Correction of the Tutorial Series N°3

Exercise 1:

Address of the variable **z**:

Variable	Address	Value
x:integer	2686748	
z:integer	2686752	
...	...	...
y:integer	2686760	
...	...	...
p:^integer	2686866	

1) The address is 2686752 because the size of an integer is 4 bytes, so the address of **z** = the address of **x** + 4.

2) The values of the variables used after the execution of each of the following instructions:

	x	z	y	p
`Var x,z,y:integer; p:^integer;`				
`x ← 5;`	5	/	/	/
`z ← 8;`	5	8	/	/
`p ← @x;`	5	8	/	2686748
`y ← p^*p;`	5	8	Error	2686748
`y ← p^*x;`	5	8	25	2686748
`y ← @p;`	5	8	Error	2686748
`y ← (@p^)^+2;`	5	8	7	2686748
`(p+1)^ ← 3;`	5	3	7	2686748
`p ← p+3;`	5	3	7	2686760
`p^ ← (p-2)^*4;`	5	3	12	2686760
`Allocate(p);`	5	3	12	Inconnu

Exercise 2 :

A curve is a set of points in the plane. We define a polygon as a curve closed upon itself.

Therefore, the polygon consists of a set of points in the plane, such that the first point of the polygon coincides with the last point.

1) Using pointers, write a procedure to reserve the necessary space to store the coordinates of **n** points (where **n** is passed as a parameter), input the coordinates of these points, and display whether they form a polygon.
2) Test this procedure in a main algorithm.

Solution:

```
    Algorithm example;
    Type   Point = Record
                   Begin
                   x, y: Real;
                   End;
           p_Point = ^Point;
    Var n:integer;
1)  Procedure testPolygon(n: Integer);
    Var p, pStart, pEnd: p_Point; i: Integer;
    Begin
    pStart ← Nil;
    pEnd ← Nil;
    For i ← 1 to n Do
        Begin
```

```
        Allocate(p);
        Read(p^.x, p^.y);
        If pStart = Nil then
                pStart ← p;
        End;
    pEnd ← p;
    If pStart≠Nil and pEnd≠Nil and pStart^.x=pEnd^.x and pStart^.y=pEnd^.y
      then WriteLn("These points form a polygon")
    Else  WriteLn("These points do not form a polygon");
    End;
```

```
2)  Begin
    Read(n);
    testPolygon(n);
    End.
```

Exercise 3 :

Using pointer formalism:

1) Write a C language function, **`fillArray`**, that reads the number of elements in an array of floats, allocates memory for the entire array accordingly, reads the elements of the array from the keyboard, and returns the address of the array. The function should also return the number of elements to the calling program in a variable passed by reference.
2) Write the main function, that fills an array by calling the previous function and displays its elements.

Solution:

```
1)    float* fillArray(int* n){
        float* t,*p;
        printf("Enter the number of elements:");
        scanf("%d",&(*n));
        t=malloc(*n*sizeof(float));
        p=t;
        while(p<t+(*n)){
            printf("Enter an element: ");
            scanf("%f",&(*p));
            p++;
        }
        return t;
    }
2)    int main(){
        int n;float* t,*p;
        t=fillArray(&n);
        p=t;
        while(p<t+n){
            printf("%.2f\n",*p);
            p++;
        }
    }
```

Exercise 4:

Let **`L`** be a linked list of integers. We want to determine if all elements in **`L`** are equal. To address this, a method involves finding the minimum and maximum values in **`L`**. If these two values are equal, it indicates that all elements share the same value. While not the optimal solution, it provides an approach.

1) Write a function to return the minimum value in a linked list **L** passed as a parameter.

2) Write a function to return the maximum value in a linked list **L** passed as a parameter.

3) Write a function that takes a list of integers as input and returns whether all elements are equal.

Solution:

```
    Type   List=^node;
           node=Record
                Begin
                val:integer;
                next:List;
                End;

1)  Function Minimum(L: List): Integer; //Assuming the list is not empty
    Var min: Integer; p: List;
    Begin
    min ← L^.val;
    p ← L^.next;
    While p ≠ Nil Do
        Begin
        If p^.val < min Then
              min ← p^.val;
        p ← p^.next;
        End;
    Minimum ← min;
    End;

2)  Function Maximum(L: List): Integer; //Assuming the list is not empty
    Var max: Integer; p: List;
    Begin
    max ← L^.val;
    p ← L^.next;
    While p ≠ Nil Do
        Begin
        If p^.val > max Then
             max ← p^.val;
        p ← p^.next;
        End;
    Maximum ← max;
    End;

3)  Function areEqual(L: List): Boolean;
    Var min, max: Integer;
    Begin
    If L = Nil Then
          areEqual ← False
    Else   Begin
           min ← Minimum(L);
           max ← Maximum(L);
           If min = max then areEqual ← True
           Else areEqual ← False;
           End;
    End;
```

Exercise 5:

The objective of this exercise is to merge two linked lists, **L1** and **L2**, to create a new list composed of the elements of **L1** followed by those of **L2**. At the end, both **L1** and **L2** should be empty.

