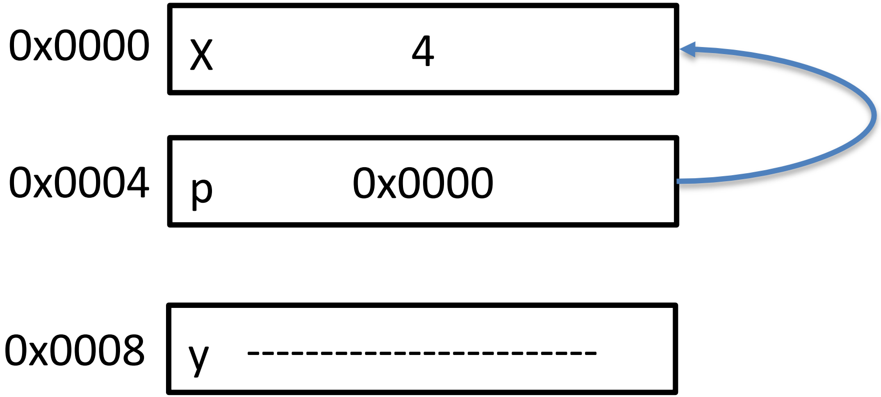


0x0000 X 4

0x0004 p 0x0000

# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



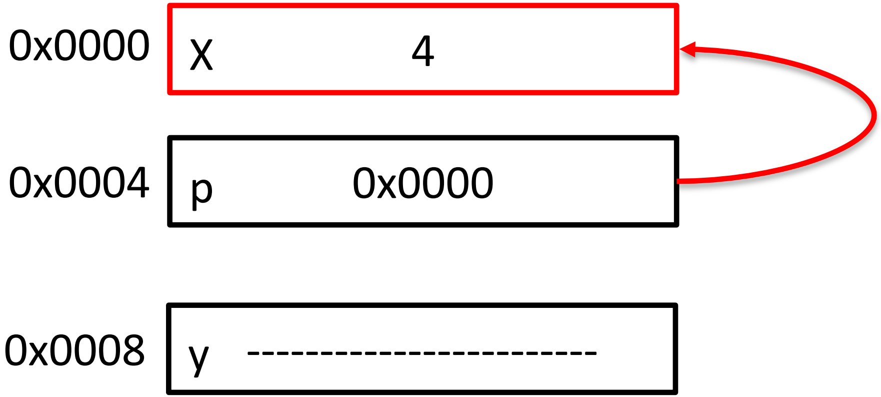
0x0000 x 4

0x0004 p 0x0000

0x0008 y ------------------------

# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



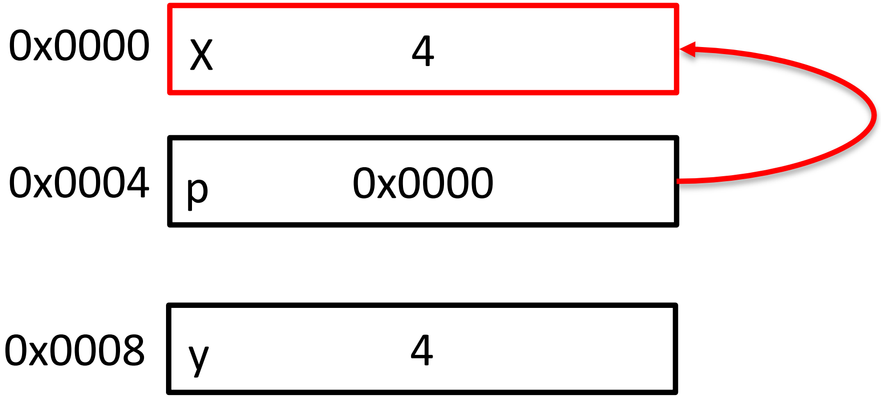
0x0000 x 4

0x0004 p 0x0000

0x0008 y ------------------------

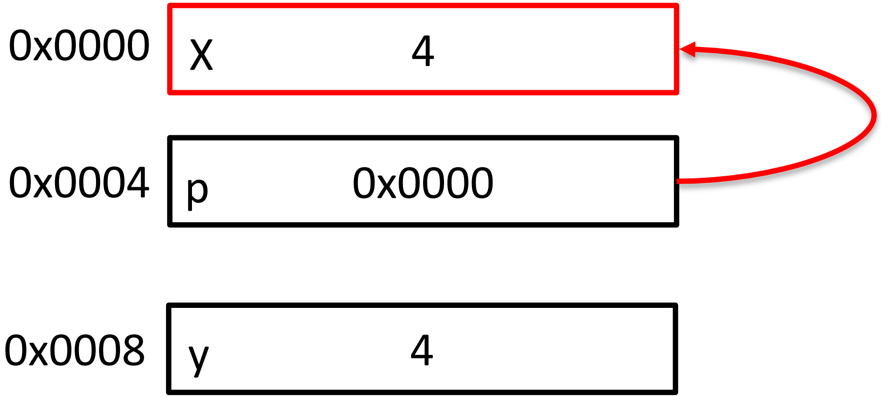
# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



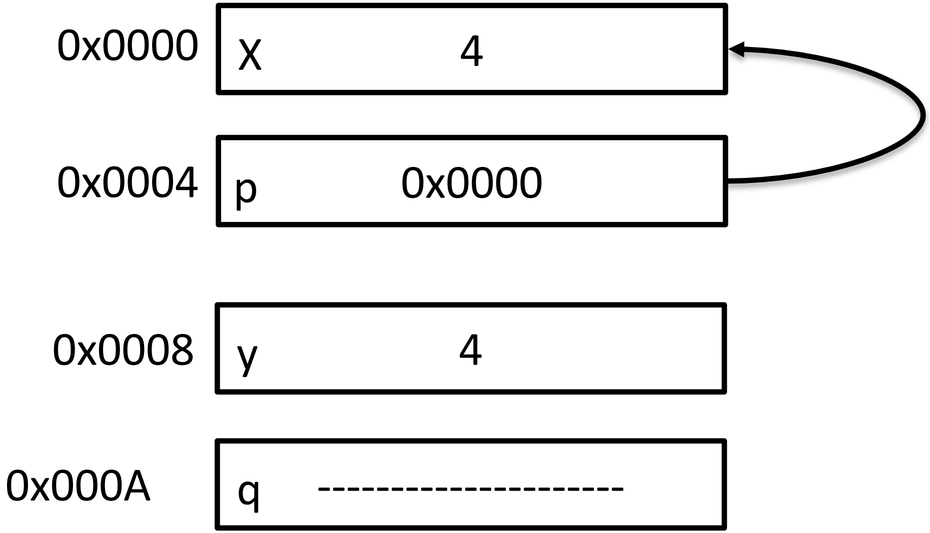
# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



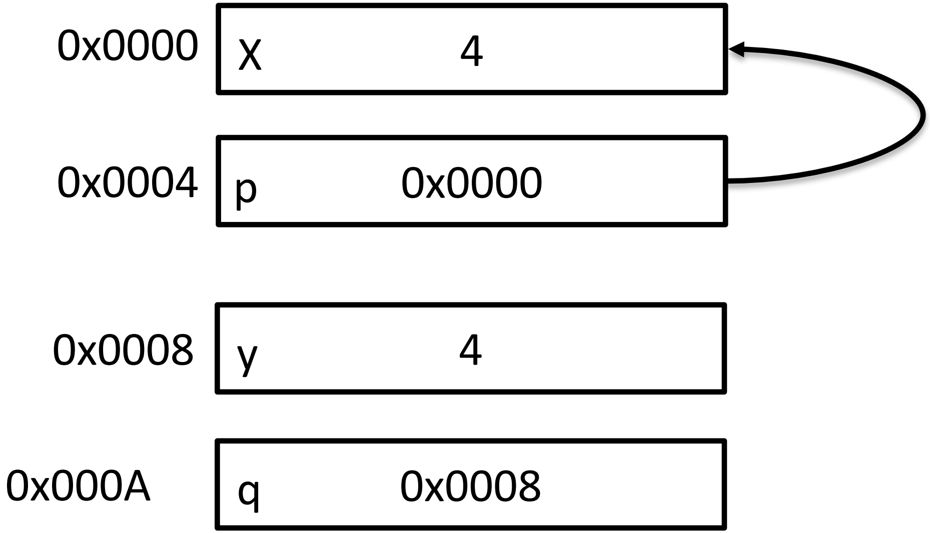
# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



