A C Primer (7): Pointer Arithmetic & Structures

Ion Mandoiu
Laurent Michel
Revised by M. Khan and J. Shi

Pointers are addresses

- The **value** of a pointer is a **byte address**
  - Unsigned integer used to number bytes in memory
  - Range is between

    0x00000000 and 0xFFFFFFFF [32-bit]

    0x0000000000000000 and 0xFFFFFFFFFFFFFFFF [64-bit]
- Corollary
  - If a pointer is an integer, you can do ***arithmetic…***
  - To ***compute*** other addresses

```
int * p = malloc(sizeof(int)*10);
int * q = p + 1;
```

Pointer Addition Example

Suppose p is a pointer to an int, and its value is 1000

p + 1 is not the next byte address.

It is the address of next item (of type int)

Address	Value		To access values		
1020		p + 5	*(p+5)	OR	p[5]
1016		p + 4	*(p+4)	OR	p[4]
1012		p + 3	*(p+3)	OR	p[3]
1008		p + 2	*(p+2)	OR	p[2]
1004		p + 1	*(p+1)	OR	p[1]
1000		p = 1000	*p	OR	p[0]
996		p - 1	*(p-1)	OR	p[-1]
992		p - 2	*(p-2)	OR	p[-2]

Adding a pointer and an integer

- Add an integer to a pointer, the result is a pointer of the same type
  - It is different from regular integer addition
  - The integer is **automatically scaled** by the size of the type pointed to

- Suppose p is a pointer to type T, and k is an integer

  Then both `p + k` and `k + p`
    - Are valid expressions that evaluate to a pointer to type T
    - Have a byte address equal to

```
(unsigned long)(address stored in p) + k * sizeof(T)
```

C standard does not allow arithmetic on void *

gcc has an extension, treating sizeof(void) as 1

Pointers Subtraction

- Subtract one pointer from another: both must have the same type
- The result is the number of data items between the two pointers!
    - Not the number of bytes

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

int main() {
   int *p = malloc(sizeof(int)*10);
   int *last = p + 9;
   int  dist = last - p; // both are int *
   printf("Distance is %d\n",dist);
  free(p);
  return 0;
}
```

Output

```
$ gcc ptrsub.c
$ ./a.out
Distance is 9
$
```

Pointer comparisons

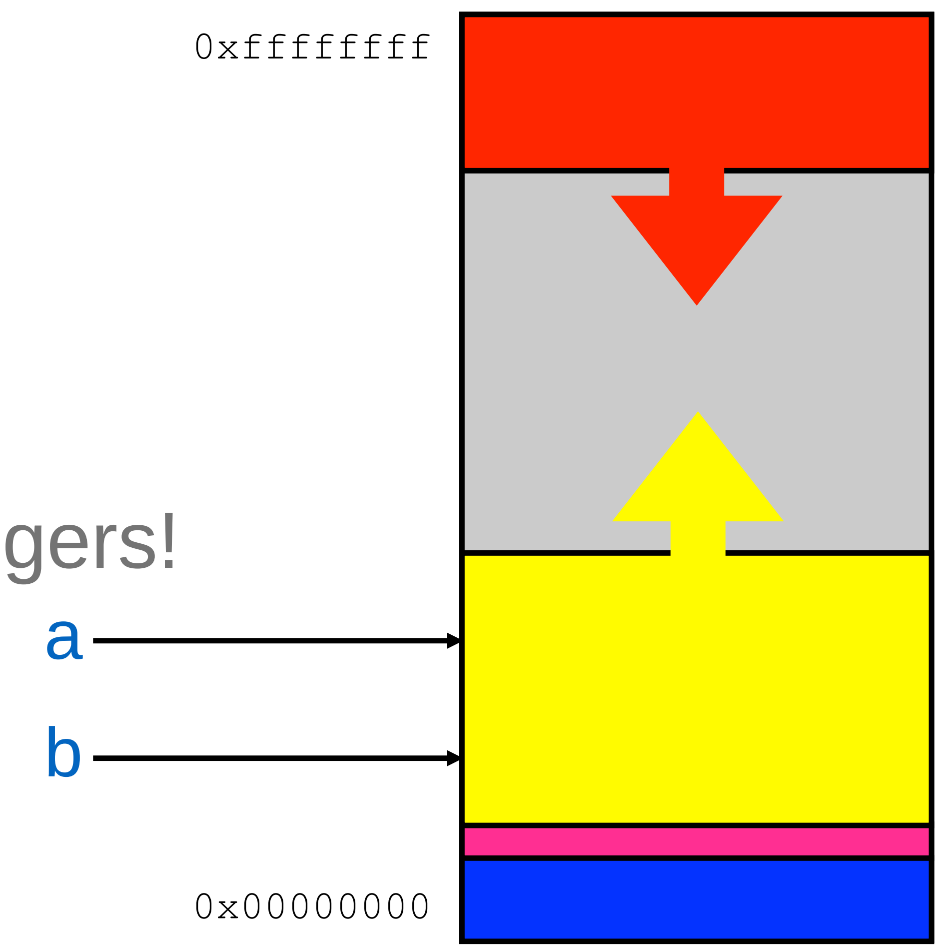
- You can also compare two pointers

  `<    >    <=    >=    !=    ==`

- Purpose
  - Check boundary conditions in arrays
  - Manually manage memory blocks
- Semantics
  - Simply based on memory layout!
  - Compare bits in pointers as unsigned integers!

Check if you are done:

