



COLLEGE  
DE PARIS  
SUPÉRIEUR  

---

DAKARTECH

---

Programmation orientée objets  
[C++]

Dr. Pape Abdoulaye BARRO

# Rappel sur les bases

- ❑ Les variables, les opérateurs et les opérations
- ❑ Structures de contrôles
- ❑ Tableaux et pointeurs
- ❑ Fonctions et récursivité
- ❑ Quelques algorithmes de tri et de recherche
- ❑ **Fichiers**
- ❑ Structures
- ❑ Listes chaînées, Piles, Files

# Rappel

## Fichiers

Jusque-là, nous ne savons que lire et écrire sur une console. Dans cette section, nous allons apprendre à interagir avec les fichiers.

- Par définition, un fichier est une suite d'informations stocker sur un périphérique (disque dur, clé USB, CDRom, etc. ...).
- On peut accéder à un fichier soit en lecture seule, soit en écriture seule ou soit enfin en lecture/écriture.
- Pour pouvoir manipuler les fichiers en C++, il va falloir inclure la bibliothèque fstream (`#include <fstream>`) qui signifie "file stream" ou "flux vers les fichiers" en français.
- **Il existe 2 types de fichiers:**
  - les fichiers textes qui contiennent des informations sous forme de caractères. Ils sont lisibles par un simple éditeur de texte.
  - les fichiers binaires dont les données correspondent en général à une copie bit à bit du contenu de la RAM. Ils ne sont pas lisibles avec un éditeur de texte.



# Rappel

## Fichiers: Manipulation des fichiers textes

Lorsqu'on inclut `fstream`, il ne faut pas inclure `iostream` car ce fichier est déjà inclut dans `fstream`.

### ❑ Lecture d'un fichier texte:

- ❑ Pour ouvrir un fichier en lecture, la syntaxe est la suivante:

```
ifstream nom_fichier ("chemin_vers_le_fichier");
```

- Pour savoir si le fichier a bien été ouvert en lecture, la méthode `is_open()` est utilisée. Elle renvoie `true` si le fichier est effectivement ouvert.

```
Ex: nom_fichier.is_open();
```

- Pour fermer le fichier, on fait: `nom_fichier.close();`
- Pour tester si on est arrivé à la fin du fichier, on fait: `nom_fichier.eof();`
- La lecture dans un fichier se fait :
  - **caractère par caractère** avec `get()`:
    - `char a;`
    - `nom_fichier.get(a);`
  - **Ligne par ligne**, en utilisant `getline()`:
    - `string ligne;`
    - `getline(nom_fichier , ligne); //On lit une ligne`

# Rappel

## Fichiers: Manipulation des fichiers textes

- **Mot par mot** avec les chevrons >> :
  - `nom_fichier >> variable1 [>> variable2>> ...];`
  - ❖ Ici, l'espace et le saut de ligne sont des séparateurs

```
# include <fstream>
# include <string>
using namespace std;
int main(void)
{
    string nom;
    string prenom;
    string tel;
    ifstream f ("data.txt"); // ouverture du fichier en lecture

    f >> nom >> prenom >> tel;
    while(!f.eof()) // tant qu'on n'est pas arrivé à la fin du fichier
    {
        cout << nom << " \t" << prenom << " \t" << tel << "\n"; // on affiche
        f >> nom >> prenom >> tel; // on lit les informations suivantes
    }
    f.close();

    return 0;
}
```

# Rappel

## Fichiers: Manipulation des fichiers textes

- Ecrire dans un fichier texte:
  - La création d'un nouveau fichier ou l'écriture dans un fichier existant se fait comme suit:
    - ❑ **ofstream** *nom\_fichier* ("chemin\_vers\_le\_fichier");
  - L'écriture dans un fichier se fait au moyen des chevrons << :
    - ❑ **nom\_fichier** <<"cheikh"<<" "<<"diop"<<" "<<"772220202"<<"\n";
      - ❖ Il va falloir écrire le séparateur soi-même.
  - Pour pouvoir écrire à la fin d'un fichier, il faut le spécifier lors de l'ouverture en ajoutant un deuxième paramètre à la création du flux :
    - ❑ **ofstream** *nom\_fichier* ("chemin\_vers\_le\_fichier", ios::app);
      - ❑ **app** pour dire "append" en anglais et qui signifie "ajouter à la fin".