0x0000 X 4

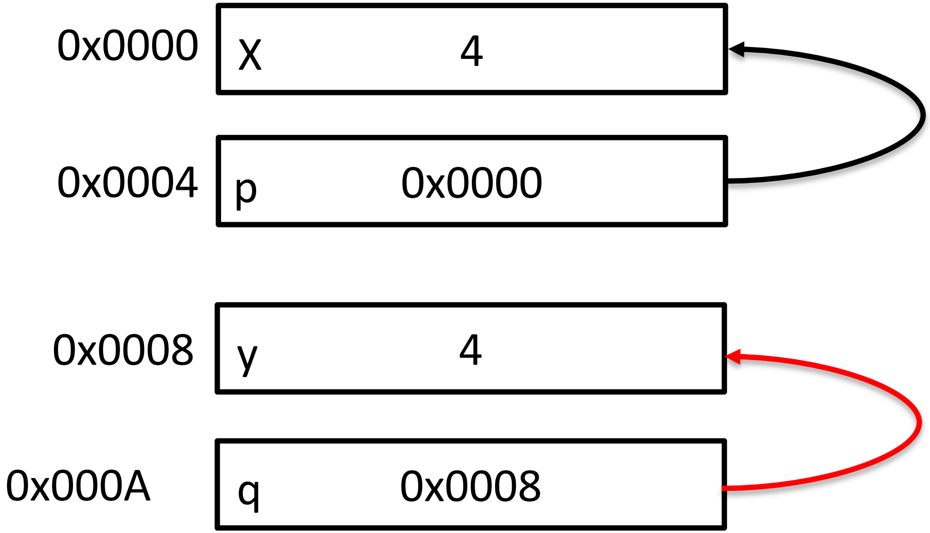
0x0004 p 0x0000

0x0008 y 4

0x000A q 0x0008

# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



0x0000 x 4

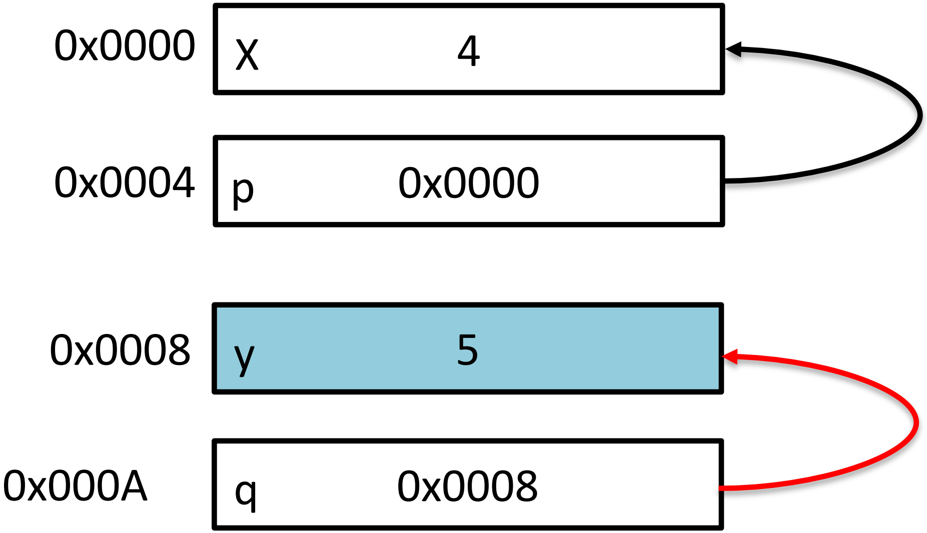
0x0004 p 0x0000

0x0008 y 4

0x000A q 0x0008

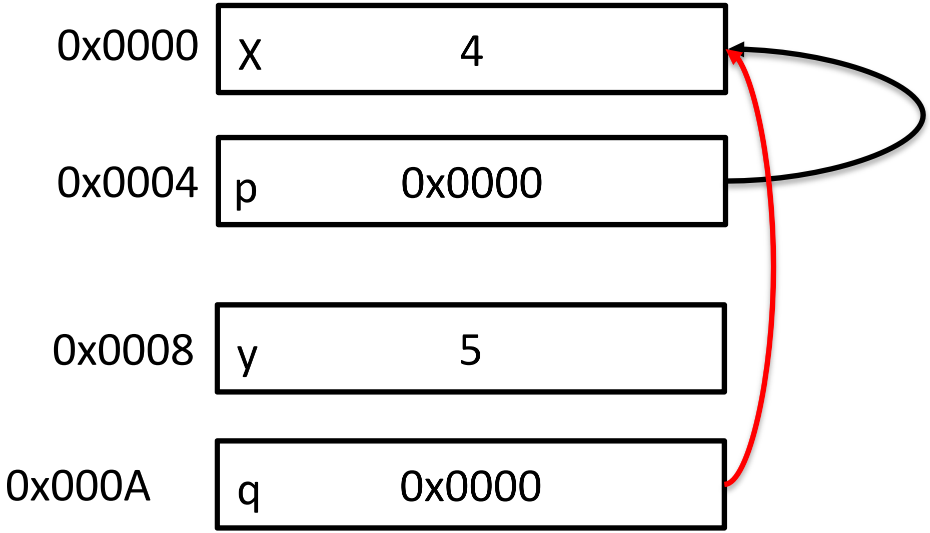
# POINTERS

```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



# POINTERS

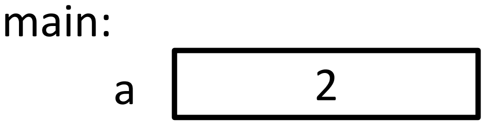
```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y
*q = *p + 1;
q = p;
```



# EVERYTHING IN C IS PASS BY VALUE

# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```

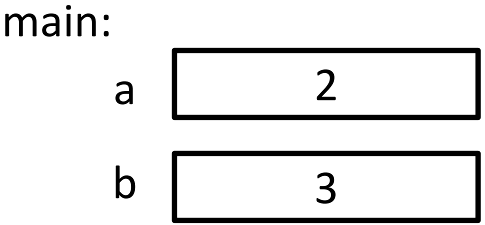


main:

a 2

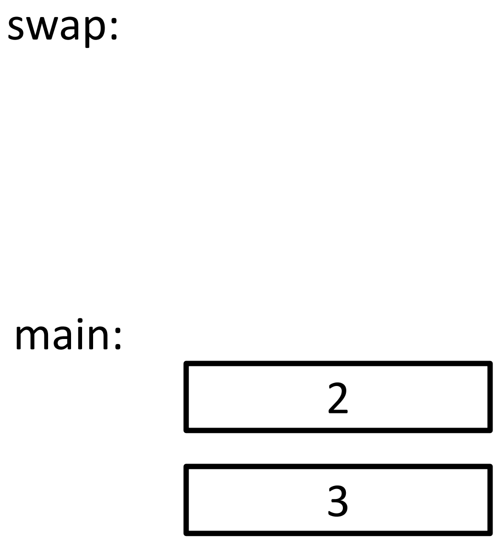
# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```



# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```



swap:

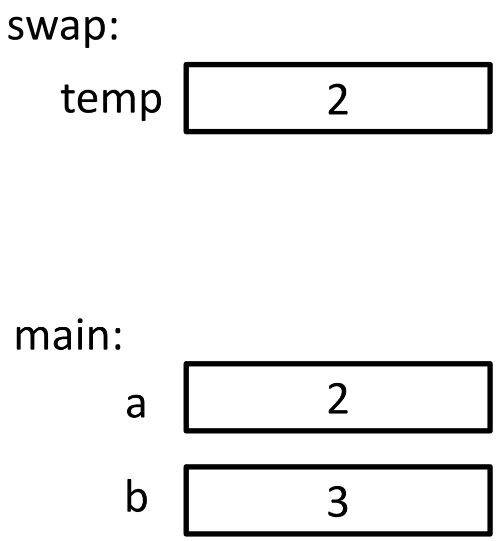
main:

2

3

# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```



swap:

temp 2

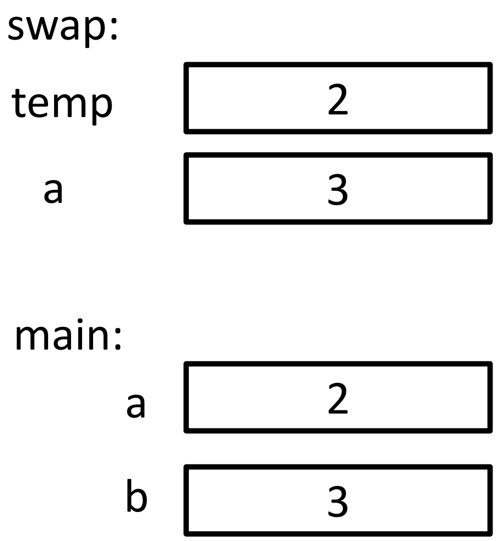
main:

a 2

b 3

# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```



swap:

temp 2

a 3

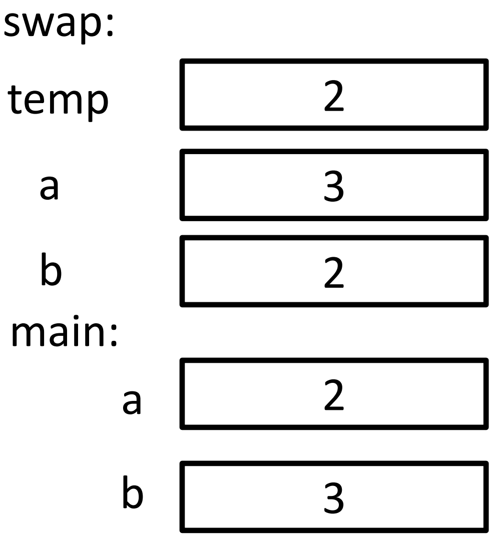
main:

a 2

b 3

# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```



swap:

temp 2

a 3

b 2

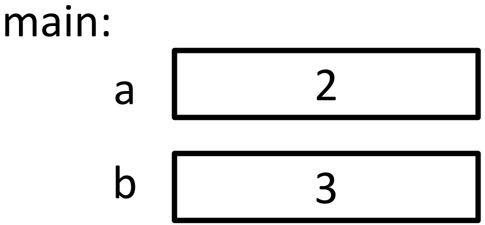
main:

a 2

b 3

# SWAP EXAMPLE (BAD)