```
while (b < a) {
    // do something
}
```



Effect of casting types ?

- If you cast a pointer type…
  - Any subsequent pointer arithmetic will use the type you chose
- Do not want scaling? Casting a pointer to (**char ***)
  - Because sizeof(char) is 1

```
int * t;
char * p = (char *) t + 8;   // 8 is not scaled
char * q = (char *) (t + 8); // 8 is scaled
```

Arrays and pointers

- Arrays and pointers can often be used interchangeably

```
int  a[10];
int *p = a;

// all of the following evaluate to the value of a[0]
*p                p[0]        *a            a[0]

// all of the following evaluate to the value of a[1]
*(p+1)            p[1]        *(a+1)        a[1]

// all of the following evaluate to the address of a[0]
// type is int *
p                 &p[0]        a            &a[0]
```

"array of int" becomes "pointer to int" (array decay)

Example: arrays and pointers

- Equivalent ways of initializing an array

```
int a[10], *p = a;  // not *p = a; it is int *p; p = a;

for(int i=0; i<10; i++) a[i] = i;  // array indexing

for(int i=0; i<10; i++) p[i] = i;  // indexing via pointer

for(int i=0; i<10; i++) *(p+i) = i; // explicit pointer arithmetic

for(i=0; i<10; i++) i[p] = i;    // obfuscated but valid C!

for(i=0; i<10; i++) *p++ = i;    // common pointer use idiom
```

```
int * tp = p;
p ++;
*tp = i;
```

Arrays and pointers are NOT the same

```
int   a[10];
int   *p = a; // a is converted to int *

// a is still an array after &
&a   // pointer to array of 10 int's  int (*)[10]
&p   // pointer to a pointer to int   int **

// a is still an array after sizeof
sizeof(a) // 40 because a is an array of 10 int's
sizeof(p) // 8  because p is a pointer

p++;   // can increment p
a++;   // cannot increment a; this will not compile
       // Similar to n++ vs 2++
```

Example: Arrays and pointers are NOT the same

```
#include <stdio.h>

void foo(int *x)
{
   printf("%lu\n", sizeof(x));
}

int main()
{
   int a[10], *p;

   p = a;  // a is converted to int *
   // a is still an array in sizeof
   printf("%lu %lu\n", sizeof(a), sizeof(p));
   foo(a); // a is converted to int *
}
```

Output

```
% gcc array.c
% ./a.out
40 8
8
```

Structures

- Mechanism to define new types
  - Also known as “compound types”
  - Used to aggregate related variables of different types
- Structures type declaration
  - Structures can have a **type name**
  - Can have “members” of any type
    - Basic types
    - Pointers
    - Arrays
    - Other structures
- Structure variable definition
  - Specifies **variable name**

```
struct student_grade {
    char *name;
    int  id;
    char grade[3];
};
…
struct student_grade student1;
struct student_grade
    student2, student3;
struct student_grade all[200];
```

Structure example

```
struct Person {
   int      age;
   char gender;
};

int main(){
   struct Person p;

   p.age = 44;
   p.gender = 'M';

   struct Person q = {44, 'M'};

   return 0;
}
```

Structure *type declaration*

Structure *variable definition*

Syntax for field access
similar to Java and Python

Structure *variable definition*
and *initialization*

Example: Array of Structures

- Member name is a char array
- Caveats
  - Names cannot be more than 31 characters long
  - Four persons in family
    - Indexed 0..3
- Array of structures for the whole family
  - Nested initializers