# Rappel

## Fichiers: Manipulation des fichiers textes

Exemple:

```
# include <fstream>
# include <string>
using namespace std;
int main(void)
{
    string nom;
    string prenom;
    string tel;

    ofstream f ("data.txt"); // ouverture du répertoire en écriture
    for(int i=0; i<10; i++){
        cout << "\n p"<< i+1 << ":\n"
        cout << "nom:";
        cin >> nom;
        f << nom << " ";
        cout << "\n prenom:";
        cin >> prenom;
        f << prenom << " ";
        cout << "\n tel:";
        cin >> tel;
        f << tel << "\n";
    }
    f.close();

    return 0;
}
```

# Rappel

## Fichiers

### EXERCICES D'APPLICATIONS

- Application 24:

Écrire un programme qui écrit dans le fichier data.txt le texte suivant:

- Bonjour les étudiants!
- Bonjour Professeur
- Comment allez-vous?
- Nous allons bien merci et de votre côté?
- Ça va bien aussi, merci!

- Application 25:

Soit le fichier data.txt précédemment créé, écrire un programme permettant de lire puis d'afficher son contenu.



# Rappel sur les bases

- ❑ Les variables, les opérateurs et les opérations
- ❑ Structures de contrôles
- ❑ Tableaux et pointeurs
- ❑ Fonctions et récursivité
- ❑ Quelques algorithmes de tri et de recherche
- ❑ Fichiers
- ❑ **Structures**
- ❑ Listes chaînées, Piles, Files

# Rappel

## Structures

- Plus haut, nous avons vu les tableaux qui sont une sorte de regroupement de données de même type. Il serait aussi intéressant de regrouper des données de types différents dans une même entité.
- Nous allons donc créer un nouveaux type de données (plus complexes) à partir des types que nous connaissons déjà: **les structures**.
  - Une structure permet donc de rassembler sous un même nom, des informations de type différent. Elle peut contenir des données **entières, flottantes, tableaux, caractères, pointeurs, structure, etc...**. Ces données sont appelés les membres de la structure.
  - **Exemple**: la carte d'identité d'une personne: (nom, prenom, date\_de\_naissance, lieu\_de\_naissance, quartier, etc... ).

# Rappel

## Structures: déclaration

Pour déclarer une structure, on utilise le mot clé `struct`. Syntaxe:

```
struct nomStructure {  
    type_1 nomMembre1 ;  
    type_2 nomMembre2 ;  
    ...  
    type_n nomMembren ;  
};
```

**Exemple :**

```
struct Personne {  
    int age;  
    double poids;  
    double taille;  
};
```

Une fois la structure déclarée, on pourra définir des variables de type structuré.

**Exemple:**

`Personne` massamba, mademba;

Massamba pourra accéder à son age en faisant `massamba.age`.



# Rappel

## Structures: déclaration

### Exemple

```
#include<iostream>

using namespace std;

struct Personne
{
    int age;
    double poids;
    double taille;
};

int main(){
    Personne massamba;
    massamba.age=25;
    massamba.poids=90,5;
    massamba.taille=185,7;
    cout << " Massamba a " << massamba.age << " ans, il pèse " <<
    massamba.poids << " kg et il fait " << massamba.taille << " cm de long ." <<
    endl;
    return 0;
}
```

# Rappel

## Structures: initialisation

Dans l'exemple précédent, nous avons attribué une valeur champ après champ. Ce qui peut s'avérer long et peu pratique.

Il est en fait possible d'initialiser les champs d'une structure au moment de son instantiation grâce à l'opérateur {}.

