# Programming and Algorithms

## C: Arrays and structures.

Jean-Bernard Hayet

CIMAT, A.C.

# Variable type modifiers

# Variables and memory

In a program, variables are associated to some **well specified memory space**. This space may vary:

1. in size,

```
int  a     = 1;
double b = 2.0;
```

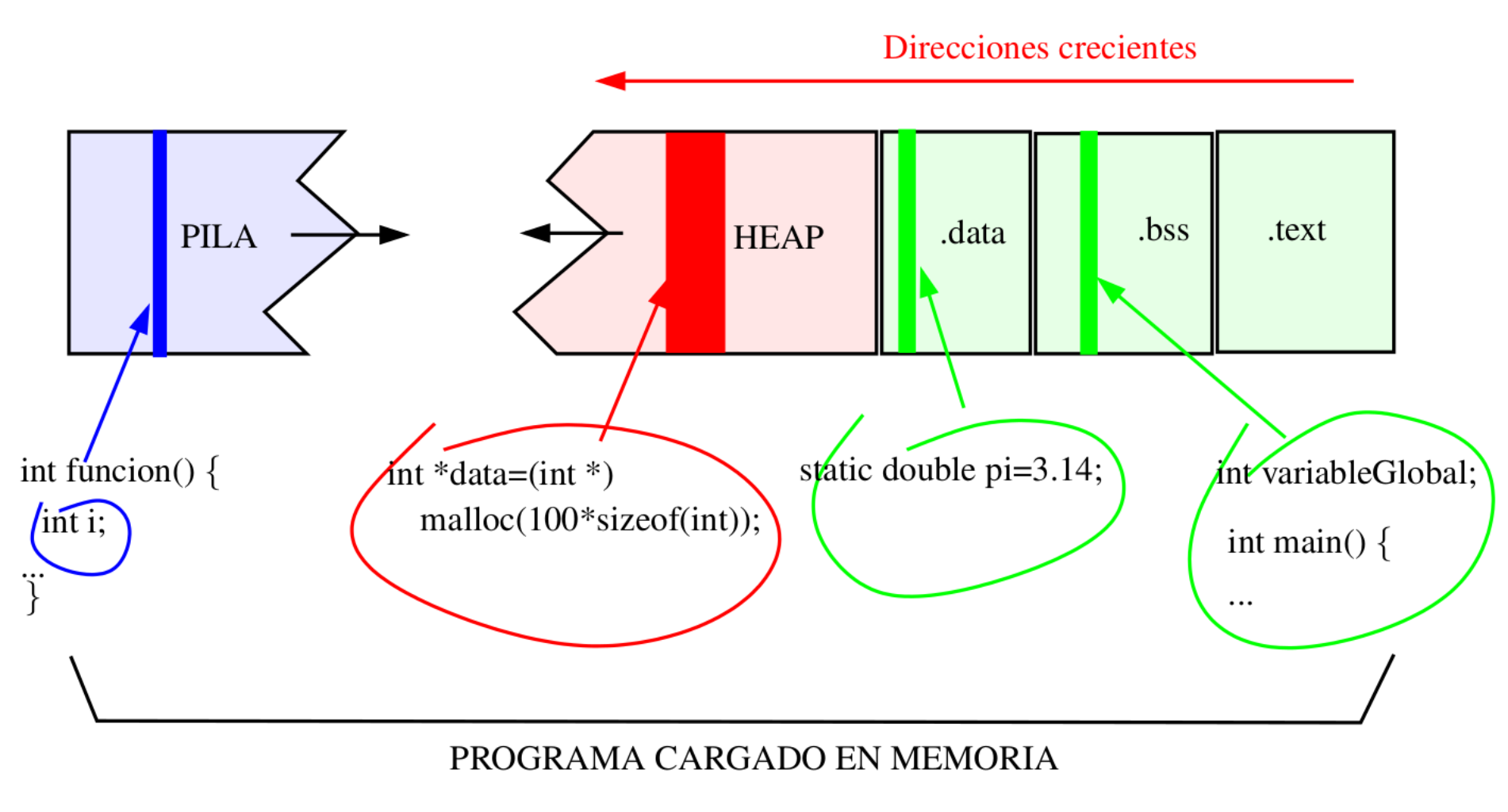
2. in position in memory,
3. in the possibility to modify the content at that position.

# Variables and memory

Three types of memory segment:

- **statically** allocated memory (specified **before execution**)
- memory on **stack**: for variables defined within a function,
- memory on the **heap**: for dynamic allocation.

# Variables and memory



# Variables and memory

- The last two types are, in a way, **short-lived**.
- In the case of memory on the heap, **the responsibility for creating/liberating memory is the one of the programmer**.
- In the case of memory on the stack, it is **automatic**.