```
void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(a, b);
    return 0;
}
```



main:

| | |
|---|---|
| a | 2 |
| b | 3 |

# WHAT IF WE PASS AN ADDRESS BY VALUE

# EVERYTHING IN C IS PASS BY VALUE

```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main() {
    int x = 2;
    myFunc(&x);
    printf("%d", x);
    return 0;
}
```

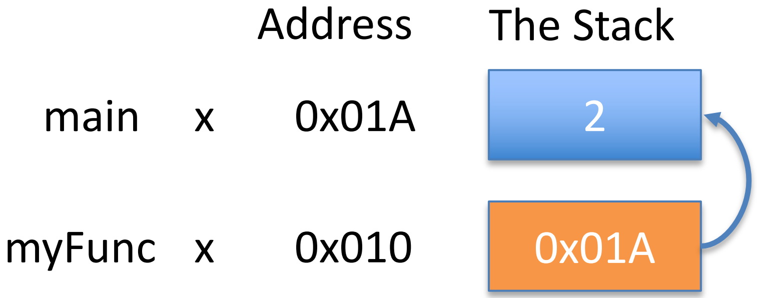
# EVERYTHING IN C IS PASS BY VALUE

```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main() {
    int x = 2;
    myFunc(&x);
    printf("%d", x);
    return 0;
}
```

| | | Address | The Stack |
|---|---|---|---|
| main | x | 0x01A | 2 |

# EVERYTHING IN C IS PASS BY VALUE

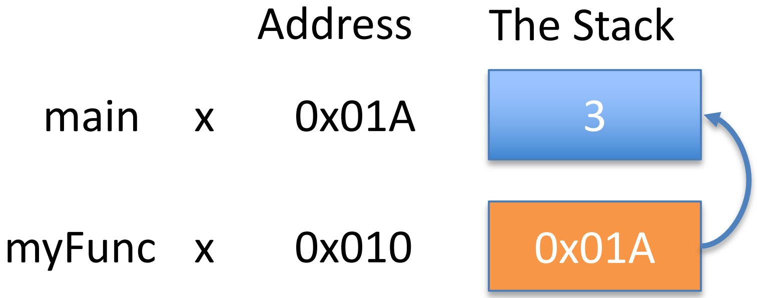
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main() {
    int x = 2;
    myFunc(&x);
    printf("%d", x);
    return 0;
}
```



| | | Address | The Stack |
|---|---|---|---|
| main | x | 0x01A | 2 |
| myFunc | x | 0x010 | 0x01A |

# EVERYTHING IN C IS PASS BY VALUE

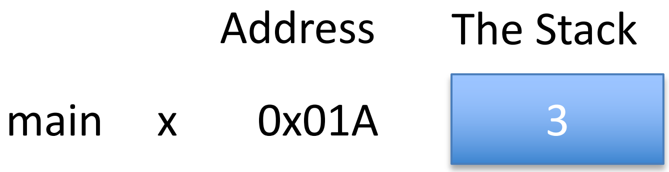
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main() {
    int x = 2;
    myFunc(&x);
    printf("%d", x);
    return 0;
}
```



|  |  | Address | The Stack |
|---|---|---|---|
| main | x | 0x01A | 3 |
| myFunc | x | 0x010 | 0x01A |

# EVERYTHING IN C IS PASS BY VALUE

```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

int main() {
    int x = 2;
    myFunc(&x);
    printf("%d", x);
    return 0;
}
```

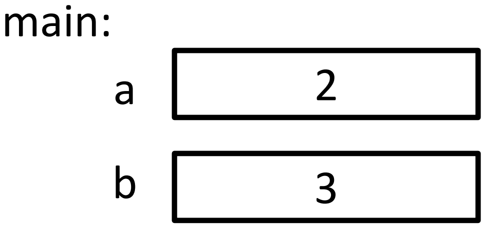


|  |  | Address | The Stack |
|---|---|---|---|
| main | x | 0x01A | 3 |

# LET’S FIX THIS.

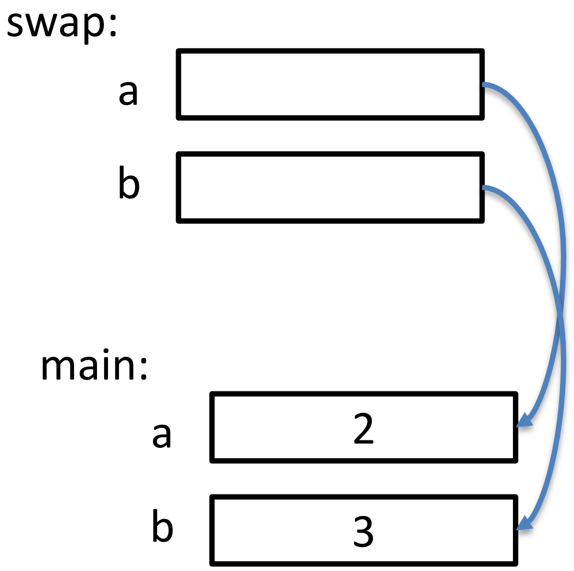
# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



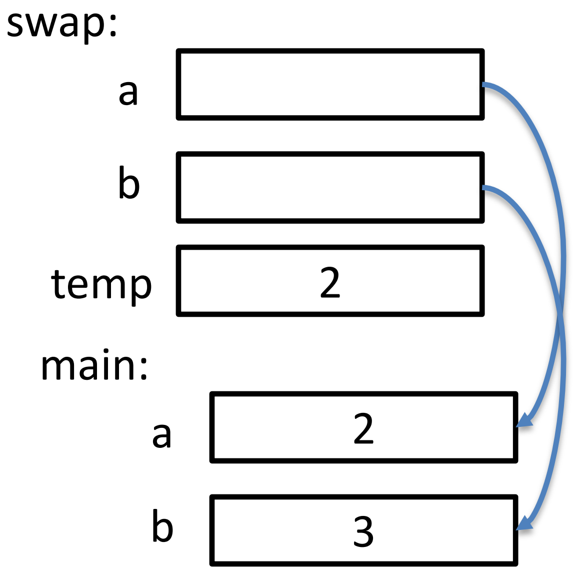
# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



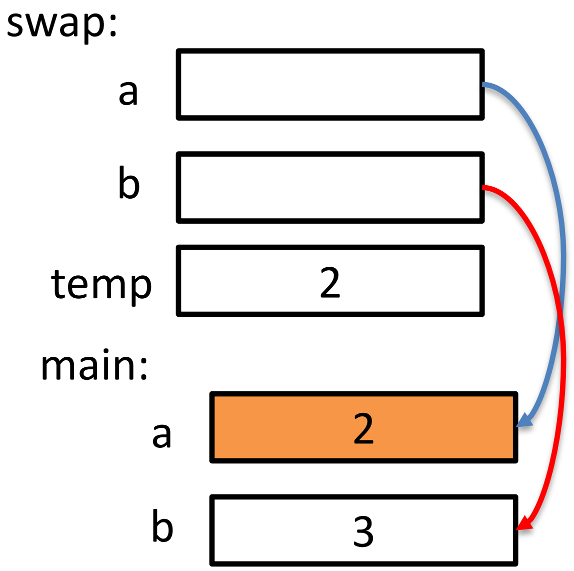
# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



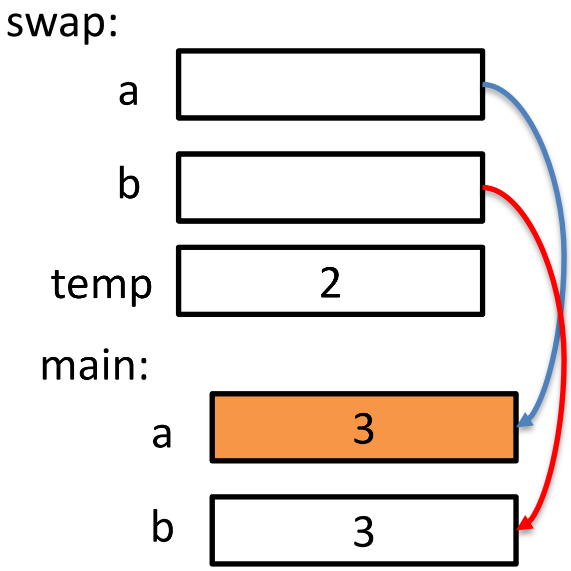
# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



swap:

a

b

temp 2

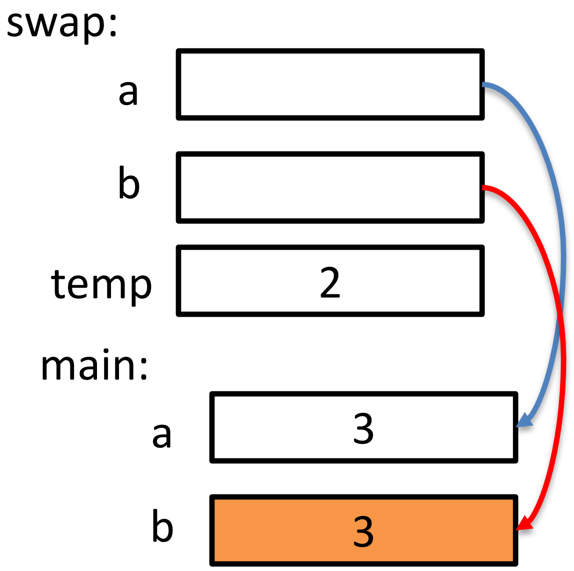
main:

a 3

b 3

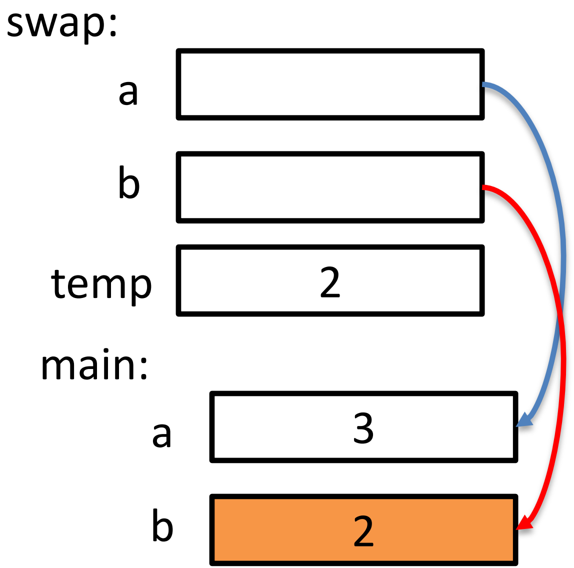
# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



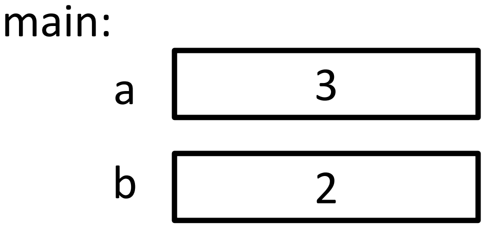
# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```



# SWAP EXAMPLE (FIXED)