### Exemple:

#### person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct Personne
{
    int age;
    double poids;
    double taille;
};

#endif
```

```
#include <iostream>
#include "person.hh"
using namespace std;
int main()
{
    Personne massamba={25, 90.5, 185.7};

    cout << " Massamba a "
    << massamba.age << " ans, il pèse "
    << massamba.poids << " kg et il fait "
    << massamba.taille << " cm de long ."
    << endl;

    return 0;
}
```

20/04/2024

# Rappel

## Structures et tableau

Une structure peut contenir un tableau. De ce fait, un espace mémoire lui sera réservé à sa création.

### Exemple:

person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct Personne
{
    char nom[20];
    int age;
    double poids;
    double taille;
};

#endif
```

```
#include<iostream>
#include"person.hh"
using namespace std;
int main()
{
    Personne m={"Massamba", 25, 90.5, 185.7};

    cout << m.nom <<" a "
    << m.age << " ans, il pèse "
    << m.poids << " kg et il fait "
    << m.taille << " cm de long ."
    << endl;

    return 0;
}
```



# Rappel

## structures imbriquées

Il est possible de créer des tableaux contenant des instances d'une même structure.

### Exemple:

#### person.hh

```
#ifndef __PERSON_HH__
#define __PERSON_HH__

struct date {
    int jour;
    int mois;
    double annee;
};

struct Personne
{
    char nom[20];
    date date_de_naissance;
    double poids;
    double taille;
};

#endif
```

```
#include<iostream>
#include"person.hh"
using namespace std;
int main()
{
    Personne m[2]={
        {"Massamba", {8,8,2008}, 25.0, 185.7),
        {"Mafatou", {5,5,2010}, 30.6, 175.3)
    };

    cout << m[0].nom <<" est né en"
    << m[0].date_de_naissance.annee << " , il
    pèse "
    << m[0].poids << " kg et il fait "
    << m[0].taille << " cm de long ."
    << endl;

    return 0;
}
```

20/04/2024

# Rappel

## structures et fonctions

Une structure peut être passer à une fonction.

### Exemple:

```
#include<iostream>
using namespace std;

struct date
{
    int jour;
    int mois;
    double annee;
};

struct Personne
{
    char nom[20];
    date date_de_naissance;
    double poids;
    double taille;
};

// la suite →
```

```
void saisirUser(Personne &p){
    cout << "Tapez le nom : ";
    cin >> p.nom;
    // ...
    cout << "Tapez la taille: ";
    cin >> p.taille;
}

int main(){
    Personne p;
    cout << "SAISIE DE P" << endl;
    saisirUser(p);

    cout << p.nom << " est né en "
    << p.date_de_naissance.annee << ", il pèse "
    << p.poids << " kg et il fait "
    << p.taille << " cm de long ."
    << endl;

    return 0;
}
```

# Rappel

## fonctions membres

On peut ajouter une fonction dans une structure.

### Exemple:

```
#include<iostream>
#include<cmath>
using namespace std;

struct date
{
    int jour;
    int mois;
    double annee;
};

struct Personne
{
    char nom[20];
    date date_de_naissance;
    double poids;
    double taille;
    double inMas(double p, double t);
};

double Personne::inMas(double p, double t)
{
    return p/pow(t;2);
}

// la suite →
```

```
void saisirUser(Personne &p){
    cout << "Tapez le nom : ";
    cin >> p.nom;
    // ...
    cout << "Tapez la taille: ";
    cin >> p.taille;
}

int main(){
    Personne p;
    cout << "SAISIE DE P" << endl;
    saisirUser(p);

    cout << p.nom << " est né en "
    << p.date_de_naissance.annee << ", il pèse "
    << p.poids << " kg, il fait "
    << p.taille << " cm de long et son IMC est de : "
    << p.inMas(p.poids, p.taille)
    << endl;

    return 0;
}
```



# Rappel

## Structures

### EXERCICES D'APPLICATIONS

- Application 26:

Soit la structure suivante:

```
struct point {  
    char a ;  
    int x, y ;  
}
```

Écrire un programme faisant appel à une fonction recevant en argument l'adresse d'une structure de type point et qui renvoie une structure de même type correspondant à un point de même nom et de coordonnées opposées. Afficher les deux points.

