# COMPUTER SYSTEMS AND ORGANIZATION

## C compilation

Daniel G. Graham Ph.D

University of Virginia | ENGINEERING

# Contents

1. Escape room
2. C basics
3. Manual/Info Pages
4. Printf and Scanf

# CMP INSTRUCTIONS

cmp A , B

jx

B op A

J[op]

Notice order is swapped

B > A

jg

B <= A

jle

B < A

jl

# ESCAPE ROOM FUN

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the Escape Room so that it returns true. Assume that we can supply an integer as input.

# ESCAPE ROOM FUN

```
escapeRoom:
  leal (%rdi,%rdi), %eax
  cmpl $5, %eax
  jg .L3
  cmpl $1, %edi
  jne .L4
  movl $1, %eax
  ret
.L3:
  movl $1, %eax
  ret
.L4:
  movl $0, %eax
  ret
```

What must be passed to the Escape Room so that it returns true

First param > 2 or == 1

# C MAIN ENTRY

```c
#include <stdio.h>

int main(void)
{
   puts("Hello World");
   return 0;
}
```

What is this return 0;

It is a status code.

# C MAIN ENTRY

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

int main(void)
{
   printf("Hello World\n");
   return EXIT_SUCCESS;
}
```

# WHEN WOULD WE USE STATUS CODE

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  if (puts("Hello, world!") == EOF) {
    return EXIT_FAILURE;
    // code here never executes
  }
  return EXIT_SUCCESS;
  // code here never executes
}
```

(Let's do a quick demo of the manual/info page. Looking up a couple of things.

- Point out return value
- Know bugs section
- The section on library and include statements

)

# LET'S DO A QUICK EXAMPLE WITH DEBUGGING

Let's also check out the power of lldb, looking at the assembly associated with the puts functions.

```
clang –g puts.c -o puts.out
```

-g : let's us do line level debugging.

# TYPES IN C

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

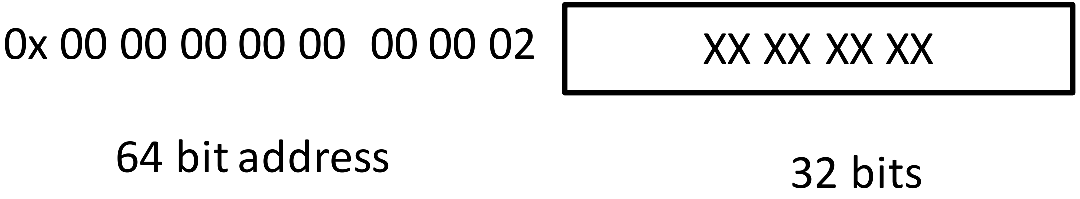
```
int x = 3;
int number_of_bytes = sizeof(x);

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;



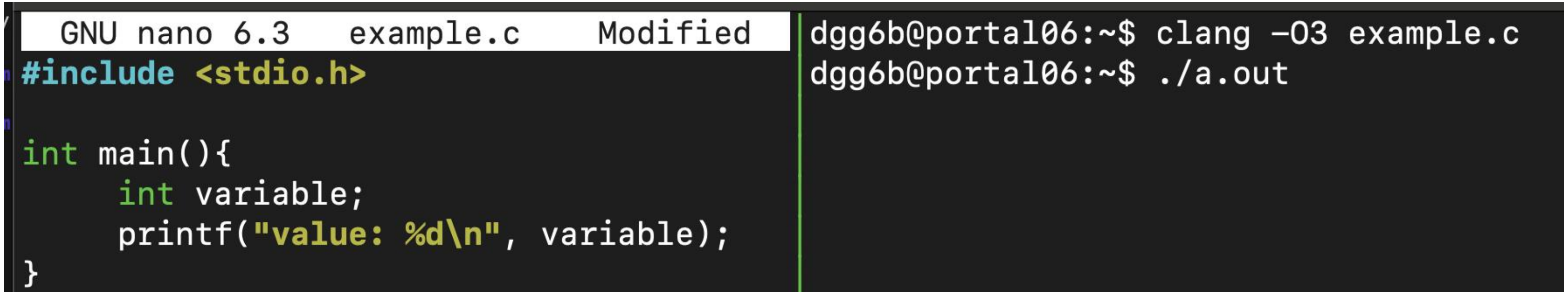
0x 00 00 00 00 00  00 00 02

XX XX XX XX

64 bit address

32 bits

# WHAT GETS PRINTED?

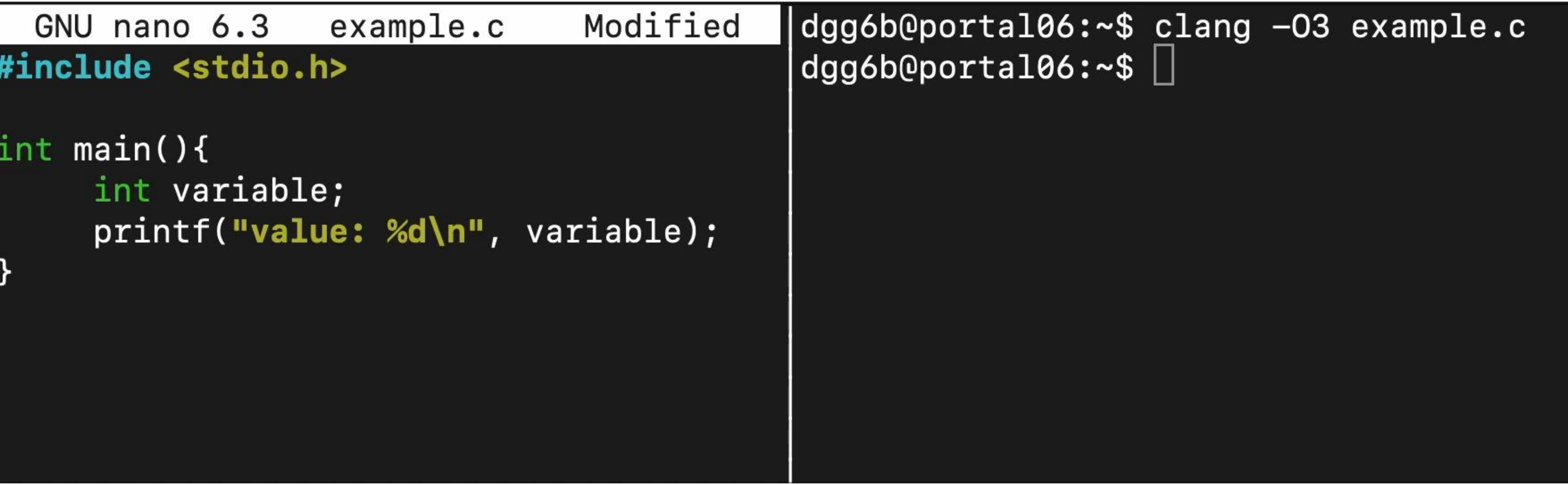


```
  GNU nano 6.3       example.c        Modified
#include <stdio.h>

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 GETS PRINTED?