```c
#include <stdlib.h>

struct Person {
   char   name[32];
   int    age;
   char   gender;
};

int main()
{
   struct Person family[4] = {
      {"Alice",34,'F'},
      {"Bob",40,'M'},
      {"Charles",15,'M'},
      {"David",13,'M'}
   };
   int juniorAge = family[3].age;
   return 0;
}
```

typedef

- Struct names can become long
- C provide the ability to define type abbreviations
  - **typedef** declaration
    - Give existing type new type name
- Make code more readable

```
struct Person {
   char   name[32];
   int    age;
   char   gender;
};
typedef struct Person TPerson;
int main()
{
    TPerson family[4];
    ...
    return 0;
}
```

- Structure and typedef declarations often combined

```
typedef struct Person {
   char   name[32];
   int    age;
   char   gender;
} TPerson;
```

Operations on struct

```
TPerson a, b, c;

…
a = b;
c = searchPerson("name");
```

- Assignment
  - All struct members copied
- Can be passed to functions
  - **By value**
  - Even if some members are arrays!
- Can be returned from a function

- If pass by value, cannot change members in functions
- Passing or returning large structures can be costly

Use pointers to structures!

Pass structure by reference

```
typedef struct Person {
   char   name[32];
   int    age;
   char   gender;
} TPerson;

TPerson * init_Person(TPerson *p, char * name, int age, char gender)
{
   strcpy(p->name, name);  // (*p).name
   p->age = age;           // (*p).age
   p->gender = gender;     // (*p).gender
   return p;
}

// Study the demo code is in the demo repo
// especially the lines that copy name
```

Structure Alignment

- Structure members are ***aligned*** for the natural types

Alignment requirements on x64 architecture	
char	1
short	2
int	4
long	8
float	4
double	8

```
struct struct_random {
    char    x[5];// bytes 0-4
    int     y;   // bytes 8-11
    double  z;   // bytes 16-23
    char    c;   // byte 24
};      // Total size 32
```

```
struct struct_sorted {
    double  z;   // bytes 0 - 7
    int     y;   // bytes 8 - 11
    char    x[5];// bytes 12 - 16
    char    c;   // byte  17
};       // Total size 24
```

Study the remaining slides yourself.

Also study the demo code!

Arrays and pointers

- Copying arrays

```
// using array indexing
void copy_array0(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    target[i] = source[i];
}

// using pointers
void copy_array1(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    *target++ = *source++;
}
```

```
int * tp = source;
source ++;
*target = *tp;
target ++;
```

Typecasting Pointers

- C lets you cast pointers in any way you like
- You can “forge” pointers to point wherever you wish….

```
float f;
char p = * (char *) &f;     // 1st byte representing f
char ch = * (char *)1000;  // access any byte in memory
```

That’s what makes C very attractive for

low-level programming

That is also very **powerful** and thus **dangerous**!

Returning more than one value from functions

- Use references (caller prepares the storage)

```
long int strtol (const char* str, char** endptr, int base);
```

- Use arrays (caller prepares the storage)

```
int pipe(int pipefd[2]);
```

- Return a structure (costly if structure is large)
- Return a pointer (to array or structure)
  - Must be dynamically allocated or static. RTM (read the manual)!

```
char * strdup(const char *str1);

struct tm *localtime( const time_t *time );
```

- Use global variables (**DON'T!**)

```
errno
```

Typedef

```
// Example of typedef. Think about how you would define a variable
typedef    int     BOOL;
typedef    char    name_t[100];
typedef    char    *Pointer;
```

Self-referential structures