- Application 27:

En considérant la structure de type point de l'application 24, écrire pour chaque cas de figure, un programme appelant une fonction **afficher** qui prend en argument une structure de type point en le transmettant par:

- Par valeur
- Par adresse
- Par référence

La fonction affichera le point et ses coordonnées comme suit: « *le point A de coordonnées x=5 et y=7* ».

# Rappel sur les bases

- ❑ Les variables, les opérateurs et les opérations
- ❑ Structures de contrôles
- ❑ Tableaux et pointeurs
- ❑ Fonctions et récursivité
- ❑ Quelques algorithmes de tri et de recherche
- ❑ Fichiers
- ❑ Structures
- ❑ **Listes chaînées, Piles, Files**

# Rappel

## Listes chaînées, Piles, Files

- Lorsqu'une structure contient une donnée avec un pointeur vers un élément de même composition, on parle alors de liste chaînée.
  - ❑ Les listes chaînées sont basées sur les **pointeurs** et sur les **structures**;
  - ❑ Quand une variable pointeur ne pointe sur aucun emplacement, elle doit contenir la valeur **Nil** - Not In List (qui est une adresse négative).

Par **définition**, une liste chaînée est une structure linéaire qui n'a pas de dimension fixée lors de sa création.

- Ses éléments de même type sont éparpillés dans la mémoire et sont reliés entre eux par des pointeurs;
- Chaque élément (dit **nœud**) est lié à son successeur. **Chaque prédécesseur contient le pointeur du successeur**;
- Le dernier élément de la liste ne pointe sur rien (**Nil**);
- La liste est uniquement accessible via sa **tête** de liste qui est son premier élément.



# Rappel

## Listes chaînées, Piles, Files



- Tête est le pointeur contenant l'adresse du premier élément alors que chaque nœud est une structure avec une case contenant la **valeur à manipuler** (20, 3, -11, 0 et 6) et une case contenant l'**adresse de l'élément suivant**;
- Contrairement au tableau, les éléments n'ont aucune raison d'être voisins ni ordonnés en mémoire;
- Selon la mémoire disponible, il est possible de rallonger ou de raccourcir une liste;
- Pour accéder à un élément de la liste, il faut toujours débuter la lecture de la liste par son premier élément dans le pointeur duquel est indiqué la position du deuxième élément. Dans le pointeur du deuxième élément de la liste on trouve la position du troisième élément. Ainsi de suite jusqu'à obtenir la position de l'élément... ;
- Pour **ajouter**, **supprimer** ou **déplacer** un élément il suffit d'allouer une place en mémoire et de mettre à jour les pointeurs des éléments.

# Rappel

## Listes chaînées, Piles, Files

Il existe différents types de listes chaînées :

- *Liste chaînée simple* constituée d'éléments reliés entre eux par des pointeurs;
- *Liste doublement chaînée* où chaque élément dispose de deux pointeurs pointant respectivement sur l'élément **précédent** et l'élément **suivant**. Ceci permet donc la lecture dans les deux sens;
- *Liste circulaire* où le **dernier** élément pointe sur le **premier** élément de la liste.

# Rappel

## Listes chaînées, Piles, Files

### exemples: insertion/suppression par l'avant

```
#include<iostream>
using namespace std;

int main(){
    int pos_noeud, num_noeud;
    typedef struct noeud
    {
        int data; // pour stocker l'information
        noeud *suivant; // reference au noeud suivant
    };
    // Insertion par l'avant
    noeud *tete = NULL;
    // premier noeud
    noeud *noeud1 = new noeud;
    noeud1->data=10;
    noeud1->suivant=tete;
    tete = noeud1;
    // deuxième noeud
    noeud *noeud2 = new noeud;
    noeud2->data=20;
    noeud2->suivant=tete;
    tete = noeud2;
    // Affichage
    cout<<"TETE -> ";
    while(tete!=NULL)
    {
        cout<< tete->data <<" -> ";
        tete = tete->suivant;
    }
    cout<<"NULL";

    return 0;
}
```

```
// suppression par l'avant
noeud *cellule=new noeud;
cellule=tete;
tete=cellule->suivant;
delete cellule;
```