```
void swap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b = 3;
    swap(&a, &b);
    return 0;
}
```

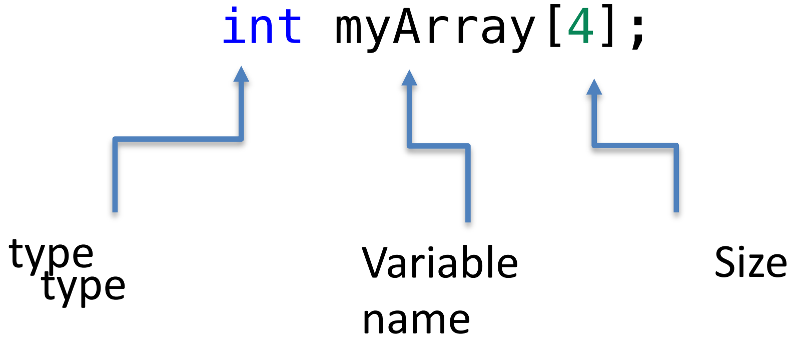


main:

| | |
|---|---|
| a | 3 |
| b | 2 |

# ARRAYS IN C

# THIS ONE WAY TO DECLARE AND ARRAY



```
int myArray[4];
```

type

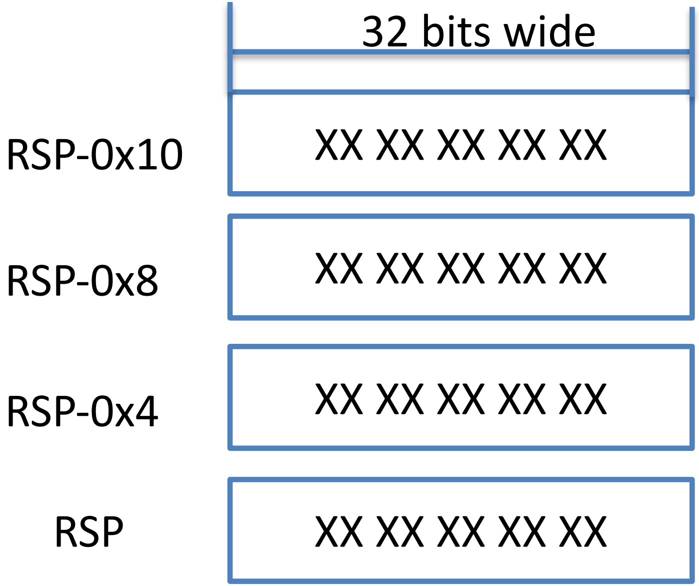
type

Variable name

Size

# THIS IS HOW ARRAYS ARE REPRESENTED IN MEMORY

```
int myArray[4];
```



32 bits wide

RSP-0x10 XX XX XX XX XX

RSP-0x8 XX XX XX XX XX

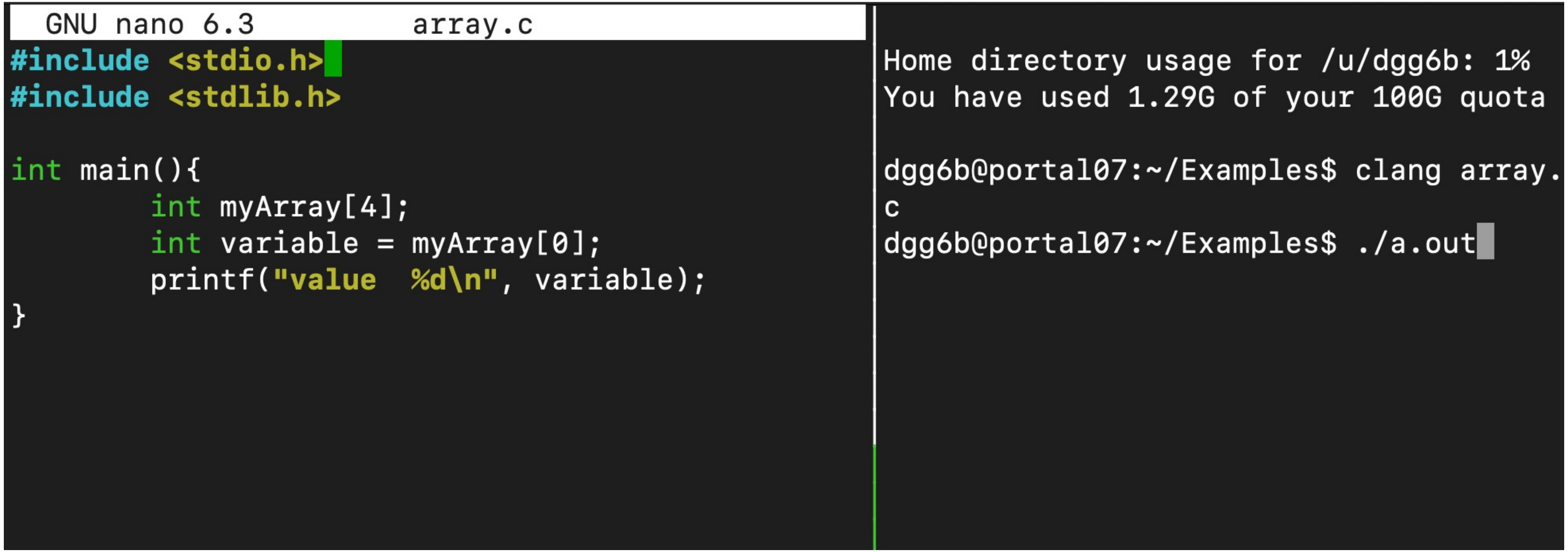
RSP-0x4 XX XX XX XX XX

RSP XX XX XX XX XX

# THIS IS HOW YOU ACCESS AND ELEMENT

```
int myArray[4];

int variable = myArray[0];
```

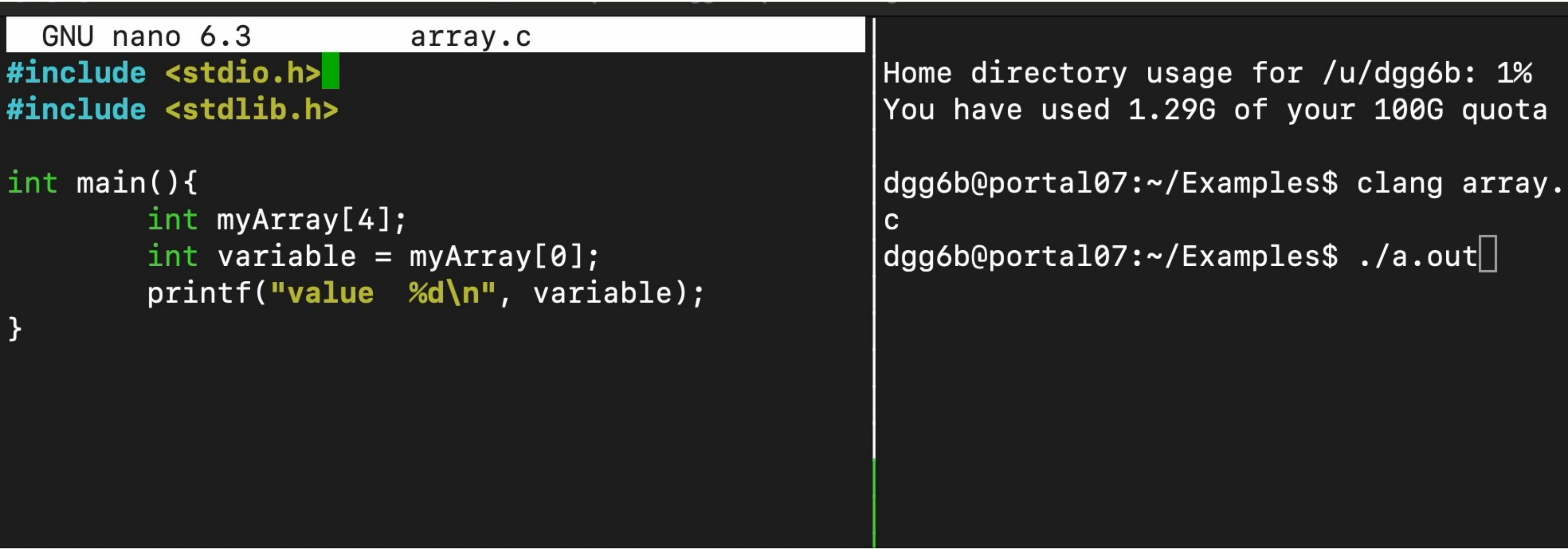
# WHAT DO WE THINK THIS WILL PRINT