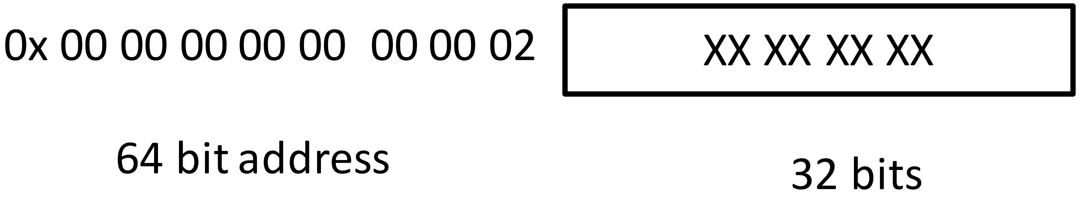
```
 GNU nano 6.3    example.c     Modified
#include <stdio.h>

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

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

Try not to 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

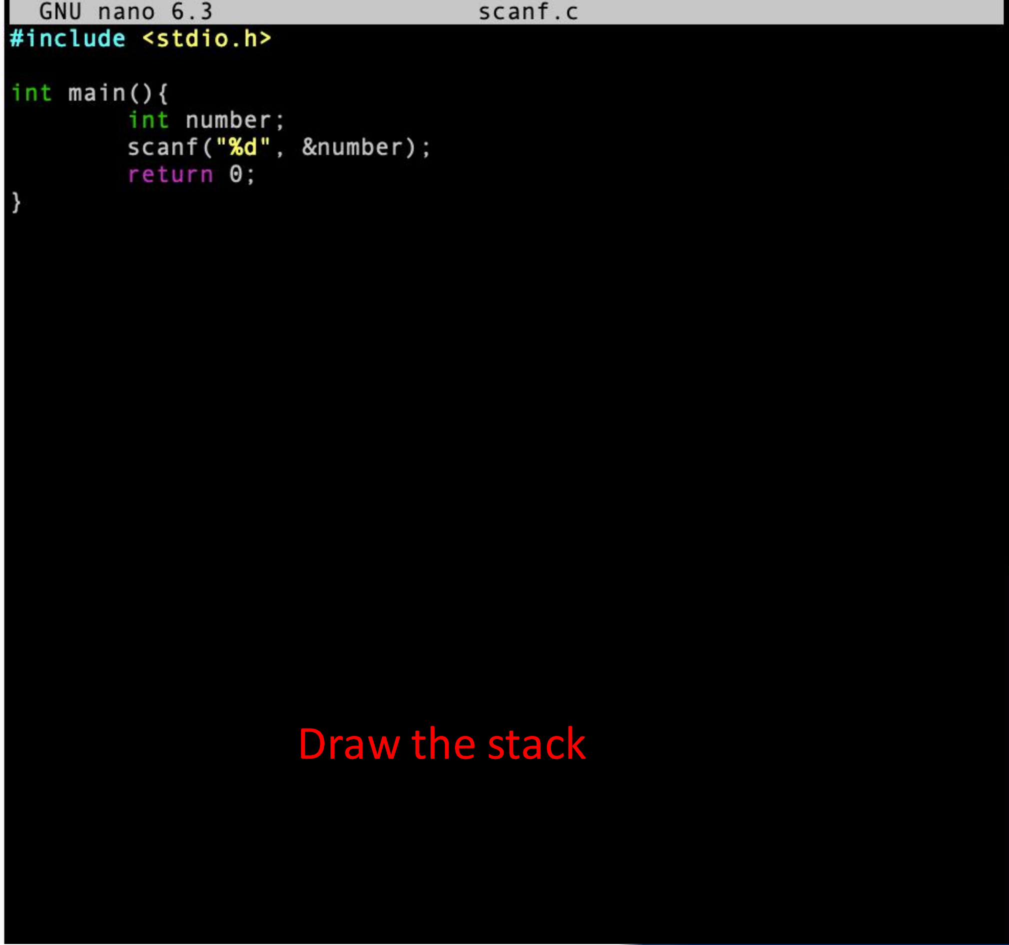
32 bits

# WHAT IF WE RUN IT WITHOUT OPTIMIZATIONS?

Quick Demo?

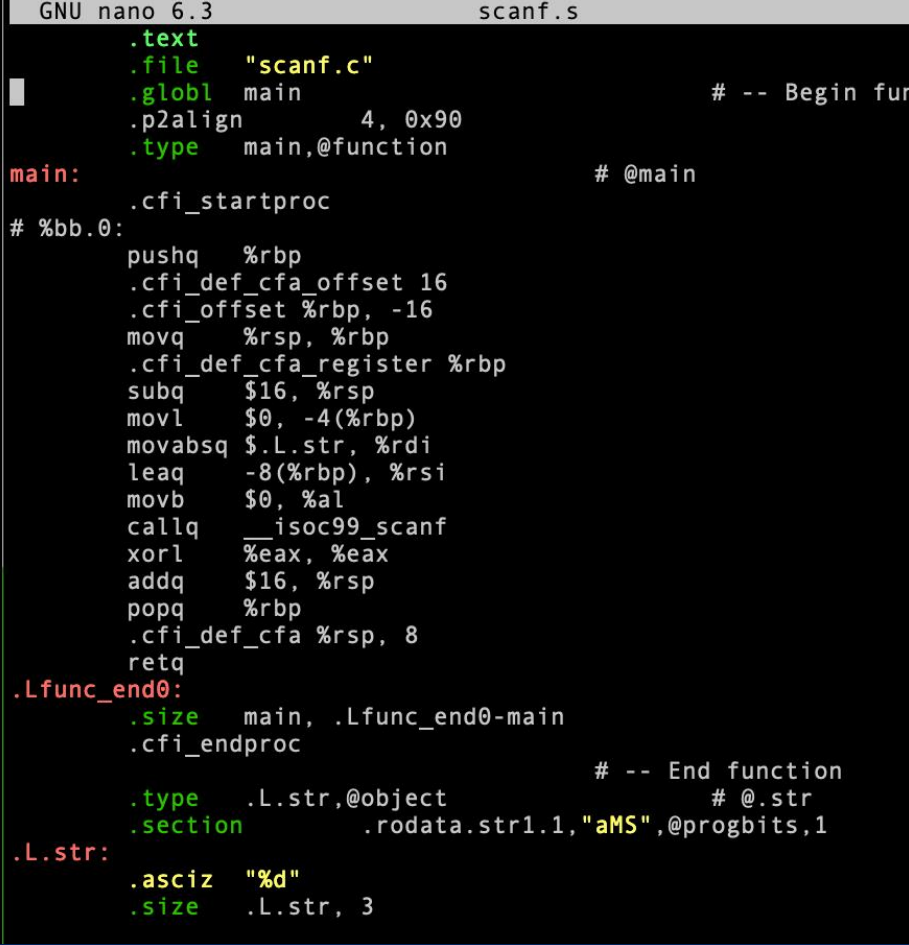
Do we always want to optimize?

# SCANF AND THE STACK



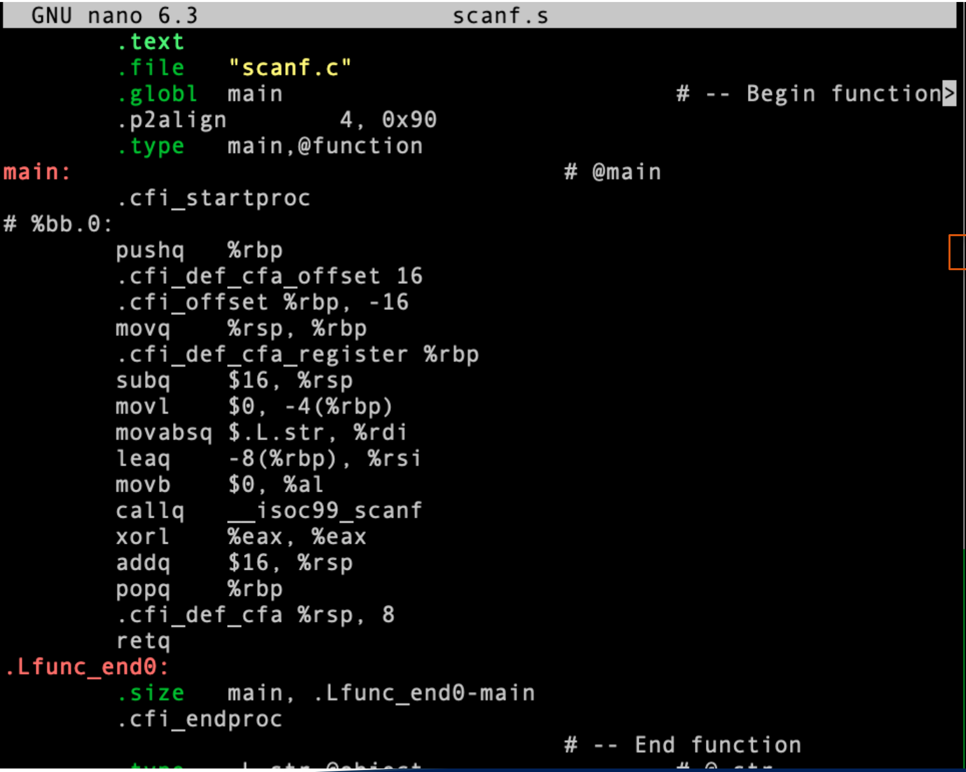
```
  GNU nano 6.3                    scanf.c
#include <stdio.h>

int main(){
        int number;
        scanf("%d", &number);
        return 0;
}
```

Draw the stack

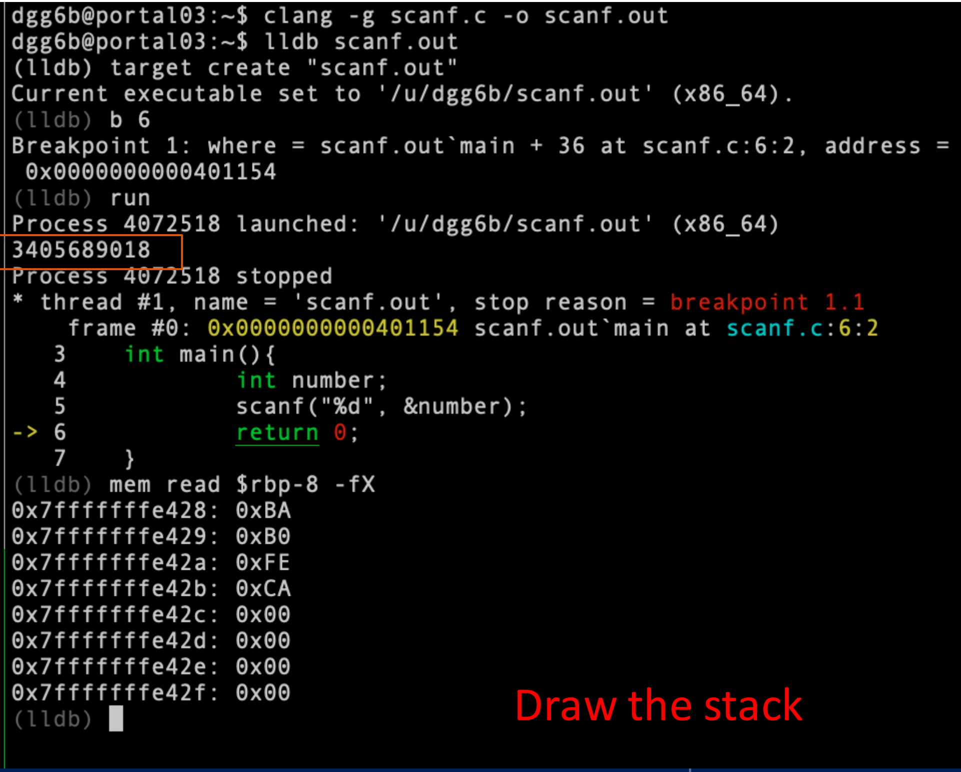


```
  GNU nano 6.3                          scanf.s
        .text
        .file   "scanf.c"
        .globl  main                            # -- Begin fun
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movabsq $.L.str, %rdi
        leaq    -8(%rbp), %rsi
        movb    $0, %al
        callq   __isoc99_scanf
        xorl    %eax, %eax
        addq    $16, %rsp
        popq    %rbp
        .cfi_def_cfa %rsp, 8
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .type   .L.str,@object                  # @.str
        .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:
        .asciz  "%d"
        .size   .L.str, 3
```

# SCANF WRITES THE INPUT THE ADDRESS

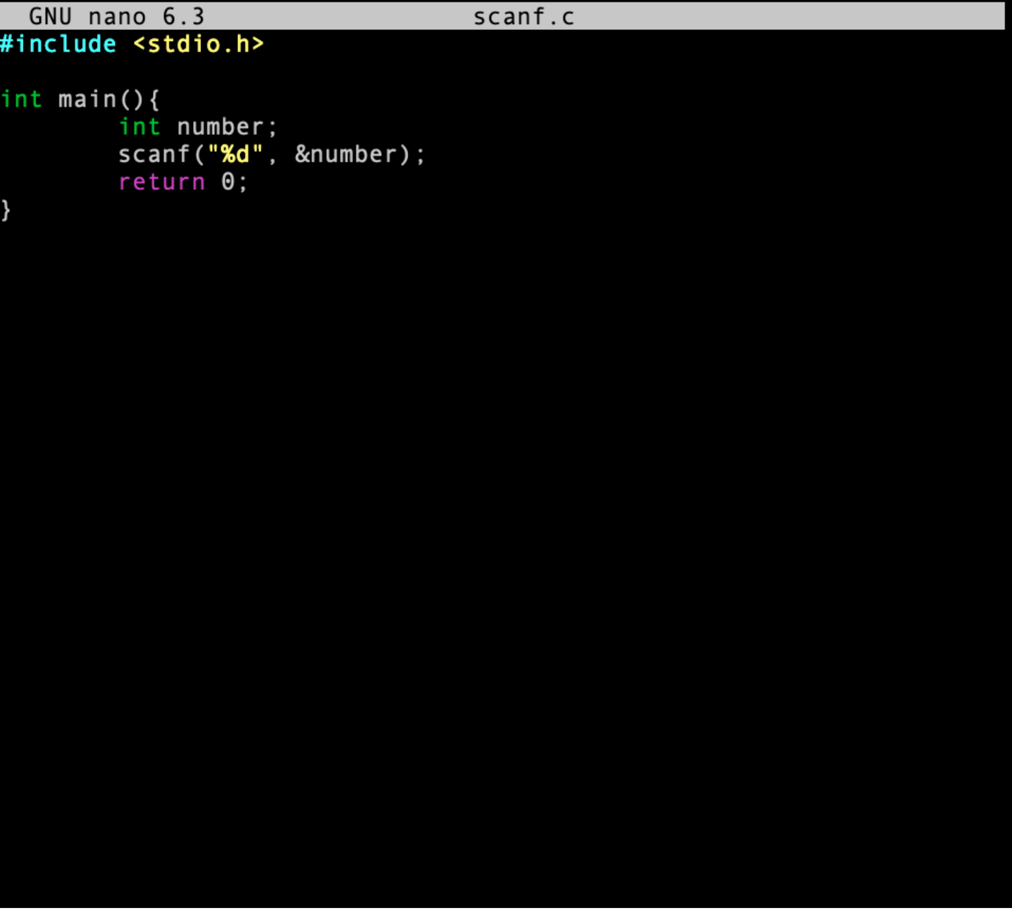


