C Basics

Angel Noé Martínez González

July 2, 2015

Composed Types I

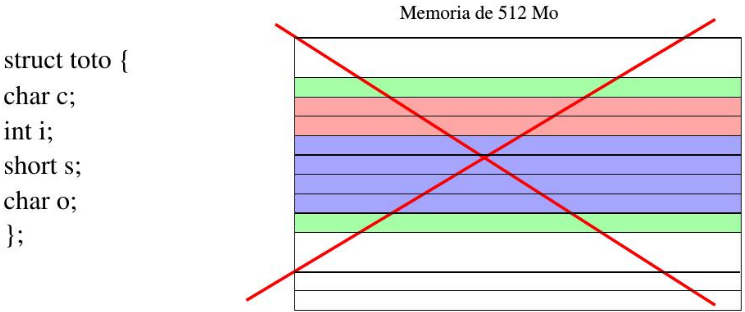
Typedef Definitions

```
typedef struct personStruct {
unsigned int edad;
unsigned int altura;
}persona;

persona angel, jorge;
```

COMPOSED TYPES II

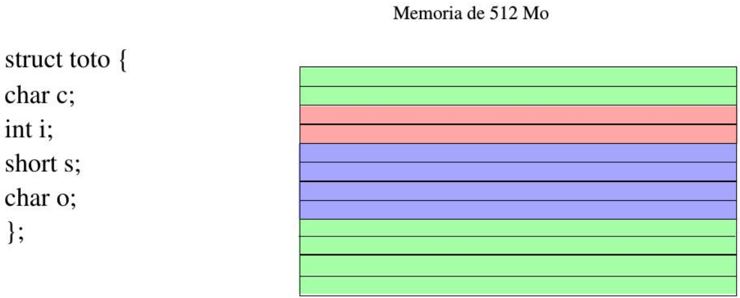
DATA ALIGNMENT



What's the size?

COMPOSED TYPES III

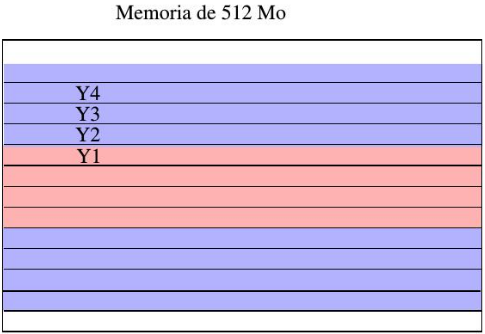
The compiler pads with extra bytes, but why?



The compiler adds extra data to align and to facilitate access.

COMPOSED TYPES IV

The machine access to memory by buses of size depending on the granularity of the OS: 2, 4, 8 and 16 bytes.



Thinking the organization of data helps alignment. Can you think a best ordering in toto struct?

Pointers I

A pointer is a type that refers a data in memory, contains

- Memory address
- The way to read data, i.e. data type.
- Access to the value example

```
int a=10;
int* a_ptr = &a;
*a_ptr = a*a;
a_ptr[0] = *a_ptr + 1;
```

- Operator priority level

Pointers II

```
int a=10;
int* a_ptr = &a;
int c = a**b**b;
printf("%d", c);
```

- Composed types pointers

```
person jorge;
person* jorge_ptr=&jorge;
jorge_ptr->edad = 30;
(*jorge_ptr).edad = 30;
```

Pointers III

Arithmetic: a pointer is an integer, we can then

- Add an integer to a pointer, which result a translated pointer in memory.
- Subtract an integer to a pointer, which result a translated pointer in memory.
- Subtract a pointer to a pointer, which result in the memory between these pointers.

All these results are given in units corresponding to the data of the pointers not just bytes.

POINTERS IV

When data structures are too big it can be very costly to pass variables to functions (it will copy). We can then pass the address of the variable to avoid data to be copied.

```
typedef struct{
int dummy;
...
} MyBigData;

void my_usage_func(MyBigData* data) {
// Some code
data->dummy;
}


MyBigData data;

my_usage_func(&data);
```

Pointers V

Double Pointers: a pointer has a given type, has a value stored in memory then we can define a pointer to a pointer

```
int a=10;
int* a_ptr=&a;
int** a_ptr_ptr = &a_ptr;
```

Pointers VI

Pointers of Functions

Memory Allocation I

A dynamic array survives outside the function call. Function malloc takes as parameter the number of bytes to be allocated and return a void *

```
int* myarray
      = (int*)malloc(sizeof(int)*sizeOfMyArray);
```

we need to cast the return value to the data type of our array. There is not initialization value for the arrays.

MEMORY ALLOCATION II

The function calloc does initialize the values to 0.

```
int* myarray
      = (int*)calloc(sizeof(int)*sizeOfMyArray);
```

Memory Allocation III

We need to be sure that our memory request was succesfull, on the other case malloc and calloc returns NULL

```
int* myarray
      = (int*)calloc(sizeof(int)*sizeOfMyArray);

if(myarray==NULL)
// Do something to manage the error
```

MEMORY ALLOCATION IV

MULTIPLE DYNAMIC ARRAYS

```
int** mydoublePointerArray =
      (int**)malloc(sizeof(int*)*myFirstDimentionSi
int i;
for(i=0; i<myFirstDimentionSize; i++)
    mydoublePointerArray[i] = (int*)malloc(sizeof(i
mydoublePointerArray[x][y] = z;
// Some code
for(i=0; i<myFirstDimentionSize; i++)
    if(mydoublePointerArray[i]!=NULL)
        free(mydoublePointerArray[i])

if(mydoublePointerArray!=NULL)
```

Memory Allocation V

```
free(mydoublePointerArray);
```

Always verify if the array points to some valid memory location.

Memory Allocation VI

Once we have finished using the memory, we need to return it to the system

```
if(myarray!=NULL)
    free(myarray)
```

Always verify if the array points to some valid memory location.