



PROGRAMMATION ORIENTÉE OBJETS

C++

Présentation et objectif du cours

- ❑ **Organisation du travail (48h)**
 - ❑ Cours magistral (24h)
 - ❑ TD/TP (24h)
- ❑ **Evaluation**
 - ❑ Contrôles
 - ❑ Examen
- ❑ **Outils de travail**
 - ❑ Visual C++
 - ❑ Dev-C ++
- ❑ **Prérequis**
 - ❑ Quelques notions en C

Généralités

- le C++ est un langage à la fois procédural et orienté objet contrairement au C qui est seulement procédural.
 - Un langage est dit procédural s'il permet seulement la définition des données grâce à des variables, et des traitements grâce aux fonctions. il sépare données et traitements sur ces données.
 - Un langage est dit orienté objet lorsqu'il offre un mécanisme de classe rassemblant données et traitements.
- Le paradigme de programmation orientée objet implique une méthode différente pour concevoir et développer des applications.

Généralités

Historique

- Le C++ était initialement nommé **C with Classes** (C avec classes) par son développeur **Bjarne Stroustrup**, qui a eu l'idée, en 1979, d'améliorer le langage C pour sa thèse de doctorat.
- En 1984 le nom du langage passa de C with classes à celui de « C++ ».
- Dès le départ, le langage ajoutait à C la notion de classe (avec encapsulation des données), de classe dérivée, de vérification des types renforcés (typage fort), d'argument par défaut,
- En 1989, c'est la sortie de la version 2.0 de C++ avec l'ajout des nouvelles fonctionnalités, telle que l'héritage multiple, les classes abstraites, les fonctions membres statiques, les fonctions membres constantes, et les membres protégés.
- Au cours de son évolution, des fonctionnalités nécessaires ont été ajoutées:
 - les fonctions C traditionnelles telles que printf et scanf et autres, sont remplacées;
 - parmi les ajouts les plus importants, il y avait la Standard Template Library (*stl*).
- En 2003, une version de C++ corrigée est publiée, prenant en compte les erreurs remontés par les utilisateurs.

Généralités

Fonctionnalités et inclusion de fichier d'en-tête

- On peut considérer que C++ « est du C » avec un ajout de fonctionnalités.
 - ▣ Mais, Il serait utile d'avoir à l'esprit qu'un programmes syntaxiquement corrects en C peut ne pas l'être en C++.
- L'utilisation d'une bibliothèque peut se faire par l'intermédiaire de la directive **#include** (suivie du nom du fichier d'en-tête) comme en C.
 - ▣ Aussi, depuis le **C++20**, le mot clé **import** peut servir à des fins similaires.
- Dans ce cours, nous parlerons de Classe, d'Objet, de Méthode, d'Encapsulation, de Constructeur/Destructeur, de Surcharge, d'Héritage et de Polymorphisme.

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

Les variables, les opérateurs et les opérations

- Une variable est un espace mémoire nommé, de taille fixée, prenant au cours du déroulement d'un programme, un nombre indéfini de valeurs différentes.
- Un programme tourne généralement autour des variables;
- Le changement de valeur se fait par l'opération d'affectation.
- La variable diffère de la notion de constante qui, comme son nom l'indique, ne prend qu'une unique valeur au cours de l'exécution du programme.

Rappel

Les variables, les opérateurs et les opérations

- Dans la plupart des langages de programmation, avant de manipuler une **variable**, il faut préalablement déclarer son **type**. C'est à dire que la variable en question ne pourra changer de valeur que dans l'intervalle défini par le type qui lui est assigné.
- Le tableau ci-dessous, donne les noms des types et leur plage:

type	Min	Max
Signed char	-127	127
int	-32 767	32 767
long	-2 147 483 647	2 147 483 647
float	-1×10^{37}	1×10^{37}
double	-1×10^{37}	1×10^{37}

Rappel

Les variables, les opérateurs et les opérations

- `type` *<nom_variable>* permet de déclarer une variable. Mais le nommage des variables est régi par les règles suivantes :
 - elle commence par une lettre ;
 - les espaces sont interdits. On peut utiliser 'underscore' pour cela ;
 - on ne peut pas utiliser des accents ;
 - on peut utiliser des minuscules, des majuscules et des chiffres.
- Pour déclarer une constante, il faut utiliser le mot `const` devant le `type` et il est obligatoire de lui donner une `valeur` au moment de sa `déclaration`.
- On affiche le contenu d'une variable avec '`cout`' puis chevrons ouvrant '<<' (ex: `cout<<variable;`).
- Il est possible d'afficher la valeur de plusieurs variables dans un seul `cout`. Il vous suffit pour cela de les séparer par des chevrons ouvrants (ex: `cout<<variable 1<<variable 2<<...<<variable n;`).
- On récupère les caractères saisis du tampon clavier avec `cin` au moyen d'opérateur d'entrée (chevrons fermants) '>>' (ex: `cin>>variable;`).

Rappel

Les variables, les **opérateurs** et les opérations

Un opérateur est un outil qui permet d'agir sur une variable ou d'effectuer des calculs. Il existe plusieurs types d'opérateurs:

- L'**affectation** qui confère une valeur à une variable ou à une constante. Il est représenté par le symbole '=' ;
- Les **opérateurs arithmétiques** qui permettent d'effectuer des opérations arithmétiques entre opérandes numériques :

+	addition
-	soustraction
*	multiplication
/	division
%	modulo

Rappel

Les variables, les opérateurs et les opérations

■ Les opérateurs relationnels :

>	supérieur
<	inférieur
>=	Supérieur ou égal
=<	Inférieur ou égal
==	égal
!=	différent

■ Les opérateurs logiques :

- Opérateur unaire : « ! » (négation) ;
- Opérateurs binaires : « && » (conjonction), « | » (disjonction).
- La **concaténation** : qui permet de créer une chaîne de caractères à partir de deux chaînes de caractère en les mettant bout à bout. Il est représenté par le symbole « + ».
- Opérateur **ternaire**: Il permet l'affectations du type.
 - Syntaxe: Si **condition** est vraie alors **variable** vaut **valeur**, sinon **variable** vaut **autre valeur**.
 - Exemple: `int a = (b > 0) ? 10 : 20;`

Rappel

Les variables, les opérateurs et les opérations

- Les opérateurs de manipulation de bit :
 - & : ET bit à bit
 - | : OU bit à bit
 - ^ : OU Exclusif bit à bit
 - << : Décalage à gauche
 - >> : Décalage à droite
 - ~ : Complément à un (bit à bit)
- La fonction sizeof est utilisée pour connaître la taille en mémoire d'une variable passé en paramètre.
 - int a = 1; sizeof(a) donne 4; double a = 3,14; sizeof(a) donne 8.
- sur les entiers et les réels : addition, soustraction, multiplication, division, division entière, puissance, comparaisons, modulo ;
- sur les booléens : comparaisons, négation, conjonction, disjonction ;
- sur les caractères : comparaisons ;
- sur les chaînes de caractères : comparaisons, concaténation

Rappel

Les variables, les **opérateurs** et les **opérations**

Lors de l'évaluation d'une expression, la priorité de chaque opérateur permet de définir l'ordre d'exécution des différentes opérations. Pour changer la priorité d'exécution, on utilise les parenthèses.

- Ordre de priorité décroissante des opérateurs arithmétiques et de concaténation :
 - «*», «/» ;
 - « % »
 - «+» et «-» ;
 - «+» (concaténation).

- Ordre de priorité décroissante des opérateurs logiques :
 - « ! »
 - « && »
 - « | | »

Rappel

Les variables, les opérateurs et les opérations

EXERCICES D'APPLICATIONS

□ Application 1 :

Ecrire un algorithme/programme permettant de déclarer deux variables de type réel, de saisir les valeurs, de calculer et d'afficher leur somme, produit et moyenne.

□ Application 2 :

Ecrire un algorithme/programme qui permet de permuter les valeurs de A et B sans utiliser de variable auxiliaire.

□ Application 3 :

Ecrire un algorithme/programme permettant de déclarer trois variables A, B, C de type réel, d'initialiser leurs valeurs et ensuite d'effectuer la permutation circulaire des trois variables.

□ Application 4 :

Ecrire un algorithme/programme qui permet de saisir les paramètres d'une équation du second degré et de calculer son discriminant delta.

□ Application 5 :

Ecrire un algorithme/programme qui à partir de la valeur saisie du côté d'un carré donné, permet de calculer son périmètre et sa surface et affiche les résultats à l'écran.

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 conditionnelles et les structures itératives

- Un ordinateur exécute un programme de manière séquentielle. Pour lui doter de l'intelligence relative afin d'être capable d'effectuer des choix ou des boucles sur un bloc d'instructions et de casser cette linéarité, il va falloir utiliser les structures de contrôle.
- Parmi les structures de contrôle nous avons :
 - LES STRUCTURES CONDITIONNELLES
 - LES STRUCTURES ITERATIVES

Rappel

Structures conditionnelles et les structures itératives

- if, syntaxes:

```
if(condition) {  
    /*instructions*/  
}
```

- Condition étant une expression booléenne

- Exemple:

-
- if(jour != 7) {
- cout <<"Je vais à l'école"<<endl;
- }
-

Rappel

Structures conditionnelles et les structures itératives

- if...else, syntaxes:

```
if(condition) {  
    /*instructions*/  
} else {  
    /*instructions*/  
}
```

- Exemple:

-
- if(jour != 7 && greve==0) {
- cout <<"Je vais à l'école"<<endl;
- } else {
- cout <<"Il n'y a pas école"<<endl;
- }
-

Rappel

Structures conditionnelles et les structures itératives

□ if imbriquées, syntaxes:

```
if(condition1) {  
    /*instructions*/  
} else if(condition 2){  
    /*instructions*/  
} else if(condition 3){  
    /*instructions*/  
}  
...  
else if(condition n){  
    /*instructions*/  
} else {  
    /*instructions*/  
}
```

□ Exemple:

```
• ...  
• if(jour==1) {  
•     cout <<"Lundi"<<endl;  
• } else if(jour==2){  
•     cout <<"Mardi"<<endl;  
• } else if(jour==3){  
•     cout <<"Mercredi"<<endl;  
• }  
• ...  
• else if(jour==6){  
•     cout <<"Samedi"<<endl;  
• } else {  
•     cout <<"Dimanche"<<endl;  
• }  
• ...
```

Rappel

Structures conditionnelles et les structures itératives

□ Structure à choix multiple, syntaxes:

```
switch (expression)
{
    case valeur1:
        {instruction 1};
        break;
    case valeur2:
        {instruction 2};
        break;
    ...
    default:
        {suite_instruction} ;
}
```

□ Exemple:

```
• ...
• switch(jour)
• {
•     case 1:
•         cout <<"Lundi"<<endl;
•         break;
•     case 2:
•         cout <<"Mardi"<<endl;
•         break;
•     case 3:
•         cout <<"Mercredi"<<endl;
•         break;
•     ...
•     case 6:
•         cout <<"Samedi"<<endl;
•         break;
•     default:
•         cout <<"Dimanche"<<endl;
•     }
• ...
```

Rappel

Structures conditionnelles et les structures itératives

EXERCICES D'APPLICATIONS

□ Application 6 :

Écrivez un programme qui calcule les solutions réelles d'une équation du second degré $ax^2+bx+c = 0$ en discutant la formule:

- Utilisez une variable d'aide d pour la valeur du discriminant $b^2 - 4*a*c$ et décidez à l'aide de d , si l'équation a une, deux ou aucune solution réelle. Utilisez des variables du type entier pour a , b et c . Affichez les résultats et les messages nécessaires sur l'écran.

□ Application 7 :

Écrivez un programme qui permet de calculer la superficie d'un cercle, d'un rectangle ou d'un triangle. L'utilisateur saisira "C", "R" ou "T" selon la superficie de la figure qu'il souhaite calculer, ensuite il saisira les dimensions.

