Chapter 3: Pointers and Linked Lists

1. Introduction

We saw in the first semester that the program is a set of data and a set of instructions where that data is stored in memory as variables.

A **variable** is a location in memory that has a storage address, name, type, and value.

- **Address**: Each variable stored in memory has an address that indicates its location. It is a natural number that identifies the first byte in which the variable is located. Usually, it is written in the hexadecimal system 16, such as: 0x5A63
- **Name:** An identifier used by the programmer to refer to the stored value and the name of the variable is manipulated instead of the address. E.g. weight
- **Type:** Everything in a computer is made up of 0s and 1s. The type determines how to translate them, as well as the size that should be reserved in memory, i.e. the number of bits and operations allowed. Example: int (32-bit)
- **Value:** This is the content of the bits that make up its value, and it's usually the thing that changes during program execution, such as: 15

When the program executes and encounters a variable declaration statement, such as int age ;), the program instructs the operating system (Windows) to reserve a memory space of size x (depending on the type). And after reservation, the system returns the memory address that can be used as a variable.
To get the value of the variable, you just have to write its name, but to get its address, i.e. its location in memory**, in algorithm we put the @ symbol before the variable name** , and in C we put the & symbol before the variable name .

Example:
```
write("value of age=", age," its address=",@age);
printf("value of age = %d its address = %p",age,&age);
```

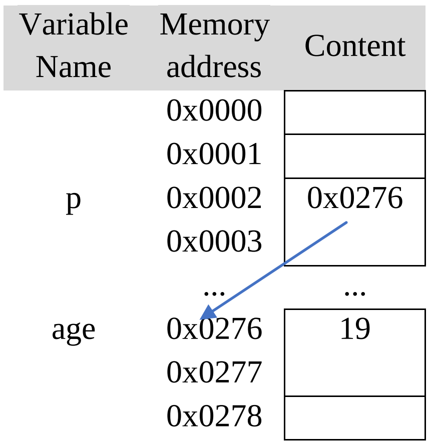
%p is a format for treating the &age value as an address in memory, i.e. a number written in hexadecimal 16. We can use %d to see it in decimal. Here, age is the value of the variable and &age is its address in memory where it can change each time we run the program.

2. Pointers

A **pointer** is a variable whose value points to an address in the computer's memory. This address is either a variable or a program. It's used to pass parameters by address, dynamically reserve memory, or define recursive types (lists, stacks, and queues), and it has other uses.

The Creation

Example:
Memory can be thought of as an array numbered from 0 to memory capacity -1
In the following example, two variables have been reserved, the first is the integer age located at address 0x0276 and contains the value 19 here 0x means that the number is written in the hexadecimal system 16 (0x0276 = 630 in the decimal system). The second variable is p and its value is 0x0276, which represents the location of the age. So we say that p points to age.



To create a pointer variable, in the algorithm we add the symbol ^ in front of the variable type.

Where it takes the following form:

```
var p1, p2 :^type
```

To create a pointer variable in C, we add * before the variable name

```
Type
```

Here ^ or * indicates that the variable is of the pointer type, i.e. a memory address, while type is the type of the contents of that location.
Example : We declare six variables x and y of integer type, p1 and p2 of type pointer to integer, z of type real, and pz of type pointer to real.

`int x,*p1,y,*p2;`
`float z,*pz;`
	`Var x, y: integer  p1, p2: ^ integer`
	`z : real pz : ^real`

When declaring a variable, it has an undefined value, so it is recommended that it be set to **NULL** in uppercase, which means that the pointer is nowhere (defined inside stdio.h, which represents the number 0)

```
p1= NULL;
```

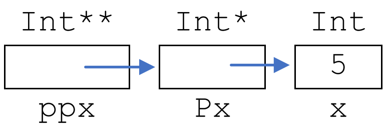
The variable p1 can take the address of variable x or the value of variable p2, but it cannot take the address of variable z, the address of p2, or the value of pz.