```
  GNU nano 6.3                     scanf.s
        .text
        .file   "scanf.c"
        .globl  main                            # -- Begin function>
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        movl    $0, -4(%rbp)
        movabsq $.L.str, %rdi
        leaq    -8(%rbp), %rsi
        movb    $0, %al
        callq   __isoc99_scanf
        xorl    %eax, %eax
        addq    $16, %rsp
        popq    %rbp
        .cfi_def_cfa %rsp, 8
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
```

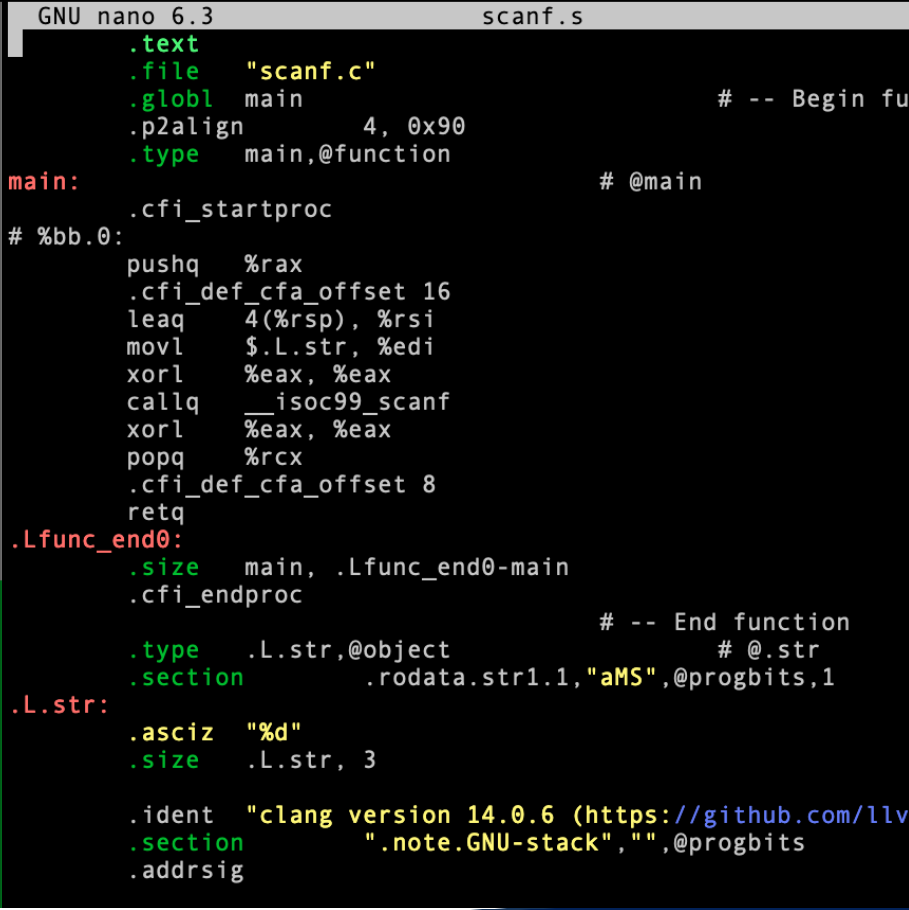


```
dgg6b@portal03:~$ clang -g scanf.c -o scanf.out
dgg6b@portal03:~$ lldb scanf.out
(lldb) target create "scanf.out"
Current executable set to '/u/dgg6b/scanf.out' (x86_64).
(lldb) b 6
Breakpoint 1: where = scanf.out`main + 36 at scanf.c:6:2, address =
 0x0000000000401154
(lldb) run
Process 4072518 launched: '/u/dgg6b/scanf.out' (x86_64)
3405689018
Process 4072518 stopped
* thread #1, name = 'scanf.out', stop reason = breakpoint 1.1
    frame #0: 0x0000000000401154 scanf.out`main at scanf.c:6:2
   3    int main(){
   4            int number;
   5            scanf("%d", &number);
-> 6            return 0;
   7    }
(lldb) mem read $rbp-8 -fX
0x7fffffffe428: 0xBA
0x7fffffffe429: 0xB0
0x7fffffffe42a: 0xFE
0x7fffffffe42b: 0xCA
0x7fffffffe42c: 0x00
0x7fffffffe42d: 0x00
0x7fffffffe42e: 0x00
0x7fffffffe42f: 0x00
(lldb)
```

Draw the stack



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

int main(){
        int number;
        scanf("%d", &number);
        return 0;
}
```



```
  GNU nano 6.3                              scanf.s
        .text
        .file   "scanf.c"
        .globl  main                            # -- Begin fu
        .p2align        4, 0x90
        .type   main,@function
main:                                   # @main
        .cfi_startproc
# %bb.0:
        pushq   %rax
        .cfi_def_cfa_offset 16
        leaq    4(%rsp), %rsi
        movl    $.L.str, %edi
        xorl    %eax, %eax
        callq   __isoc99_scanf
        xorl    %eax, %eax
        popq    %rcx
        .cfi_def_cfa_offset 8
        retq
.Lfunc_end0:
        .size   main, .Lfunc_end0-main
        .cfi_endproc
                                        # -- End function
        .type   .L.str,@object                  # @.str
        .section        .rodata.str1.1,"aMS",@progbits,1
.L.str:
        .asciz  "%d"
        .size   .L.str, 3

        .ident  "clang version 14.0.6 (https://github.com/llv
        .section        ".note.GNU-stack","",@progbits
        .addrsig
```

# THIS DECLARES A POINTER

```
int *pointer;
```

0x 00 00 00 00 00  00 00 06

XX XX XX XX XX XX XX XX

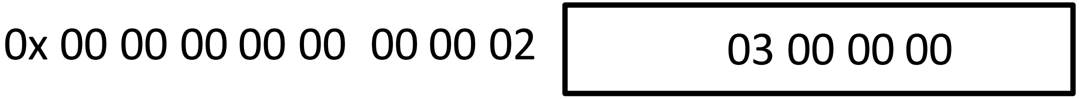
64 bit address

64 bit value

Be careful with uninitialized pointers: if referenced to without setting, it will lead to a memory error

# THIS INITIALIZES A VARIABLE

int variable = 3;

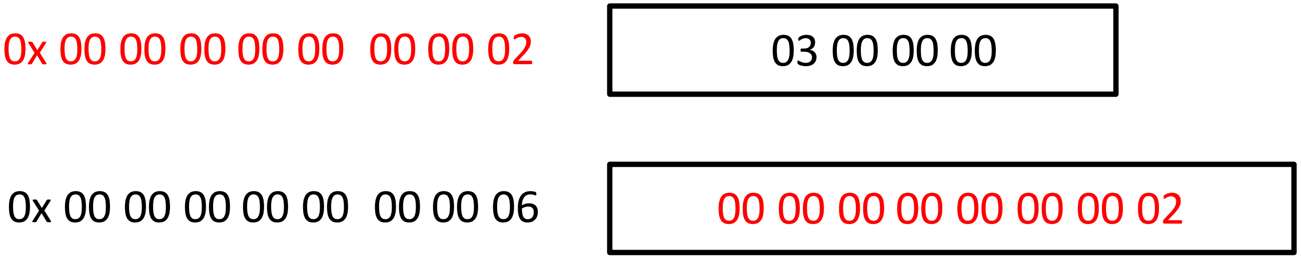


0x 00 00 00 00 00  00 00 02

03 00 00 00

# THIS INITIALIZES A POINTER

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



0x 00 00 00 00 00  00 00 02

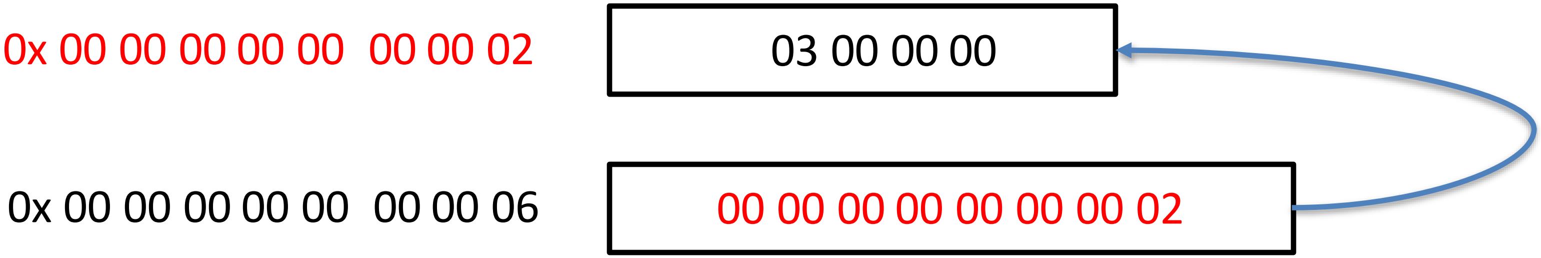
03 00 00 00

0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 02

# THIS INITIALIZES A POINTER

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

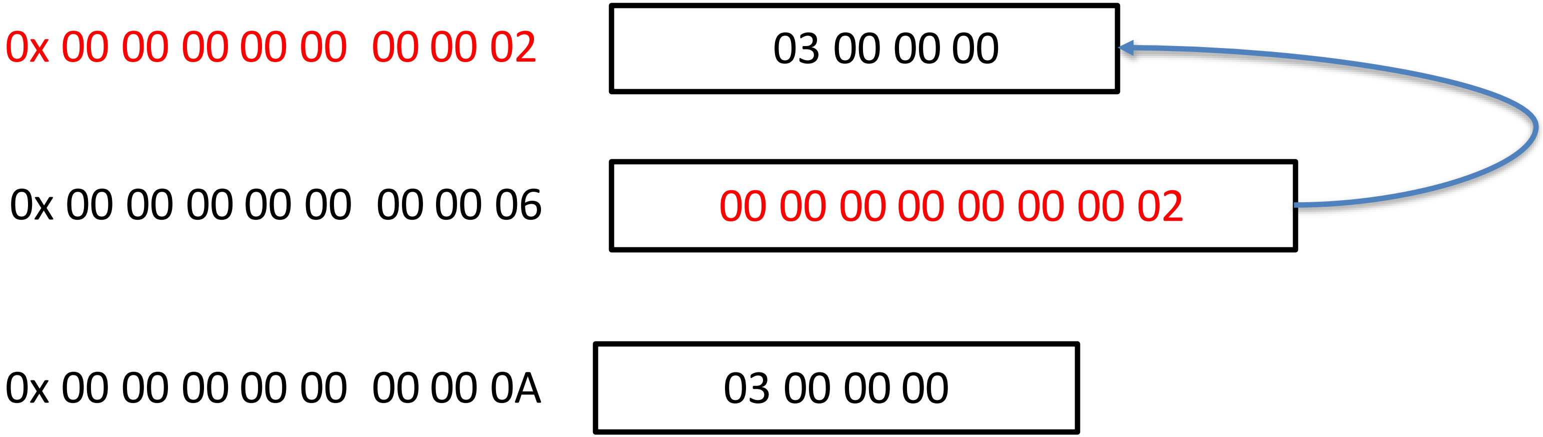


0x 00 00 00 00 00  00 00 02 | 03 00 00 00

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

# DEREFERENCE VALUE (USE)

int variable2 = *pointer;



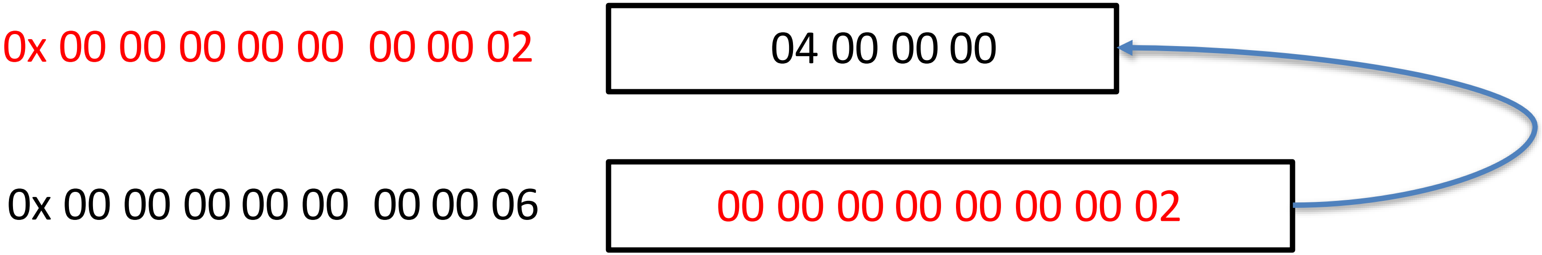
0x 00 00 00 00 00  00 00 02 — 03 00 00 00