Selon le choix de l'utilisateur, le programme doit pouvoir lui demander de saisir les dimensions appropriées.

Afficher ensuite à l'écran selon son choix la superficie demandée

Rappel

Structures conditionnelles et **les structures itératives**

- Supposons qu'on veut afficher tous les nombres entiers comprises entre 9 et 999. il va falloir faire:

```
cout<<"9";  
cout<<"10";  
...  
cout<<"999";
```

Une tache répétitive fastidieuse. D'où la nécessité de trouver une solution alternative.

- Une itération consiste en la répétition d'un blocs d'instructions jusqu'à ce qu'une certaine condition soit vérifiée.

Il en existe 2 sortes:

- Le nombre d'itérations est connu d'avance
- Le nombre d'itération dépend du résultat précédemment obtenue.

Rappel

Structures conditionnelles et **les structures itératives**

- **for**, syntaxe:

```
for (initialisation ; condition ; incrémentation){  
    /*instructions*/  
}
```

- Exemple:

-
- int compteur;
- **for** (compteur = 9; compteur < 1000 ; compteur++)
- {
- cout << compteur << endl;
- }
-

Rappel

Structures conditionnelles et les structures itératives

□ while, syntaxe:

```
while(condition ){  
    /*instructions*/  
}
```

□ Exemple:

-
- `int result(0), i(1), n;`
- `cout << "Entrez un entier naturel ?" << endl;`
- `cin >> n;`
- `while(i<=n)`
- `{`
- `result = result + i;`
- `i = i+1;`
- `}`
- `cout << "Somme =" << result << endl;`
-

Rappel

Structures conditionnelles et **les structures itératives**

- `do...while`, syntaxe:

```
do {  
    /*instructions*/  
} while(condition_de_reprise );
```

- le contenu de la boucle sera toujours lu au moins une fois.
- Exemple:
 -
 - `int` nombre(0);
 - `do`
 - `{`
 - `cout << "veuillez entrer un entier ?" << endl;`
 - `cin >> nombre;`
 - `} while (nombre < 0);`
 -

Rappel

Structures conditionnelles et les structures itératives

EXERCICES D'APPLICATIONS

□ Application 8:

Écrivez un programme qui calcule les solutions réelles d'une équation du second degré $ax^2+bx+c = 0$ en discutant la formule:

- Utilisez une variable d'aide d pour la valeur du discriminant $b^2 - 4*a*c$ et décidez à l'aide de d , si l'équation a une, deux ou aucune solution réelle. Utilisez des variables du type entier pour a , b et c . **On suppose que les valeurs saisies sont non nulles**. Affichez les résultats et les messages nécessaires sur l'écran

□ Application 9:

Ecrire un programme qui permet de faire les opérations suivantes :

- Ecrire un programme qui affiche la somme des n premiers entiers naturels. La valeur de n est saisie au clavier lors de l'exécution.
- Ecrire un programme qui affiche la somme des entiers compris entre les entiers d et f . Les valeurs de d et f sont saisies au clavier lors de l'exécution.
- Ecrire un programme qui affiche la somme des valeurs absolues des entiers compris entre les entiers relatifs d et f . Les valeurs de d et f sont saisies au clavier lors de l'exécution.

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

Tableaux, tableaux dynamiques (vector/string) et pointeurs

- Un tableau est une liste d'éléments ayant le même type et désignés sous le même nom et accessibles par indices.
 - Il est déclaré comme suit:
 - `type nom [taille] ;`
 - Exemple:
 - `double notes[10] ;`
 - En lecture, nous avons:
 - `cin>> notes[i]; // i=0,1,2,...,9`
 - En écriture, nous avons:
 - `cout<<notes[i]<<endl; // i=0,1,2,...,9`

Rappel

Tableaux, tableaux dynamiques (vector/string) et pointeurs

- On peut avoir un tableau Bidirectionnel(ou multidimensionnel). Dans ce cas, il est déclaré comme suit:
 - `type nom [ligne][colonne] ;`
 - Exemple:
 - `double matrice[2][3] ;`
 - En lecture, nous avons:
 - `cin>> matrice[1][2]; // i=0,1 et j=0,1,2`
 - En écriture, nous avons:
 - `cout<< matrice[i][j] <<endl; // i=0,1 et j=0,1,2`

Rappel

Tableaux, tableaux dynamiques (vector/string) et pointeurs

- On peut attribuer un nom à un type, dans ce cas, on utilise la déclaration `typedef`:
- Sa forme générale est: **typedef** <déclaration>
 - C'est très pratique pour nommer certains types de tableaux.
 - Exemple:
 - **typedef** `int` `Matrice`[2][3]; // définit un type Matrice
 - On peut l'utiliser pour déclarer ensuite, son équivalent :
 - `Matrice` M;

Rappel

Tableaux, **tableaux dynamiques** (**vector/string**) et pointeurs

Un tableau dynamique est un tableau dont la taille peut varier.

- Avec `vector`, la syntaxe est la suivante: **`vector<TYPE> nom(TAILLE);`**
// Il va falloir inclure la bibliothèque `<vector>`
- Quelques fonctions utiles:
 - `push_back ()`: ajout à la fin du vecteur
 - `pop_back ()`: retire de la fin du vecteur
 - `size()` : retourne le nombre d'éléments du vecteur
 - `erase()` : supprime un éléments ou un intervalle d'un vecteur et déplace les éléments suivants.

Rappel

Tableaux, tableaux dynamiques (vector/string) et pointeurs

□ Exemples:

- Créer un tableau vide: `vector <double> tab;`
- Créer un tableau de 10 éléments: `vector <int> tab(10);`
- Créer un tableau de 10 éléments initialisés à 0:
 - ❖ `vector <int> tab(10, 0);`
 - ❖ `tab.push_back(3);` // ajout du 11^{ème} case au tableau de valeur 3;
 - ❖ `tab.pop_back();` // suppression de la dernière case du tableau;
 - ❖ `const int taille(tab.size());` // variable contenant la taille du tableau;
 - ❖ `tab.erase(tab.begin()+5);` // suppression du 6^{ième} élément;
 - ❖ `tab.erase(tab.begin(), tab.begin()+5);` // suppression des 5 premiers éléments;

Rappel

Tableaux, **tableaux dynamiques** (**vector/string**) et pointeurs

Il est également possible de créer des tableaux multidimensionnels de taille variable en utilisant les vector.

- Syntaxe pour 2D: **vector<vector<TYPE>> nom**; // nous avons plus tôt un tableau de ligne.

Exemples:

- ❖ `vector<vector<int>> mat;`
- ❖ `mat.push_back(vector(3));` //ajout d'une ligne de 3 cases;
- ❖ `mat[0].push_back(4);` //ajout d'une case contenant 4 à la 1^{ière} ligne du tableau;
- ❖ `mat[0][2] = 6;` // change la valeur de la cellule (0, 2) du tableau;

Rappel

Tableaux, **tableaux dynamiques (vector/string)** et pointeurs

Une chaîne de caractère est en réalité un tableau de caractères. Ce qui veut dire qu'il a beaucoup de point communs avec les vector.

- Pour pouvoir utiliser la classe standard, il faut rajouter la bibliothèque `<string>`;
 - Elle embarque toutes les opérations de base sur les chaînes:
 - **Déclaration:** `string s1; string s2="Hello";`
 - **Saisie et Affichage:** `cin>>s1; cout<<s2;`
 - **Concaténation:** `string s3=s1+s2;`
 - ❖ `s3.size();` // pour connaître le nombre de lettres;
 - ❖ `s3.push_back("!");` // pour ajouter des lettres à la fin;
 - ❖ `s3.at(i);` // pour récupérer le i-ème caractère;
 - ❖ `getline(cin, s4);` // pour saisir une chaîne de caractères en utilisant le passage à la ligne comme séparateur (notre chaîne de caractères peut alors comporter des espaces);