Valid Transactions	Invalid transactions	The Explanation
`p1=&x;`	`p1=x;`	p1 is a pointer and x is an integer
`p2=p1;`	`p1=&z;`	p1 is an integer pointer and &z is a real address
`pz=&z;`	`pz=p1;`	pz is a pointer to a real and p1 is a pointer to an integer

	`p2=&p1 ;`	P2 is a pointer to an integer, but **&p1** is the address of a pointer to an integer.
	`p1=&(0x0276);`	Must be a variable, not a number.

We need to differentiate between the address stored in the pointer and the address of the pointer itself, because the pointer is a variable that has an address like the rest of the variables, and therefore its address can be assigned to another pointer, but in this case the second type of pointer must be the address of a pointer of the first type.

For example: x is of integer type (int), and px contains the address of x, so its type is (int*) and ppx contains the address of px, so its type is (int**) as shown in the following diagram:



It is declared as follows:

```
int x,*px,**ppx;
x=5;
px=&x;
ppx=&px;
```

typedef can be used to create new types and the above statement becomes something like this:

```
typedef int* pint;
typedef int** ppint;
pint px;
ppint ppx;
```

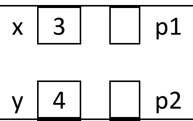
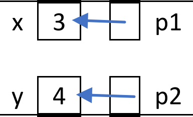
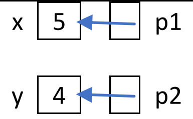
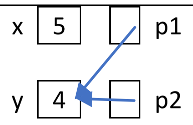
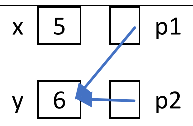
Usage:

It's rare that we treat memory addresses as direct numbers, but we treat them as addresses for existing variables. To get the address of a variable, we use the @ operation in the algorithm or & in the C programming language **before** the variable name, and to retrieve the value of the variable (Dereference) from its address stored in a pointer, we use the symbol ^ **after** the variable name in the algorithm and *** before the** name of the variable in the C programming language.

```
p←@x ⇒ p^ ⇔ x
p=&  x ⇒ *p ⇔ x
```

Example:

C	The Algorithm	memory	The Explanation
`int`	Var x, y:		

`x,*p1,y,*p2;`	integer p1, p2 : ^ integer		
`x=3;` `y=4;`	x←3 y←4		
`p1=&x;` `p2=&y;`	p1←@x p2←@y		Here p1 contains the address of x and p2 contains the address of y
`*p1=5;`	p1^←5		We assign the number 5 to the variable whose address is at p1, and at this point it is the variable x, as if the variable x had a second name, which is *p1 can be replaced by the x=5 statement;
`p1=p2;`	p1 p2←		We assign the value of p2, which represents the address of y, to p1, so that y, *p1, and *p2 become the same variable at that time.
`*p1=6;`	p1^←6		We assign the digit 6 to the variable whose address is in p1 and at this point it is the variable y can be replaced by the y=6 statement; or *p2=6;

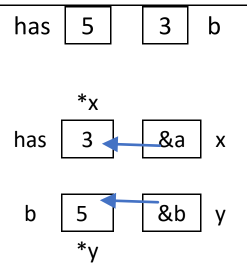
Notes:

- To understand pointers, it is always recommended to draw variables, where the pointer carries an arrow to the variable that carries its address, and we symbolize the pointer that has a value of NULL, i.e. it does not point to any place with ⧅
- A pointer is always of a simple type, while the variable whose address it contains can be of a complex type (array or structure).
- Attempting to retrieve the value of an uninitialized pointer or a NULL value causes the program to terminate.
  - A value (variable address) must be assigned to the pointer before attempting to retrieve the value it points to.
  - Before you retrieve the value that the pointer is pointing to, you must make sure that it is not null.
- It is now possible to understand the passing of parameters by address in subroutines.