```
  GNU nano 6.3              array.c
#include <stdio.h>
#include <stdlib.h>

int main(){
        int myArray[4];
        int variable = myArray[0];
        printf("value  %d\n", variable);
}
```

```
Home directory usage for /u/dgg6b: 1%
You have used 1.29G of your 100G quota

dgg6b@portal07:~/Examples$ clang array.
c
dgg6b@portal07:~/Examples$ ./a.out
```

# WITH OR WITHOUT OPTIMIZATIONS



```
  GNU nano 6.3              array.c
#include <stdio.h>
#include <stdlib.h>

int main(){
        int myArray[4];
        int variable = myArray[0];
        printf("value  %d\n", variable);
}
```

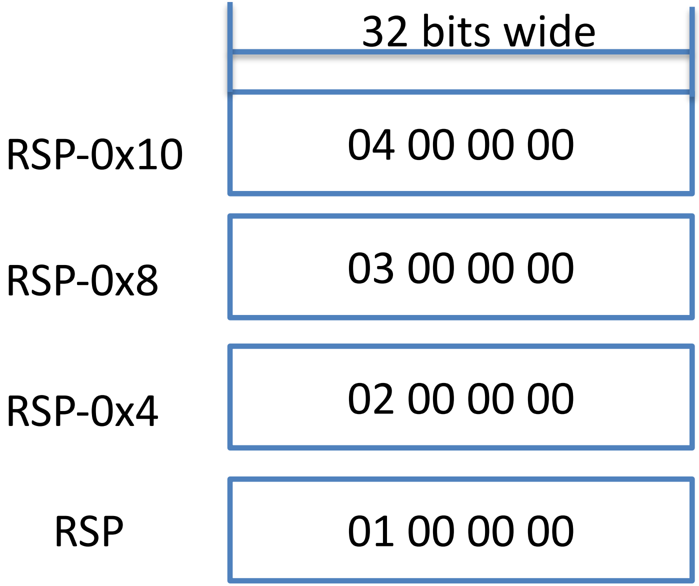
```
Home directory usage for /u/dgg6b: 1%
You have used 1.29G of your 100G quota

dgg6b@portal07:~/Examples$ clang array.
c
dgg6b@portal07:~/Examples$ ./a.out
```

# THIS IS HOW YOU SET A VALUE IN ARRAY

```
int myArray[4];

myArray[0] = 3;
```

# INITIALIZING ARRAYS WHEN THEY ARE DEFINED

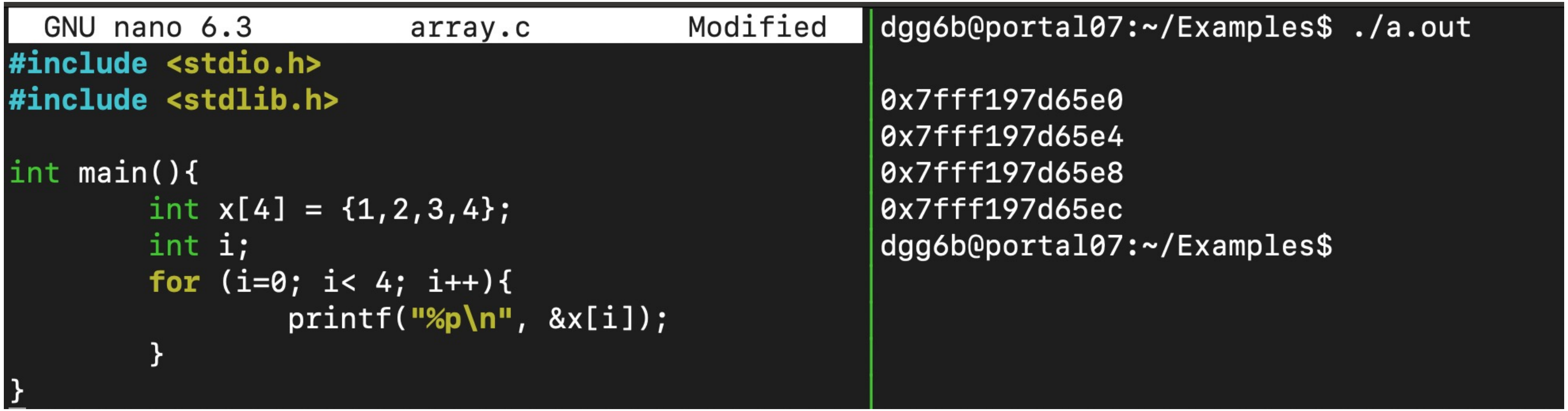
```
int x[4] = {1,2,3,4};
```



32 bits wide

| | |
|---|---|
| RSP-0x10 | 04 00 00 00 |
| RSP-0x8 | 03 00 00 00 |
| RSP-0x4 | 02 00 00 00 |
| RSP | 01 00 00 00 |

# PRINTING ADDRESS



```
  GNU nano 6.3              array.c              Modified
#include <stdio.h>
#include <stdlib.h>

int main(){
        int x[4] = {1,2,3,4};
        int i;
        for (i=0; i< 4; i++){
                printf("%p\n", &x[i]);
        }
}
```

```
dgg6b@portal07:~/Examples$ ./a.out

0x7fff197d65e0
0x7fff197d65e4
0x7fff197d65e8
0x7fff197d65ec
dgg6b@portal07:~/Examples$
```

# ARRAY SYNTAX AND POINTERS

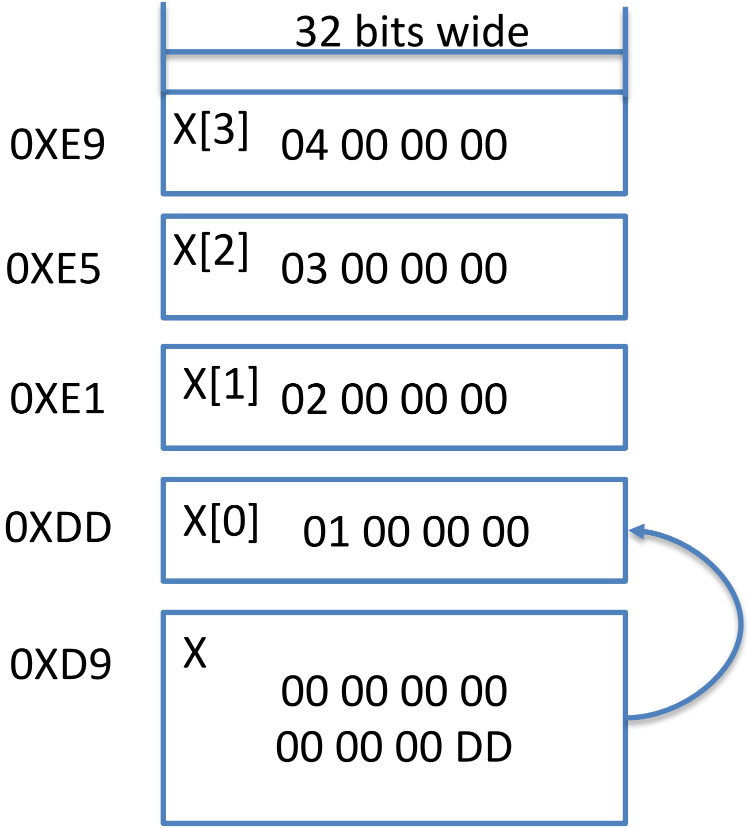
```
int x[4] = {1,2,3,4}
```

What does X really store?
Understanding this question is the key to understanding pointers.

# ARRAY SYNTAX AND POINTERS