Rappel

Tableaux, tableaux dynamiques (vector/string) et pointeurs

Exemple:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string prenom("Masamba");
    cout << "Je suis" << prenom << "et toi ?" << endl;
    prenom[2] = 'd';
    prenom[3] = 'e';
    cout << "moi c'est" << prenom << "!" << endl;
    return 0;
}
```

Rappel

Tableaux, tableaux dynamiques (vector/string) et pointeurs

EXERCICES D'APPLICATIONS

□ Application 10:

Ecrivez un programme qui lit au clavier une suite de nombres entiers positifs ou nuls et qui les affiche dans l'ordre inverse de leur lecture. La frappe d'un nombre négatif indique la fin de la série. Nous avons des raisons de penser qu'il n'y aura pas plus de 100 nombres.

□ Application 11 :

On considère un tableau `tab` de `N` entiers. Ecrire un programme permettant:

- a) de compter le nombre d'éléments nuls de `tab`
- b) de chercher la position et la valeur du premier élément non nul de `tab`
- c) de remplacer les éléments positifs par leur carré

□ Application 12 :

Ecrire un programme qui permet de saisir des nombres entiers dans un tableau à deux dimensions `TAB[10][20]` et de calculer les totaux par ligne et par colonne dans des tableaux `TOTLIG[10]` et `TOTCOL[20]`.

□ Application 13 :

Ecrire un programme qui permet de chercher une valeur `x` dans un tableau à deux dimensions `t[m][n]`. Le programme doit aussi afficher les indices ligne et colonne si `x` a été trouvé.

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

Un pointeur est une variable spéciale qui peut contenir l'adresse d'une autre variable.

- Si un pointeur P contient l'adresse d'une variable N, on dit que '*P pointe sur N*'.
- Les pointeurs et les noms de variables ont presque le même rôle (à *exception près*):
 - Ils donnent accès à un espace mémoire.
 - Un pointeur peut '*pointer*' sur différentes adresses tant que le nom d'une variable reste toujours lié à la même adresse.

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

□ **Déclaration**

- Syntaxe: `<type> *<nom>`

- **Exemple**: `int *p;`

□ **Affectation**

- Syntaxe: `<pointeur> = &(variable);`

- **Exemple**:

- `int n=10;`
- `int *p(0);`
- `p=&n;`

□ **Manipulation**

- `cout << "entrer une valeur";`
- `cin >> *p; // écrire dans la case mémoire pointée par p`
- `cout << "La valeur est : " << *p<< endl;`
- `cout << "L'adresse est : " << p<< endl;`

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

- Pour demander manuellement une case mémoire, on utilise l'opérateur `malloc` qui signifie « Memory ALLOCation ».
- `malloc` est une fonction ne retournant aucune valeur (`void`) (on n'en reviendra plus tard) :
 - `void* malloc(size_t nombreOctetsNecessaires);`
 - **Exemple:**
 - `int* p= NULL;`
 - `p = malloc(sizeof(int));`
- On peut libérer la ressource après usage via l'opérateur `free`
- *Free également est une fonction ne revoyant aucune valeur.*
 - `void free(void* p);`
 - **Exemple:** `free(p);`

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

L'allocation dynamique d'un tableau est un mécanisme très utile. Elle permet de demander à créer un tableau ayant exactement la taille nécessaire (pas plus, ni moins).

- Si on veut créer un tableau de n élément de type `int` (par exemple), on fera appel à `malloc`.
 - **Exemple:**
 - `int *t = NULL;`
 - `t = (int *)malloc(n*sizeof(int));`
- Autres fonctions
 - `calloc`: identique à `malloc` mais avec initialisation des cases réservées à 0.
 - `void* calloc(size_t taille, size_t nombreOctetsNecessaires);`
 - `realloc` : permet d'agrandir une zone mémoire déjà réservée
 - `void* realloc(void* tableau, size_t nombreOctetsNecessaires);`
 - **Exemple:**
 - `t = (int *) calloc (taille, sizeof(int));`
 - `taille = taille+10;`
 - `t=(int *) realloc(t, taille*sizeof(int));`

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

- En C++, comme avec `malloc` et `calloc`, il est possible de demander manuellement une case mémoire, en utilisant l'opérateur `new`.
- La syntaxe est la suivante: `<pointeur> = new type`
 - **Exemple:**
 - `int *p(0);`
 - `p = new int;`
- On peut accéder à la case et modifier sa valeur
 - **Exemple:** `*p = 10;`
- On peut libérer la ressource après usage via l'opérateur `delete`
 - **Exemple:** `delete p;`

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

On peut donc créer un tableau dynamique avec l'opérateur `new[taille]`. L'utilisation de `delete[]` permettra alors de détruire un tableau précédemment alloué.

□ **Pour le cas d'un tableau unidimensionnel, voici, ci-dessous, une illustration:**

- `int i, taille;`
- `...`
- `cout << " Entrez la taille du tableau: ";`
- `cin >> taille;`
- `int *t;`
- `t = new int[taille];`
- `...`
- `delete[] t;`

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

□ **Pour le cas d'un tableau à deux dimensions, voici, ci-dessous, une illustration:**

- `int **t;`
- `int nColonnes;`
- `int nLignes;`
- ...
- `t = new int* [nLignes];`
- `for (int i=0; i < nLignes; i++)`
 `t[i] = new int[nColonnes];`
- ...
- `delete[] t;`

Rappel

Tableaux, tableaux dynamiques (vector/string) et **pointeurs**

EXERCICES D'APPLICATIONS

□ Application 12 :

Ecrivez un programme déclarant une variable *i* de type `int` et une variable *p* de type pointeur sur `int`. Affichez les dix premiers nombres entiers en :

- n'incrémentant que `*p`
- n'affichant que *i*

□ Application 13 :

Écrire un programme qui lit un entier *n* au clavier, alloue un tableau de *n* entiers initialisés à 0, remplir le tableau par des valeurs saisies au claviers et affiche le tableau.

□ Application 14 :

Ecrire un programme qui place dans un tableau *T* les *N* premiers nombres impairs, puis qui affiche le tableau. Vous accéderez à l'élément d'indice *i* de *t* avec l'expression `*(t + i)`.

□ Application 15 :

Ecrivez un programme qui demande à l'utilisateur de saisir un nombre *n* et qui crée une matrice *T* de dimensions *n***n* avec un tableau de *n* tableaux de chacun *n* éléments. Nous noterons *tij*=0 *j*-ème élément du *i*-ème tableau. Vous initialiserez *T* de la sorte : pour tous *i*, *j*, *tij*=1 si *i*=*j* (les éléments de la diagonale) et *tij*=0 si *i*≠*j* (les autres éléments). Puis vous afficherez *T*.

□ Application 16 :

Écrire un programme allouant dynamiquement un emplacement pour un tableau d'entiers, dont la taille est fournie par l'utilisateur. Utiliser ce tableau pour y placer des nombres entiers lus également au clavier. Créer ensuite dynamiquement un nouveau tableau destiné à recevoir les carrés des nombres contenus dans le premier. Supprimer le premier tableau, afficher les valeurs du second et supprimer le tout.

□ Application 17 :

Écrire un programme qui demande à l'utilisateur de lui fournir un nombre entier entre 1 et 7 et qui affiche le nom du jour de la semaine ayant le numéro indiqué (lundi pour 1, mardi pour 2, ... dimanche pour 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

Fonctions et récursivité

Lorsque l'Algorithme à écrire devient de plus en plus important (volumineux), des difficultés d'aperçu global sur son fonctionnement se posent. Il devient très difficile de coder et de devoir traquer les erreurs en même temps.

- Il est donc utile de découper le problème en de sous problème,
- de chercher à résoudre les sous problèmes (sous-algorithmes),
- puis de faire un regroupement de ces sous-algorithmes pour reconstituer une solution au problème initial.

Un sous-algorithme est une partie d'un algorithme. Il est d'habitude déclaré dans la partie entête et est réutiliser dans le corps de l'algorithme.

- Un sous-algorithme est un algorithme. Il possède donc les même caractéristiques d'un algorithme.

Rappel

Fonctions et récursivité

- Un sous-algorithme peut utiliser les variables déclarés dans l'algorithme. Dans ce cas, ces variables sont dites globales. Il peut également utiliser ses propres variables. Dans ce cas, les variables sont dites locales. Ces dernières ne pourront alors être utilisable qu'à l'intérieur du sous-programme et nulle part ailleurs (**notion de visibilité**). Ce qui signifie que leur allocation en mémoire sera libérer à la fin de l'exécution du sous-programme.
- Un sous-programme peut être utilisable plusieurs fois avec éventuellement des paramètres différents.
- **Un sous-algorithme peut se présenter sous forme de fonction ou de procédure:**
 - Une fonction est un sous-algorithme qui, à partir de donnée(s), calcul et rend à l'algorithme un et un seul résultat;
 - alors qu'en général, une procédure affiche le(s) résultat(s) demandé(s).

Rappel

Fonctions et récursivité

□ Syntaxe d'une fonction (en Algorithmme)

Fonction Nom_Fonction (Nom_Paramètre:Type_paramètre;...): type_Fonction ;

Variable

Nom_variable : Type_variable ;
...
} // Variables locales

Début

...
Instructions ;
...
Nom_Fonction ← resultat ;
} // Corps de la fonction

Fin ;

Un appel de fonction est une expression d'affectation de manière à ce que le résultat soit récupéré dans une variable globale de même type:

Nom_variable_globale ← Nom_Fonction (<paramètres>) ;

Rappel

Fonctions et récursivité

▣ Exemple de fonction (en Algorithme)

Algorithme Calcul_des_n_premiers_nombres_entiers;

Variable

I, Som, N : entier;

Fonction Somme: entier ;

Variable

S : entier ;

Debut /*Début de la fonction*/

S \leftarrow 0 ;

Pour I \leftarrow 1 à N Faire

S \leftarrow S + I;

FinPour ;

Somme \leftarrow S

Fin /*Fin de la Fonction */;

Debut /*Début de l'algorithme*/

Som \leftarrow Somme ;

Ecrire ('La somme des ', N, 'premiers nombres est', Som) ;

Fin /*Fin de l'algorithme*/

Rappel

Fonctions et récursivité

□ Syntaxe d'une procédure (en Algorithmique)

Fonction Nom_Procedure (Nom_Paramètre:Type_paramètre;...) ;

Variable

Nom_variable : Type_variable ;

...

} // Variables locales

Début

...

Instructions ;

} // Corps de la fonction ...

Fin ;

L'appel d'une procédure peut être effectué en spécifiant, au moment souhaité, son nom et éventuellement ses paramètres; cela déclenche l'exécution des instructions de la procédure.

Rappel

Fonctions et récursivité

▣ Exemple de procédure (en Algorithmme)

Algorithmme Calcul_des_n_premiers_nombres_entiers;

Variable

I, Som, N : entier;

Procédure Somme ;

Debut /*Début de la procédure*/

Som \leftarrow 0 ;

Pour I \leftarrow 1 à N Faire

Som \leftarrow Som + I;

FinPour ;

Ecrire ('La somme des ', N, 'premiers nombres est', Som) ;

Fin /*Fin de la Fonction */;

Debut /*Début de l'algorithmme*/

Somme ;

Fin /*Fin de l'algorithmme*/

Rappel

Fonctions et récursivité

□ **Mode de passages de paramètres:** passage par valeur

On distingue deux types de passage de paramètres: par valeur et par variable (dite aussi par référence ou encore par adresse).

- Le mode de **passage par valeur** qui est le mode par défaut, consiste à copier la valeur des paramètres effectifs dans les variables locales issues des paramètres formels de la fonction ou de la procédure appelée.
 - Dans ce mode, nous travaillons pas directement avec la variable, mais avec une copie. Ce qui veut dire que le **contenu** des paramètres effectifs n'est pas modifié. À la fin de l'exécution du sous-programme, la variable conservera sa valeur initial.
 - **Syntaxe:**
 - `Procédure nom_procédure (param1:type1 ; param2, param3:type2) ;`
 - `Fonction nom_fonction (param1:type1 ; param2:type2):Type_fonction ;`

Rappel

Fonctions et récursivité

- **Mode de passages de paramètres:** passage par valeur

- **Exemple d'application**

Algorithme valeur_absolue_d-un_nombre_entier;

Variable

val: entier;

Procedure Abs(nombre: entier);

Debut /*Début de la procédure*/

Si nombre < 0 Alors

nombre \leftarrow - nombre;

FinSi ;

Ecrire (nombres) ;

Fin /*Fin de la Fonction */;

Debut /*Début de l'algorithme*/

Lire (val);

Abs (val);

Ecrire (val);

Fin /*Fin de l'algorithme*/

- Ici, val reprend sa valeur initiale. Il a juste servi de données pour Abs.

Rappel

Fonctions et récursivité

□ Mode de passages de paramètres: passage par adresse

Dans le mode de passage par variable, il s'agit pas simplement d'utiliser la valeur de la variable, mais également son emplacement mémoire.

- Le paramètre formel se substitue au paramètre effectif tout au long de l'exécution du sous-programme et à la sortie il lui transmet sa nouvelle valeur.
- Un tel passage se fait par l'utilisation du mot-clé **Var**.
- **Syntaxe:**
 - `Procédure nom_procédure (Var param1:type1 ; param2, param3:type2) ;`
 - `Fonction nom_fonction (Var param1:type1; param2:type2):Type_fonction ;`

Rappel

Fonctions et récursivité

- **Mode de passages de paramètres:** passage par adresse

- **Exemple d'application**

Algorithme valeur_absolue_d-un_nombre_entier;

Variable

val: entier;

Procedure Abs(Var nombre: entier);

Debut /*Début de la procédure*/

Si nombre < 0 alors

nombre \leftarrow - nombre;

FinSi ;

Ecrire (nombres) ;

Fin /*Fin de la Fonction */

Debut /*Début de l'algorithme*/

Lire (val);

Abs (val);

Ecrire (val);

Fin. /*Fin de l'algorithme*/

- Ici, val prend une nouvelle valeur.

Rappel

Fonctions et récursivité

Fonctions en c++

- Une fonction est un bloc paramétré et nommé
- Permet de découper un programme en plusieurs modules.
- Dans certains langages, on trouve *deux sortes de modules*:
 - Les *fonctions*, assez proches de la notion mathématique
 - Les *procédures* (Pascal) ou sous-programmes (Fortran, Basic) qui élargissent la notion de fonction.
- En C/C++, il n'existe qu'une seule sorte de module, nommé fonction
 - **Syntaxe:**

```
typeDeRetour nomFonction([arguments]){  
    //instructions;  
}
```


Rappel

Fonctions et récursivité

□ Exemple:

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

int abs(int nombre)
{
    if (nombre<0)
        nombre=-nombre;
    return nombre; // Valeur renvoyée
}

int main()
{
    int val, valAbs;
    cout << "Entrez un nombre : ";
    cin >> val;
    valAbs = abs(val); // Appel de la
fonction et affectation
    cout << "La valeur absolue de" <<
val << "est" << valAbs << endl;
    return 0;
}
```

- L'instruction **return** permet à la fois de fournir une valeur de retour et à mettre fin à l'exécution de la fonction.
- Dans la déclaration d'une fonction, il est possible de prévoir pour un ou plusieurs arguments (obligatoirement les derniers de la liste) des **valeurs par défaut** ;
 - elles sont indiquées par le signe **=**, à la suite du type de l'argument.
Exemple: **float op**(char, float=1.0, float=1.0);

- Une fonction peut ne pas renvoyer de valeur. Dans ce cas, le type de la fonction est **void**.
 - Exemple:

```
void abs(int nombre)
{
    if (nombre<0)
        nombre=-nombre;
}
```
- Lorsqu'une fonction s'appelle elle-même, on dit qu'elle est « **récursive** » (on y reviendra).

Rappel

Fonctions et récursivité

Fonctions en c++: passage par valeur

Supposons que l'on souhaite faire une permutation de deux entiers a et b.