Example

C	memory	The Explanation

```
void exchange(int *x, int *y){
int t;
t=*x;
*x=*y;
*y=t;
}
int a=5,b=3;
exchange(&a,&b);
```



Here x and y are two pointers and when calling the function we assign x the address of variable a i.e. x=&a and y the address of variable b i.e. y=&b and inside the function exchange to obtain the variable whose address x carries we use the operation * where *x at this moment represents the variable a and *y represents the variable b

3. Pointer Operations

Suppose that P and Q are pointers and i is an integer. The following table summarizes the operations that can be performed on pointers:

Algorithm operation	Operation C	Type of 2nd Operator	Type of result	Example	Observation
+	+	Int	Pointer	P + i	Returns a pointer to the ith element after P in an array
	++		Pointer	P++	Returns a pointer to the next immediately P element in an array
-	-	Int	Pointer	P – i	Returns a pointer to the ith element before P in an array
	--		Pointer	P--	Returns a pointer to the element immediately preceding P in an array
-	-	Pointer of the same type	Int	P - Q	Returns the number of items between P and Q where P and Q should point to the same array
=	==	Pointer	Boolean	P == Q	This is true if P and Q have the same address, i.e. they point to the same place
≠	!=	Pointer	Boolean	P != Q	This is true if P and Q are different
^	*		Value Type	*P	To retrieve the value whose address it contains

4. Dynamic Memory Management

The method we know so far for reserving variables in memory is called static reservation, where the variable is declared at the beginning of the program, and the compiler reserves the necessary memory automatically, and the variable is not removed until the end of the execution of the program (or subroutine in the case of a local variable). But sometimes we need to reserve an amount of memory, whether it's an array with N elements, for example, and N can only be known at runtime, so we declare a pointer and when N becomes available, we reserve the array.
The developer has a set of functions that allow them to manage memory dynamically, i.e., during runtime.

In algorithm:

There are three procedures for dynamic memory management:

allocate() to reserve an array where it takes as a parameter the name of the pointer (name of the array) and the number of elements .1

```
Allocation(nom_tab,nb_elements)
```
Example:
```
Allocation(T,10)
```

reallocation() to change the size of the array, either by increasing or decreasing, and takes as a parameter the name of the pointer (the name of the array) and the new number of elements (new size), it preserves the values of the previously reserved elements and removes the excess or adds new elements to the array .2

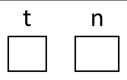
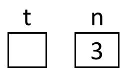
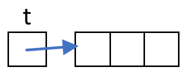
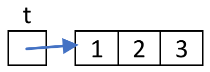
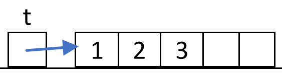
```
reallocate (nom_tab, nouvelle_taille)
```
Example:
```
Reallot(T,15)
```

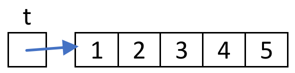
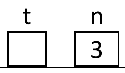
dealallocate() to delete the reserved array with **allocate()** and takes as a parameter the name of the pointer (name of the array) .3

```
deallocate (nom_tab)
```
Example:
```
deallocate
```

After creating an array t by allocate(), its elements can be accessed by square brackets [ ] or by the retrieval operation ^, where we know that the pointer t contains the address of the first element t[0] i.e. @t[0]= t and t^=t[0] and to get The address of the second element t[1] adds 1 to t i.e. @t[1] t+1 ⇔and (t+1)^ ⇔ t[1] so the address of t[i] is t+i. i.e. @t[i]⇔(t+i) and (t+i)^ ⇔t[i].

Example:

algorithm	**memory**	**The Explanation**
var t : ^real n:integer		A pointer t and a variable n representing the number of its elements are declared
beginning write("enter number of elements") read(n)		Let n take 3
Allocation(t ,n)		allocate() reserves an array of three elements and sets its address to t
t[0←] 1 t[1] 2 t[←2] ←3 t^ 1 (t+1)^ ←2 (t+2)^ ←3←		We fill in the table where we can use the square brackets [ ] or use ^ where t[i] ⇔ (t+i)^
reallouer(t,n+2)		Calling reallouer() resizes the array to 5

t[3←] 4 t[4] ←5 (t+3)^ 4 (t+4)^ ←5 ←		We fill in the two added elements
deallocate		We call dealdeal() to remove the array

In C

Memory management in C is a little different than algorithms, and before we can learn more, we need to learn sizeof and type switching.

