REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de M'sila
Faculté des Mathématiques et de l'Informatique
Département d'informatique

جامعة المسيلة
كلية الرياضيات والإعلام الآلي
قسم الإعلام الآلي

# Chapter 3: Part 1

## Pointers and linked lists

Algorithmics and data structure 2

**Presented by** : Dr. Benazi Makhlouf
**Academic year** : 2023/2024

# Chapter 03 Content:

1. Introduction
2. Pointers
3. Pointer operations
4. Dynamic memory management
5. …

# 1. Introduction

- A **program** consists of both data and instructions.
- **Data** is stored in memory as variables.
- A **variable** is a designated memory space. It has a name, a type, a value, and a storage address.
- The **memory address** is a natural number indicating the location of a variable. It's commonly expressed in hexadecimal (e.g., 0x5A63).
- When a **variable is declared** in the program, the operating system is requested to allocate a specific amount of memory, depending on the variable's type. and returns the memory address for the variable's use.

# The address

- To obtain the value of a variable: Simply write its name.
- To obtain its address (location in memory): in algorithm you put the symbol '**@**' before the name of the variable, and in C, you use the reference operator "&" before the name of the variable.

**Example** :

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

- In C, `**%p**` is a format specifier used to print the value of a pointer, typically an address. When you use `%p` with `&age`, it will print the address of the variable `age` in hexadecimal.
- If you want to see the address in decimal, you can use `**%d**`.
- Note that the memory address of the `age` variable can vary each time the program is run due to memory allocation by the operating system.

# 2. Pointers

A **pointer** is a variable whose value is a memory address.

The **memory address** stored in a pointer can point to either another variable or program.

**Uses of Pointers:**

- Passing Parameters by Address:
- Dynamic Memory Allocation:
- Recursive Type Definitions: such as linked lists, stacks, queues,….
- …

**Example** :

- **age**: An integer variable its value 19.
- **p**: A pointer variable.
- The variable 'age' is located at the memory address 0x0276 (16) or 630 (10).
- 'p holds the value 0x0276, which is the memory address where the variable age is stored. We say "**p points to age**".

| Nom de variable | adresse mémoire | Contenu |
|---|---|---|
| | 0x0000 | |
| | 0x0001 | |
| p | 0x0002 | 0x0276 |
| | 0x0003 | |
| | … | … |
| age | 0x0276 | 19 |
| | 0x0277 | |
| | 0x0278 | |

# Declaration

- In **algorithmic** pseudocode, the pointer type is indicated with the caret symbol (**^**) placed **in front of the type**. For example: `**var** p1, p2: ^type`.
- In **C**, the pointer type is indicated with the asterisk (*) placed **before the variable name.** For example: `type *p1, *p2;`.
- The symbols (**^** or *) indicate that the variable is of pointer type, representing a memory address, while the **type** specifies the content type at that memory location.
- It is a good practice to initialize pointers to **NULL** when they are declared. In C, this is often done to indicate that the pointer is not currently pointing to a valid memory location.
- In C, `**NULL**` is defined in `<stdio.h>`. The common definition is `#define NULL 0`.

# Example 1

```
Var x: integer          p1, p2: ^integer
    z: real             pz : ^real
```

**In C**

```
    int   x, * p1, * p2;
    float z, * pz ;
```

- p1 can take the address of x or the value of p2 but not the address of z nor the value of pz nor the address of p2

**Valid operations**

```
p1 = &x; p2 = p1; pz = &z;
```

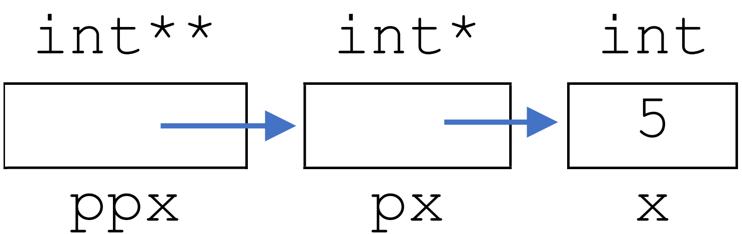
**Invalid operations**

```
  p1 = x; x = p1; p1 = &z;
  pz = p1; P2 = &p1; p1 = &(0x0276);
```

- You need to distinguish between the memory address stored in the pointer, which points to another variable, and the memory address of the pointer itself, representing the location where the pointer variable is stored.

# Example 2

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 :




**Declaration** :

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

Declaration using **typedef** :

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

# Use

It is rare that memory addresses are manipulated directly like numbers. But, we manipulate the addresses of existing variables.

**In algorithm**

- We use the @ operator **before** the variable name to retrieve its address.
- We use the **^** operator **after** the variable name to retrieve the value of the variable (Dereference) from its address stored in a pointer.