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

04 00 00 00

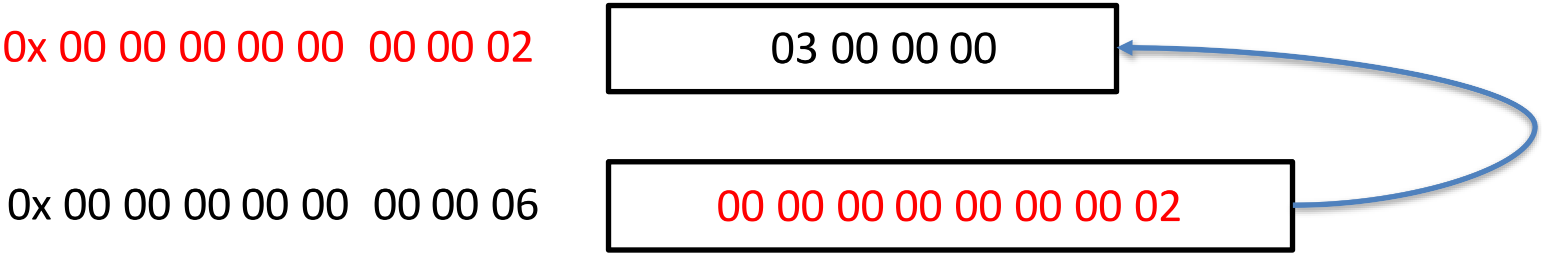
0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 02

*pointer = 4;

# ASSIGNMENT POINTER

int *pointer = &variable;



0x 00 00 00 00 00  00 00 02

03 00 00 00

0x 00 00 00 00 00  00 00 06

00 00 00 00 00 00 00 02

*pointer = 3;

# 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

Location on the stack

| 00 00 00 00 00 00 00 00 |
|---|

Value at that location

Reserve a memory location on the stack to store an address

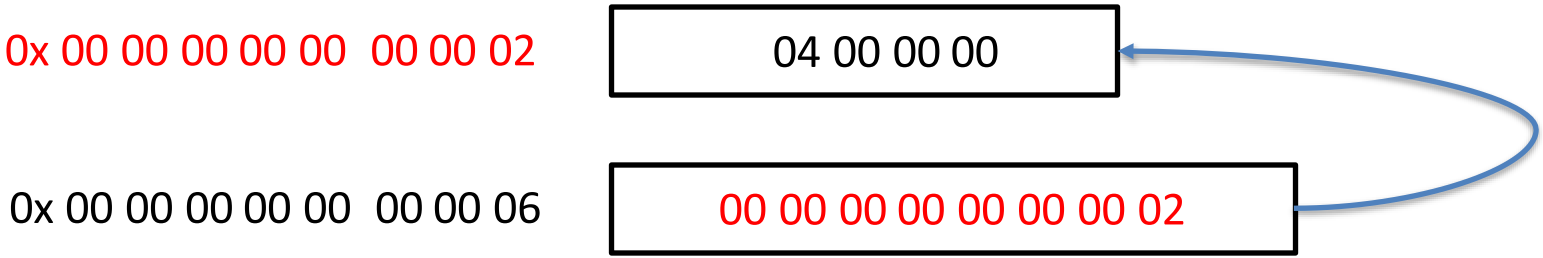
# POINTER RULES RULE 2

```
*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

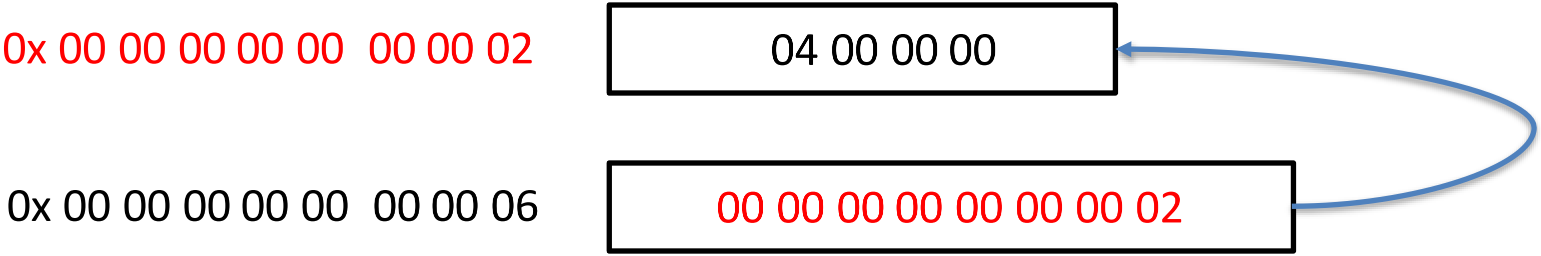
= *p

- * 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

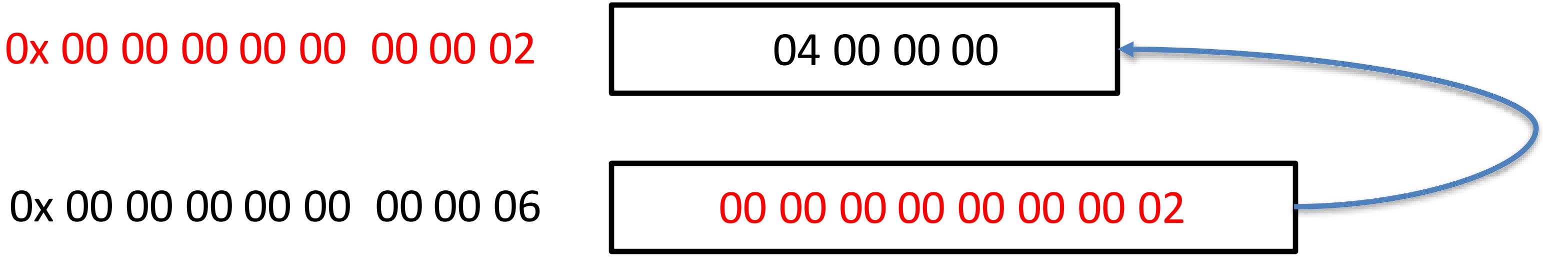
04 00 00 00

0x 00 00 00 00 00 00 00 06

00 00 00 00 00 00 00 02

# POINTER RULES RULE 3

= 4



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

# FINAL RULE

```
=0x…0006
```

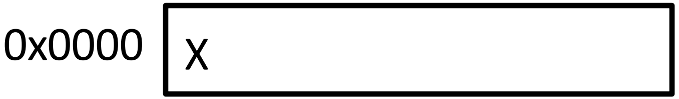
0x 00 00 00 00 00  00 00 06

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;
```



# 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 |

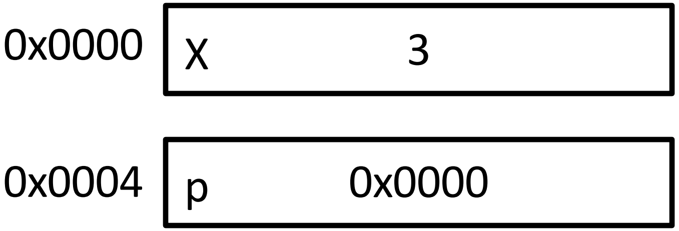
# POINTERS

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

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

# POINTERS

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

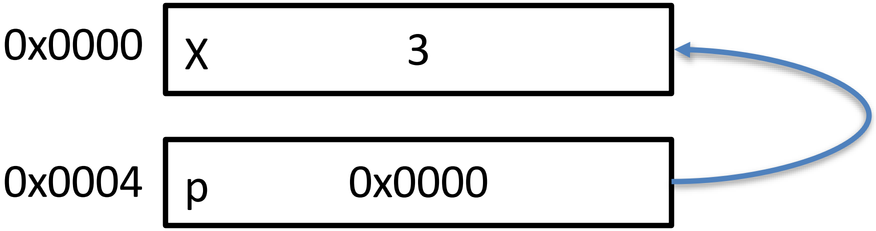


0x0000 | X | 3

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;
```

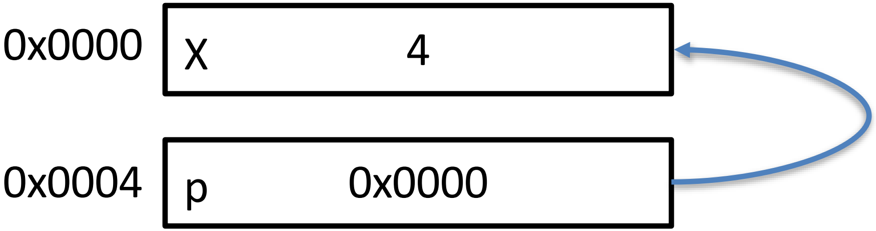


0x0000 | x | 3

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;
```