4.1.The "sizeof" operation

A variable takes up more or less memory space depending on its type. As a variable of type char takes one byte, whereas a variable of type int requires two or four bytes, depending on version C. To find out the size required for a type, we use sizeof(), which takes the name of the variable or the name of the type to return the number of bytes it needs in memory.

```
int sizeof(type);
```

Example:

```
float t[20];
printf("char: %d bytes\n", sizeof(char));
printf("int : %d bytes\n", sizeof(int));
printf("double: %d bytes\n", sizeof(double));
printf("the size of t: %d bytes\n", sizeof(t));
printf("the size of t:%d bytes\n", 20*sizeof(float));
```

that displays on the screen

```
char: 1 byte
int: 4 bytes
Double: 8 bytes
T size: 80 bytes
T size: 80 bytes
```

The size of an array can be found by multiplying the size of a single element by the number of elements.

4.2.Type Change: Typing/Casting

Sometimes we need to convert a specific value from one type to another, and to force the compiler to change the type of a specific value, we use the following formula:

```
(type) expression
```

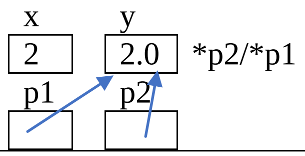
Where the expression is converted to type

Example 1

`int A=8,B=3;`	
`float R=A/B;`	Since operators A and B are integers, the /

	operation performs an integer division R=8/3
`printf("no casting R=%f \n",R);`	poster no casting R=2.000000
`R=(float)A/B;`	We convert the value of A (not the variable A) to a real number, and then we do the dividing process, so that the operation becomes R = 8.0/3
`printf("with casting R=%f \n",R);`	poster with casting R=2.6666666

Example 2

`int x,*p1;`	An integer and an integer pointer
`float y=2,*p2;`	A real number and a pointer to a real number
`x=(int)y;`	It converts the value of y to an integer and puts it in x, so x takes the value 2
`p2=&y;`	p2 takes the address of y
`p1=(int*)p2;`	Converting the address of a float to the address of an int, but the address of the variable remains in both variables, which is the address of y 
`printf("x=%d \n",x);`	Displays x=2
`printf("*p2=%f \n",*p2);`	Displays *p2=2.000000 the same as y
`printf("*p1=%d \n",*p1);`	Poster *p1=1073741824 Because translating the bits of a real number into an integer does not give the same number

4.3.Memory Management in C

Dynamic memory management in C is done using four functions defined in the stdlib library:

- malloc()) ‹**memory allocation** This means to reserve memory) It instructs the operating system to reserve the required amount of memory.

```
void * malloc(int size);
```

It takes as a parameter the required memory size (the number of bytes) and returns a pointer to the memory that has been reserved, or returns NULL if the process fails because the required size is not available.

Example:

```
float *t;

t=(float *)malloc(10*sizeof(float));
```

`t=`	`(float *)`	`malloc(`	`10*`	`sizeof(`	`float`	`));`
Table Name	Convert to Pointer Type	To reserve the table	Number of items	The size of each element	Type of each element	

- free(), to return memory previously reserved by the operating system's malloc so that it can be used by other programs.

```
void free( void * pointer );
```

Takes a previously reserved pointer as a parameter. It is recommended that you set the pointer to NULL after calling free to ensure that the pointer is nowhere to be found and to avoid any errors.

Example:

```
free(t);
```

- realloc(), to change the size of the reserved memory, either by increasing or decreasing.

```
void * realloc(void * pointer, int nouvelle_taille);
```

Where the function calls malloc to reserve a new place of the size of the nouvelle_taille, then copies all the values from the "pointer" array to the new location (or deletes the extra elements if the nouvelle_taille is smaller than the old size), then deletes the old reserved array by calling free, and if the operation succeeds, it returns a pointer to the new location otherwise returns NULL.

Example:

```
t=(float*)realloc(t, 20*sizeof(float));
```

- calloc(), like malloc, except that it puts zeros in the reserved memory.

```
void * calloc(int nb_element, int taille_element);
```

It takes nb_element, which represents the number of items in the table, and taille_element, which represents the size of a cell, and returns a pointer to the placeholder.