□ **Exemple:**

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

void permute(int a, int b)
{
    int tempon = a;
    a = b;
    b = tempon;
}

int main()
{
    int a=2, b=6;
    cout << "a: " << a << " b: " << b
    << endl; // avant
    permute(a, b);
    cout << "a: " << a << " b: " << b
    << endl; // après
    return 0;
}
```

Après exécution, on constate qu'on a pas le résultat attendu.

- Par défaut, le passage des arguments à une fonction se fait par valeur.
- Pour remédier à cela, il faut passer par adresse ou par référence.

Rappel

Fonctions et récursivité

Fonctions en c++: passage par adresse

Pour modifier le paramètre réel, on passe son adresse plutôt que sa valeur.

□ **Exemple:**

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

void permute(int *a, int *b)
{
    int tempon = *a;
    *a = *b;
    *b = tempon;
}

int main()
{
    int a=2, b=6;
    cout << "a: " << a << " b: " << b << endl; // avant
    permute(&a, &b);
    cout << "a: " << a << " b: " << b << endl; // après
    return 0;
}
```

Rappel

Fonctions et récursivité

Fonctions en c++: passage par référence

On peut également passer les paramètres par référence:

□ **Exemple:**

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

```
void permute(int& a, int& b)
{
    int tempon = a;
    a = b;
    b = tempon;
}
```

```
int main()
{
    int a=2, b=6;
    cout << "a: " << a << " b: " << b
    << endl; // avant
    permute(a, b);
    cout << "a: " << a << " b: " << b
    << endl; // après
    return 0;
}
```

- Ici, le compilateur se charge de la gestion des adresses:
 - le paramètre formel est un alias de l'emplacement mémoire du paramètre réel.

Rappel

Fonctions et récursivité

Fonctions en c++: passage par référence

On peut faire passer un tableau en paramètre. Nous avons dans ce cas, deux cas de figure: par pointeur ou par semi-référence

□ **Exemple:** par pointeur

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

void affiche(int *tableau, int taille)
{
    for(int i=0; i<taille; i++)
        cout << tableau[i] << "    " <<
endl;
}

int main()
{
    int tab[5] = {1, 2, 3, 4, 5};
    affiche(tab, 5);
    return 0;
}
```

□ **Exemple:** par semi-référence

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

void affiche(int tableau[], int taille)
{
    for(int i=0; i<taille; i++)
        cout << tableau[i] << "    " <<
endl;
}

int main()
{
    int tab[5] = {1, 2, 3, 4, 5};
    affiche(tab, 5);
    return 0;
}
```

Rappel

Fonctions et **récurtivité**

Récurtivité: définitions

On appelle récurtivité tout sous-programme qui s'appelle dans son traitement.

- Il est impératif de prévoir une condition d'arrêt puisque le sous-programme va s'appeler récurtivement. sinon, il ne s'arrêtera jamais.
 - ▣ On teste la condition,
 - ▣ Si elle n'est pas vérifié, on lance à nouveau le sous-programme.

Rappel

Fonctions et **ré**cursivité

Exemples

Algorithme:

```
fonction factoriel(n : entier): entier
Début
    Si(n<2) alors
        retourner 1
    Sinon
        retourner n*factoriel(n-1)
    Fin si
Fin
```

C++:

```
int factoriel(int n)
{
    if(n<=1)
        return 1;
    else
        return(n*factoriel(n-1));
}
```

Rappel

Fonctions et **ré**curtivité

Il est également possible qu'un sous-programme appelle un second qui appelle le premier. On dit que la **ré**curtivité est indirecte, cachée, **croisée** ou mutuelle.

Exemples

Algorithme:

fonction **pair** (n : entier) : booléen

Début

 Si(n=0) alors

 retourner VRAI

 Sinon Si(n=1) alors

 retourner FAUX

 Sinon

 retourner **impair**(n-1)

 Fin si

Fin

fonction **impair** (n : entier) : booléen

Début

 Si(n=1) alors

 retourner VRAI

 Sinon Si(n=0) alors

 retourner FAUX

 Sinon

 retourner **pair**(n-1)

 Fin si

Fin

Rappel

Fonctions et récursivité

EXERCICES D'APPLICATIONS

□ Application 18 :

Ecrire un programme qui appelle trois fonctions:

- Une fonction affiche « Toc toc ! » et qui ne possède ni argument, ni valeur de retour;
- Une deuxième qui affiche « entrée » un ou plusieurs fois (une valeur reçue en argument) et qui ne renvoie aucune valeur;
- Une troisième qui fera comme la première mais un ou plusieurs fois (une valeur reçue en argument) et qui retourne cette fois-ci la valeur de 0.

□ Application 19 :

- a) Ecrire un programme utilisant une fonction qui reçoit en argument 2 nombres flottants et un caractère (opération), et qui fournit le résultat du calcul demandé.
- b) Proposer le même programme mais cette fois-ci, la fonction ne disposera plus que de 2 arguments en nombres flottants. L'opération est précisée, cette fois, à l'aide d'une variable globale.

□ Application 20 :

Ecrire un programme utilisant une fonction qui fournit en valeur de retour la somme des éléments d'un tableau d'entiers. Le tableau ainsi que sa dimension sont transmis en argument.

□ Application 21 :

Ecrire un programme faisant appel à une fonction qui ne renvoie aucune valeur et qui détermine la valeur maximale et la valeur minimale d'un tableau d'entiers. Proposer deux solutions: l'une utilisant effectivement cette notion de référence, l'autre la « simulant » à l'aide de pointeurs.

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

Quelques algorithmes de tri

Etant donné une collection d'entier placés dans un tableau. L'idée fondamentale est de trier le tableau dans l'ordre croissant.

- Les opérateurs de comparaison (\leq , \geq , $>$, $<$, ...) sont activement utilisés.
- On peut citer quelques algorithmes de tris:
 - Tris élémentaires (tris naïfs)
 - Tri par insertion
 - Tri par sélection
 - ...
 - Tris avancés (Diviser pour régner)
 - Tri fusion
 - Tri rapide
 - ...

Rappel

Quelques algorithmes de tri - Tri par insertion

Le tri par insertion consiste à pré-trier une liste afin d'entrer les éléments à leur bon emplacement dans la liste triée. à l'itération i , on insère le i -ième élément à la bonne place dans la liste des $i-1$ éléments qui le précède.

□ Principe:

- On commence par comparer les deux premiers éléments de la liste et de les trier dans un ordre;
- puis un troisième qu'on insère à sa place parmi les deux précédents;
- puis un quatrième qu'on insère à sa place parmi les trois autres;
- ainsi de suite jusqu'au dernier.

Rappel

Quelques algorithmes de tri - Tri par insertion

Considérons un tableau d'entiers de n éléments à trier.

❑ Algorithme

Pour (i allant de **2** à **n**) faire

$j \leftarrow i$;

tampon \leftarrow tab[i] ;

 Tant que ($j > 1$ ET tab[$j-1$] > **tampon**) faire

 Tab[j] \leftarrow tab[$j-1$];

$j \leftarrow j-1$;

 Fin tant que

 Tab[j] \leftarrow **tampon**;

Fin pour

❑ Complexité

Pour apprécier la complexité de cet algorithme, il suffit d'analyser le nombre de comparaisons effectué ainsi que le nombre d'échange lors du tri. On remarque qu'il s'exécute en $\Theta(n^2)$.

$$\text{➤ } n(1 + 2 + 3 + \dots + n) = n(1 + 2 + 3 + \dots + n) = n(1 + 2 + 3 + \dots + n) = n(1 + 2 + 3 + \dots + n) \rightarrow \Theta(n^2)$$

Rappel

Quelques algorithmes de tri - Tri par sélection

Le tri par sélection consiste à rechercher le minimum parmi les éléments non triés pour le placer à la suite des éléments déjà triés.

□ Principe:

- Il suffit de trouver le plus petit élément et le mettre au début de la liste;
- Ensuite, de trouver le deuxième plus petit et le mettre en seconde position;
- Puis, de trouver le troisième plus petit élément et le mettre à la troisième place;
- Ainsi de suite jusqu'au dernier.

Rappel

Quelques algorithmes de tri - Tri par sélection

Considérons un tableau d'entiers de n éléments à trier.

❑ Algorithme

Pour (i allant de **1** à **$n-1$**) faire

 Pour (j allant de $i+1$ à n) faire

 Si ($\text{Tab}[i] > \text{tab}[j]$) alors

tampon \leftarrow $\text{tab}[i]$;

$\text{tab}[i] \leftarrow \text{tab}[j]$;

$\text{tab}[j] \leftarrow$ **tampon**;

 Fin Si

 Fin pour

Fin pour

❑ Complexité

On remarque qu'il s'exécute en $\Theta(n^2)$.

Rappel

Quelques algorithmes de tri - Tri par fusion

Le tri par fusion consiste à fusionner deux tableaux triés pour former un unique tableau trié. Il s'agit d'un algorithme “diviser-pour-régner”.

□ Principe:

- ❖ Etant donné un tableau $\text{tab}[n]$:
 - si $n=1$, retourner le tableau tab ;
 - Sinon:
 - ✓ Trier le sous-tableau $\text{tab}[1 \dots n/2]$;
 - ✓ Trier le sous-tableau $\text{tab}[n/2 + 1 \dots n]$;
 - ✓ Fusionner ces deux sous-tableaux...

Rappel

Quelques algorithmes de tri - Tri par fusion

Considérons un tableau d'entiers de n éléments à trier.

□ Programme en C++

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

void triFusion(int, int, int [], int []);
int main() {
    int n, i, tab[100], tmp[100];
    cout<<" Entrez le nombre d'elements du tableau: ";
    cin>>n;
    cout<<" Entrez "<<n<<" entiers : ";
    for (i = 0; i < n; i++)
        cin>>tab[i];
    triFusion(0, n-1, tab, tmp);
    cout<<" Tableau trie : ";
    for(i = 0; i < n; i++) {
        cout<<" "<<tab[i];
    }
    cout<<"\n";
    return 0;
}
```

□ Complexité

On remarque qu'il s'exécute en $\Theta(n \log_2 n)$ opérations.

```
void triFusion(int debut, int fin, int tab[], int tmp[]) {
    if(fin <= debut)
        return;
    int milieu = (debut + fin) / 2;
    triFusion(debut, milieu, tab, tmp);
    triFusion(milieu+1, fin, tab, tmp);
    int g = debut;
    int d = milieu+1;
    for(int i=debut; i<=fin; i++){
        if(g == milieu+1){
            tmp[i] = tab[d];
            d++;
        }else if (d == fin+1) {
            tmp[i] = tab[g];
            g++;
        }else if (tab[g] < tab[d]) {
            tmp[i] = tab[g];
            g++;
        }else {
            tmp[i] = tab[d];
            d++;
        }
    }
    for(int i = debut; i <= fin; i++) {
        tab[i] = tmp[i];
    }
}
```

Rappel

Quelques algorithmes de tri - Tri rapide

Le tri rapide ou encore tri de Hoare (du nom de l'inventeur) est aussi un tri basé sur le principe “diviser-pour-régner”.

□ Principe:

- Il consiste à placer un élément du tableau (le pivot) à sa place définitive, en permutant tous les éléments qui lui sont inférieurs à gauche et ceux qui lui sont supérieurs à droite (le partitionnement).
- Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement.

Rappel

Quelques algorithmes de tri - Tri rapide

Considérons un tableau d'entiers de n éléments à trier.

□ Programme en C++

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

```
void tri_rapide(int[],int, int);
int main() {
    int n, i, tab[100];
    cout<<" Entrez le nombre elements du tableau: ";
    cin>>n;
    cout<<" Entrez "<<n<<" entiers : ";
    for (i = 0; i < n; i++)
        cin>>tab[i];
    tri_rapide(tab, 0, n);
    cout<<" Tableau trié : ";
    for(i = 0; i < n; i++) {
        cout<<" "<<tab[i];
    }
    cout<<"\n";
    return 0;
}
```