```
struct Person {
   int    age;
   char   gender;
   char   name[32];
   struct Person * parents; // A pointer to this type of struct
} person1, person2;  // Can define variables here
```

Self-referential structures - 2

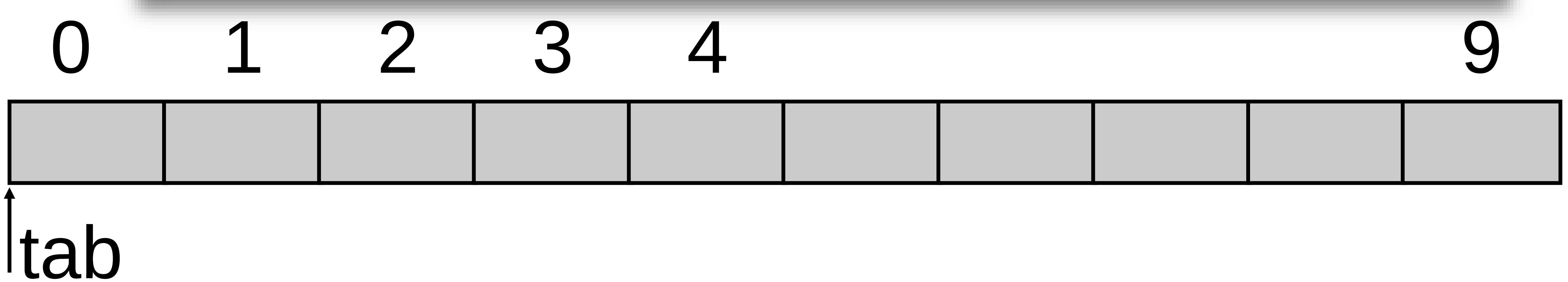
```
struct student {
   char   name[128];
   // Can have a pointer to a struct defined later.
   // However, you cannot define an array of book here (e.g. books[8])
   struct book * books;
};
struct book {
   char   title[128];
   struct student * owner;
   struct book * next; // A pointer to this type of struct
};
```

Example of Pointer Arithmetic

- Simple illustration

```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]    = 10;
   int *p    = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```

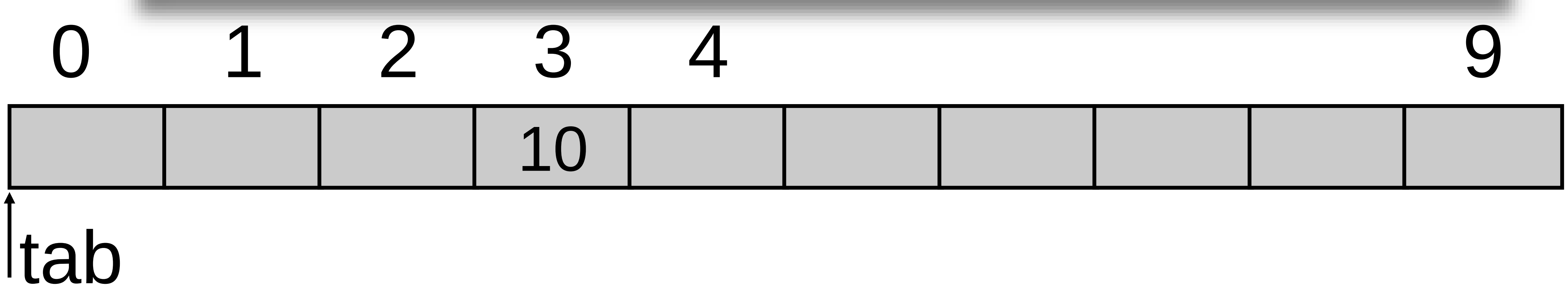


Example

- Simple illustration

```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]    = 10;
   int *p    = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```

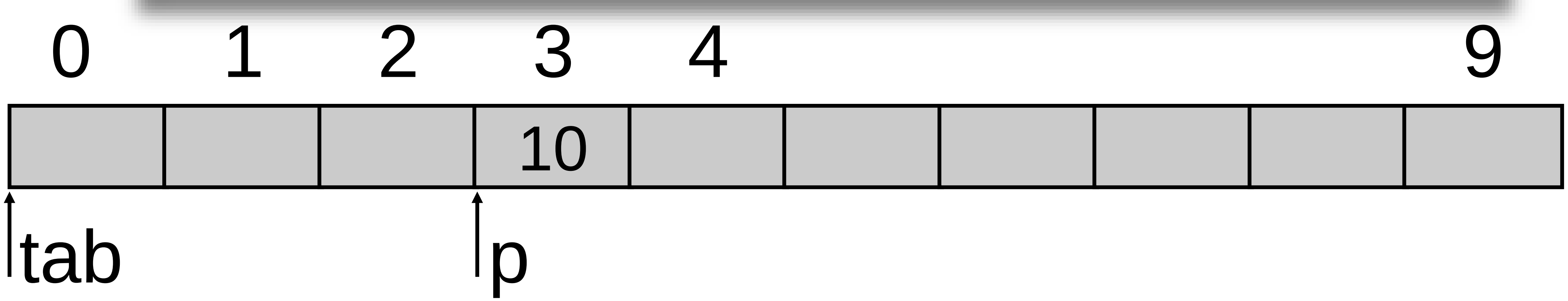


Example

- Simple illustration