Example:

```
t=(float*)calloc(10,sizeof(float));
```

Observation:

- In the function lesson, we saw that void means that the function returns nothing, but void* means that the function returns a pointer of type undefined.
- The void* type must be converted to the pointer type that will contain the address by placing the pointer type in parentheses before the malloc, calloc, and realloc function names, but this conversion is not necessary in C++.
- To use these functions, you must call the stdlib or alloc library using the following statement:

```
#include <stdlib.h>
#include <alloc.h>
```

The `sizeof` operation is not a function, so parentheses can be omitted.

When we reserve memory, we follow these steps:

1. We reserve memory with malloc.
2. We make sure that the booking process has completed successfully by using **`if`** `(pointer! =` **`NULL)`**
3. When we are done using the placeholder, we return the memory to the system via free

Example

C	The Explanation
`#include <stdio.h>` `#include <stdlib.h>`	Inclusion of the STDLIB library
`int main(void) {` `char *str;`	Declaring a char pointer
`str = (char *) malloc(4*sizeof(char));`	Book a table for 4 characters
`str[0]='A'; str[1]='S'; str[2]='D';` `str[3]='\0';`	We populate the array with the string "ASD" using [ ] and the symbol '\0' to indicate the end of the string.
`*str='A'; *(str+1)='S'; *(str+2)='D';` `*(str+3)='\0';`	We populate the array with the literal string "ASD" using the retrieval operation * where *(str+i) ⇔ str[i]
`printf("String is %s\n Address is %p\n", str, str);`	To display the string and its address, where we note that & is not used because str is an address
`str = (char *) realloc(str, 5*sizeof(char));`	Changed the capacity of the table from 4 to 5
`str[3]='2'; str[4]='\0';`	We fill in the last two characters so that the string becomes "ASD2"
`*(str+3)='2'; *(str+4)='\0';`	
`printf("String is %s\n New address is %p\n", str, str);`	Displays the string "ASD2" and its new address
`free(str);` `return 0;` `}`	Return Reserved Memory

4.4. Pointers and matrices in C

C-matrices are an array in which each element is an array. We want to create an M[3][4] matrix with 3 rows and 4 columns.

Suppose we have 3 arrays M0, M1, M2

```
float M0[4],M1[4],M2[4] ;
```

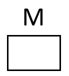
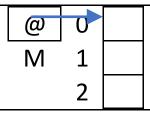
These tables can be created using pointers

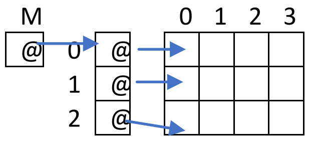
```
float *M0,*M1,*M2;
M0=(float *)malloc(4*sizeof(float));
M1=(float *)malloc(4*sizeof(float));
M2=(float *)malloc(4*sizeof(float));
```

Note that M0, M1 and M2 are all of the same type (float *), so they can be replaced by an array M of type (float *).

```
float * M[3];
for(int i=0; i<3; i++)
  M[i]=(float *)malloc(4*sizeof(float));
```

Now, pointers can be used to create table M

C	memory	The Explanation
`float **M;`		An M pointer is declared to be of type float **
`M=(float**) malloc( 3*sizeof(float*));`		
		Array M is created, which contains 3 elements that represent the number of

		rows, the type of each element is float *
`for(int i=0; i<3; i++)` `M[i]=(float*) malloc(4*sizeof(float));`		
		We create 3 tables, each of which represents a row in the matrix. 4 is the number of columns, and the type of each column is float. *(M+i) can be used instead of M[i]

Any element of the matrix can be accessed by using [] or by using the retrieval operator * where
M[i][j] ⇔ *(M[i]+j)
M[i][j] ⇔ *(*(M+i)+j)

using typedef

```
typedef float ** matrix;
typedef float * table;
matrix M;
M=(matrix)malloc(3* sizeof(table));
for(int i=0; i<3; i++)
  M[i]=(table) malloc(4*sizeof(float));
```

Note : A static array in C is a constant memory address that cannot be changed.

Example:

```
int *p,t[10];
```

`p=t;`	Correct because t is the address of the first element
`t=p;`	Not accepted because t is a constant that cannot be changed.