**In C,**

- We use the **&** operator **before** the variable name to obtain its address.
- We use the ***** operator **before** the variable name to retrieve the value of the variable from its address stored in a pointer.
- `p ← @x` ⇒ `p^` ⇔ `x`
- `p=&x` ⇒ `*p` ⇔ `x`

# Example

```
int x,*p1,y,*p2;
x=3; y=4;

p1=&x ;
p2=&y ;

*p1=5;

p1=p2;

*p1=6;
```

x 3 p1
y 4 p2

x 3 p1
y 4 p2

x 5 p1
y 4 p2

x 5 p1
y 4 p2

x 5 p1
y 6 p2

**Algorithm** exmpl
**Var** x, y: integer
p1, p2: ^ integer
**begin**
x ← 3
y ← 4
p1 ← @x
p2 ← @y
p1^ ← 5
p1 ← p2
p1^ ← 6

# Comments

- To understand pointers, you have to draw the variables. the pointer carries an arrow towards the variable which holds its address.

**NULL** value

- While a pointer is always of a **simple type**, the variable whose address it holds can be of a **composite** type, such as an array or structure.
- Attempting to retrieve the value of an uninitialized or **NULL** pointer can result in program termination. It is essential to assign a valid value (variable address) to the pointer before attempting to access the value it points to.
- Before retrieving the value indicated by the pointer, it is crucial to verify that the pointer is not **NULL**.

# passing parameters by address

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

swap( &a, &b );
```

*x

| a | 3 | | &a | x |
|---|---|---|---|---|
| b | 5 | | &b | y |

*y

# 3. Pointer operations

Suppose **P** and **Q** are two pointers and i is an integer

| Op | Type of the 2nd operant | Type of result | |
|---|---|---|---|
| **+** | int | Pointer | |
| **P+i** | Returns a pointer to the $i^{th}$ element after P in an array | | |
| **++** | | Pointer | |
| **P++** | Returns a pointer to the immediately following element P in an array | | |
| **-** | int | Pointer | |
| **P–i** | Returns a pointer to the $i^{th}$ element before P in an array | | |
| **--** | | Pointer | |
| **P--** | Returns a pointer to the element immediately preceding P in an array | | |

# 3. Pointer operations

Suppose **P** and **Q** are two pointers and i is an integer

| Op | Type of the 2nd operant | Type of result |
| --- | --- | --- |
| - | Pointer of the same type | int |
| P-Q | Returns the number of elements between P and Q where P and Q must point to the same array | |
| == | Pointer | Boolean |
| P == Q | is true if P and Q have the same address, that is, they point to the same location | |
| != | Pointer | Boolean |
| P != Q | It is true if P and Q are different | |
| * | | Value type |
| * P | To retrieve the value whose address it contains | |

# 4. Dynamic memory management

- **Static** reservation involves the compiler automatically allocating memory for declared variables, retained until the end of the program or subprogram execution.

- **Dynamic** reservation occurs during program execution when a specific amount of memory needs allocation. For example, to reserve an array of ‘N’ elements, a pointer is declared, and when N is known, the array is dynamically allocated.

- Developers use a set of functions to manage memory dynamically during program execution, providing flexibility for memory allocation and deallocation.

# Algorithmic management

There are three **procedures**:

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

**Syntax:** `allocate( arr_name, number_elements )`

**Example:** `allocate(t,10)`

2. **reallocate** () allows you to modify the size of the array, whether to increase or decrease it.

**Syntax:** `reallocate( arr_name , new_size )`

**Example:** `reallocate(t,15)`

3. **deallocate** () to remove the reserved array with **allocate ()**

**Syntax:** `deallocate( arr_name )`

**Example:** `deallocate(t)`

# access

in algorithmic pseudocode, after creating an array 't' using the allocate() function, its elements can be accessed either through square brackets [ ] or the fetch operation ^.

The pointer 't' holds the address of the first element, t[0] i.e.:

@t[0] ⇔ t and t^ ⇔ t[0]

@t[1] ⇔ t+1 and (t+1)^ ⇔ t[1]

For the general case of t[i]:

@t[i] ⇔ (t+i) and ( t+i )^ ⇔ t[i]

# Example 1/2

| algorithme | mémoire |
| --- | --- |
| var t : ^réel<br>n :entier | t n |
| **début**<br>ecrire("entrer le nombre des éléments")<br>lire(n) | t n<br>3 |
| allouer(t ,n) | t |

# Example 2/2

| | |
|---|---|
| t[0] ←1 t[1] ←2 t[2] ←3<br>t^ ←1 (t+1)^ ←2 (t+2)^ ←3 | t → [1 \| 2 \| 3] |
| reallouer(t,n+2) | t → [1 \| 2 \| 3 \| \| ] |
| t[3] ←4 t[4] ←5<br>(t+3)^ ←4 (t+4)^ ←5 | t → [1 \| 2 \| 3 \| 4 \| 5] |
| désallouer(t) | t [ ] n [3] |

# The “sizeof” operation

To find out the size of a type or variable in bytes

**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));
```