```
int x[4] = {1,2,3,4}
```

X is location in memory that holds the address of first element in the array X



32 bits wide

0XE9 X[3] 04 00 00 00

0XE5 X[2] 03 00 00 00

0XE1 X[1] 02 00 00 00

0XDD X[0] 01 00 00 00

0XD9 X 00 00 00 00
00 00 00 DD

# SETTING VALUES IN ARRAYS USING POINTERS

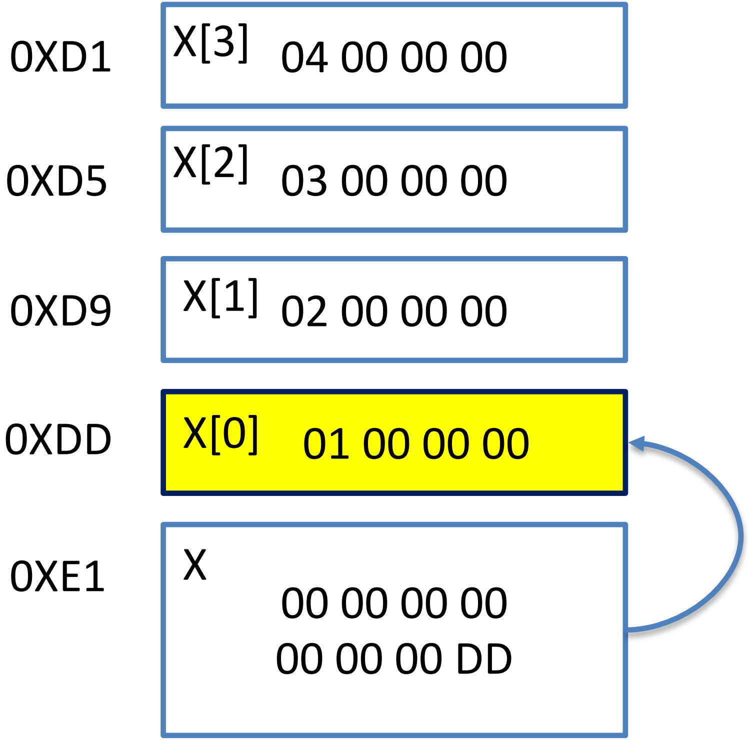
```
int x[4] = {1,2,3,4}

*x = 7;
```

Go to address X points to an update it to 7;

# SETTING VALUES IN ARRAYS USING POINTERS

```
int x[4] = {1,2,3,4};

*x = 7;
```

Go to address X points to an update it to 7;



0XD1 X[3] 04 00 00 00

0XD5 X[2] 03 00 00 00

0XD9 X[1] 02 00 00 00

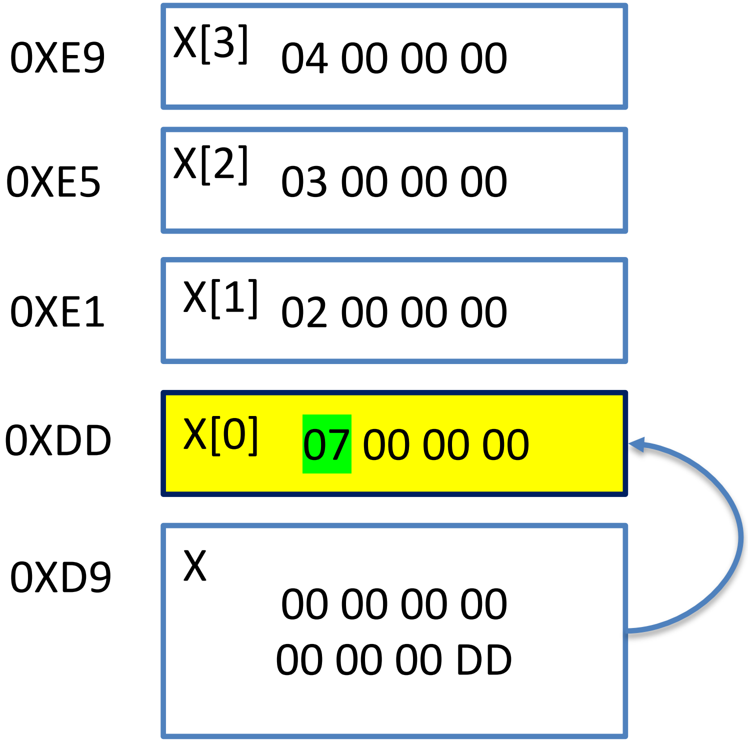
0XDD X[0] 01 00 00 00

0XE1 X 00 00 00 00 00 00 00 DD

# SETTING VALUES IN ARRAYS USING POINTERS

```
int x[4] = {1,2,3,4}

*x = 7;
```

Go to address X points to an update it to 7;



0XE9 X[3] 04 00 00 00

0XE5 X[2] 03 00 00 00

0XE1 X[1] 02 00 00 00

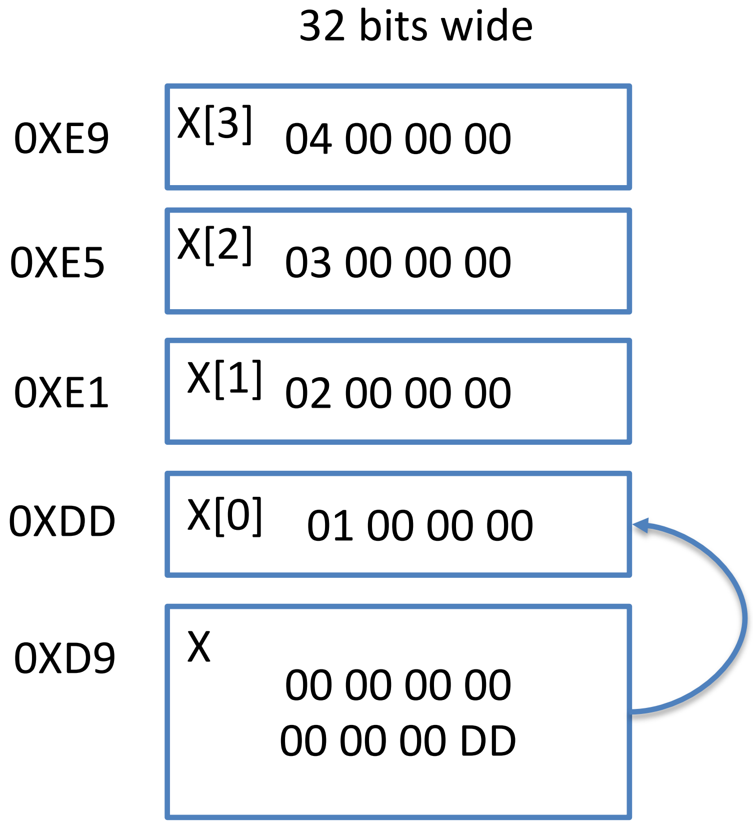
0XDD X[0] 07 00 00 00

0XD9 X 00 00 00 00 00 00 00 DD

# ARRAY SYNTAX AND POINTERS

```
int x[4] = {1,2,3,4};

*(x + 1) = 7;
```

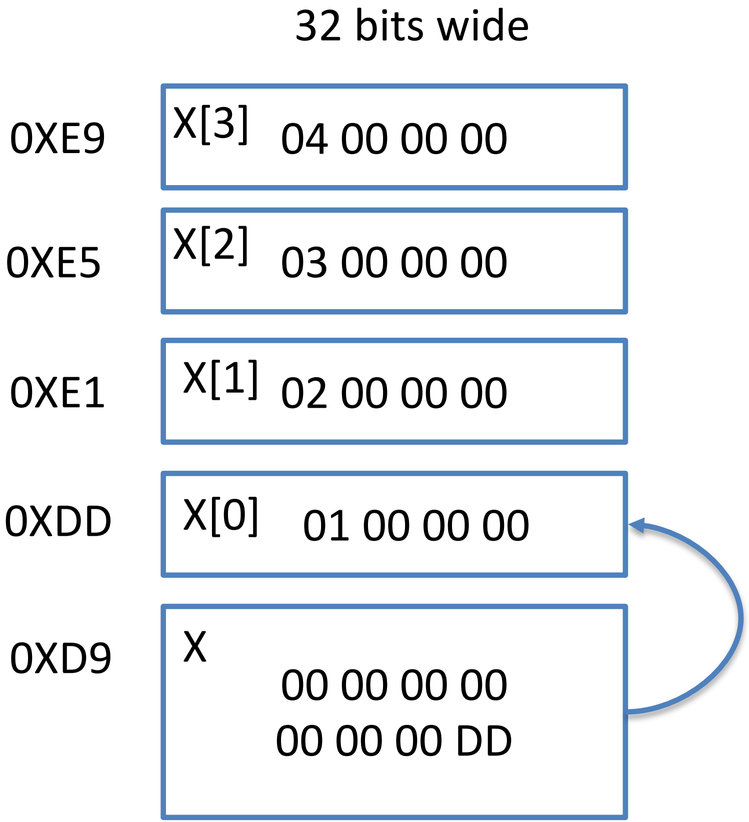
Should we do:
0xDD + 1 =  0xDE
Or
0xDD + 4 = 0xE1



# ARRAY SYNTAX AND POINTERS

```
int x[4] = {1,2,3,4};

