Programming and Algorithms

C: Pointers and dynamic memory allocation.

Jean-Bernard Hayet

CIMAT, A.C.

Pointers

Pointers

A **pointer** is a variable that refers to some data in memory. It includes:

1. An address in memory where the data is stored.
2. The type of the referred data: How the bytes in that place in memory should be read.

It is used as:

```
type *var;
```

where "type" can be any type, including a pointer type (pointer to pointer, or multiple pointer).

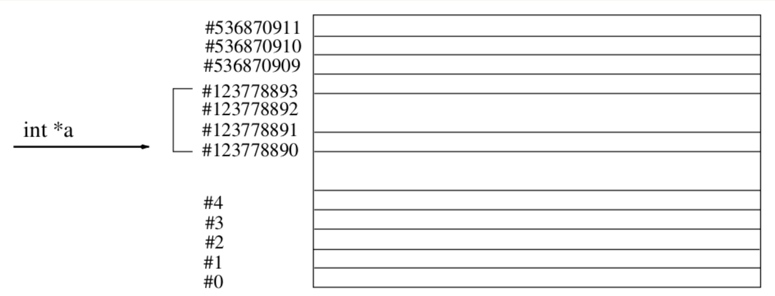
Pointers

When dealing with an initialized pointer, remember that you are handling two variables:

- The **pointer variable** (its value represents a memory address).
- The **pointed variable** (it value depends on its type).

Both variables have their memory address and their own type.

Pointers



- 123778890 is an **address**, held in the pointer variable (32/64).
- It would be similar if you were manipulating a char *.
- Arithmetic operations with the pointer will be done along packets of 4.

Pointers: Access to values

The access to the pointed value is through the **indirection operator ***:

```c
int a       = 40;
int *a_ptr = &a ;
*a_ptr      = a * a ;
```

Has a **priority level of 2**:

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

Pointers: Access to values

An equivalent form is with the **bracket operator**:

```
int  a      = 40;
int* a_ptr = &a ;
a_ptr[0]    = a * a ;
```

Pointers: Access to values

In the case of structures, there is an operator **equivalent to * for the structure and . for the access to an element**:

```
person nery ;
person *nery_Ptr = &n e r y ;
nery_Ptr->age = 25 ;
(*nery_Ptr).age = 25;
```

the last 2 lines are equivalent.

Pointers: Arithmetic

A pointer being a positive integer, some **arithmetic operators** can be applied:

- **Sum an integer**: The result is a pointer with the same type, but with its address translated in memory, as specified by the integer.
- **Subtract an integer**: same meaning.
- **Subtract two pointers**: returns an integer measuring the space in memory between the two pointers.
- **Comparison between pointers**: Compare the address, but only for pointers of the same type.

Pointers: Arithmetic

Arithmetic operations on pointers should be understood **not in bytes but in the unit corresponding to the pointed type**!

Example: if you manipulate a pointer to an int, then **the unit will be 4 bytes**.

Pointers: Arithmetic

```
int i = 15;
int *p1,*p2 ;
p1 = &i ;
p2 = p1 + 2 ;
printf("p1 = %ld \t p2 = %ld \n",p1,p2) ;
```

If p1 = 12358952, p2 =?

Pointers: Arithmetic

```
double i = 15.0 ;
double *p1,*p2 ;
p1 = &i ;
p2 = p1 + 2 ;
printf("p1 = %ld \t p2 = %ld \n",p1,p2) ;
```

If p1 = 12358952, p2 =?

Pointers: Arithmetic

Cycle over the data of a static array:

```
#define N 5
int tab[N] = { 1, 2, 6, 0, 7};
int main ( ) {
        int *p;
        for (p = &tab[0]; p <= &tab[N-1];p++)
                printf ("%d \n",*p) ;
        for (p = &tab[N-1]; p >= &tab[0];p--)
                printf ("%d \n",*p) ;
        return 0;
}
```

Pointers: Arithmetic

The bracket operator [] is equivalent to a **combination between the indirection operator and the pointer arithmetic operator +**:

```
int tab[10] ;
int *p=&tab[0] ;
p[2]    = 5;
*(p+2) = 5;
```

The last 2 lines are **equivalent**.

Pointers and functions

When passing large data (struct...) to **functions**, we may have two kind of problems:

- The arguments of a function are **copied**, and this may take some time if the argument is large.
- Inside the functions, we work **only with copies of the arguments**. We may want to modify them.

Pointers and functions

Compare:

```
typedef struct dataStruct{
        double data[100];
        int width;
        int height;
} data;
void func1(data mydata) {
        int w = mydata.width;
        ...
} ;
void func2(data *mydata) {
        int w = mydata->width ;
        ...
} ;
```

Pointers and functions

```
void isYoung(person *cons , int answer) {
        if (cons->age<30)
                answer = 1;
        else
                answer = 0;
};
int main() {
        person nery ; nery.age = 25 ;
        int young = 0;
        isYoung(&nery,young) ;
        if (young)
                printf ("nery is young") ;
        else
                printf ("nery is quite old") ;
        return 0;
}
```

Pointers and functions

For getting the return value from a function:

- we cannot pass the variable **by value** because the function will work with a copy;
- we can use the **return keyword** but sometimes the return value is too large (and is being copied);
- we can pass it as a **pointer**.

Pointers: Modifiers

A pointer being a "normal" variable after all, you can use **modifiers** as with any variable.
The notation can be tricky:

```
int a        = 3;
const int b = 2 ;
int      *c = &a ;
const int *d= &b ; // Pointer to a const int
int *const e = &a ; // Const pointer to an int
```

Pointers: Modifiers

```
const int *x = ...;
```

Cannot change the pointed variable.

```
int *const x = ...;
```

Cannot change the pointer.

```
const int *const = ...;
```

Cannot change either.