**which displays on the screen**

```
char : 1 bytes
int  : 4 bytes
double: 8 bytes
the size of t: 80 bytes
the size of t: 80 bytes
```

# Type change: type casting

To force the compiler to change the type of a specific value, we use the following formula:

```
(type) expression
```

**Example 1**

```
int A=8,B=3;
printf("no casting %f \n", A/B);
printf("with    casting    %f    \n",
(float) A/B);
```

x y

2 2.0 *p2/*p1

p1 p2

**Example 2**

```
int x, *p1;
float y=2, *p2;
x=( int )y;
p2= &y ;
p1=( int* )p2;
printf("x=%d \n", x);
printf("*p2=%f        \n",
*p2);
printf("*p1=%d        \n",
*p1);
```

# MemoryManagement in C 1/2

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

1. malloc () ‘( **memory allocation** to reserve memory. Takes the size in bytes and returns a pointer to that address or **NULL** on failure.

**Example**

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

| t= | (float*) | malloc( | 10* | sizeof( | float | )); |
|---|---|---|---|---|---|---|
| Table name | Convert to pointer type | To reserve painting | Number of elements | The size of each element | Type of each element | |

**NB** : Pointer type conversion is not necessary in C++

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

# Memory Management in C 2/2

2. The **free**() function is used to return the memory reserved by **malloc** back to the system. It takes as input a pointer to memory that was previously allocated. After calling **free**, it is recommended to assign **NULL** to the pointer to avoid using a dangling pointer.

**Example**

```
free(t);
```

3. realloc(), to change the size of reserved memory

**Example**

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

4. The **calloc**() function is similar to **malloc**(); however, it initializes the reserved memory with **zeros**. It requires the number of elements in the array and the size of each element and returns a pointer to the allocated memory..

**Example**

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

# Comments

- In the functions chapter, we learned that `void` in a function's return type means the function returns nothing. However, `void*` signifies that the function returns a pointer of undefined type.
- When working with `void*`, it's necessary to convert it to the desired pointer type by placing the pointer type in parentheses before the `malloc`, `calloc`, and `realloc` function names. This conversion is not required in C++.
- To use these functions, include the necessary libraries using :
  - #include <stdlib.h>
  - #include <alloc.h>
- When reserving memory, follow these steps:
  1. Reserve memory with `malloc`.
  2. Ensure the reservation process is successful by checking: `if (pointer != NULL)`
  3. When done using the reserved space, return the memory to the system using: `free(pointer)`

# Example

```
#include < stdio.h >
#include < stdlib.h >
int main(void) {
  char *str;
  str = (char *) malloc(4*sizeof char);
  str[0]='A'; str[1]='D'; str[2]='S'; str[3]='\0';
  // Or
  *str='A'; *(str+1)='D'; *(str+2)='S'; *(str+3)='\0';
  printf("String is %s\n Address is %p\n", str, str);
  str = (char *) realloc (str, 5*sizeof(char));
  str[3]='2'; str[4]='\0';
  // Or
  *(str+3)='2'; *(str+4)='\0';
  printf("String is %s\n New address is %p\n", str, str);
  free(str);
  return 0;
}
```

# Pointers and matrices in C 1/3

- Matrix in C is an array where each element is an array.

**For example** we will create a matrix M[3][4] with 3 rows and 4 elements in each row (4 columns).

- Suppose we have 3 arrays M0, M1, M2

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

- These arrays 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);
```

# Pointers and matrices in C 2/3

Now the pointers can be used to create array M

| C | mémoire |
|---|---|
| `float **M ;` | M |
| `M=(float**) malloc( 3*sizeof(float*));` | |
| | @ 0<br>M 1<br>2 |
| `for(int i=0;i<3;i++)`<br>`  M[i]=(float*) malloc(4*sizeof(float));` | |
| | M 0 1 2 3<br>@ 0 @<br>1 @<br>2 @ |

# Pointers and matrices in C 3/3

**using typedef**

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

Any element of the array can be accessed using **[]** or using the retrieval operator ***** where

```
M[i][j]  ⇔  *(M[i]+j)
M[i][j]  ⇔  *(*( M+i )+j)
```

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

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

```
p=t;  // Correct because t is the address of the first element
t=p;  // Incorrect because t is a constant
```

# Memory management in C++

is done using two **operators** :

There are two syntaxes: one for simple types and one for arrays.

1. **new** : to reserve memory. returns a pointer to this address or **NULL** on failure.

**Example**

```
float *p, *t;
p= new float;
t= new float[n];
```

2. **delete** : to return the memory reserved by **new** to the system.

**Example**

```
delete p;
delete []t;
```

End of part 1 of Chapter 03