```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]    = 10;
   int *p    = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```

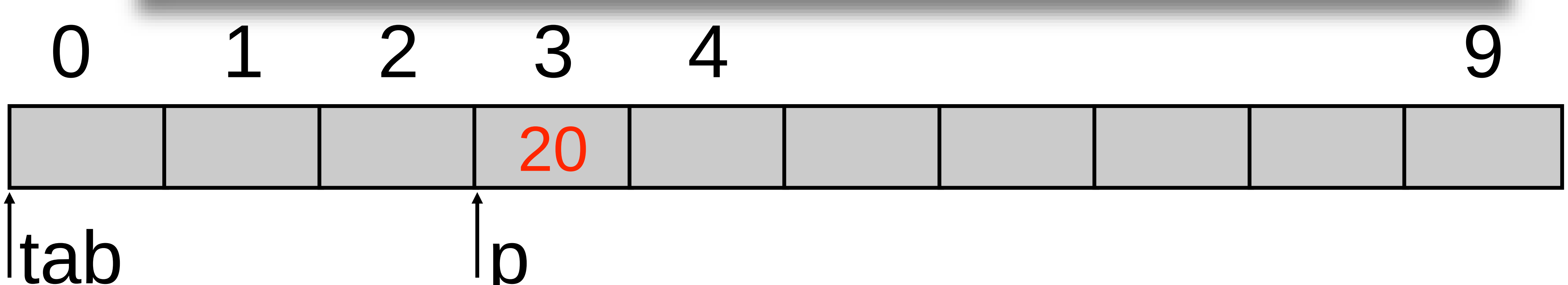


Example

- Simple illustration

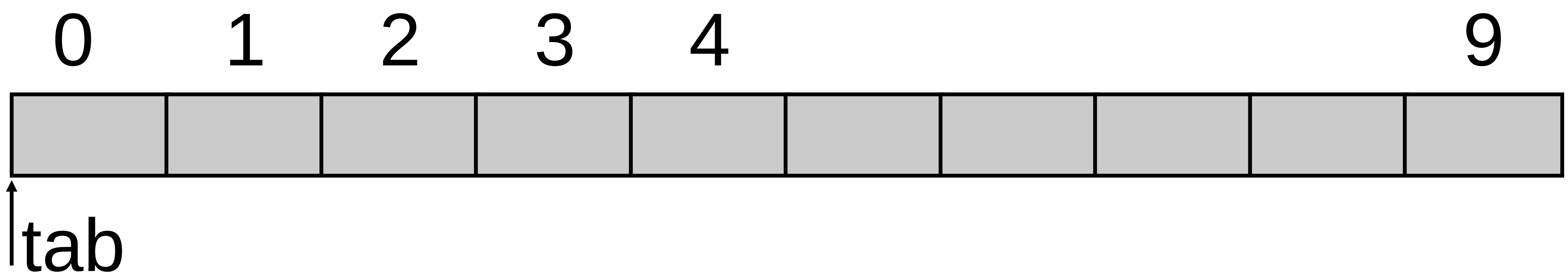
```
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]   = 10;
   int *p   = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```



But what about memory addresses?

- Same story…!



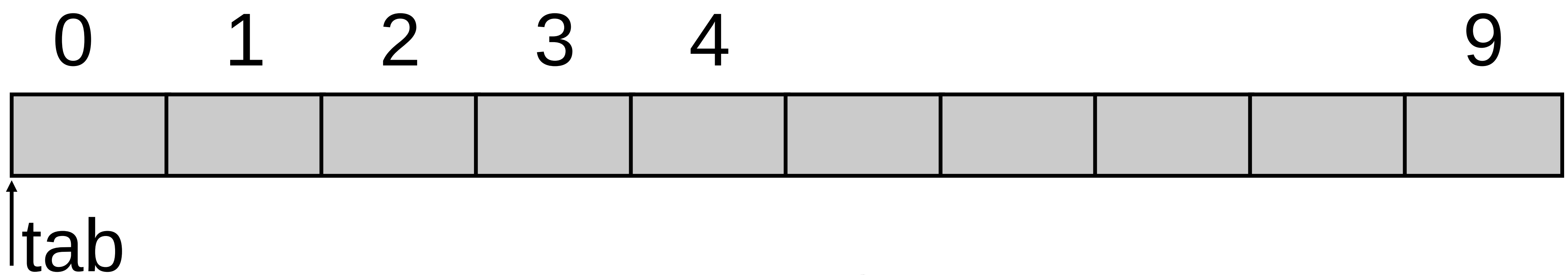
`0x11110000`

tab+1 = ?

`0x11110001` ?

But what about memory addresses?

- Same story…!



0x11110000

tab+1 = ?

~~0x11110001 ?~~

0x11110004 ?

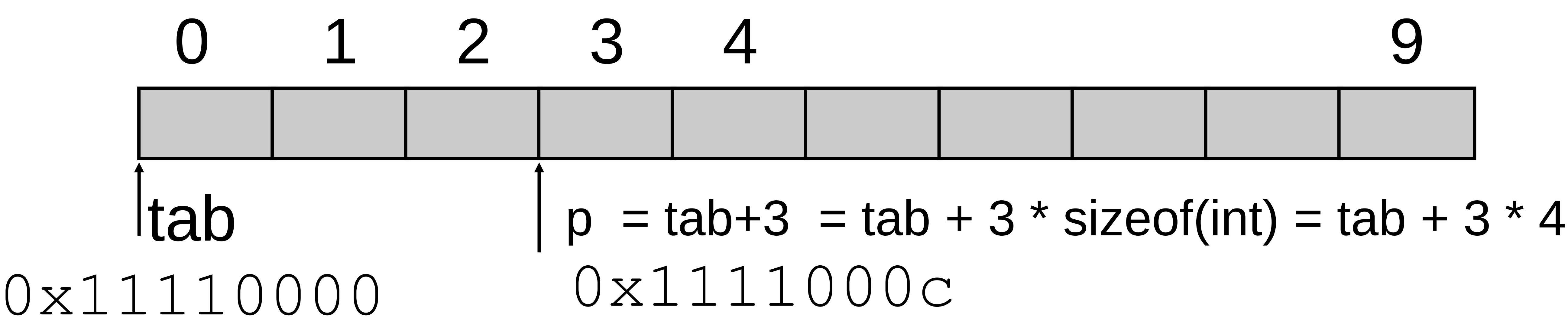
WHY?
Simply because tab is a pointer to an int and an int is 4-bytes wide!

Bottom line