```
void tri_rapide(int tab[],int deb,int fin){
    const int pivot = tab[deb];
    int position=deb;
    if (deb>=fin)
        return;
    for (int i=deb; i<fin ; i++){
        if (tab[i]<pivot){
            tab[position]=tab[i];
            position++;
            tab[i]=tab[position];
            tab[position]=pivot;
        }
    }
    tri_rapide(tab,deb, position);
    tri_rapide(tab, position+1,fin);
}
```

□ Complexité

On remarque qu'il s'exécute en $\Theta(n^2)$ dans le pire des cas. Mais elle peut être en $\Theta(n \log_2 n)$ en moyenne.

Rappel

Quelques algorithmes de recherche d'un élément- **recherche laborieuse**

Soit x l'élément à rechercher dans un tableau t de n entiers.

- ❑ Principe:
 - On parcourt complètement le tableau et pour chaque élément, on teste l'égalité avec x .
 - En cas d'égalité, on mémorise la position.

- ❑ Algorithme

.....

indice $\leftarrow 0$;

Pour i allant de 1 à n faire

 Si ($t[i]=x$) alors

 indice $\leftarrow i$;

 Fin Si

Fin Pour

retourner indice ;

.....

Rappel

Quelques algorithmes de recherche d'un élément- **recherche séquentielle**

Soit x l'élément à rechercher dans un tableau t de n entiers.

□ Principe:

- On parcourt séquentiellement le tableau jusqu'à trouver l'élément dans une séquence.
- Si on arrive à la fin sans le trouver c'est qu'il n'est pas contenu dans la séquence.

□ Algorithme

.....

Pour i allant de 1 à n faire

 Si ($t[i]=x$) alors

 retourner i ;

 Fin Si

Fin Pour

retourner 0;

.....

Rappel

Quelques algorithmes de recherche d'un élément- **recherche dichotomique**

Soit x l'élément à rechercher dans un tableau t **ordonné** de n entiers.

□ Principe:

- On compare l'élément à rechercher avec celui qui est au milieu du tableau.
- Si les valeurs sont égales, la tâche est accomplie sinon on recommence dans la moitié du tableau pertinente.

□ Algorithme

.....

bas $\leftarrow 1$;

haut $\leftarrow \text{taille}(t)$;

position $\leftarrow -1$;

Repeter

Si ($x = t[\text{milieu}]$) alors

 position $\leftarrow \text{milieu}$;

Sinon Si ($t[\text{milieu}] < x$) alors

 bas $\leftarrow \text{milieu} + 1$

Sinon

 haut $\leftarrow \text{milieu} - 1$

Fin Si

jusqu'à ($x = t[\text{milieu}]$ OU bas $>$ haut)

retourner position

.....

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 la 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 par:
`nom_fichier >> variable1 [>> variable2>> ...];`
 - ❖ Ici, l'espace et le saut de ligne sont des séparateurs

Rappel

Fichiers: Manipulation des fichiers textes

Exemple:

```
# 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 par:
 - **nom_fichier** <<"cheikh"<<" "<<"diop"<<" "<<"772220202"<<"\n";
 - ❖ Il va falloir écrire le séparateur soi-même.

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 22:

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 23:

Soit le fichier data.txt précédemment créé, écrire un programme 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 {}.

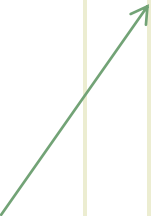
Exemple:

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;
}
```

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;
}
```

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 24:

Soit la structure suivante:

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

Ecrire 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 25:

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

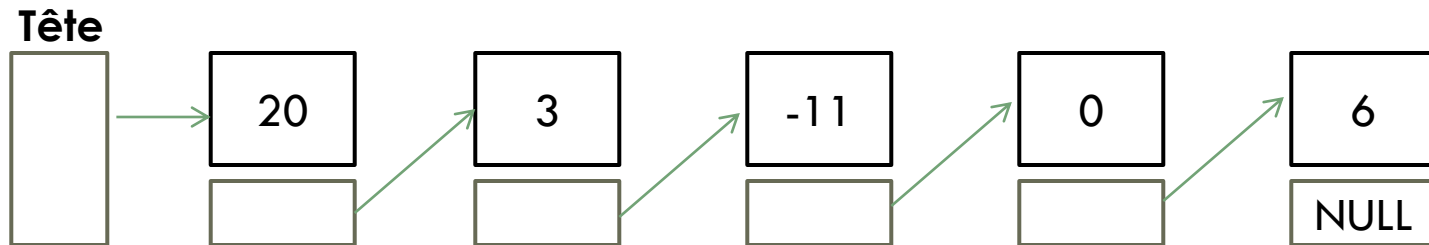
- Lorsqu'une structure contient une donnée avec un pointeur vers un élément de même composition, on parle alors de liste chaînée.
 - Les listes chaînées sont basées sur les pointeurs et sur les structures;
 - Quand une variable pointeur ne pointe sur aucun emplacement, elle doit contenir la valeur **Nil** - 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 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 débiter 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;
    }
}
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; // reference 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";

    return 0;
}
```

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

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 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. syntaxe: `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(depiler): `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)
{
    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();
        }
    }
}
```

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. syntaxe: `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(depiler): `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 element: ";
    cin >> n;
    while(n>0){
        file.push(n);
        cout << "entrer un autre element: ";
        cin >> n;
    }
    // affichage
    if(file.size()==0){
        cout <<"la file est vide ";
    }else if(file.size()==1){
        cout<<"la file contient un element 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 elements sont : " << endl;
        while(!file.empty()){
            cout << file.front() << " ";
            file.pop();
        }
    }

    return 0;
}
```

Rappel

Listes chaînées, Piles, Files

EXERCICES D'APPLICATIONS

- Application 27:
 - Ecrire un programme permettant de créer et de lire une liste chaînée d'entiers et affiche ensuite ses éléments.

- Application 26:
 - 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.

La POO

- ❑ **Notion de Classe**
- ❑ Fonctions membres
- ❑ Constructeurs, destructeurs et initialisation d'objet
- ❑ Héritage simple
- ❑ Héritage multiple
- ❑ Fonctions virtuelles et le polymorphisme
- ❑ **La gestion des exceptions**
- ❑ **Généralités sur la STL**

Notion de Classe

Dans cette section, nous abordons véritablement les possibilités de la P.O.O en C++, qui comme introduit initialement est entièrement basée sur le concept de Classe.

Une **classe** est un type de données dont le rôle est de rassembler sous un même nom à la fois données et traitements. La notion de classe n'est pas trop loin de la notion de structure:

- La déclaration d'une classe est presque similaire de celle d'une structure.
 - Il suffit de remplacer le mot clé **struct** par le mot clé **class**;
 - Puis de préciser les fonctions ou données membres publics avec le mot clé **public** et les membres privés avec le mot clé **private**.

- Syntaxe:

```
class X
{
    private :
        ...
    public :
        ...
    private :
        ...
};
```

Classe

Déclaration

En C++, la programmation d'une classe se fait en trois phases: déclaration, définition et utilisation.

- **Déclaration:** c'est la partie interface de la classe. Elle se fait dans un fichier dont le nom se termine par .h ou .hpp, .H, ou .h++ (appelé fichier d'entête).

- La syntaxe est la suivante:

```
class Nom_de_la_classe {  
    public:  
        // déclarations des données et fonctions-membres publiques  
    private:  
        // déclarations des données et fonctions-membres privées  
};
```

- **Exemple:** // [Personne.hpp](#)

```
class Personne {  
    public:  
        Personne(string, string);  
        void toString();  
    private:  
        string p_nom;  
        string p_prenom;  
};
```

Classe

Définition

- **Définition:** c'est la partie implémentation de la classe. Elle se fait dans un fichier dont le nom se termine par .cc, .c++, .c ou .cpp. Ce fichier contient les définitions des fonctions-membres de la classe.

- **Exemple:** `//Personne.cpp`

```
#include <iostream>
```

```
#include "Personne.hpp"
```

```
// constructeur (Il permet d'initialiser une nouvelle personne)
```

```
Personne::Personne(string prenom, string nom){
```

```
    p_prenom = prenom;
```

```
    p_nom = nom;
```

```
}
```

```
// une méthode (fonctions-membres)
```

```
void Personne::toString(){
```

```
    cout <<"Je m'appel"<<p_prenom<<" "<<p_nom<<endl;
```

```
}
```

Classe

Utilisation

- **Utilisation:** Elle se fait dans un fichier dont le nom se termine par .cc, .c++, .c ou .cpp. Ce fichier contient le traitement principal (la fonction *main*).

- **Exemple:** `// test.cpp`

```
#include <iostream>
#include "Personne.hpp"

// traitement principal
void main(){
    // appel implicite du constructeur
    Personne p("Nabi" , "Barro");

    p.toString();
}
```

Classe

Constructeurs et destructeurs

- Un **constructeur** est une fonction-membre déclarée du même nom que la classe, et sans type.

Syntaxe : `Nom_de_la_classe(<paramètres>);`

- **Fonctionnement**: à l'exécution, l'appel au constructeur produit un nouvel objet de la classe, dont on peut prévoir l'initialisation des données-membres dans la définition du constructeur.

Exemple: `personne p("Nabi" , "Barro");`

- Dans une classe, il peut y avoir plusieurs constructeurs à condition qu'ils diffèrent par le nombre ou le type des paramètres. *Un constructeur sans paramètre s'appelle **constructeur par défaut**.*
- Un **destructeur** est une fonction-membre déclarée du même nom que la classe mais précédé d'un tilde (~) et sans type ni paramètre.

Syntaxe : `~Nom_de_la_classe();`

- **Fonctionnement**: à l'issue de l'exécution d'un bloc, le destructeur est automatiquement appelé pour chaque objet de la classe déclaré dans ce bloc. Cela permet par exemple de programmer la restitution d'un environnement, en libérant un espace mémoire alloué par l'objet.

Classe

Constructeurs et destructeurs

L'exemple ci-dessous est un petit programme mettant en évidence les moments où sont appelés respectivement le constructeur et le destructeur d'une classe.

- Nous créons une classe `test` définissant que ces deux fonctions membres et une donnée membre **`num`** qui sera initialisée par le constructeur nous permettant d'identifier l'objet en question.

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

class test
{
    public :
        int num;
        test (int) ; // déclaration constructeur
        ~test () ; // déclaration destructeur
};
```

```
test::test (int n)
{
    num = n ;
    cout << "Appel du constructeur,
avec num=" << num<<endl;
}
```

```
test::~~test ()
{
    cout << "Appel du destructeur, avec
num=" << num<<endl;
}
```

Classe

Constructeurs et destructeurs

- Nous créons des instances de type test à deux endroits différents: dans la fonction **main** d'une part, dans une fonction **fct** appelée par **main** d'autre part.

```
main()
{
    void fct (int) ;
    test a(1) ;

    for (int i=1 ; i<=2 ; i++)
        fct(i) ;
}

void fct (int n)
{
    test t(2*n) ;
}
```

-> Appel du constructeur, avec num=1
-> Appel du constructeur, avec num=2
-> Appel du destructeur, avec num=2
-> Appel du constructeur, avec num=4
-> Appel du destructeur, avec num=4
-> Appel du destructeur, avec num=1

Classe

Constructeurs et destructeurs – quelques règles usuelles

- Un **constructeur** peut éventuellement comporter un nombre quelconque d'arguments.
- un **constructeur** n'a pas de type de retour et par conséquent, ne renvoie pas de valeur (la présence de **void** est dans ce cas une erreur).
- Un **destructeur** n'a pas de type de retour et donc, ne renvoie pas de valeur. Ici aussi, la présence de **void** est une erreur.
- Les **constructeurs** et **destructeurs** peuvent être **publics** ou **privés**. En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il vaut mieux les rendre **publics**.