*(x + 1) = 7;
```

Should we do:
0xDD + 1 =  0xDE
Or
0xDD + 4 = 0xE1



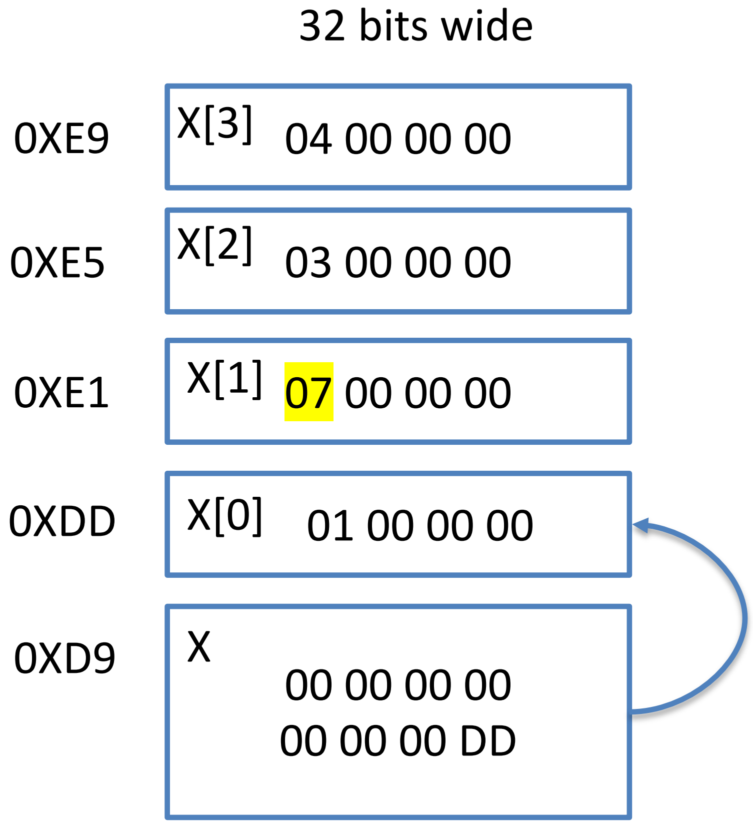
32 bits wide

| | | |
|---|---|---|
| 0XE9 | X[3] | 04 00 00 00 |
| 0XE5 | X[2] | 03 00 00 00 |
| 0XE1 | X[1] | 02 00 00 00 |
| 0XDD | X[0] | 01 00 00 00 |
| 0XD9 | X | 00 00 00 00<br>00 00 00 DD |

# ARRAY SYNTAX AND POINTERS

```
int x[4] = {1,2,3,4};

*(x + 1) = 7;
```

Should we do:
0xDD + 1 = 0xDE
Or
0xDD + 4 = 0xE1



32 bits wide

| | | |
|---|---|---|
| 0XE9 | X[3] | 04 00 00 00 |
| 0XE5 | X[2] | 03 00 00 00 |
| 0XE1 | X[1] | 07 00 00 00 |
| 0XDD | X[0] | 01 00 00 00 |
| 0XD9 | X | 00 00 00 00<br>00 00 00 DD |

# POINTER ARITHMETIC RULE

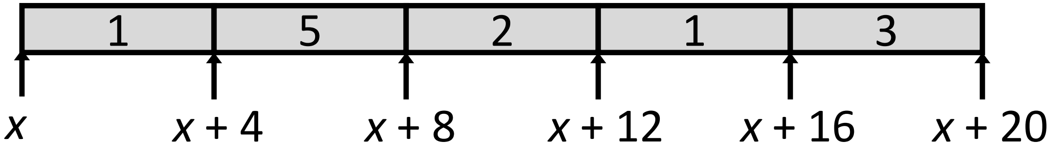
When do arithmetic operation using on pointer variables constants are treated as a multiple of size of the pointer type.

```
int *p;
p = p + 3;
```

```
long long *ll;
ll = ll - 2;
```

# ARRAY ACCESSES

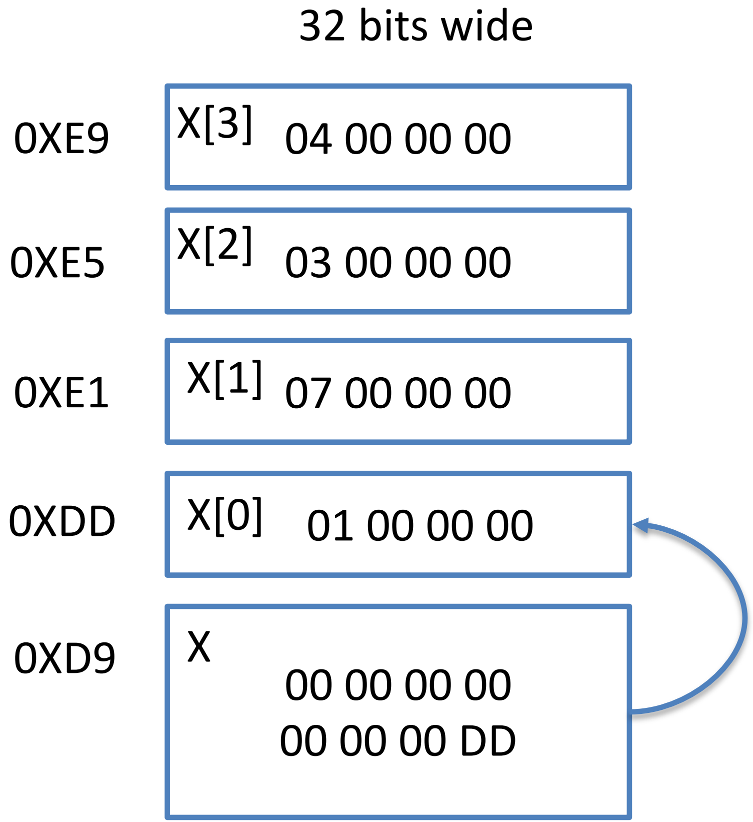
```
int val[5];
```



| Reference | Type | Value |
|---|---|---|
| **val[4]** | **int** | 3 |
| **val** | **int *** | $x$ |
| **val+1** | **int *** | $x + 4$ |
| **&val[2]** | **int *** | $x + 8$ |
| **val[5]** | **int** | ?? // Could return a value or segfault*** |
| ***(val+1)** | **int** | 5 |
| **val + *i*** | **int *** | $x + 4i$ |

# ARRAY SYNTAX AND POINTERS

```
int x[4] = {1,2,3,4};

*x = *x + 1;
```



32 bits wide

0XE9 X[3] 04 00 00 00

0XE5 X[2] 03 00 00 00

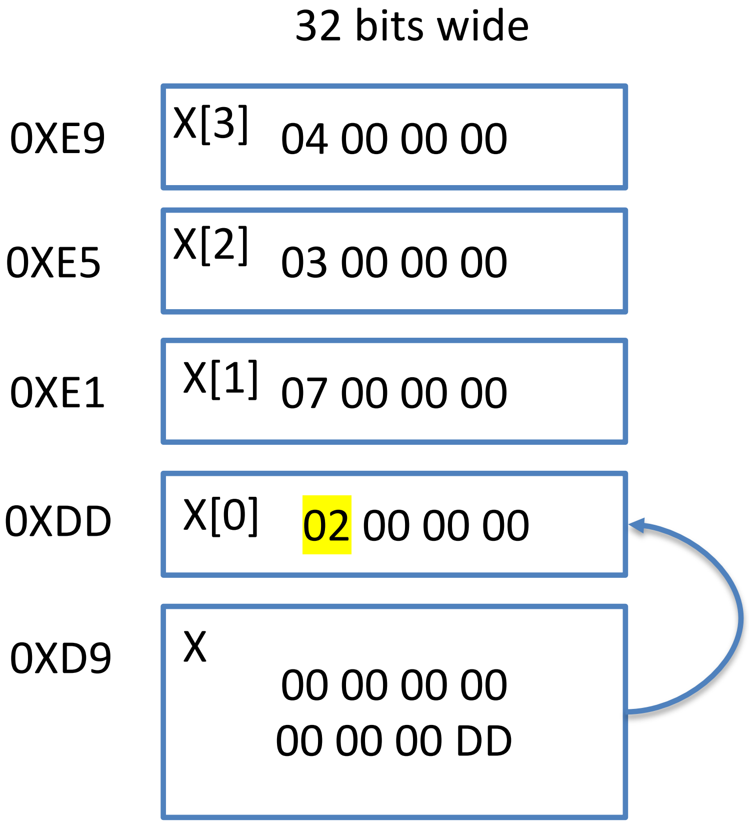
0XE1 X[1] 07 00 00 00

0XDD X[0] 01 00 00 00

0XD9 X 00 00 00 00 00 00 00 DD

# ARRAY SYNTAX AND POINTERS

```
int x[4] = {1,2,3,4};

*x = *x + 1;
```



32 bits wide

0XE9 X[3] 04 00 00 00

0XE5 X[2] 03 00 00 00

0XE1 X[1] 07 00 00 00

0XDD X[0] 02 00 00 00

0XD9 X 00 00 00 00 00 00 00 DD

# IF ARRAY ARE JUST POINTERS WHY DOES SIZEOF WORK

Well arrays aren’t of pointer types.

int * the are of type int [n]

