Algorithmic and Data Structure 2

Chapter 3

“ Pointers”

Outline

1. Introduction

Any variable manipulated in a program is stored somewhere in central memory. This memory consists of bytes, which are uniquely identified by a number called *an address*. To find a variable, it is enough to know the address of the byte where it is stored (or, if it is a variable covers several contiguous bytes, the address of the first of these bytes). For obvious reasons for readability, we often designate variables by identifiers, and not by their address. It is the compiler, which then makes the link between the identifier of the variable with its address in memory. However, it is sometimes very practical to directly manipulate a variable by its address.

Example:

int i, j; i = 3;

j = i;

If the compiler placed the variable i at address 4831836000 in memory, and the variable j at address 4831836004, we have

Object	Address	value
i	4831836000	3
j	4831836004	3

Two different variables have different addresses. The assignment i = j; only operates on the values of the variables.

2. Definition

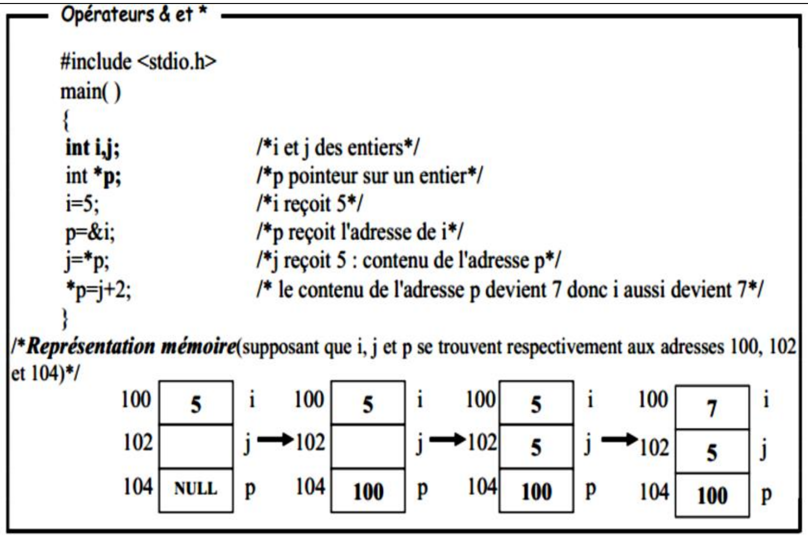
A pointer is an object whose value is equal to the address of another object. Even if the value of a pointer is always an integer (possibly a long integer), the type of a pointer depends on the type of the object it points to. This distinction is essential for interpreting the value of a pointer. Indeed, for a pointer to an object of type char, the value gives the address of the byte where this object is stored. For a pointer to an object of integer or real type, the value gives the address of the first byte where the object is stored? where an integer is stored on 2 bytes and a real is stored on 4 bytes. (in C language the number of bytes of a given type is calculated by the function: **sizeof (type)**).

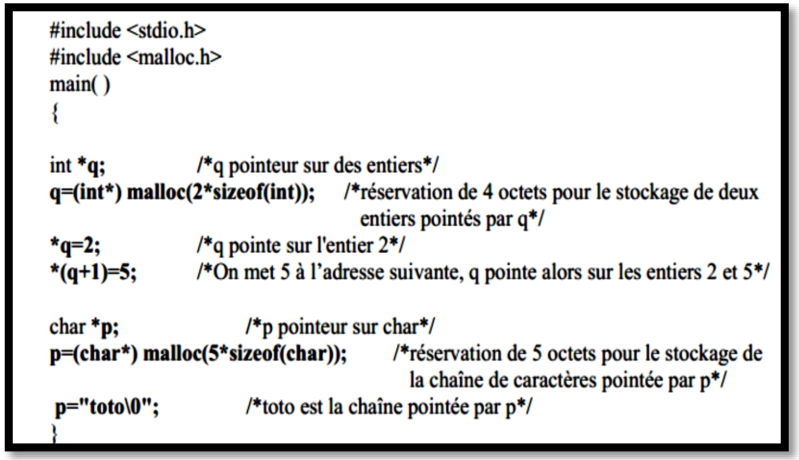
	Algorithm	**C language**

Declaration	**pointer_name: * Type;** (Type is the type of the pointed object.)
	Example :
	p1:*character; //p1 is a pointer pointing to a character object.
	p2:*integer; // p2 is a pointer pointing to an integer object.
	p3: real; //p3 is a pointer pointing to a real object.
		Type * pointer_name ; (Type is the type of the pointed object)**.**
		Example :
		char *p1;
		int *p2;
		float *p3;

After declaration and before use, a pointer must be initialized (otherwise it can point to any region of memory!):		

Initialization	**1. Assigning a null value**
	p ← NULL ;
	2. **Assigning the address of another variable**
	p ← &i ;
	Algorithm example;
	Variables i: integer; p:*integer;
	Begin
	i←3;
	p ← &i ;
		1. Assigning a null value
		p = NULL ;
		2. **Assigning the address of another variable**
		p = &i ;
		Example

	write (“the contents of the memory box pointed to by p”,*p);
	p ← 5 ;
	write (“i=”,i, “*p=”,*p);
	END
	3. The dynamic allocation of a new memory space. Dynamic allocation is the operation of reserving memory space of a defined size. **allocate(p);**
	allocate (p): reserves a memory location of the size corresponding to the type, and puts in the variable p the address of the reserved memory area.
		