Several solutions exist; the simplest is to link the last element of **L1** with the first element of **L2**, and then point the head of the "result" list to the first element of **L1**.

Write a function to implement the described process.

Solution:

```
    Function MergeLists(Var L1, L2: List): List;
    Var L, p: List;
    Begin
    If L1 = Nil Then
          Begin
          L ← L2;
          L2 ← Nil;
          End
    Else  Begin
          p ← L1;
          While p^.next ≠ Nil Do
               p ← p^.next;
          p^.next ← L2;
          L ← L1;
          L1 ← Nil;
          L2 ← Nil;
          End;
    MergeLists ← L;
    End;
```

Exercise 6:

Consider a list of characters, **L**:

1) Write a function that returns the address of the second-to-last element in a linked list **L** passed as a parameter.
2) Write a procedure that swaps the positions of the first node and the last node in a linked list **L** passed as a parameter.

Solution:

```
    Type  List=^node;
          node=Record
               Begin
               val:Character;
               next:List;
               End;
1)  Function beforeLast(L: List): List;
    Var p: List;
    Begin
    If L = Nil or L^.next = Nil Then
          beforeLast ← Nil
    Else  Begin
          p ← L;
          While p^.next^.next ≠ Nil Do
               p ← p^.next;
```

```
            beforeLast ← p;
            End;
    End;

2)  Procedure exchange(Var L: List);
    Var p, q: List;
    Begin
    p ← beforeLast(L);
    If p ≠ Nil Then
            Begin
            If p = L Then
                  Begin
                  L ← p^.next;
                  L^.next ← p;
                  p^.next ← Nil;
                  End
            Else Begin
                   q ← p^.next;
                   q^.next ← L^.next;
                   L^.next ← Nil;
                   p^.next ← L;
                   L ← q;
                   End;
            End;
    End;
```

Exercise 7:

Consider a linked list of integers, **L**:

1) Write the recursive procedure **display(L)** to display the elements of a linked list of integers passed as a parameter.
2) Write the recursive function **search(L, v)** that searches and returns whether the value **v** belongs to the list **L**.
3) Modify the previous function to return the address of the element containing the value (**Nil** if it does not belong to the list) and not just whether it exists or not.
4) Write the recursive procedure **sum(L, sEven, sOdd)** that calculates and returns the sum of even elements and the sum of odd elements in a linked list **L** passed as input.

Solution:

```
1)  Procedure display(L: List);
    Begin
    If L ≠ Nil Then
        Begin
        Write(L^.val);
        display(L^.next);
        End;
    End;

2)  Function search(L: List; v: Integer): Boolean;
    Begin
    If L = Nil Then
            search ← False
    Else if L^.val = v Then
```

```
            search ← True
    Else   search ← search(L^.next, v);
    End;

3)  Function search (L: List; v: Integer): List;
    Begin
    If L = Nil Then
            search ← Nil
    Else if L^.val = v Then
            search ← L
    Else   search ← search (L^.next, v);
    End;

4)  Procedure sum(L: List; Var sEven, sOdd: Integer);
    Begin
    If L = Nil Then
          Begin
          sEven ← 0;
          sOdd ← 0;
          End
    Else   Begin
          sum(L^.next, sEven, sOdd);
          If L^.val mod 2 = 0 Then
                sEven ← sEven + L^.val
          Else sOdd ← sOdd + L^.val;
          End;
    End;
```

Exercise 8:

Consider a list of real numbers.

1) Write the function **`insertSorted(L, v)`** that inserts an element into a linked list sorted in ascending order, ensuring that the list remains sorted after insertion.
2) Write the main algorithm that prompts the user to enter 10 real numbers and inserts them into a linked list in such a way that the resulting list is sorted in ascending order.

Solution:

```
    Algorithme insertion;
    Type   List = ^Node;
          Node = Record
                Begin
                val:Real;
                next:List;
                End;
    Var L: List; i: Integer; v: Real;

1)  Function InsertSorted(L: List; v: Real): List;
    Var p, q, r: List;
    Begin
    Allocate(p);
    p^.val ← v;
    If L = Nil or L^.val > v Then
          Begin
          p^.next ← L;
```

