III. Pointeurs, variables dynamiques, et listes chaînées

1. Pointeurs

Un pointeur est une variable dont le contenu est une adresse mémoire.

```
pointeur = adresse mémoire
```

1.1. Déclaration des variables pointeur

La déclaration d'une variable de type pointeur suit la syntaxe suivante : `type *identificateur;`

Exemple

```
int *p;       // pointeur sur un entier
char *q;      // pointeur sur un caractère
```

Les instructions (déclarations) suivantes sont équivalentes :

```
int* p;
int * p;
int *p;
```

Considérons maintenant la déclaration suivante :

```
int* p, q;
```

En fait, `p` est une variable de type pointeur `(int*)` alors que `q` est une variable de type `int`. Pour cette raison, on préfère attacher l'astérisque `(*)` au nom de la variable :

```
int *p, q;
```

Pour déclarer deux variables pointeur, ajouter `*` à chaque variable :

```
int *p, *q;
```

Considérons la déclaration suivante :

```
int x;
int *p;
```

On a dit qu'un pointeur est une variable qui contient une adresse mémoire. Alors, comment peut-on mettre l'adresse de la variable `x` dans la variable `p`?

Opérateur d'adresse (&)

Cet opérateur retourne l'adresse de son opérande.

```
int x;
int *p;
p = &x; // cette instruction attribue l'adresse de x à p
```

Maintenant, est-ce qu'on peut modifier le contenu de la variable `x` en utilisant le pointeur `p`?

Opérateur de déréférencement (*)

```
int x = 25;
int *p;
p = &x;       // stocke l'adresse de x dans p
*p = 55;      // *p: variable (mémoire) pointée par p
```

Exemple

```
int main(){
    int x = 25;
    int *p ;
    p = &x;
    cout << "p  = "<< p << endl;
    cout << "&x = "<< &x << endl;

    cout << "&p = "<< &p << endl;

    cout << "x  = "<< x << endl;
    cout << "*p = "<< *p << endl;

    *p = 55;

    cout << "x  = "<< x << endl;
    cout << "*p = "<< *p << endl;
}
```

Remarque : L'écriture suivante est incorrecte.

```
int n = 78;
int *p = &n;
*n = 24;      // Erreur de compilation
```

Initialisation des variables pointeur

```
int x;
cout << "x = " << x << endl;
```

En effet, indépendamment du fait que C++ initialise automatiquement les variables ou non, on doit toujours donner des valeurs initiales aux variables. Par exemple :

```
int x = 0;
cout << "x = " << x << endl;
```

Pour initialiser une variable pointeur, on peut utiliser soit la valeur `0` soit `NULL` :

```
int *p = 0;          // p : ne pointe sur aucun élément
// ou
int *p = NULL;       // p : ne pointe sur aucun élément
```

`NULL` représente la valeur d'un pointeur qui ne pointe sur aucun élément. En effet, `NULL` est défini à l'aide de la directive `#define` comme suit:

```
#define NULL 0
```

Remarque : Le nombre `0` est le seul nombre qui peut être directement affecté à une variable pointeur.

```
int *p = 5;  // Error: invalid conversion from 'int' to 'int*'
```

Maintenant, on sait :

- Comment déclarer une variable pointeur : `int *p;`
- Comment stocker l'adresse d'une variable dans une variable pointeur du même type que la variable : `p = &x;`
- Comment manipuler des données à l'aide des pointeurs : `*p = 50;`

Considérons les instructions suivantes :

```
int n = 78;
int *p = &n;
*p = 24;
```

Ces trois instructions sont équivalentes à :

```
int n = 78;
n = 24;
```

Alors, quel est l'avantage de l'utilisation des pointeurs? On peut accéder à ces espaces mémoire en utilisant les variables utilisées pour les créer. Dans la section suivante, on va montrer comment on peut utiliser les pointeurs. Plus spécifiquement, on va montrer comment allouer et libérer la mémoire lors de l'exécution d'un programme à l'aide des pointeurs.

2. Variables dynamiques

Une variable est dite statique lorsqu'elle est déclarée dans un programme ou sous-programme et est associée à un identificateur.

Une variable statique a un identificateur.

```
int x;
```

On peut accéder à une telle variable par l'intermédiaire de son identificateur.

```
x = 2;
```

La durée de vie d'une variable statique commence à partir du point de sa déclaration jusqu'à la fin du bloc où cette variable a été déclarée.