```
#include <stdlib.h>

int main()
{
    int *tab = (int*)malloc(s
    tab[3]    = 10;
    int *p    = tab + 3;
    printf("What is at tab+3?
    *p = 20;
    printf("What is at tab[3]? = %d\n",tab[3]);
    return 0;
}
```

The offset **3** is scaled by the compiler with the size of the type to get an address in bytes



Memory alignment requirements

- Memory used to store a value of type X **MUST**
  - **be lined-up on a multiple of natural alignment for X**
- Why?
  - Performance!
- If you do not respect alignment requirements…
  - BUS ERROR (sigbus)
  - The O.S. will <u>kill</u> your program

Good news and bad news

The C compiler handles alignment automatically 99% of the time

Programmers have to handle the rest

- When you do pointer arithmetic of course!
  - Do not assume the location of the fields
- When you call system routines with specific alignment needs
  - Your arguments must comply
  - Use compiler annotations to force specific alignments (beyond our scope, simply remember that this exists!)

Example

```c
#include <stdio.h>
struct Person {
   char   name[32];
   int    age;
   char   gender;
};
typedef struct Person TPerson;

TPerson init(char name[], int age, char gender) {
  TPerson p;
  int i;
  for(i = 0; name[i]>0; i++)
     p.name[i] = name[i];
  p.name[i] = '\0';
  p.age = age;
  p.gender = gender;
  return p;
}

void print_info(TPerson p) {
  printf("name: %s, age: %d, gender: %c\n",
         p.name, p.age, p.gender);
}
```

```c
int main() {
    TPerson family[4];
    family[0] = init("Alice",34,'F');
    family[1] = init("Bob",40,'M');
    family[2] = init("Charles",15,'M');
    family[3] = init("David",13,'M');
    print_info(family[0]);
    print_info(family[1]);
    print_info(family[2]);
    print_info(family[3]);
    family[1] = family[0];
    print_info(family[1]);
    return 0;
}
```

Output

```
./a.out
name: Alice, age: 34, gender: F
name: Bob, age: 40, gender: M
name: Charles, age: 15, gender: M
name: David, age: 13, gender: M
name: Alice, age: 34, gender: F
```