# Rappel

Listes chaînées, Piles, Files

**exemples:** Insertion à une position spécifique

```
// .....  
cout<<"Entrer la position du noeud: ";  
cin>>pos_noeud;  
noeud *curseur=new noeud;  
curseur ->suivant=tete;  
for(int i=1;i<pos_noeud;i++){  
    curseur=curseur->suivant;  
    if(curseur==NULL){  
        cout<<"La position"<<pos_noeud<<" n'est pas dans la liste"<< endl;  
        break;  
    }  
}  
if(curseur!=NULL){  
    noeud *nouveau=new noeud;  
    nouveau->data=30;  
    nouveau->suivant=curseur->suivant;  
    curseur->suivant=nouveau;  
}  
// ...
```

# Rappel

## Listes chaînées, Piles, Files

### exemples: insertion/suppression par l'arrière

```
#include<iostream>
using namespace std;

int main(){
    typedef struct noeud
    {
        int data; // pour stocker l'information
        noeud *suivant; // référence au noeud suivant
    };
    // insertion par l'arrière
    noeud *tete = NULL;
    // premier noeud
    noeud *noeud1 = new noeud;;
    tete = noeud1;
    noeud1->data=10;
    noeud1->suivant=NULL;
    // deuxième noeud
    noeud *noeud2 = new noeud;
    noeud1->suivant = noeud2
    noeud2->data=20;
    noeud2->suivant=NULL;
    // Affichage
    cout<<"TETE-> ";
    while(tete!=NULL)
    {
        cout<< tete->data <<"-> ";
        tete = tete->suivant;
    }
    cout<<"NULL";

    // suppression par l'arrière
    noeud *cellule=new noeud;
    cellule=tete;
    noeud *ancien=new noeud;
    while(cellule->suivant!=NULL)
    {
        ancien=cellule;
        cellule=cellule->suivant;
    }
    ancien->suivant=NULL;
    delete cellule;

    return 0;
}
```

# Rappel

## Listes chaînées, Piles, Files

Les **piles** et les **files** sont des *listes chaînées particulières* permettant d'ajouter et de supprimer des éléments uniquement à une des deux extrémités de la liste.

- Une structure **pile** est assimilable à une superposition d'assiettes . On pose et on prend à partir du sommet de la pile. C'est du principe **LIFO** (Last In First Out);
- Une structure **file** est assimilable à une file d'attente de caisse. Le premier client entré dans la file est le premier à y sortir. C'est du principe **FIFO** (First In First Out).



# Rappel

## Listes chaînées, Piles, Files

une **Pile** est donc un ensemble de valeurs ne permettant des insertions ou des suppressions qu'à une seule extrémité, le **sommet**.

- l'opération insertion d'un objet sur une pile consiste à *empiler* cet objet au sommet de celle-ci.
  - **Exemple**: ajouter une nouvelle assiette au dessus de celle qui se trouve au sommet.
- l'opération suppression d'un objet sur une pile consiste à *dépiler* celui-ci au sommet de celle-ci.
  - **Exemple**: supprimer ou retirer l'assiette qui se trouve au sommet.

Une pile sert essentiellement à stocker des données ne pouvant pas être traitées immédiatement.

# Rappel

## Listes chaînées, Piles, Files

une **Pile** est un enregistrement avec une variable **sommet** indiquant le sommet de la pile et une **structure données** pouvant enregistrer les données.

La manipulation d'une **pile** en C++ nécessite d'inclure la bibliothèque **stack**. Dans cette bibliothèque nous trouvons les fonctions pour:

- La déclaration: `stack<type> pile;`
- Connaitre la taille de la pile (qui nous renvoie le nombre d'élément): `pile.size();`
- Vérifier si la pile est vide ou non: `pile.empty();`
- Ajouter une nouvelle valeur à la pile(empiler): `pile.push(element);`
- Accéder au premier élément de la pile: `pile.top();`
- Supprimer la valeur se trouvant au sommet de la pile(dépiler): `pile.pop();` // ici, la pile ne doit pas être vide !