Multiple pointers

As a pointer is a variable with a type and a representation in memory, we can define a **pointer to a pointer**:

```
int a = 1;
int *a_ptr = &a;
int **a_ptr_ptr = &a_ptr;
printf("The address of a is %ld and its value is %d \n",*a_ptr_ptr,**a_ptr_ptr);
```

Can be read as (int*)* for a better understanding.

Multiple pointers

In the same way, you can define **multiple pointers**, where the pointer of degree n is in reality a simple pointer to another multiple pointer of degree $n-1$.

Segmentation fault

A pointer which is not initialized, badly initialized, or freed can be a **source of problem**s: it gives access to memory zone that is forbidden to be used by you program:

```
int main() {
        // Do not do that at home!
        int *myPointer = 0xFFFFF;
        *myPointer = 123;
}
```

Gives:

```
Segmentation fault
```

Segmentation fault

The **Segmentation fault** error is typical of a bad use of pointers and invalid access to memory
When it occurs:

- the first reaction should be to revise all the pointers;
- it can occur in a delayed way (not exactly where the code is wrong);
- use gdb or valgrind if you do not manage to spot the problem.

Dynamic memory allocation

Dynamic arrays

Sometimes, we do not know what will be the size of an array that we need. Options:

- Use a **maximum size and static array**.
- Use a **variable-length static array**. In this case and in the previous: the variables are automatic and will be liberated when leaving their scope.
- Use **dynamic allocation**: Using the malloc function, we ask for some amount of memory that is reserved on the heap; we are responsible of this memory (free it).

Dynamic arrays

```
#define N 100 ;
int* applyFunction (int *arregloIn ) {
        int arregloOut[N] ;
        for (int i=0;i<N;i++)
                arregloOut[i] = ...;
        return &arregloOut[0] ;
}
int main() {
        int arreglo[N];
        int *result = applyFunction(&arreglo[0]);
        ...
}
```

Problem?

Dynamic arrays

A dynamic array **survives out of the scope where it is defined**, contrary to the static array.

The **malloc** function is used to ask for memory, and it takes as an argument the **number of required bytes**:

```
int *dynamicArray = (int *)malloc( arraySize * sizeof ( int ));
```

The values are **not initialized**!

Cast the result.

Dynamic arrays

As an alternative, the **calloc** function takes as arguments the number of elements and the size of each.

```
int *dynamicArray = (int *) calloc ( arraySize , sizeof ( int ));
```

calloc **initializes values at 0**.

Dynamic arrays

Caution: malloc or calloc **ask** for memory but **they may not get it**! It is important to **check** the returned pointer to ensure that memory has been allocated.

In case of non-allocated memory, those functions return **NULL**:

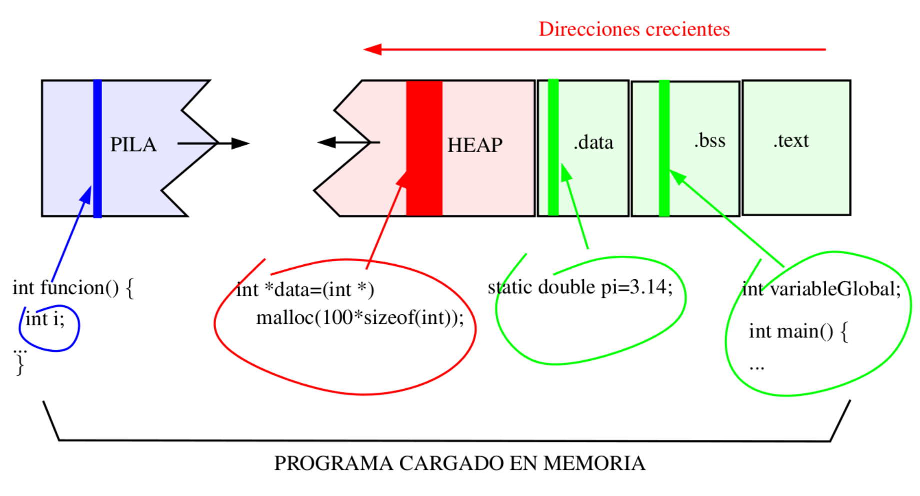
```
int *dynamicArray = (int *) calloc ( arraySize , sizeof ( int ));
if (dynamicArray == NULL)
        ...
```

Dynamic arrays

Allocated space is liberated with the free function:

```
int *dynamicArray = (int *) calloc ( arraySize , sizeof ( int ));
...
free (dynamicArray);
```

Dynamic arrays



Dynamic arrays

Remember that the memory zone that corresponds to dynamic allocation is the **heap**. It is **persistent** memory (surviving function calls).

Caution: the **unique reference to this memory is the pointer**. Don't lose the pointer (to use it in free).

Multiple arrays

The principle can be used to define **multiple arrays**: For example, in the case of a matrix, you can allocate memory for all the rows, then for each row allocate the corresponding space (columns).

Dynamic arrays

```
int **doubleDynamicArray = (int **)calloc( arraySizeX , sizeof ( int  *));
if (doubleDynamicArray==NULL)
...
for ( int i =0; i<arraySizeX ; i++) {
        doubleDynamicArray[i] = (int *) calloc ( arraySizeY , sizeof ( int ))
         if (doubleDynamicArray[i]==NULL)
         ...
}
...
for ( int i=0;i<arraySizeX;i++)
         free (doubleDynamicArray[i]) ;
free (doubleDynamicArray);
```