Classe

Visibilité des membres d'une classe

La visibilité des attributs et méthodes d'une classe est définie dans l'interface de la classe grâce aux mots-clés **public**, **private** ou **protected** qui permettent de préciser leurs types accès.

- **public**: autorise l'accès pour tous. Exemple: le constructeur ainsi que la méthode `toString()` de l'exemple précédent peuvent être utilisés partout sur une instance de **personne**.
- **private**: restreint l'accès aux seuls corps des méthodes de cette classe. Sur l'exemple précédent, les attributs privés `p_prenom` et `p_nom` ne sont accessibles sur une instance de **personne** que dans les corps des méthodes de la classe. Ainsi, la méthode `toString()` a le droit d'accès sur ses attributs `p_prenom` et `p_nom`. Elle aurait aussi l'accès aux attributs `p_prenom` et `p_nom` d'une autre instance de **personne**.
- **protected**: comme **private** sauf que l'accès est aussi autorisé aux corps des méthodes des classes qui héritent de cette classe.

Classe

Objet

Un objet est une instance d'une classe (c'est-à-dire, une variable dont le type est une classe). Un objet occupe donc de l'espace mémoire. Il peut être alloué :

- **statiquement**: dans ce cas, on met le nom de la classe suivi du nom de l'objet et éventuellement suivi par des arguments d'appel donnés à un constructeur de la classe.

Exemple: `Personne p("Nabi" , "Barro");`

- ou **dynamiquement**: dans ce cas, un pointeur sur la zone mémoire où l'objet a été alloué est retourné. Lorsqu'on n'en a plus besoin, on le libère avec l'opérateur *delete*.

Exemple:

```
personne * p = new personne("Nabi" , "Barro");  
p->toString();  
delete p;
```

Classe

Exemple complet

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

```
// déclaration de la classe
```

```
class personne
{
    public:
    personne(string, string);
    void toString();

    private:
    string p_nom;
    string p_prenom;
};
```

```
// définition des membres publics
```

```
// constructeur
```

```
personne::personne(string prenom, string nom){
    p_prenom = prenom;
    p_nom = nom;
}
```

```
// une méthode
```

```
void personne::toString(){
    cout << "Je m'appel" << p_prenom << " " << p_nom << endl;
}
```

statiquement

```
// Utilisation de la classe personne
```

```
void main(){
    // appel implicite du constructeur
    personne p("Nabi" , "Barro");
    p.toString();
}
```

dynamiquement

```
// Utilisation de la classe personne
```

```
void main(){
    // appel implicite du constructeur
    personne *p = new personne("Nabi" , "Barro");
    p->toString();
    delete p;
}
```

La POO

- ❑ Notion de Classe
- ❑ **Propriétés des fonctions membres**
- ❑ Constructeurs, destructeurs et initialisation d'objet
- ❑ Héritage simple
- ❑ Héritage multiple
- ❑ fonctions virtuelles et le polymorphisme

Propriétés des fonctions membres

Précédemment, nous avons vu les concepts fondamentaux de la notion de Classe. Ici, nous allons étudier en profondeur les fonctions membres et les quelques possibilités offertes par C++.

□ **Surdéfinition des fonctions membres**

Ces possibilités sont déjà offertes pour les fonctions (indépendantes). Il s'agit simplement d'une généralisation aux fonctions membre. Seulement, ici, il faut tenir compte de la visibilité (privée ou publique).

Exemple:

```
class personne
{
    public:
        personne();           // constructeur (sans arguments)
        personne(string);     // constructeur (avec 1 argument)
        personne(string, string); // constructeur (avec 2 arguments)
        void toString();      // fonction membre (sans arguments)
        void toString(string); // fonction membre (avec 1 argument)

        private:
            string p_nom;
            string p_prenom;
};
```

Propriétés des fonctions membres

□ Arguments par défaut

Comme pour les fonctions indépendantes, les fonctions membres peuvent disposer d'arguments par défaut.

Exemple:

```
class personne
{
    public:
        personne();           // constructeur (sans arguments)
        personne(string);     // constructeur (avec 1 argument)
        personne(string, string); // constructeur (avec 2 arguments)
        void toString(string s = "Bonjour"); // fonction membre (avec 1 argument par défaut)

    private:
        string p_nom;
        string p_prenom;
};
```

définition

```
// la méthode
void personne::toString(string s){
    cout << s << p_prenom << " " << p_nom << endl;
}
```

Propriétés des fonctions membres

□ Fonctions membres en ligne

C++ permet la définition de fonctions en ligne pour accroître l'efficacité d'un programme dans le cas de fonctions courtes. Cette propriété s'applique aussi aux fonctions membres avec quelques nuances en ce qui concerne sa mise en œuvre.

Deux cas de figure s'offrent à nous pour rendre en ligne une fonction membre.

- On peut fournir directement la définition de la fonction dans la déclaration. Dans ce cas, le qualificatif *inline* n'a pas à être utilisé;

```
class personne
{
    public:
        personne() {p_nom = ""; p_prenom = ""; }           // constructeur 1 « en ligne »
        personne(string np) {p_prenom = p_prenom= np; }    // constructeur 2 « en ligne »
        personne(string n, string p) {p_nom = n; p_prenom = p; } // constructeur 3 « en ligne »
        void toString(string s= "Bonjour");

    private:
        string p_nom;
        string p_prenom;
};
```

définition

```
// la méthode
void personne::toString(string s){
    cout << s << p_prenom << " " << p_nom << endl;
}
```

Propriétés des fonctions membres

□ Fonctions membres en ligne

- On peut aussi procéder comme pour une fonction ordinaire en fournissant une définition en dehors de la déclaration de la classe. Dans ce cas, le qualificatif *inline* doit apparaître à la fois devant la déclaration et devant l'entête;

```
class personne
{
    public:
        inline personne() ;           // constructeur « en ligne »
        ...

    private:
        string p_nom;
        string p_prenom;
};

int main()
{
    personne p ; // appel du constructeur
    p.toString(" Hello") ;
}
```

définition

// le constructeur en ligne
inline personne::personne() {p_nom = "" ; p_prenom = "" ; }

Propriétés des fonctions membres

□ Transmission d'objets en argument à une fonction membre

une fonction membre peut recevoir un ou plusieurs arguments du type de sa classe. Par exemple, supposez que nous souhaitons, au sein d'une classe `personne`, introduire une fonction membre nommée `coïncide`, chargée de voir si deux personnes ont le même nom et prénom.

```
class personne
{
    public:
        personne() {p_nom = ""; p_prenom = ""; }
        ...
        bool coincide(personne);

    private:
        string p_nom;
        string p_prenom;
};

int main()
{
    personne p1, p2 ;
    ....
    cout<< " p1 et p2 ont elles le même noms et le même prenom ? "<<p1.coïncide(p2);
}
```

définition

```
bool personne::coïncide(personne p)
{
    return ((p.p_nom == p_nom )&& (p.p_prenom == p_prenom));
}
```

Propriétés des fonctions membres

□ Mode de transmission des objets en argument

Sur l'exemple précédent, la personne p2 à été transmis par valeur à la fonction membre **coïncide**. Il serait donc possible de prévoir le mode de transmettre par adresse plutôt que la valeur, ou de mettre en place le mode de transmission par référence.

□ Transmission par adresse d'un objet

Comme pour les fonctions ordinaires, le principe reste le même. Voici une adaptation de la fonction membre **coïncide**.

```
class personne {  
    public:  
        personne() {p_nom = ""; p_prenom = "";}  
        ...  
        bool coincide(personne *);  
  
    private:  
        string p_nom;  
        string p_prenom;  
};  
  
int main(){  
    personne p1, p2;  
    ....  
    cout<< " p1 et p2 sont elles des amies ? "<<p1.coïncide(&p2);  
}
```

définition

```
bool personne::coïncide(personne * p)  
{  
    return ((p->p_nom == p_nom) && (p->p_prenom == p_prenom));  
}
```

Propriétés des fonctions membres

❑ Transmission par référence d'un objet

Comme nous l'avons vu, l'emploi des références va permettre de mettre en place une transmission par adresse, sans avoir à le gérer soi-même. Voici une adaptation de la fonction membre **coincide**.

```
class personne {  
    public:  
        personne() {p_nom = ""; p_prenom = "";}  
        ...  
        bool coincide(personne &);  
  
    private:  
        string p_nom;  
        string p_prenom;  
};  
  
int main(){  
    personne p1, p2 ;  
    ....  
    cout<< " p1 et p2 sont elles des amies ? "<<p1.coincide(p2);  
}
```

définition

```
bool personne::coincide(personne & p)  
{  
    return ((p.p_nom == p_nom) && (p.p_prenom == p_prenom));  
}
```

Propriétés des fonctions membres

■ Une fonction renvoyant un objet

Les propriétés qui s'applique sur les arguments d'une fonction membre s'applique également à sa valeur de retour qui peut être un objet. Cette objet pourra être du même type que la classe(*auquel cas la fonction aura accès à ses membres privés*) ou bien un type différent de la classe(*auquel cas la fonction n'aura accès qu'à ses membres publics*).

- La transmission par valeur suscite la même remarque que précédemment : par défaut, elle se fait par simple recopie de l'objet.
- En revanche, la transmission par adresse ou la transmission par référence risquent de poser un problème qui n'existait pas pour les arguments. *Ici, il va falloir éviter que l'objet de retour soit un objet local à la fonction car l'emplacement de ce dernier sera libéré dès la sortie de celle-ci et la fonction appelante récupérera l'adresse d'un objet qui n'existe plus.*

définition

```
personne::homonyme()  
{  
    personne p;  
    p.p_nom = p_nom;  
    p.p_prenom = p_prenom;  
    return p;  
}
```

Il est donc déconseillé d'en prévoir une transmission par référence, en utilisant cet en-tête :

```
personne & personne::homonyme ( )
```

Propriétés des fonctions membres

■ Autoréférence

En C++, nous avons le mot clé **this** utilisable uniquement au sein d'une fonction membre et désignant un pointeur sur l'objet l'ayant appelée.

- Si on reprenant l'exemple avec la fonction membre **coincide**, on pourra le réécrire comme suit:

```
bool personne::coincide(personne * p)
{
    return ((this->p_nom == p->p_nom ) && (this->p_prenom == p.p_prenom));
}
```

- Il est applicable aussi dans un constructeur initialisant les membres données. exemple:

```
class personne {
public:
    personne(string n, string p) {this->p_nom = n ; this->p_prenom = p ;}
    ...

private:
    string p_nom;
    string p_prenom;
};
```

Propriétés des fonctions membres

❑ Fonctions membres statiques

C++ permet aussi de définir des fonctions membre statiques avec le mot clé *static*. Ces fonctions membres ont un rôle totalement indépendant d'un quelconque objet. On pourra les utiliser, par exemple, pour agir sur des membres données statiques.

```
class personne {  
    public:  
        personne() {p_nom = " "; p_prenom = " ";}  
        ...  
        static void toString();  
};
```

```
private:  
    static int age;  
    string p_nom;  
    string p_prenom;  
};  
  
int personne::age= 25;  
int main(){  
    personne p;  
    personne:: toString ();  
}
```

définition

```
void personne:: toString ()  
{  
    cout << "Elle a" << age<< " ans " ;  
}
```

Propriétés des fonctions membres

❑ Fonctions membres constantes

Le concept de constance des variables s'étend aux classes, ce qui signifie qu'on peut définir des objets constants. Dans ce cas, il va falloir préciser parmi les fonctions membres, lesquelles sont autorisés à opérer sur des objets constants, sinon le compilateur rejettera la requête. Ces fonctions membres sont accompagnés du mot cle *const*.

```
class personne {  
    public:  
        personne() {p_nom = ""; p_prenom = "";}  
        void toString() const;  
        void affiche();  
    private:  
        static int age;  
        string p_nom;  
        string p_prenom;  
};
```

```
int main(){  
    personne p1;  
    const personne p2;  
    p1.affiche();    // OK  
    p1.toString (); // OK  
    p2.affiche();    // Incorrect  
    p2.toString (); // OK  
}
```

Propriétés des fonctions membres

■ Les membres mutables

Il est impossible qu'une fonction membre constante puisse modifier les valeurs des membres non statiques. Pour que cela soit possible, la donnée membre doit être précédée du mot clé mutable pour désigner qu'elle est modifiable même par des fonctions membres constantes.

```
class personne {  
    public:  
        personne() {p_nom = ""; p_prenom = "";}  
        void toString() const {age=25;}  
        void affiche();  
    private:  
        mutable int age;  
        string p_nom;  
        string p_prenom;  
};
```


Propriétés des fonctions membres

❑ Les getters et les setteurs

En se rappelant des étiquettes **public**, **private** ou **protected** indiquant la visibilité des fonctions et données membres ou précisant si celles-ci sont accessibles à partir d'autres classes ou non... Et bien cette notion est appelée **encapsulation** des données et fonctions membres, qui est une notion essentiels du concept « orienté objet »

Ainsi, des données membres portant l'étiquette **private** ne peuvent pas être manipulées directement par les fonctions membres des autres classes.