Exemple 1

```
int myfunction(void){
       int x;        // la variable statique x est créée.
} // la variable statique x est détruite.
```

Exemple 2

```
for (int i = 0; i < N; i++){ // la variable statique i est créée.
        ...
} // la variable statique i est détruite.
```

Le langage C++ permet au cours de l'exécution d'un programme de créer dynamiquement des nouvelles variables en leur allouant de la place en mémoire grâce à l'opérateur `new`. Une variable créée de cette manière (variable dynamique: `new int;`) n'a pas d'identificateur. On la désigne en utilisant une variable auxiliaire appelée pointeur.

On doit utiliser des pointeurs pour créer, utiliser et détruire des variables dynamiques. C ++ fournit deux opérateurs pour créer et détruire les variables dynamiques: `new` et `delete`. La durée de vie d'une variable dynamique commence avec l'opérateur `new` (création) et se termine avec l'opérateur `delete` (destruction).

2.1 Opérateur new

```
new type;       // alloue une seule variable de type 'type' et retourne son adresse.
new type[size]; // alloue un ensemble (tableau) de variables et retourne son adresse.
```

L'instruction suivante : `type *p = new type;`

1/ réserve un emplacement mémoire (variable) dont la taille est évaluée d'après le type `type`.

2/ range dans `p` l'adresse de l'emplacement mémoire alloué.

Exemple 1

```
int *p;
p = new int; //réserve un emplacement mémoire de type int et range son adresse dans p.
```

Exemple 2

```
int main(){
    int *p = new int;
    cout << "p = " << p << endl;
    cout << "&p = "<< &p << endl;
    cout << "*p = "<< *p << endl;

    *p = 55;

    cout << "*p = "<< *p << endl;
}
```

Remarques

1. Dans le code suivant :

```
int *p;
int x = 25;
p = &x;
```

aucun espace mémoire n'a été alloué.

2. Dans le morceau de code suivant :

```
int *p;
p = new int;
*p = 54;
p = new int;
*p = 73;
```

on a perdu l'adresse du premier espace mémoire alloué.

3. Lorsqu'une variable dynamique n'est plus nécessaire, elle doit être détruite.

2.2 Opérateur delete

```
delete pointer;     // restitue l'espace mémoire réservé à une seule variable.
delete [] pointer;  // restitue l'espace mémoire réservé à un tableau dynamique.
```

Exemple 1

```
int *p = new int;
*p = 54;
delete p;     // supprime la réservation de l'emplacement mémoire dont
              // l'adresse est indiquée dans p.
```

Exemple 2

```
int *p;
p = new int[5];
p[0] = 4;
p[1] = 15;
delete [] p;
```

Remarque : "`delete p`" ne fait que marquer comme libre l'espace mémoire pointé par `p`. Selon le système, `p` peut encore contenir l'adresse de cet espace mémoire. Donc, pour éviter ce piège, il faut affecter la valeur `NULL` au pointeur après l'opération de suppression delete:

```
delete p;
P = NULL;
```

```
int main(){
    int *p = new int;
    cout << "p = "<< p << endl;
    cout << "*p = "<< *p << endl;

    *p = 55;
    cout << "*p = "<< *p << endl;

    cout << "p = "<< p << endl;
    delete p;
    cout << "p = "<< p << endl;
}
```

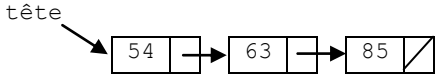
```
int main(){
    int *p = new int;
    cout << "p = "<< p << endl;
    cout << "*p = "<< *p << endl;

    *p = 55;
    cout << "*p = "<< *p << endl;

    cout << "p = "<< p << endl;
    delete p;
    p = NULL;
    cout << "p = "<< p << endl;
    //cout << "*p = " << *p << endl;
}
```

Si un pointeur `p` a la valeur `NULL`, la variable `*p` n'existe pas et une référence à `*p` provoquera une erreur fatale.

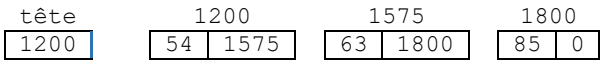
```
p = NULL;
*p = 4;       // Erreur fatale
```

3. Listes simplement chaînées

Une liste chaînée est une suite d'éléments ou composants, appelés nœuds. Chaque nœud contient l'adresse du nœud prochain (sauf le dernier).