```
int x[4] = {1,2,3,4};
```

This type is actually type int [4]

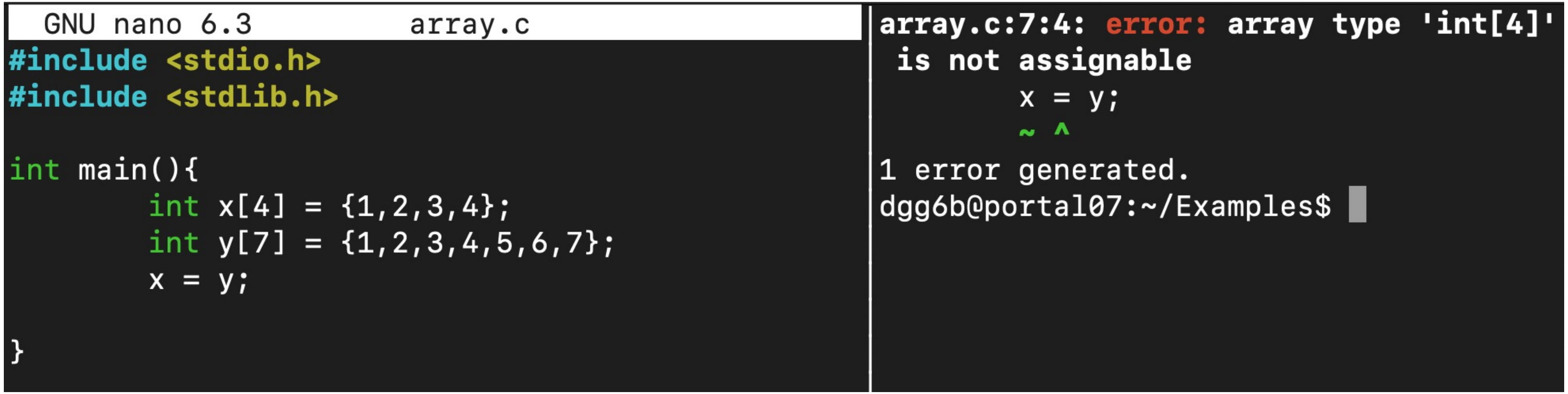
Arrays are of type int [n] and language doesn’t allow these to

# ARRAY NOT QUITE POINTS

```
int x[4] = {1,2,3,4};

int y[5] = {1,2,3,4,5};

x = y // Not allowed.

//If you want to do this you
will need to a memcpy
(memcp(x,y, sizeof(x));
```

Arrays are of type int [n] and language doesn't allow these types to be assigned

# ARRAY TYPES NOT ASSIGNABLE



```
  GNU nano 6.3                   array.c
#include <stdio.h>
#include <stdlib.h>

int main(){
        int x[4] = {1,2,3,4};
        int y[7] = {1,2,3,4,5,6,7};
        x = y;

}
```

```
array.c:7:4: error: array type 'int[4]'
 is not assignable
        x = y;
        ~ ^
1 error generated.
dgg6b@portal07:~/Examples$ 
```

# ARRAYS NOT QUITE POINTERS

Allowed the language

```
int x[4] = {1,2,3,4};
int *p;
p = x; //Same as p=&(x[0])
```

Allowed
pointer = array

Not allow by the language

```
int x[4] = {1,2,3,4};
int *p;
x = p //Not allowed ☹
```

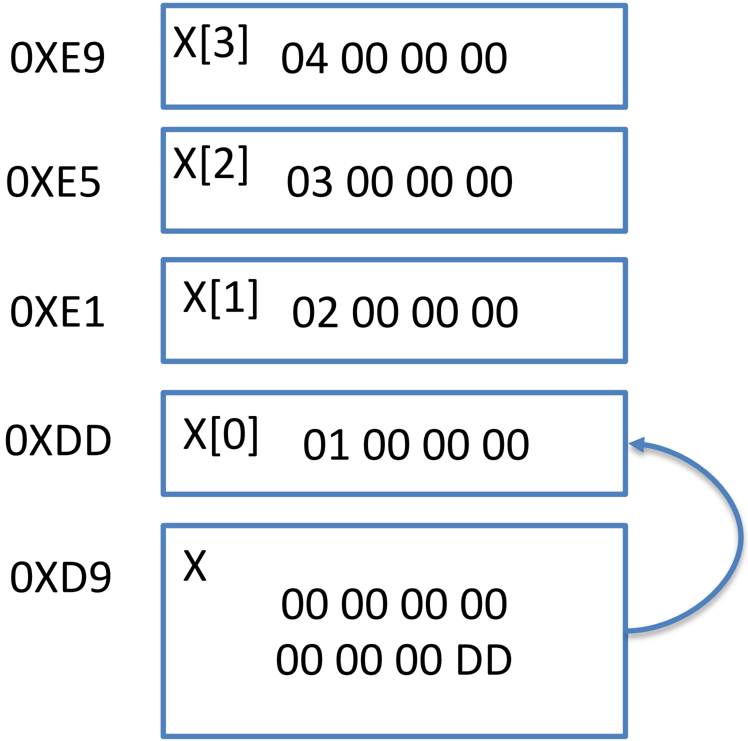
Because array types
int[4] is not assignable

# LET’S LOOK AT SOME TRICKY EXAMPLES

# TALK TO YOUR NEIGHBOR

```
*(x + 1) = *x + *(x + 1);

printf("value: %d", x[1]);
```

What does this print out?



0XE9 X[3] 04 00 00 00

0XE5 X[2] 03 00 00 00

0XE1 X[1] 02 00 00 00

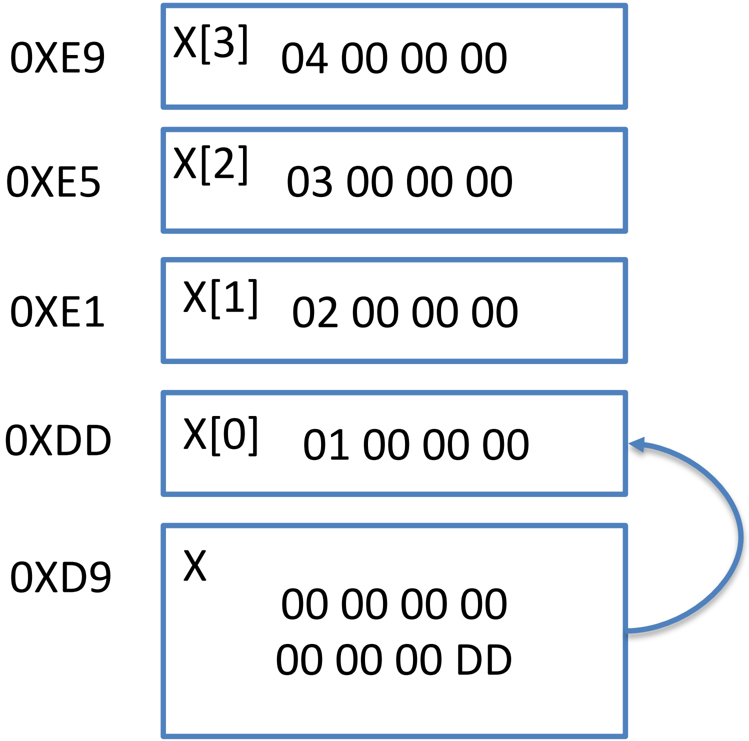
0XDD X[0] 01 00 00 00

0XD9 X
00 00 00 00
00 00 00 DD

# TALK TO YOUR NEIGHBOR

```
x = x + 1;

printf(“value: %d”, x[1]);
```

What does this print out?



0XE9 X[3] 04 00 00 00

0XE5 X[2] 03 00 00 00

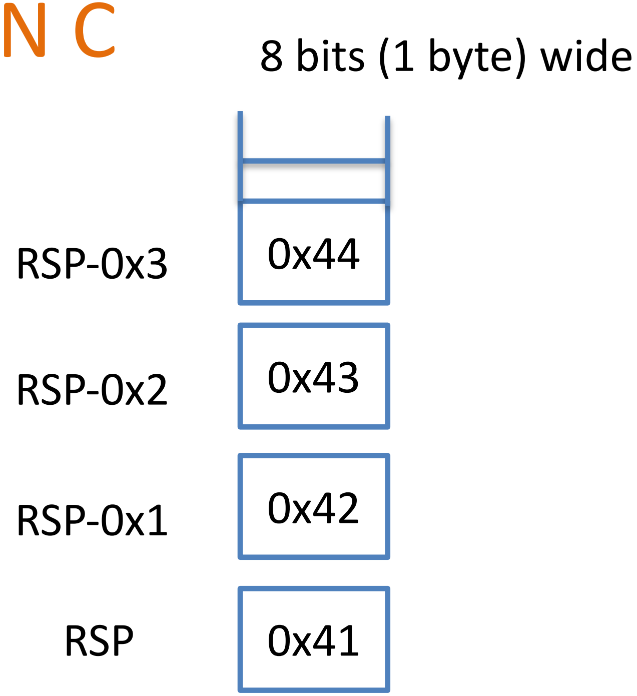
0XE1 X[1] 02 00 00 00

0XDD X[0] 01 00 00 00

0XD9 X 00 00 00 00 00 00 00 DD

# ARRAY IN C

```
char a[4] = {‘A’, ‘B’, ‘C’, ‘D’}
```



# CHAR ARRAY, AND STRING

```
char b[7] = {‘D’,‘a’,‘n’,‘i’,‘e’,‘l’,‘\0’}
```

# CHAR ARRAY, AND STRING

```
char b[7] = {‘D’,‘a’,‘n’,‘i’,‘e’,‘l’,‘\0’};
```

```
char *b = “Daniel”;
```

# NEXT TIME

1. Methods for manipulating string
2. Implementing some of these methods ourselves
3. Multidimensional arrays

8. [12 points] Consider the following C code:

```
char first[5] = {'f', 'y', 'i', '!', '\0'};
char *second = strdup("hello");
char *both[2] = {first, second};
```

What is printed for each of the following lines? If the program would crash or seg fault, write **crash**. *Hint:* `printf("%c", x);` *means "print the char stored in variable x."*

A. `printf("%c", (*both)[1]);`

B. `printf("%c", *(both[1]));`

C. `puts(&both[0][2]);`

y, h, i!

# SEGMENTATION FAULT