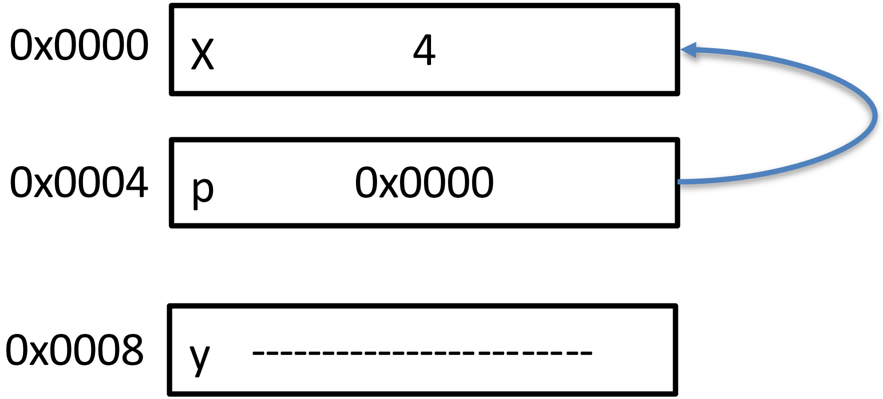


0x0000 X 4

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;
```



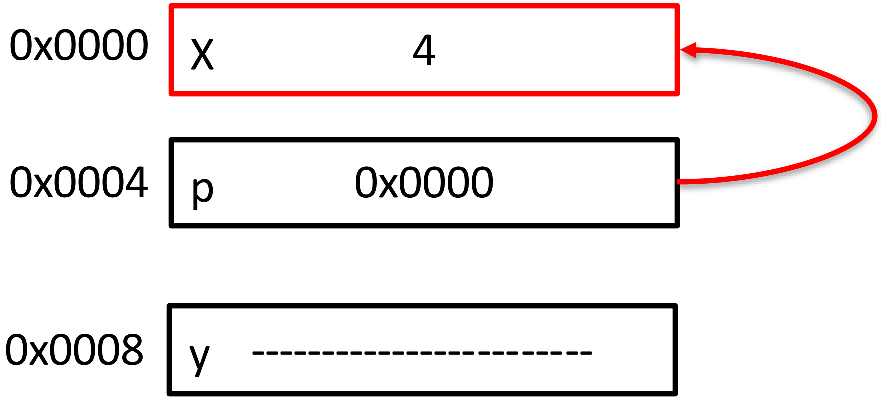
0x0000 X 4

0x0004 p 0x0000

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

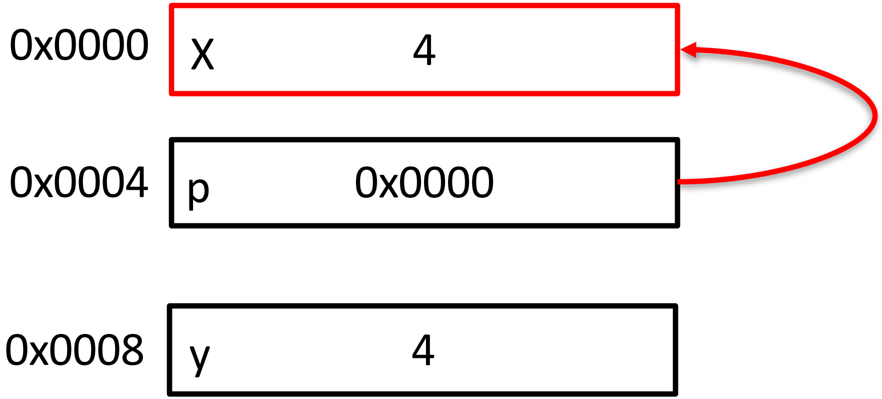
# POINTERS

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



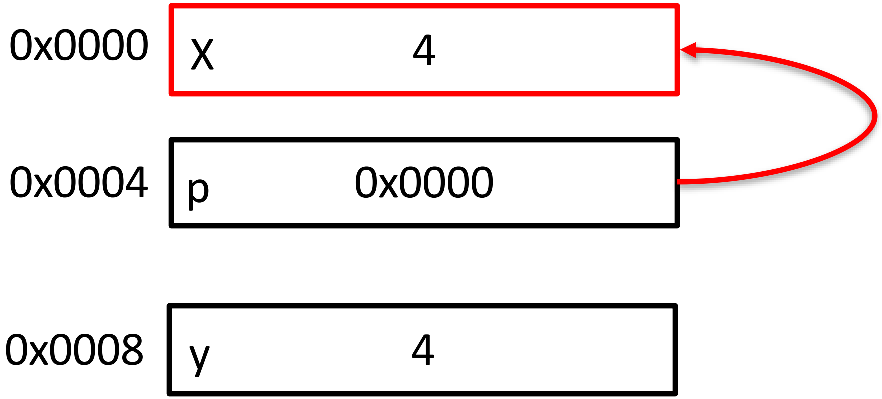
# POINTERS

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



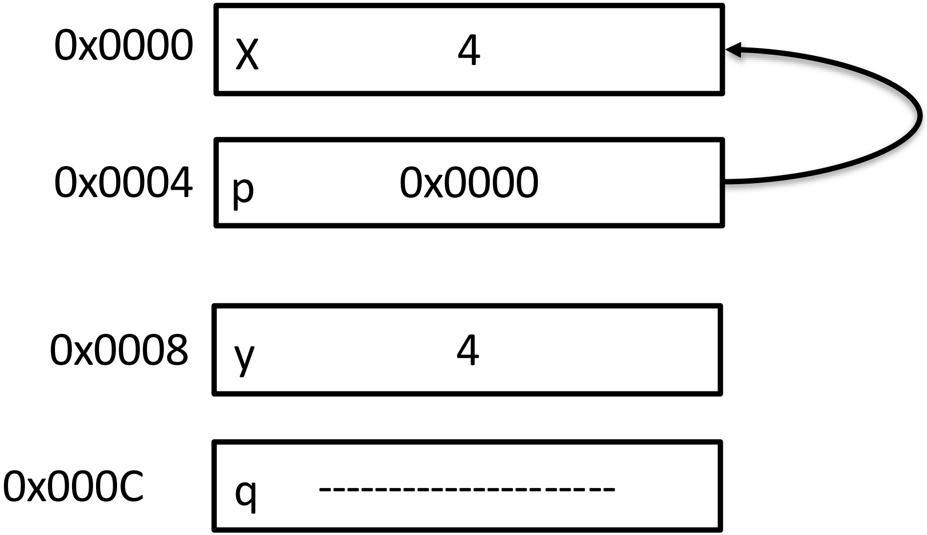
# POINTERS

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



# POINTERS

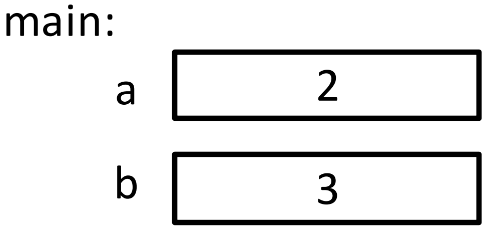
```
int x;
x = 3;
int *p;
p = &x;
*p = 4;
int y = *p;
Int *q =  &y;
*q = *p + 1;
q = p;
```



| | | |
|---|---|---|
| 0x0000 | X | 4 |
| 0x0004 | p | 0x0000 |
| 0x0008 | y | 4 |
| 0x000C | q | --------------------- |

# 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;
}
```



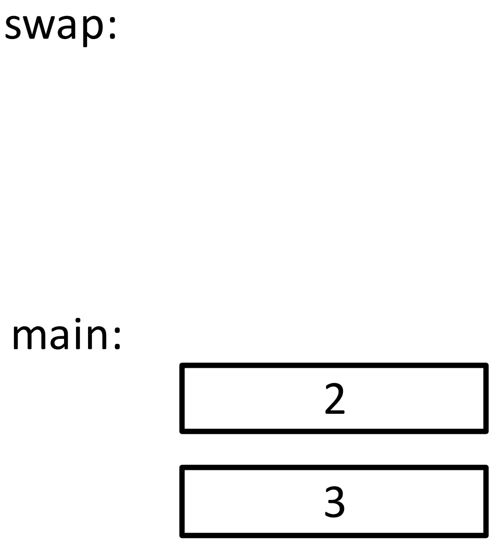
main:

a 2

b 3

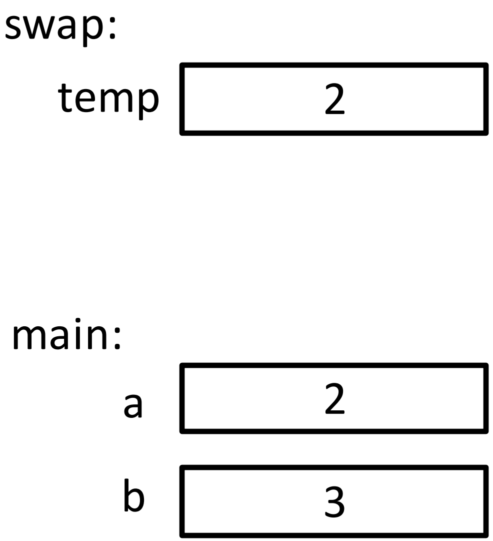
# SWAP EXAMPLE (BAD)

```
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;
}
```



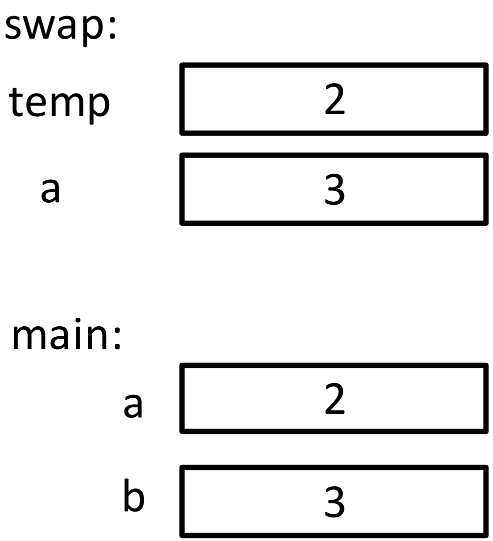
# SWAP EXAMPLE (BAD)

```
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 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

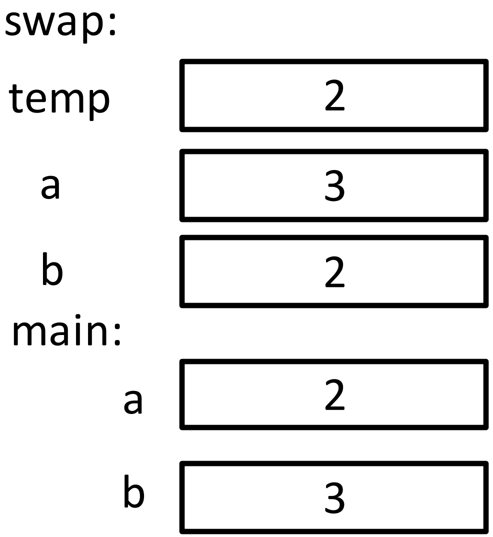
main:

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

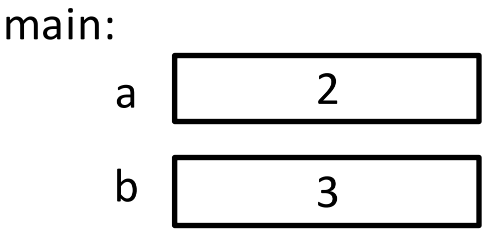
main:

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;
}
```

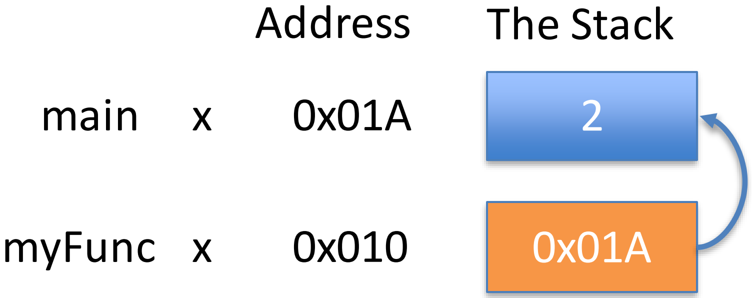
# EVERYTHING IN C IS PASS BY VALUE

```
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 |

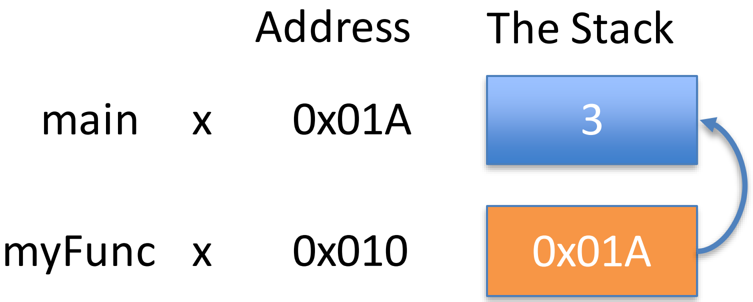
# EVERYTHING IN C IS PASS BY VALUE

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

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



# EVERYTHING IN C IS PASS BY VALUE

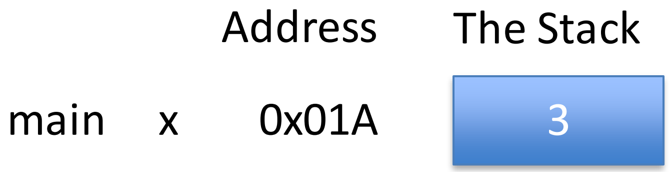
```
void myFunc(int *intPtr) {
    *intPtr = 3;
}

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;
}
```

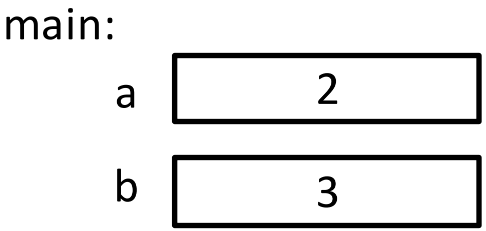


| | | Address | The Stack |
|---|---|---|---|
| main | x | 0x01A | 3 |

# 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;
}
```