		Opérateurs & et *
		#include <stdio.h>
		main( )
		{
		int i,j; /*i et j des entiers*/
		int ***p;** /*p pointeur sur un entier*/
		i=5; /*i reçoit 5*/
		p=&i; /*p reçoit l'adresse de i*/
		j=*p; /*j reçoit 5 : contenu de l'adresse p*/
		p=j+2; / le contenu de l'adresse p devient 7 donc i aussi devient 7*/
		}
		/****Représentation mémoire***(supposant que i, j et p se trouvent respectivement aux adresses 100, 102 et 104)*/
		100 **5** i → 100 **5** i → 100 **5** i → 100 **7** i
		102 j → 102 j → 102 **5** j → 102 **5** j
		104 **NULL** p → 104 **100** p → 104 **100** p → 104 **100** p
		3. Dynamic allocation in C is done through (among other things) the function **malloc(numberBytes)** of the standard library **stdlib.h**
		type* p = (type*) malloc (sizeof (type));
		Example

		
		#include <stdio.h>
		#include <malloc.h>
		main( )
		{
		
		int ***q;** /*q pointeur sur des entiers*/
		**q=(int*) malloc(2*sizeof(int));** /*réservation de 4 octets pour le stockage de deux
		                                                entiers pointés par q*/
		***q=2;** /*q pointe sur l'entier 2*/
		***(q+1)=5;** /*On met 5 à l'adresse suivante, q pointe alors sur les entiers 2 et 5*/
		
		char ***p;** /*p pointeur sur char*/
		**p=(char*) malloc(5*sizeof(char));** /*réservation de 5 octets pour le stockage de
		                                                la chaîne de caractères pointée par p*/
		 p="toto\0"; /*toto est la chaîne pointée par p*/
		}

Memory Liberation	This is the operation, which consists of freeing the allocated memory space. **Free (p);** **Free (p)**: frees the memory area whose address is in p (and makes it available for the allocation of other variables), it leaves the pointer value as is (does not erase the address which is in the pointer variable).	In C, memory is freed using the free function of the standard library stdlib.h: **void free(p);** Any memory space dynamically allocated via **malloc** (or equivalent) must be deallocated using free.
Pointer arithmetic	The value of a pointer being an integer, we can apply a certain number of classic arithmetic operators to it. The only valid arithmetic operations on pointers are: • The addition of an integer to a pointer. The result is a pointer of the same type as the starting pointer; • The subtraction of an integer from a pointer. The result is a pointer of the same type as the starting pointer; • The difference between two pointers both pointing to objects of the same type. The result is an integer.	

Example

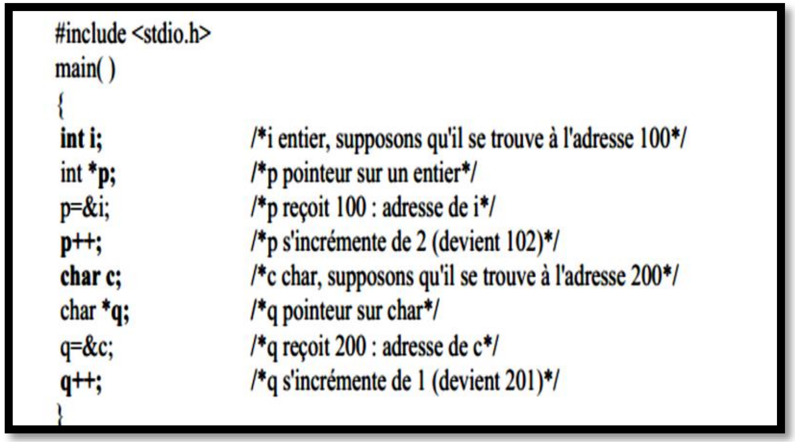
If k: integer; p:*type;

The expression "p+k" designates a pointer to an object of type integer whose value is equal to the value of p incremented by k*sizeof(type).

// sizeof(type) is size(type).

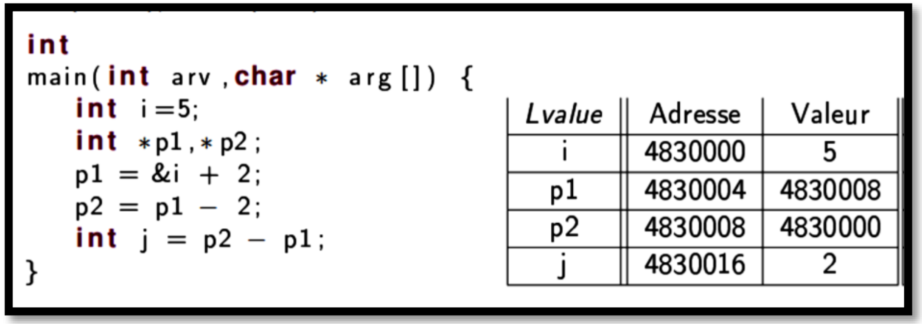
This is the same case for subtracting an integer from a pointer and for the increment and decrement operators ++ and--.