# const

With the const modifier, the modified variable becomes an rvalue after initialization:

```
const int a = 1;
...
a = 3;
```

Does not compile:

```
error: assignment of read-only variable a
```

# static/automatic

For variables defined within functions: **Makes them exist out of the stack**.

```
void myFunction() {
        int i;
        static counter = 0;
        ...
        counter++;
}
```

Opposed to automatic (default behavior) which is that variables within blocks/functions disappear at leaving these blocks.

# register

- This modifier makes that the variable, when possible, is kept in a **CPU register** (not in memory).
- Useful when, **locally, a variable is used many times** (e.g. an index in a for loop).
- Compilers may decide by themselves when to put this modifier.
- By definition, it makes no sense using the operator &.

# extern

- Default storage class of all **global variables**.
- With the extern modifier, this is a variable declaration without memory allocation.
- Without the explicit extern keyword, a global variable is **initialized automatically with default value**.
- Default initial value is zero.

# extern

```
extern int i=10;
int main(){
        return 0;
}
```

vs.

```
int i=10;
int main(){
        return 0;
}
```

vs.

```
extern int i;
int main(){
        return 0;
}
```

# scope

- **Scope** refers to the visibility of variables (through their names) from one part of a program to another part of that program.
- **Local scope** means that variables are accessible only in the current block of code (and its nested blocks).
- **Global scope** means that variables can be seen large chunks of code.
- There may have conflicts...

# Static arrays

# Static arrays

It is a set of **homogeneous data** (with the same type), stored in a **contiguous space in memory**. They do not need to be freed/allocated.

Declaration:

```
type arrayName[arraySize] ;
```

# Static arrays

The access to the elements is with the **bracket operator**, with index values between 0 y $N-1$, where $N$ is the array size (different from Matlab).

Don't forget that:

```
int myArr[10];
myArr [10] = -1;
```

compiles without problem. The error may occur at execution.

# Static arrays

For basic one-dimensional arrays:

```
int myArr[10];
myArr[3]=-1;
```

corresponds to write 4 bytes in memory, at position:

```
&myArr[0]+3*sizeof(int)
```

```
#define N 10
int main() {
        int tab[N] ;
        ...
        for (int i=0;i<N;i++)
                printf ("tab[%d] = %d\n",i,tab[i]) ;
        return 0;
}
```

Use defines to avoid referring to numerical values in the code.

# Static arrays

An array is similar to a pointer to the first data of the array; it is **constant** (not lvalue). In particular, copying an array into another implies copying each element:

```
#define N 10
int main()
{
        int tab1[N], tab2[N] ;
        ...
        for (int i=0;i<N;i++)
                tab1[i]=tab2[i] ;
        return 0;
}
```

# Static arrays

Equivalence:

```
A[n-1]    = 10;
*(A+n-1)= 10;
```

# Static array initialization

type arrayName[arraySize] = {a1, a2, ... aN} ;

where $a1$, $a2$, ..., $aN$ are numerical values.

- It is possible to put $n < N$ values, and in that case only the first $n$ elements will be initialized.
- Un-initialized elements are not necessarily 0! This only happens if (1) the array is a global variable or (2) if the array is declared as static.
- You can initialize without specifying the size:

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

# Static array initialization

For strings, caution to the initialization

```
char toto[5] = "hello" ;
```

Won't work: You need one more space!

# Static array initialization

A string, to be considered as such, should end with \0.

You can initialize an array without specifying the size (it will be deduced by the number of elements passed in the initialization).

```
char toto[]="hello";
 printf ("Array size %d\n",sizeof(toto)) ;
```

# Multidimensional arrays

Same as in 1D:

type arrayName[dim1 ]...[ dimK];

and the access is through

arrayName[i1 ]...[ iK ];

# Multidimensional arrays

```
int main() {
        int i ;
        int A[2] ;
        int B[2][2] ;
        printf("%lx %lx \n",(long)&A[0],(long)&A[1]);
        for (i=0;i<4;i++)
                printf("%lx ",(long)&B[i/2][i%2]) ;
}
```

```
jbhayet@Barkoxe:~$ ./a.out
7fff5fe9ce98 7fff5fe9ce9c
7fff5fe9cea0
7fff5fe9cea4
7fff5fe9cea8
7fff5fe9ceac
```

# Variable length arrays

Recent standards allow to use variable-lengths arrays:

```
int test (int len, char data[len][len] ) {
        return 1;
}
```

More details in:

http://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html

# Structures

# Structures

Combines heteregeneous data:

```
struct person {
        unsigned int age;
        unsigned int height;
        char name[256];
};
struct person tonio, elba;
```

The different elements can be accessed through the . operator:

```
unsigned int age = elba.age ;
```

# Structures

No arithmetic/comparison operators.