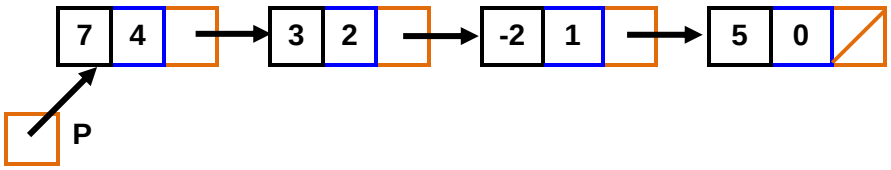
```
            L ← p;
            End
      Else  Begin
            q ← L;
            r ← L^.next;
            While (r ≠ Nil) and (r^.val < v Do
                  Begin
                  q ← r;
                  r ← r^.next;
                  End;
            q^.next ← p;
            p^.next ← r;
            End;
      InsertSorted ← L;
      End;

2)    Begin
      For i ← 1 To 10 Do
            Begin
            Read(v);
            L ← InsertSorted(L,v);
            End;
      End.
```

Exercise 9:

Polynomials can be represented by a linked list, where each term of the polynomial is stored in a node of the list containing the degree (degree) of the term and the corresponding coefficient (coeff).

For example, the polynomial $7x^4 + 3x^2 - 2x + 5$ is represented as follows:



1) Provide the data structure **`Poly`** allowing to represent a polynomial.
2) Write the function **`evaluate(P, x)`** that evaluates a polynomial **`P`** for a given value **`x`**. The polynomial and the value **`x`** are passed as arguments.

Solution:

```
1)    Type  Poly = ^Node;
            Node = Record
                  Begin
                  degree,coeff:Real;
                  next:Poly;
                  End;

2)    Function Evaluate(P: Poly; x: Real): Real;
      Var q: Poly; s: Real;
      Begin
      s ← 0;
      q ← P;
      while q ≠ Nil Do
            Begin
```

```
        s ← s + (q^.coeff) * x^(q^.degree);
        q ← q^.next;
        End;
    Evaluate ← s;
    End;
```

Additional Exercises

Exercise 10:

Always using the pointer formalism, extend Exercise 3 by adding a procedure that determines and returns the maximum and minimum values of an array of real numbers of any size. Therefore, four parameters need to be considered: the array, its size, the maximum value, and the minimum value.

Solution:

```
void MinMax(float* t,int n,float* min,float* max){
    float* p;
    *min = *t;
    *max = *t;
    for(p=t+1; p<t+n; p++){
        if(*p < *min) *min = *p;
        else if(*p > *max) *max = *p;
    }
}
```

Exercise 11:

Consider the sequence u_n defined by:

$u_0 = 1$ and $u_{n+1} = 3u_n + 1$

Using pointer formalism:

1) Write a function in C that takes an integer **n** as a parameter and returns an array containing the first **n** terms of the sequence u_n.
2) Write the main program that inputs an integer **n** and displays the first **n** terms of the sequence u_n. These terms should be calculated and returned by the previous function.

Solution:

```
int* suite(int n){
    int* t,*p;int i;
    t=malloc(n*sizeof(int));
    p=t;
    *p=1;
    p++;
    while(p<t+n){
      *p=3**(p-1)+1;
      p++;
    }
    return t;
}
main(){
    int n,i,*t,*p;
    scanf("%d",&n);
    t=suite(n);
    for(p=t;p<t+n;p++)
       printf("%d ",*p);
```

```
}
```

Exercise 12:

Let **L1** and **L2** be two lists of integers.

Write a procedure **DisplayCommon(L1, L2)** that displays all elements common to both lists **L1** and **L2**.

Solution:

```
Procedure DisplayCommon(L1, L2: Liste);
Var p, q: Liste;
Begin
p ← L1;
While p ≠ Nil Do
   Begin
   q ← L2;
   While q ≠ Nil Do
      Begin
      If p^.val = q^.val Then
           Write(p^.val);
      q ← q^.next;
      End;
   p ← p^.next;
   End;
End;
```

Exercise 13:

1) Write the function **isSorted(L)** to determine if a linked list **L** passed as a parameter is sorted (in ascending order) or not. The function should return a boolean value.
2) Write the function **searchSorted(L, v)** that searches if a value **v** exists in a sorted linked list **L**. This function should perform the minimum number of tests possible.

Solution:

```
1)  Function isSorted(L: List): Boolean;
    Var p, q: List; sorted: Boolean;
    Begin
    If L = Nil Then
        isSorted ← False
    Else  Begin
          sorted ← True;
          p ← L;
          q ← L^.next;
          While q ≠ Nil and sorted = True Do
               Begin
               If p^.val < q^.val Then
                    Begin
                    p ← q;
                    q ← q^.next;
                    End
               Else sorted ← False;
               End;
          isSorted ← sorted;
          End;
```

```
    End;
2)  Function searchSorted(L: List; v: Integer): Boolean;
    Var p: List;
    Begin
    p ← L;
    While p ≠ Nil and p^.val < v Do
        p ← p^.next;
    If p = Nil or p^.val > v Then
        searchSorted ← False
    Else searchSorted ← True;
    End;
```