Example on addition



```
#include <stdio.h>
main( )
{
int i;                /*i entier, supposons qu'il se trouve à l'adresse 100*/
int *p;               /*p pointeur sur un entier*/
p=&i;                 /*p reçoit 100 : adresse de i*/
p++;                  /*p s'incrémente de 2 (devient 102)*/
char c;               /*c char, supposons qu'il se trouve à l'adresse 200*/
char *q;              /*q pointeur sur char*/
q=&c;                 /*q reçoit 200 : adresse de c*/
q++;                  /*q s'incrémente de 1 (devient 201)*/
}
```

Example on the difference of two pointers



```
int
main(int arv ,char * arg[]) {
    int i=5;
    int *p1,*p2;
    p1 = &i + 2;
    p2 = p1 - 2;
    int j = p2 - p1;
}
```

Lvalue	Adresse	Valeur
i	4830000	5
p1	4830004	4830008
p2	4830008	4830000
j	4830016	2

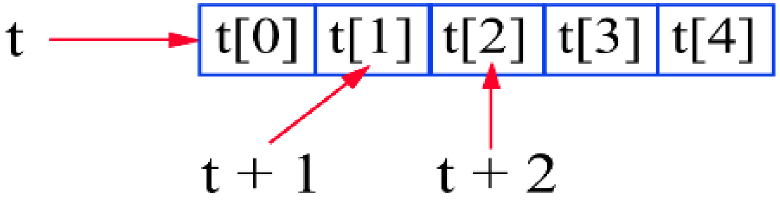
Pointers and arrays

Any one-dimensional array can be replaced by a pointer to its first element.

Equivalences:

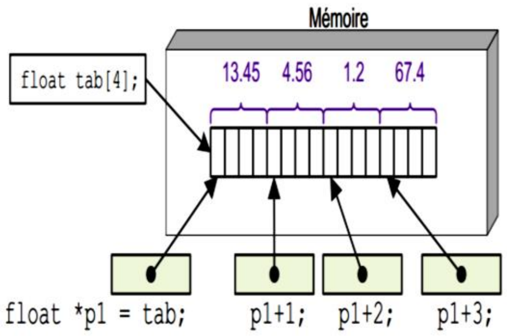
- t equivalent to t +0 equivalent to &t[0];
- t+i equivalent to &t[i];

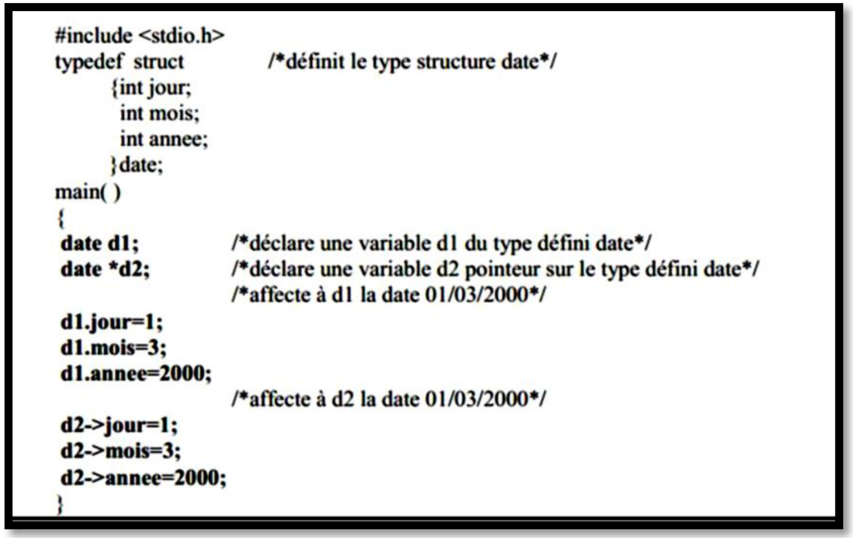
- *t equivalent to t[0];
- *(t+i) equivalent to t[i]



The following loops are equivalent

for i ← 1 to n do
Read (t[i]) ;
i← i +1 ;
endfor
	for i ← 1 to n do
	Read (*(t +i)) ;
	i← i +1 ;
	endfor
		for p1 ←t to t+n do
		Read (*p1) ;
		p1← p1 +1 ;
		endfor



Pointers and records	When using a pointer to a record. Its fields will be referenced by the name of the pointer followed by an arrow then the name of the field in question.	**Example** 

```
#include <stdio.h>
typedef  struct            /*définit le type structure date*/
        {int jour;
         int mois;
         int annee;
        }date;
main( )
{
 date d1;              /*déclare une variable d1 du type défini date*/
 date *d2;             /*déclare une variable d2 pointeur sur le type défini date*/
                       /*affecte à d1 la date 01/03/2000*/
 d1.jour=1;
 d1.mois=3;
 d1.annee=2000;
                       /*affecte à d2 la date 01/03/2000*/
 d2->jour=1;
 d2->mois=3;
 d2->annee=2000;
}
```