In particular, to check equality, all the data should be checked (with a tailored comparison function)

```
if (tonio==elba) {
```

makes no sense. But in C++ it is possible to give it sense.

# Structures

The size dedicated to each element can be specified:

```
struct person {
        unsigned int age: 7;
        unsigned int height: 25;
        char name[256];
};
struct person tonio, elba;
```

Size?

# Union

A **union** contains memory space not for all the elements but only for the largest one: It allows to "see" the memory through different types.

```
union person {
        unsigned int age;
        unsigned int height;
        char name[256] ;
} tonio, elba ;
tonio.age    = 24;
tonio.height= 175;
 printf ("Tonio is %d years old \n",tonio.age);
```

What is the output? What is the union size? How large would be the corresponding struct?

# Enumerations

A type that enumerates different values for an object and allows to have some semantics.

```
enum sportEnum { Rugby, Football, Tennis, Golf} ;
```

In C, there is no checking of the values:

```
enum sportEnum {
        Rugby ,
        Football  ,
        Tennis  ,
        Golf
};
enum sportEnum s1=1, s2=4, s3=Football;

 printf ("s1 %d s2 %d s3 %d \n" , s1, s2 ,  dep3 )
```

# Enumerations

**typedef** allows to define new type names, which is useful to avoid repeating the full struct name at each variable declaration.

```
typedef struct personStruct {
        unsigned int age ;
        unsigned int height;
        char name[256] ;
} person;
person nery, adolfo, cuahutemoc;
```

# Anonymity

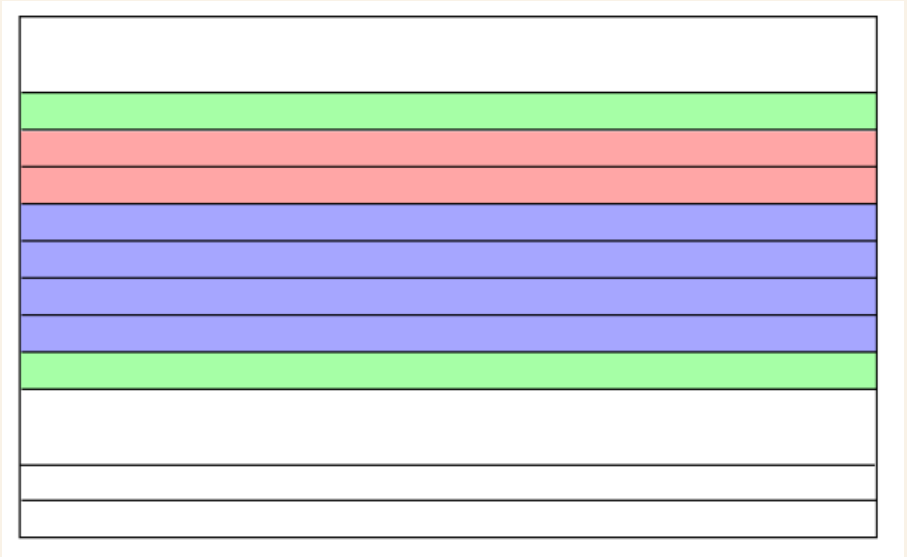
Since C11,

```
typedef struct personStruct {
        union {
                unsigned int age ;
                unsigned int height;
        }
        char name[256] ;
} person;
person nery;
nery.age = 30;
```

# Data alignment

# Structure alignment

```
struct toto {
        char c ;
        int i ;
        short s ;
        char o ;
};
```



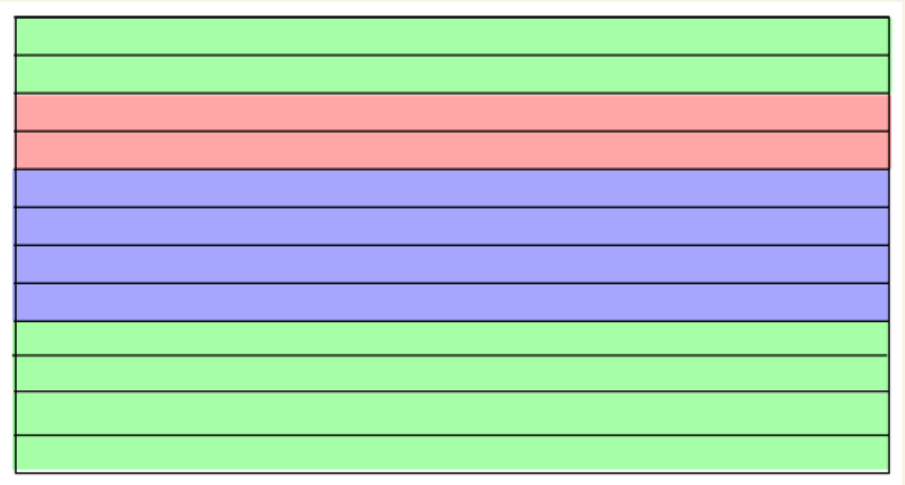
# Structure alignment

```
struct toto {
        char c ;
        int i ;
        short s ;
        char o ;
};
int main() {
        printf("%d\n",(int)sizeof(struct toto)) ;
}
```

```
jbhayet@Barkoxe:~$ ./a.out
12
```