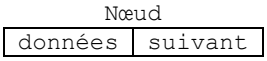


ou



Une liste chaînée a un début ou tête et une fin ou queue. La propriété principale d'une liste chaînée est la nature dynamique de ses éléments. Les nœuds peuvent êtres ajoutés ou enlevés à tout moment. Comme le nombre de nœuds est variable l'utilisation des pointeurs s'impose.

Chaque nœud d'une liste chaînée comporte deux parties.



- Données : dépend de l'application. Ce champ contient les données qui peuvent être une variable simple (`int, double, ...`), une structure (`struct`) ou un `pointeur` sur une structure.
- Suivant : ce champ est un pointeur qui indique l'adresse du nœud suivant dans la liste. Le type du pointeur est le nœud lui-même.

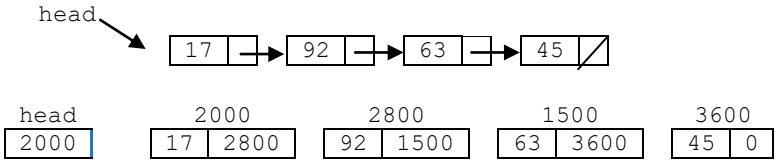
Chaque nœud doit être déclaré comme une classe ou une structure.

```
struct node{
       int data;
       node* next;
};

node* head;
```

Une autre façon de déclaration :

```
typedef struct node* ptr;
struct node{
       int data;
       ptr next;
};

ptr head;
```

Exemple




```
head = 2000
head->data = 17
head->next = 2800
head->next->data = 92
head->next->next = 1500
head->next->next->data = 63
head->next->next->next = 3600
```

3.1 Création

```
#include<iostream>

using namespace std;

typedef struct node* ptr;
struct node{
       int data;
       ptr next;
};

void createListBackward(ptr& head){
     head = NULL;
     cout << "Creation of a linked list." << endl;
     int data;
     cout << "Enter a value (0 to exit): "; cin >> data;
     while (data){ // while(data) means while(data != 0)
                ptr p = new node;
                p->data = data;
                p->next = head;
                head = p;
                cout << "Enter a value (0 to exit): "; cin >> data;
     }
}


void displayList(ptr head){
     cout << "Your list:" << endl;
     while (head){
           cout << head->data << " ";
           head = head->next;
     }
}

int main(){
    ptr head;
    createListBackward(head);
    displayList(head);
}
```

Une autre façon pour créer une liste chaînée.

```
void createListBackward (ptr& head){
     head = NULL;
     while (true){
                int data;
                cout << "Enter a value (0 to exit):"; cin >> data;
                if (!data)  // if (data == 0)
                    break;
                ptr p = new node;
                p->data = data;
                p->next = head;
                head = p;
     }
}
```

Dans cette deuxième fonction `createListBackward()`, on a évité la répétition des instructions :

```
cout << "Enter a value (0 to exit): "; cin >> data;
```

Comme déjà mentioné, en programmation, il est toujours souhaitable d'éviter les situations où on exprime deux fois (ou même plusieurs fois) de façon indépendante la même chose à des endroits différents. Parce que, par exemple, si on veut modifier une partie, on pourrait oublier de modifier l'autre.

Exercice : Écrire un sous-programme qui réalise la création d'une liste chaînée en respectant l'ordre initial des éléments.

3.2 Insertion en tête de liste

```
//  cette fonction permet d'insérer un nouvel élément au début de la liste.
void insertFront(ptr& head, int data){
	ptr p = new node;
	p->data = data;
	p->next = head;
	head = p;
}
```

3.3 Insertion en queue de liste

```
// Cette fonction permet d'insérer un nouvel élément à la fin de la liste.
void insertBack(ptr& head, int data){
	ptr p = new node;
	p->data  = data;
	p->next = NULL;
	if(head){
		ptr q = head;
		while(q->next){ q = q->next; }
		q->next = p;
	}
	else { head = p; }
}
```

3.4 Suppression de la tête

```
// Cette fonction supprime l'élément qui se trouve au début de la liste.
void deleteFront(ptr& head){
	if(head){
		ptr p = head;
		head = head->next;
		delete p;
	}
}
```

3.5 Suppression de la queue

```
// cette fonction supprime l'élément se trouvant à la fin de la liste.
void deleteBack(ptr& head){
	if(head){
		if (head->next){
			ptr p = head;
			while(p->next->next){p = p->next;}
			delete p->next; p->next = NULL;
		}
		else {
			delete head; head = NULL;
		}
	}
}
```

3.6 Recherche d'un élément

```
ptr search(ptr head, int data){
       while(head && head->data != data){ head = head->next; }
       return head;
}
```

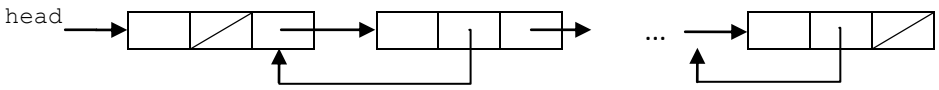
Cette fonction permet de déterminer si un élément figure dans une liste chaînée. Elle retourne un pointeur sur l'élément si l'élément existe dans la liste, sinon elle retourne le pointeur `NULL`.