Pour que ces dernières soient manipulables, il faut définir :

- ❑ des fonctions membres **publique** appelées **accesseurs** ou **getter** en anglais, permettant d'accéder aux données membres privées en vue de lire leur contenu;
- ❑ des fonctions membres **publique** appelées **mutateurs** ou **setter** en anglais permettant, de modifier les données membres privées en vue de changer leur valeur.

Propriétés des fonctions membres

Les getters et les setteurs

□ Getters

Un Getter ou Accesseur est donc une fonction membre permettant de récupérer le contenu d'une donnée membre étiquetée **private**. Pour sa définition:

- Son type de retour doit correspondre au type de la variable à renvoyer;
- Il ne doit pas naturellement posséder d'arguments;
- Son nom doit conventionnellement précéder par le préfixe **Get** afin de faire ressortir sa fonction première.

Syntaxe:

```
class NomClasse{  
    private :  
        TypeVariable NomVariable;  
    public :  
        TypeVariable GetVariable();  
};  
TypeVariable NomClasse::GetVariable(){  
    return NomVariable;  
}
```

Exemple:

```
class personne  
{  
    public:  
        personne(string, string);  
        string GetNom();  
    private:  
        string nom;  
};  
  
string personne::GetNom(){  
    return nom;  
}
```

Propriétés des fonctions membres

Les getters et les setteurs

❑ Setters

Un Setter ou mutateur est donc une fonction membre permettant de **modifier** le contenu d'une donnée membre étiquetée **private**. Pour sa définition:

- ❑ Il doit prendre en paramètre, la valeur (de même type) à assigner à la donnée membre;
- ❑ C'est un void, Il ne doit donc pas renvoyer de valeur;
- ❑ Son nom doit conventionnellement précéder par le préfixe **Set** afin de faire ressortir sa fonction première.

Syntaxe:

```
class NomClasse{
    private :
        TypeVariable NomVariable;
    public :
        void SetVariable(TypeVariable);
};
NomClasse::SetVariable(TypeVariable NomVar){
    NomVariable=NomVar;
}
```

Exemple:

```
class personne
{
    public:
        personne(string, string);
        void SetNom(string nom);
    private:
        string _nom;
};

personne::SetNom(string nom){
    _nom=nom;
}
```

Propriétés des fonctions membres

Les fonctions amies

□ Les fonctions amies

Avec l'encapsulation, une fonction non membre ne peut pas accéder aux données privées ou protégées d'un objet. Heureusement, en c++, il existe un mécanisme permettant d'accéder aux données privées ou protégées à partir de fonctions non membres: la notion de fonction et/ou de classe amie.

- Une **fonction indépendante amie** est une fonction normale déclaré dans une classe avec le mot clé **friend** et pouvant accéder aux données étiquetée **private** de la classe.

Syntaxe:

```
class NomClasse{
    public:
        friend typeDeRetour NomFonction(paramètres);
    private:
        TypeVariable NomVariable;
};

typeDeRetour NomFonction(paramètres){
    // instructions
}
```

- A sa définition le mot clé **friend** n'est utilisé.

Exemple:

```
#include <iostream>
using namespace std;
class personne
{
    public:
        personne(){nom="NoName"}
        friend void toString(personne&);
    private:
        string nom;
};

void toString(personne& p){
    cout << p.nom<< endl;
}

int main() {
    personne p;
    toString(p);
    return 0;
}
```

Propriétés des fonctions membres

Les fonctions amies

- On peut également avoir une **fonction membre d'une classe**, qui est aussi **amie dans une autre classe**.

Exemple:

```
#include <iostream>
using namespace std;
class B;
class A
{
    public:
        void toString(B&);
};
class B
{
    public:
        B(){value=5;}
        friend void A::toString(B& b);
    private:
        int value;
};
```

```
// suite
```

```
void A::toString(B& b){
    cout << b.value << endl;
}

int main() {
    A a;
    B b;
    a.toString(b);

    return 0;
}
```

Propriétés des fonctions membres

Les classes amies

□ Les classes amies

Une classe amie comme une fonction amie peut accéder aux membres privés et protégés d'une autre classe dans laquelle elle est déclarée comme ami.

Exemple:

```
#include <iostream>
using namespace std;
class B;
class A
{
    public:
        void toString(B&);
};
class B
{
    public:
        B(){value=5;}
        friend class A;
    private:
        int value;
};
```

// suite

```
void A::toString(B& b){
    cout << b.value << endl;
}

int main() {
    A a;
    B b;
    a.toString(b);

    return 0;
}
```

La POO

- ❑ Notion de Classe
- ❑ Fonctions membres
- ❑ **Constructeurs, destructeurs et initialisation d'objet**
- ❑ Héritage simple
- ❑ Héritage multiple
- ❑ fonctions virtuelles et le polymorphisme

Constructeurs, destructeurs et initialisation d'objet

Précédemment, nous avons vu les quelques propriétés des fonctions membres. Ici, nous allons étudier en profondeur les constructeurs/destructeurs et l'initialisation des objets en C++.

□ **Constructeurs et destructeurs**

Les constructeurs et le destructeur sont des fonctions membres particulières des classes.

■ **Constructeurs :**

Un constructeur permet de définir un ou des comportements particuliers lors de l'instanciation d'une classe. Ils permettent aussi d'initialiser correctement un nouvel objet. Ils portent tous le même nom que la classe avec des particularités sur le nombre et le type de paramètres.

Le constructeur par défaut ou constructeur sans paramètre qui est appelé lors de l'instanciation d'un objet sans argument d'appel.

Exemple: // appel du constructeur par défaut

- `Personne p1 ;` // ou bien
- `Personne* p2 = new Personne;`

Un tel constructeur est aussi appelé lors de l'instanciation d'un tableau d'objets

Exemple: // pour chaque personne du tableau, appel du constructeur par défaut

- `Personne tabP1[5];` // ou bien
- `Personne* tabP2 = new Personne[5];`

Constructeurs, destructeurs et initialisation d'objet

■ destructeurs:

Un destructeur permet la destruction ou la désallocation d'un objet. Il est définie implicitement pour toutes les classes. Il ne fait rien par défaut, mais on peut lui donner un comportement spécifique.

■ Appel des constructeurs statiquement

- ❖ Supposons qu'un objet possède qu'un seul constructeur, sa déclaration doit obligatoirement comporter les arguments correspondants.

Exemple:

```
class personne {  
    public:  
    personne(string, string);  
    ...  
    private:  
    string p_nom;  
    string p_prenom;  
};
```

- La déclaration suivante est correcte:
personne p("Nabi", "BARRO");
- Les déclarations suivantes sont incorrectes:
personne p1;
personne p2("Nabi");

Constructeurs, destructeurs et initialisation d'objet

- ❖ S'il existe plusieurs constructeurs, la déclaration doit comporter les arguments requis du constructeur appelé.

Exemple:

```
class personne {  
    public:  
    peronne( );  
    personne(string, string);  
    ...  
    private:  
    string p_nom;  
    string p_prenom;  
};
```

- ❑ Les déclarations suivantes sont correctes:

```
personne p1;  
personne p2("Nabi" , "BARRO");
```

- ❑ La déclaration suivante est incorrecte:

```
personne p("Nabi");
```

- ❖ S'il n'existe pas de constructeur, alors la déclaration suivante est acceptable:

- ❑ `personne p; // il va falloir faire attention à ne pas faire personne p() car ceci est une déclaration pour fonction.`

Constructeurs, destructeurs et initialisation d'objet

□ Appel des constructeurs dynamiquement

- ❖ Reconsidérons la class personne (*sans constructeur*) suivante:

```
class personne {  
    public:  
    void toString();  
    ...  
    private:  
    string p_nom;  
    string p_prenom;  
};
```

- Il est donc possible de créer dynamiquement un emplacement mémoire de type personne.
 - Exemple: `personne * p;`
- Puis maintenant d'affecter son adresse à **p** par:
 - `p = new personne; // ou new personne()`
- On pourra, alors accéder aux fonctions membres soit via l'opérateur `->` :
 - `p -> toString();`
- Ou en faisant comme suit:
 - `(*p).toString();`
- L'opérateur *delete* sera utilisé pour appeler le destructeur afin de libérer l'emplacement mémoire correspondant:
 - `delete p;`

Constructeurs, destructeurs et initialisation d'objet

□ Appel des constructeurs dynamiquement

- ❖ Reconsidérons la class personne (*avec constructeur*) suivante:

```
class personne {  
    public:  
    personne(string, string);  
    void toString();  
    ...  
    private:  
    string p_nom;  
    string p_prenom;  
};
```

- Il est donc possible de créer dynamiquement un emplacement mémoire de type personne.
 - Exemple: `personne * p;`
- ici, pour que *new* puisse appeler un constructeur disposant d'arguments, il est nécessaire qu'il dispose des informations correspondantes.
 - `p = new personne("Nabi" , "BARRO");`

Constructeurs, destructeurs et initialisation d'objet

□ Initialisation d'objet

En C++, munie d'un initialiseur, on peut fournir, sous une forme peu naturelle, des arguments pour un constructeur. Le langage n'impose aucune restriction sur le type de l'initialiseur qui pourra, éventuellement, être du même type que l'objet initialisé.

Supposons la classe `personne` avec un constructeur:

```
class personne {  
    public:  
    personne();  
    personne(string prenom){p_prenom=prenom; p_nom="BARRO";}  
    personne (string, string);  
    ...  
    private:  
    string p_nom;  
    string p_prenom;  
};
```

- Les déclarations et initialisations suivantes sont valables et équivalentes:
 - `personne p("zeynab");`
 - `personne p= "zeynab";`
- on peut aussi initialiser un objet avec un autre de même type:
 - `personne p1;`
 - `personne p2=p1;`
 - `personne p3(p1);`
- Pour le cas d'un constructeur à deux ou plusieurs arguments, l'initialisation prévue est la suivante:
 - `Personne p("Souleymane", "BARRO");`

Notion de classe – fonctions membre – constructeurs et destructeurs

EXERCICES D'APPLICATIONS

□ Application 28:

Réaliser une classe *point* permettant de manipuler un point d'un plan.

On prévoira :

- un constructeur recevant en arguments les coordonnées (*float*) d'un point ;
- une fonction membre *deplace* effectuant une translation définie par ses deux arguments (*float*) ;
- une fonction membre *affiche* se contentant d'afficher les coordonnées cartésiennes du point.

Les coordonnées du point seront des membres donnée privés.

On écrira séparément :

- un fichier source constituant la *déclaration* de la classe ;
- un fichier source correspondant à sa *définition*.
- un petit programme d'essai (*main*) déclarant un point, l'affichant, le déplaçant et l'affichant à nouveau.