# Rappel

## Listes chaînées, Piles, Files exemple

```
#include<iostream>
#include<stack>
using namespace std;
int main(){
    int n;
    stack<int> pile;
    // remplissage
    cout << "veuillez saisir un element: ";
    cin >> n;
    while(n>0){
        pile.push(n);
        cout << "entrer un autre element: ";
        cin >> n;
    }
    // affichage
    cout << endl;
    if(pile.size()==0){
        cout << "la pile est vide ";
    }else if(pile.size()==1){
        cout << "la pile contient un element qui est: "<< pile.top();
    }else{
        cout << "la pile contient " << pile.size() << " elements que sont : " << endl;
        while(!pile.empty()){
            cout << pile.top() << " ";
            pile.pop();
        }
    }

    return 0;
}
```



# Rappel

## Listes chaînées, Piles, Files

### Pile et fonction

Passer une pile en paramètre à un sous-programme se fait par références.

- **Exemple:** remplissage(`stack<int>& pile`) , affichage(`stack<int>& pile`);

```
...
void remplissage(stack<int>& pile)
{
    cout << "veuillez saisir un element: ";
    cin >> n;
    while(n>0){
        pile.push(n);
        cout << "entrer un autre element: ";
        cin >> n;
    }
}

...
void affichage(stack<int>& pile)
{
    if(pile.size()==0){
        cout <<"la pile est vide ";
    }else if(pile.size()==1){
        cout<<"la pile contient un element qui est: "<<pile.top();
    }else{
        cout <<"la pile contient " << pile.size() << " elements que sont : " << endl;
        while(!pile.empty()){
            cout << pile.top() << " ";
            pile.pop();
        }
    }
}
```

# Rappel

## Listes chaînées, Piles, Files

Une **File** est donc un enregistrement avec une variable **Début** indiquant le premier élément, **Queue** indiquant le dernier élément et une **structure données** pouvant enregistrer les données.

La manipulation d'une **file** en **C++** nécessite d'inclure la bibliothèque **queue**. Dans cette bibliothèque nous trouvons les fonctions pour:

- La déclaration: `queue<type> file;`
- Connaitre la taille de la file (qui nous renvoie le nombre d'élément): `file.size();`
- Vérifier si la file est vide ou non: `file.empty();`
- Ajouter une nouvelle valeur à la pile(empiler): `file.push(element);`
- Accéder au premier élément de la file: `file.front();`
- Accéder au dernier élément de la file: `file.back();`
- Supprimer le premier élément de la file(dépiler): `file.pop();` // ici, la file ne doit pas être vide !

# Rappel

## Listes chaînées, Piles, Files

```
#include<iostream>
#include<queue>
using namespace std;
int main(){
    int n;
    queue<int> file;
    // remplissage
    cout << "veuillez saisir un élément: ";
    cin >> n;
    while(n>0){
        file.push(n);
        cout << "entrer un autre élément: ";
        cin >> n;
    }
    // affichage
    if(file.size()==0){
        cout << "la file est vide ";
    }else if(file.size()==1){
        cout<<"la file contient un élément qui est: "<<file.front();
    }else{
        cout <<"la file contient " << file.size() << endl;
        cout << "Le premier élément est : " << file.front() << endl;
        cout << "Le dernier élément est : " << file.back() << endl;
        cout << "Les éléments sont : " << endl;
        while(!file.empty()){
            cout << file.front() << " ";
            file.pop();
        }
    }
    return 0;
}
```



# Rappel

Listes chaînées, Piles, Files

## EXERCICES D'APPLICATIONS

- Application 28:
  - Ecrire un programme permettant de créer et de lire une liste chaînée d'entiers et affiche ensuite ses éléments.
- Application 29:
  - Ecrire un programme demandant à l'utilisateur la taille de la pile puis la remplir. Afficher par la suite la pile.
  - Faire la même chose pour le cas d'une file.

À suivre ...

Feedback sur:  
[pape.abdoulaye.barro@gmail.com](mailto:pape.abdoulaye.barro@gmail.com)