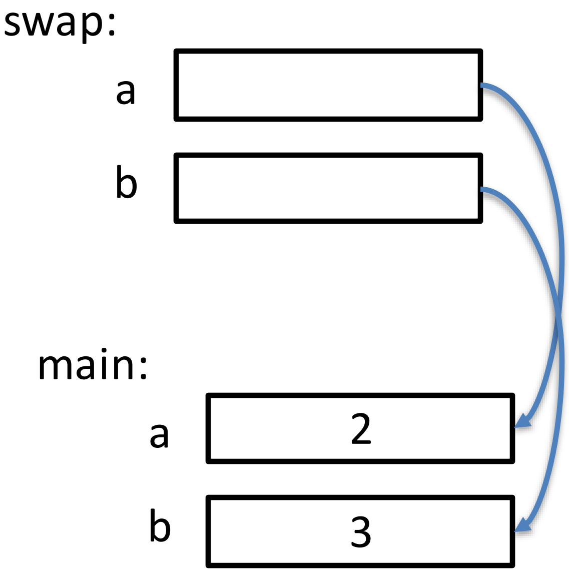
main:

a 2

b 3

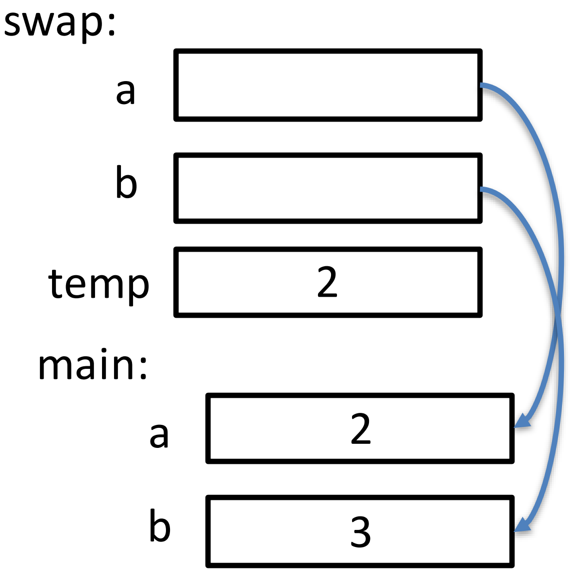
# SWAP EXAMPLE (FIXED)

```
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;
}
```



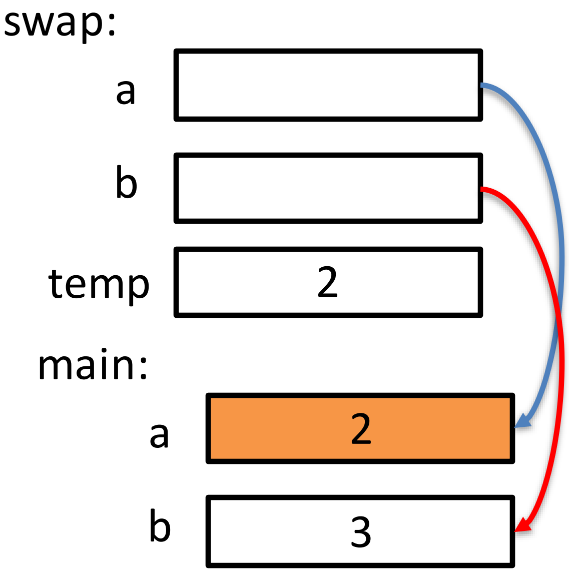
# SWAP EXAMPLE (FIXED)

```
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;
}
```



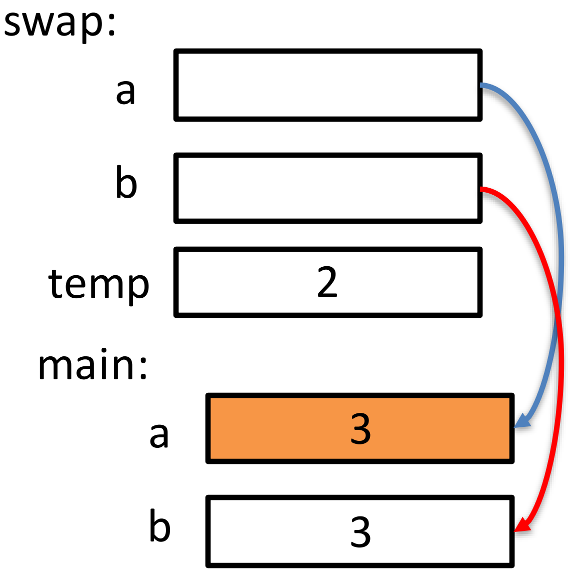
# SWAP EXAMPLE (FIXED)

```
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;
}
```



# 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

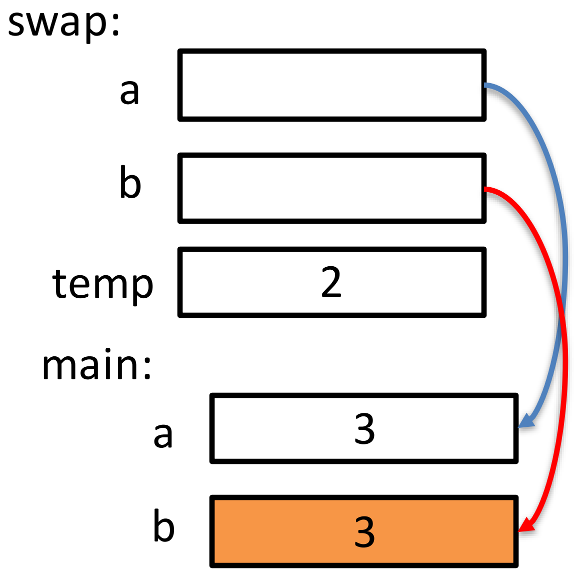
main:

a 3

b 3

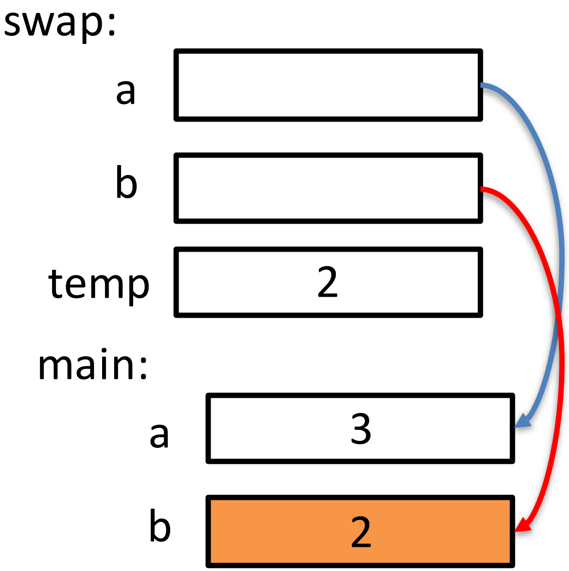
# SWAP EXAMPLE (FIXED)

```
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:

a

b

temp 2

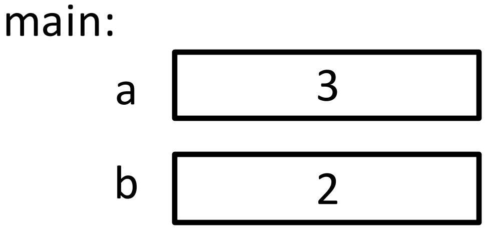
main:

a 3

b 2

# 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;
}
```



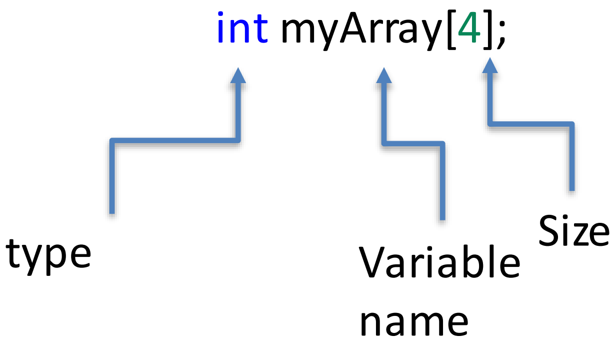
main:

a 3

b 2

# ARRAYS IN C

# THIS ONE WAY TO DECLARE AND ARRAY



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

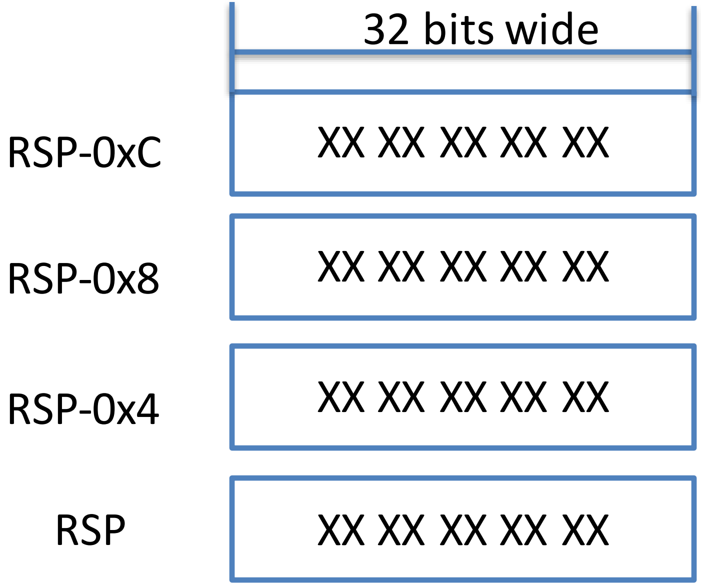
type

Variable name

Size

# THIS IS HOW ARRAYS ARE REPRESENTED IN MEMORY

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



32 bits wide

RSP-0xC XX XX XX XX XX

RSP-0x8 XX XX XX XX XX

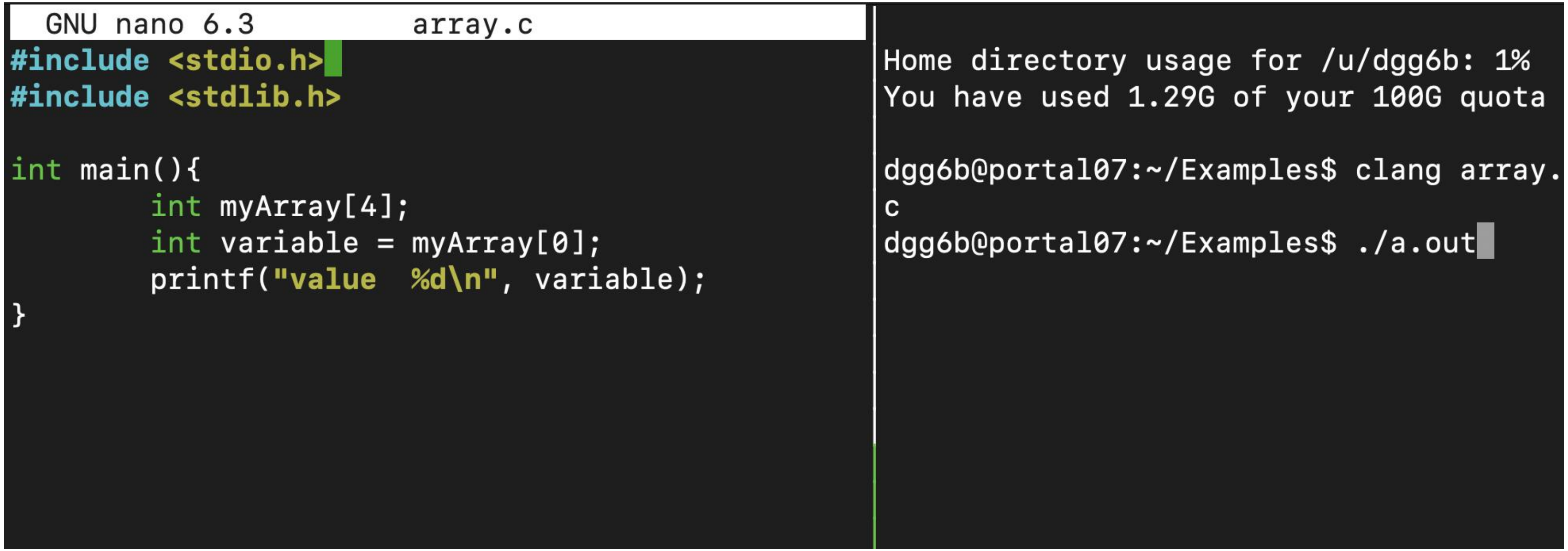
RSP-0x4 XX XX XX XX XX

RSP XX XX XX XX XX

# THIS IS HOW YOU ACCESS AND ELEMENT

```
int myArray[4];

int variable = myArray[0];
```

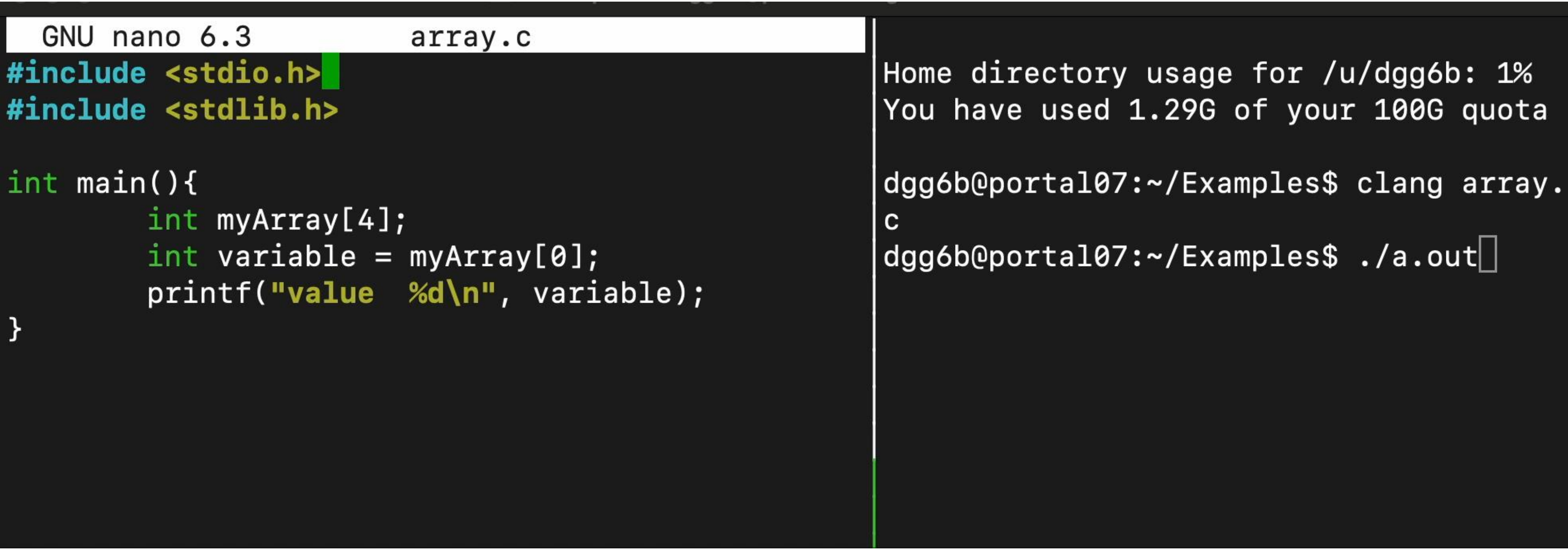
# WHAT DO WE THINK THIS WILL PRINT



```
  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);
}
```

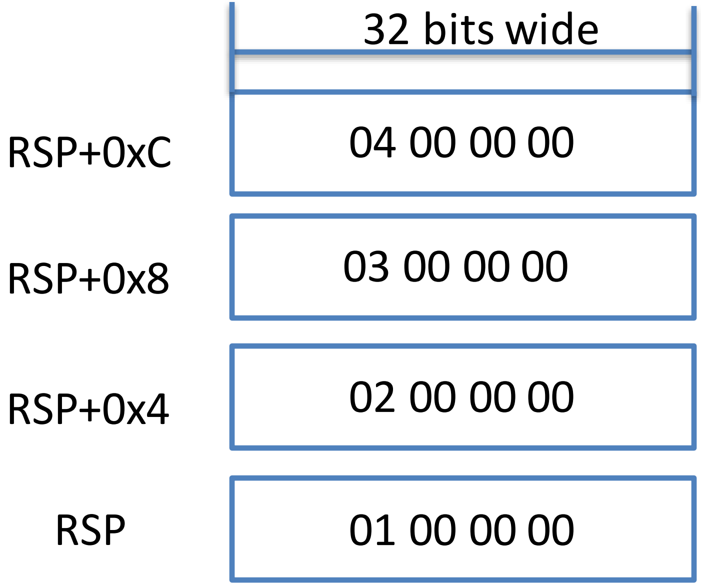
```
Home directory usage for /u/dgg6b: 1%
You have used 1.29G of your 100G quota

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

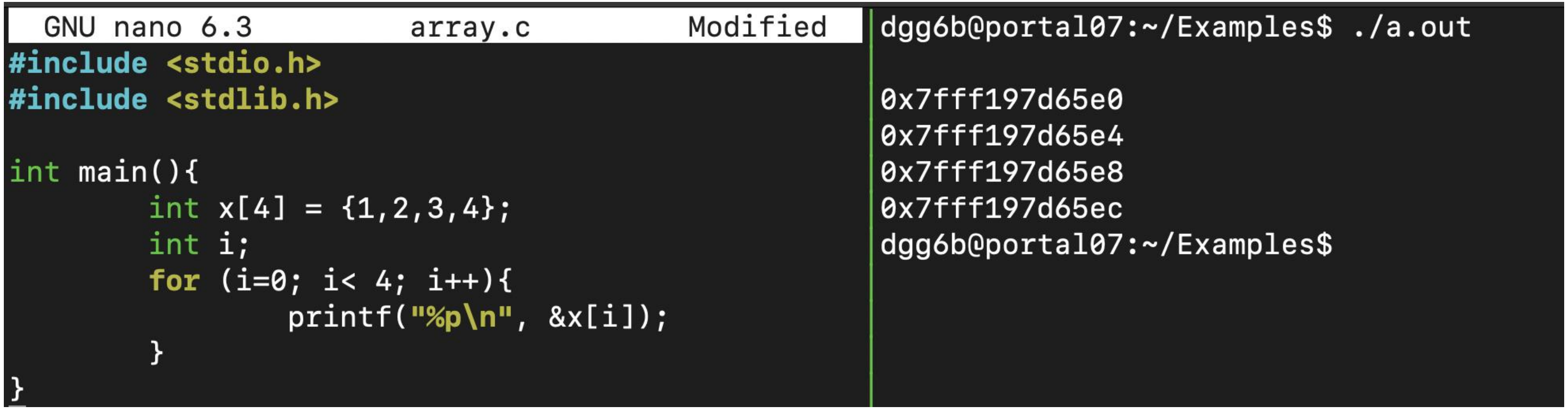
```
int x[4] = {1,2,3,4};
```



32 bits wide

| | |
|---|---|
| RSP+0xC | 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

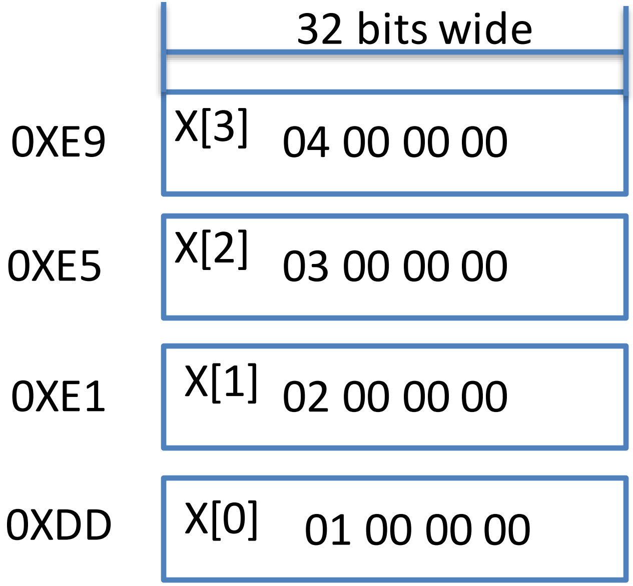
```
int x[4] = {1,2,3,4};
```

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 |

# 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 and 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 |

# 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 |

# ARRAY SYNTAX AND POINTERS

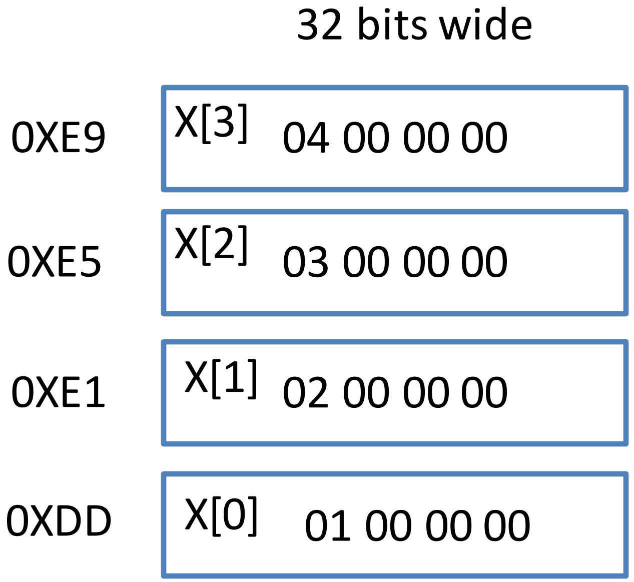
```
int x[4] = {1,2,3,4};