Remarquez bien que dans cette fonction si `head` est passée par variable, l'appelant récupère la modification et la tête de la liste sera perdue.

Exercice : Écrire une fonction qui permet de déterminer si un élément figure dans une liste chaînée triée. La fonction retourne un pointeur sur l'élément, si l'élément existe dans la liste, sinon elle retourne le pointeur `NULL`.

4. Listes doublement chaînées

On appelle liste doublement chaînée une liste dans laquelle chaque élément (nœud) possède un pointeur vers son successeur et un pointeur vers son prédécesseur. C'est-à-dire que chaque nœud contient l'adresse du nœud suivant (sauf le dernier nœud), et chaque nœud contient l'adresse du nœud précédent (sauf le premier nœud).




```
typedef struct node* ptr;
struct node{
       int data;
       ptr prev;
       ptr next;
};
```

4.1 Insertion en tête

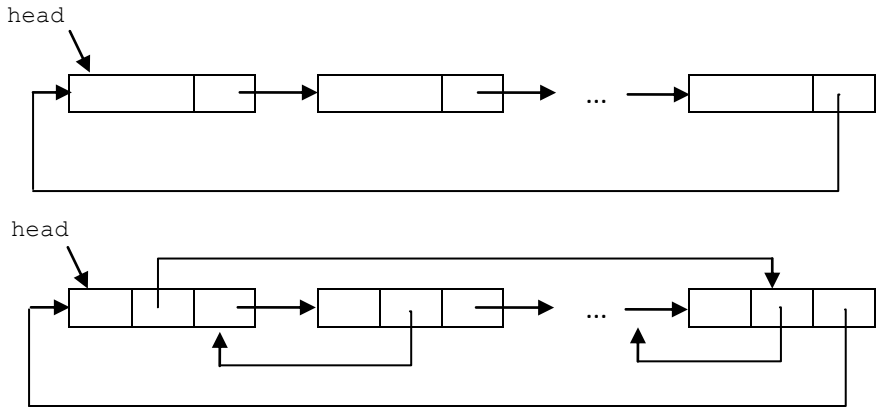
```
void insertFront(ptr& head, int data){
       ptr p = new node;
       p->data = data;
       p->prev = NULL;
       p->next = head;
       if (head != NULL){ head->prev = p; }
       head = p;
}
```

4.2 Suppression de la tête

```
void deleteFront(ptr& head){
    if(head != NULL){
              ptr p = head;
              head = head->next;
              delete p;
              if (head != NULL){ head->prev = NULL; }
    }
}
```

5. Listes circulaires

On appelle liste circulaire toute liste dans laquelle le dernier élément possède un pointeur sur la tête de la liste. Les listes simplement chaînées et les listes doublement chaînées peuvent être circulaires.



6. Exercices

1. Écrire un sous-programme qui permet de supprimer tous les nœuds d'une liste chaînée.
2. Écrire un sous-programme qui détermine si une liste chaînée est vide. Le sous-programme doit retourner true si la liste est vide, sinon il retourne false.
3. Écrire un sous-programme qui détermine le nombre de nœuds dans une liste chaînée.
4. Écrire une fonction qui permet de déterminer si un élément figure dans une liste chaînée triée. La fonction retourne un pointeur sur l'élément, si l'élément existe dans la liste, sinon elle retourne le pointeur NULL.
5. Écrire un sous-programme qui élimine les éléments redondants dans une liste simplement chaînée d'entiers.
6. Écrire un sous-programme qui concatène deux listes chaînées de chaînes de caractères.
7. Écrire un sous-programme qui concatène deux listes chaînées triées.
8. Écrire un sous-programme qui permet d'ajouter un élément (nombre) à une liste simplement chaînée triée d'entiers. Si le nombre existe déjà le sous-programme ne doit rien faire.