# Structure alignment

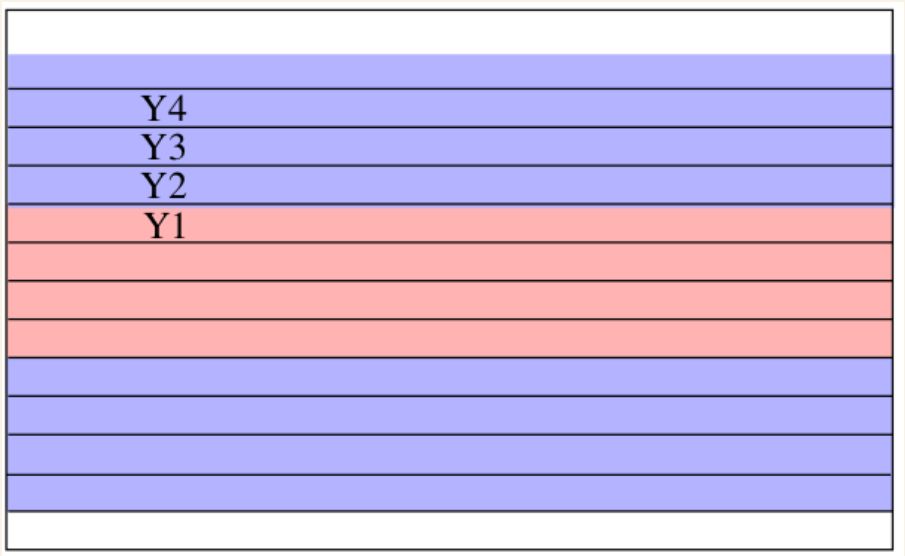
This is not this way!
Instead:



The compiler adds bytes that will not been used (named padding).

# Structure alignment

How the computer deals with the memory:



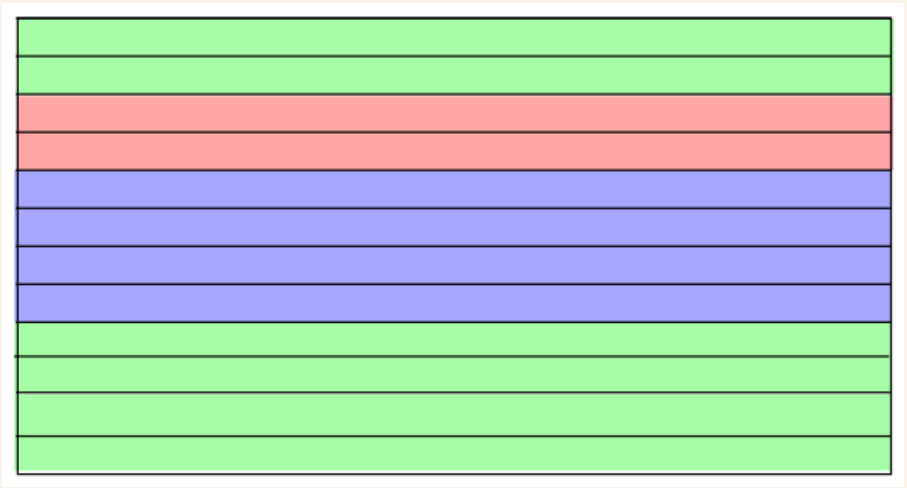

The machine has access to memory through cabling named bus, which has some width (multiple bytes). It can be 2, 4 (32-bits machines), 8, 16...

# Structure alignment

- Some machines in the past simply could not process unaligned data.
- In any case, processing capacities were lost.
- Also applies to local variables in a function.
- Two levels: intra-structure; inter-structure.

# Structure alignment

- Done by the compiler.
- Makes access easier.



# Structure alignment

- A single octet is aligned on any address.
- A short is aligned on even addresses.
- int, float are aligned on multiple of 4.
- structure between 1 and 4 octets $\rightarrow$ 4 octets.
- long, double aligned on 8 octets.