*(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 |

# ARRAY SYNTAX AND POINTERS

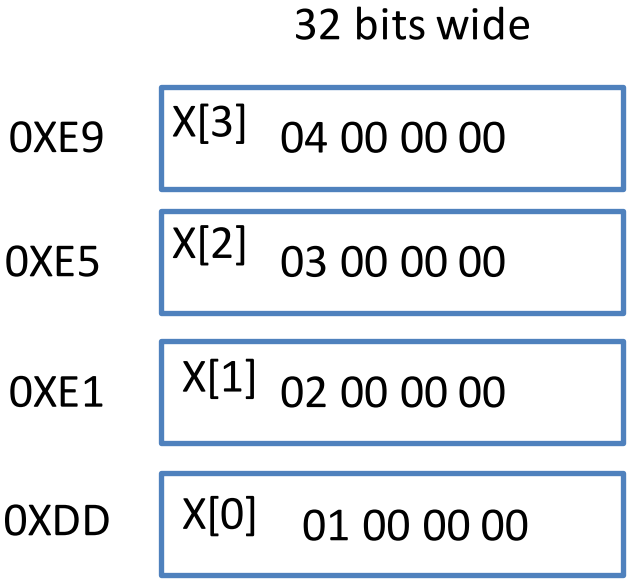
```
int x[4] = {1,2,3,4};

*(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 |

# ARRAY SYNTAX AND POINTERS

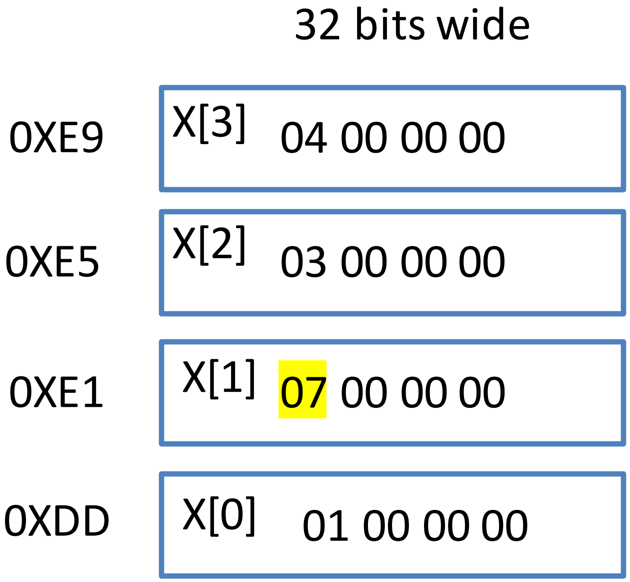
```
int x[4] = {1,2,3,4};

*(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

# POINTER ARITHMETIC RULE

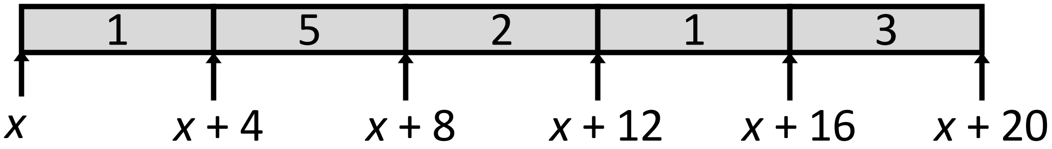
When do arithmetic operation using on pointer variables constants are treated as a multiple of size of the pointer type.

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

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

# ARRAY ACCESSES

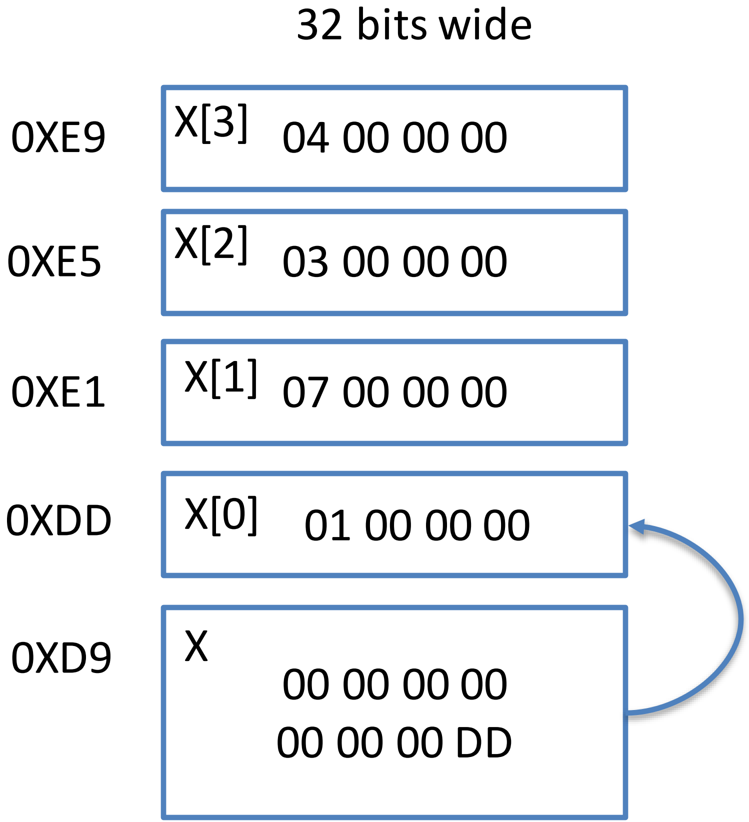
`int val[5];`



| 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 *`** | $x + 4\,i$ |

# ARRAY SYNTAX AND POINTERS

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

*x = *x + 1;
```

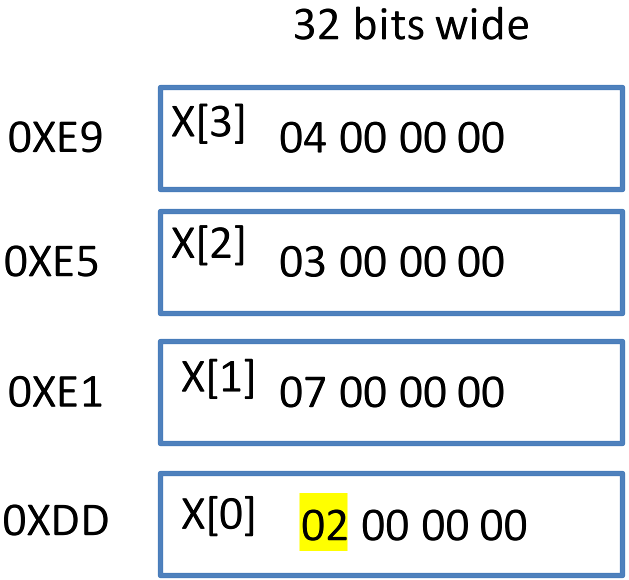


32 bits wide

| | | |
|---|---|---|
| 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<br>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

# 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]

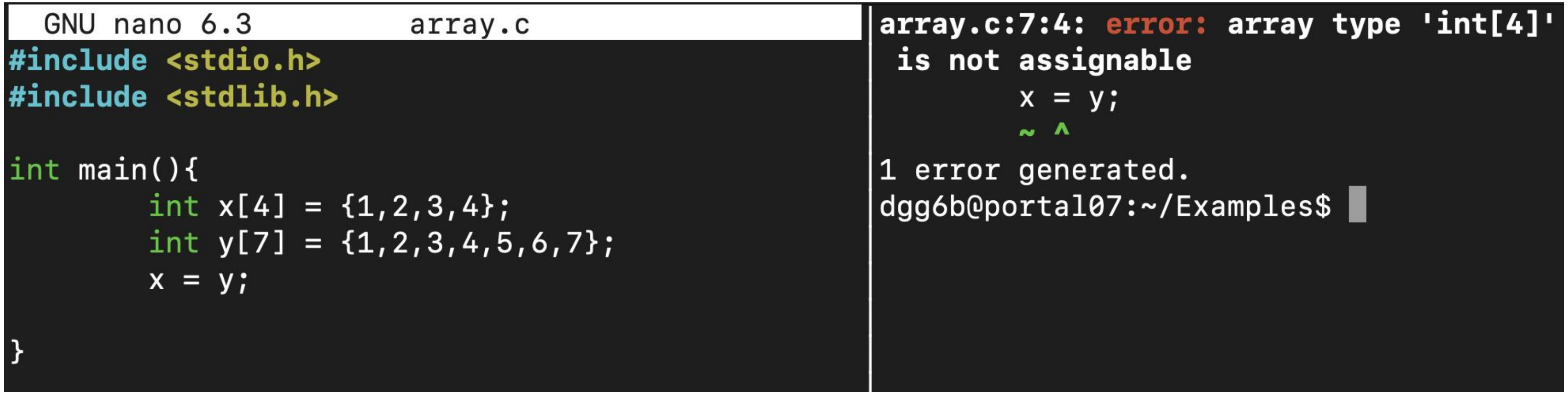
Arrays are of type int [n] and language doesn't allow these to be assigned

# ARRAY NOT QUITE POINTERS

```
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
(memcpy(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 by the language

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

Allowed

pointer = array

Not allowed by the language

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

Because array types

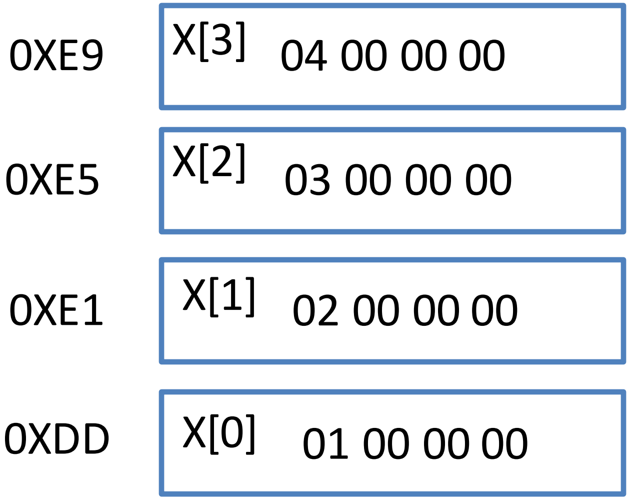
int[4] is not assignable

# 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

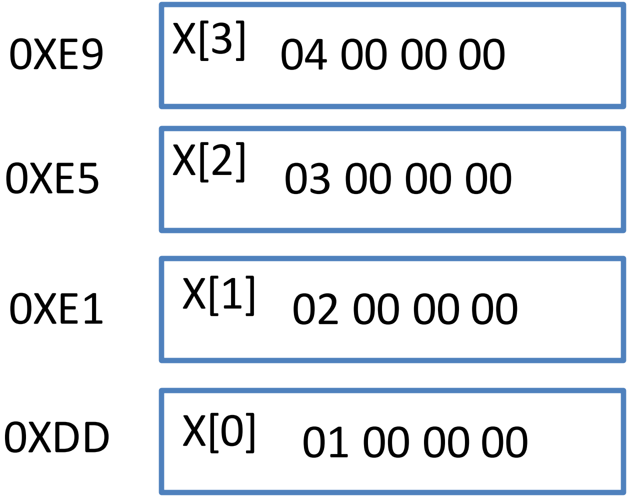
0XE1 X[1] 02 00 00 00

0XDD X[0] 01 00 00 00

# TALK TO YOUR NEIGHBOR

```
x = x + 1;

printf(“value: %d”, x[1]);
```

What does this print out?

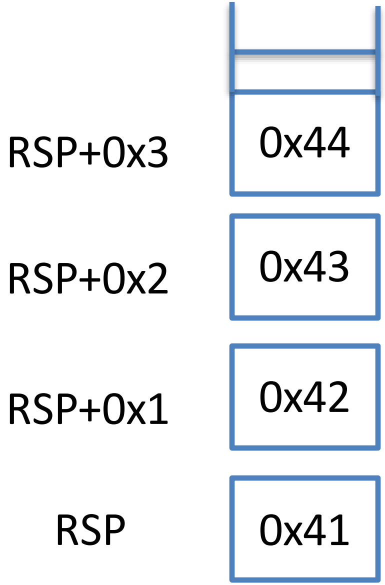


| | | |
|---|---|---|
| 0XE9 | X[3] | 04 00 00 00 |
| 0XE5 | X[2] | 03 00 00 00 |
| 0XE1 | X[1] | 02 00 00 00 |
| 0XDD | X[0] | 01 00 00 00 |

# ARRAY IN C

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

8 bits (1 byte) wide



| | |
|---|---|
| RSP+0x3 | 0x44 |
| RSP+0x2 | 0x43 |
| RSP+0x1 | 0x42 |
| RSP | 0x41 |

# 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”;
```

The & of an array is the & of its first element (i.e., &array == &(array[0])).