□ Application 29:

Réaliser une classe *point*, analogue à la précédente, mais ne comportant pas de fonction *affiche*. Pour respecter le principe d'encapsulation des données, prévoir deux fonctions membre publiques (nommées *abscisse* et *ordonnée*) fournissant en retour l'abscisse et l'ordonnée d'un point. Adapter le petit programme d'essai précédent pour qu'il fonctionne avec cette nouvelle classe.

La POO

- ❑ Notion de Classe
- ❑ Fonctions membres
- ❑ Constructeurs, destructeurs et initialisation d'objet
- ❑ **Héritage simple**
- ❑ Héritage multiple
- ❑ fonctions virtuelles et le polymorphisme

Héritage simple

Le principe est d'utiliser la déclaration d'une classe (appelée classe de base ou classe parente) comme base pour déclarer une seconde classe (appelée classe dérivée). La classe dérivée héritera de tous les membres (données et fonctions) publique de la classe parente.

▣ **Exemple de déclaration de la classe parente:**

```
class ClasseParente {  
    public:  
    void fonctionParente(string stringParente);  
    protected:  
    int valeurParente;  
  
};
```

▣ **Exemple de déclaration de la classe dérivée de la classe parente:**

```
// héritage public  
class ClasseDerivee : public ClasseParente {  
    public:  
    void fonctionDerivee(string stringDerivee);  
    protected:  
    int valeurDerivee;  
  
};
```

Un objet de la classe Dérivée possède alors ses propres données et fonctions-membres, plus les données-membres et fonctions-membres héritées de la classe parente.

Héritage simple

Après avoir créer la classe dérivée, il est donc possible de déclarer des objets de type `ClasseDerivee` de manière usuelle :

`ClasseDerivee` c, d ;

Chaque objet aura accès :

- aux méthodes publiques de la classe dérivée;
- aux méthodes publiques de la classe de base.

Exemple:

```
#include <iostream>
#include "personne.hpp"
using namespace std ;
```

```
class toubab: public personne
{
    string couleur;
    public :
        void teint(string c="blanche") { couleur = c ; }
};
```

```
int main()
{
    toubab t ;
    t.defn("Natacha","Lacroix") ;
    t.teint("Marron") ;
    t.toString () ;
}
```

Héritage simple

Utilisation des membres de la classe de base dans une classe dérivée

L'exemple précédent sur la classe `toubab` ne nous renseigne pas sur la couleur de peau de la personne après appel de la méthode `toString()`.

Pour remédier à cela, deux méthodes sont possibles:

- La première consiste à écrire une nouvelle fonction membre publique dans `toubab`, qui est censée afficher à la fois le nom, le prénom ainsi que la couleur de peau.

```
void toubab::affiche()  
{  
    cout << "Je m'appel" << p_prenom << " " << p_nom << endl;  
    cout << "je suis de teint " << couleur << endl;  
}
```

- Mais cette méthode signifierai que `affiche()` a accès aux membres privés de `personne` (ce qui est contraire au principe d'encapsulation). Par conséquent: **"une méthode d'une classe dérivée n'a pas accès aux membres privés de sa classe de base"**.

Héritage simple

Utilisation des membres de la classe de base dans une classe dérivée

- **Par contre**, comme une méthode de la classe dérivée a accès aux membres publics de sa classe de base, la fonction membre `affiche()` de la classe `toubab` pourra alors faire appel à la fonction membre `toString()` de la classe `personne`.

```
void toubab::affiche()
{
    toString();
    cout<< "je suis de teint " << couleur <<endl ;
}
```

- **D'une manière générale**, on aurait pu définir une fonction `init(<paramètres>)` permettant d'initialiser les trois données membres de `toubab`:

```
void toubab::init(string prenom, string nom, string clr)
{
    defn(prenom, nom);
    couleur = clr;
}
```

Héritage simple

exemple

Exemple complet:

```
#include <iostream>
#include <string>
#include "personne.hpp"
using namespace std ;

class toubab: public personne
{
    string couleur;
    public :
        void teint(string c="blanc") { couleur = c ; }
        void affiche();
        void init(string, string, string);
};

void toubab::affiche()
{
    toString();
    cout<< "je suis de teint " << couleur <<endl ;
}

void toubab::init(string prenom, string nom, string clr)
{
    defn(prenom, nom);
    couleur = clr;
}
```

```
int main()
{
    toubab t ;
    t.init("Natacha","Lacroix", "Blanc") ;
    t.affiche();
}
```

Héritage simple

La redéfinition des membres d'une classe dérivée

- Dans la classe dérivée, nous avons défini une méthode `affiche()` qui fait pratiquement la même chose que la méthode `toString()` de la classe de base. On aurait pu seulement redéfinir la méthode `toString()` dans la classe dérivée, mais dans ce cas, on ne pourra plus appeler à l'intérieur de celle-ci, la méthode `toString()` de la classe parente comme on a l'habitude de le faire: on fera `personne::toString()` pour véritablement localiser la bonne méthode.

Exemple:

```
#include <iostream>
#include "personne.hpp"
using namespace std ;

class toubab: public personne
{
    string couleur;
public :
    void teint(string c="blanc") { couleur = c ; }
    void toString();
    void init(string, string, string);
};

void toubab::toString()
{
    personne::toString();
    cout<< "je suis de teint " << couleur <<endl ;
}
```

```
int main()
{
    toubab t ;
    t.init("Natacha","Lacroix", "Blanche") ;
    t.toString(); // définit dans toubab
    t.personne::toString() // définit dans personne
}
```

Héritage simple

Constructeurs, destructeurs et l'héritage

Quand un objet est créé, si cet objet appartient à une classe dérivée, le constructeur de la classe parente est d'abord appelé. Quand un objet est détruit, si cet objet appartient à une classe dérivée, le destructeur de la classe parente est appelé après.

Voici un exemple:

```
#include<iostream.h>
class GrandPere
{    // données membres
    public:
        GrandPere();
        ~GrandPere();
};
class Pere : public GrandPere
{    // données membres
    public:
        Pere();
        ~Pere();
};
class Fils : public Pere
{    // données membres
    public:
        Fils();
        ~Fils();
};
```

```
void main()
{
    Fils *junior;
    junior = new Fils;
    // appels successifs des constructeurs
    // de GrandPere, Pere et Fils
    .....
    delete junior;
    // appels successifs des destructeurs
    // de Fils, Pere et GrandPere
}
```

Héritage simple

Constructeurs, destructeurs et l'héritage

- Il est possible d'utiliser un constructeur de la classe de base pour définir un constructeur de la classe dérivée (**mécanisme de transmission d'informations entre constructeurs**):

Exemple:

```
#include<iostream>
class Rectangle{
public:
    Rectangle(int lo, int la);
    void toString();
protected:
    int longueur, largeur;
};
Rectangle::Rectangle(int lo, int la){
    longueur = lo;
    largeur = la;
}
void Rectangle::toString(){
    cout <<"surface= "<<longueur*largeur<<endl;
}

-----
class Carre : public Rectangle {
public:
    Carre(int cote);
};
Carre::Carre(int cote) : Rectangle(cote, cote) {
}
```

```
void main()
{
    Carre *monCarre;
    Rectangle *monRectangle;
    monCarre = new Carre(5);
    monCarre->toString(); // affiche 25
    ...
    monRectangle = new Rectangle(5, 10);
    monRectangle->toString(); // affiche 50
}
```

Héritage

Le constructeur de copie et l'héritage

Rappel: Un **constructeur de copie** est généralement utilisé lorsqu'il s'agit d'initialiser un objet par un autre de même type ou lors de la transmission d'un objet en paramètre ou en retour à une fonction.

En supposant que la classe B dérive de la classe A, 2 situations s'offrent alors à nous:

- ❑ Soit B ne définit pas de constructeur de copie. Dans ce cas, le constructeur de copie de A sera appelé pour les membres données correspondants et le constructeur de copie par défaut de B (on prévoira des informations pour le constructeur de A).
 - ❑ Exemple: `B (B & b) : A (...)`
- ❑ Soit B définit un constructeur de copie (en supposant que A aussi à définit un constructeur de copie). Dans ce cas, on pourra effectuer une conversion implicite de la classe B dans la classe A.
 - ❑ Exemple: `B (B & b) : A (b){...}`

Héritage

Le constructeur de recopie et l'héritage - Exemple

```
// inclure les bibliothèques requises
```

```
using namespace std ;
```

```
class personne {
public:
    personne(string p, string n){
        p_prenom = p; p_nom = n;
        cout<<"Prenom"<<p_prenom<<"Nom"<<p_nom<<endl;
    }
    personne(personne & p){
        p_prenom = p.p_prenom; p_nom = p.p_nom;
        cout<<p_prenom<<" "<<p_nom<<endl;
    }
private:
    string p_nom; string p_prenom;
};

class toubab : public personne {
    string couleur;
public :
    toubab(string p, string n, string c):personne(p, n){
        couleur = c;
        cout<< "Couleur de peau:" << couleur <<endl ;
    }
    toubab(toubab & t):personne(t){
        couleur = t.couleur;
        cout<< "teint:" << couleur <<endl ;
    }
};
```

```
void fct (toubab t){
    cout << "Fin !" <<endl;
}
```

```
int main()
{
    void fct(toubab);
    toubab t("Natacha", "Lacroix", "Blanche");
    fct(t);
}
```

- Prenom Natacha Nom Lacroix
- Couleur de peau: Blanche
- Natacha Lacroix
- teint: Blanche
- Fin !

Héritage simple

EXERCICES D'APPLICATIONS

- Application 30:
- Application 31:

La POO

- ❑ Notion de Classe
- ❑ Fonctions membres
- ❑ Constructeurs, destructeurs et initialisation d'objet
- ❑ Héritage simple
- ❑ **Héritage multiple**
- ❑ fonctions virtuelles et le polymorphisme

Héritage multiple

Il est possible de faire dériver une classe de plusieurs autres classes simultanément (**héritage multiple**). Exemple: une classe C peut hériter de la classe A et de la classe B. Une instance de la classe C possèdera alors à la fois les données et fonctions-membres de la classe A et celles de la classe B.

Exemple:

```
class A {  
.....  
};  
  
class B {  
.....  
};  
  
class C : public A, public B {  
.....  
};
```

- À la création de C, les constructeurs des classes parentes sont appelés: celui de A, puis celui de B.
- À la destruction, les destructeurs des classes parentes sont appelés, celui de B, puis celui de A.
- il peut arriver que des données ou fonctions-membres des classes A et B aient le même nom. Dans ce cas, il faut utiliser l'opérateur de porté comme suit: A::var ou B::var pour faire la distinction;

Héritage multiple

En guise d'illustration, nous allons recréer la classe `toubab`, dérivant de la classe `personne` et de la classe `couleur`.

- Soit la classe `personne` et la classe `couleur`:

```
class personne {
public:
    personne();
    ~personne(){cout<<"End personne!"<<endl;}
    personne(string p, string n){p_prenom = p; p_nom = n; }
    p_personne(string p, string n){p_prenom = p; p_nom = n; }
    void toString(){
        cout<<"Prenom"<<p_prenom<<"Nom"<<p_nom<<endl;
    }
private:
    string p_nom; string p_prenom;
};
```

```
class couleur {
public:
    c_couleur(string c){ clr = c; }
    ~couleur(){cout<<"End couleur!"<<endl;}
    void toString(){
        cout<<"teint"<<clr<<endl;
    }
private:
    string clr;
};
```

- La classe `toubab` sera alors déclarer comme suit:

```
class toubab : public | private | protected personne, public | private | protected couleur
{
    // traitement
};
```

Héritage multiple

En guise d'illustration, nous allons recréer la casse toubab, dérivant de la classe `personne` et de la classe `couleur`.

- Soit la classe `personne` et la classe `couleur`:

```
class personne {  
    public:  
        personne();  
        ~personne(){cout<<"End personne!"<<endl;}  
        personne(string p, string n){p_prenom = p; p_nom = n; }  
        void p_personne(string p, string n)  
        {p_prenom = p; p_nom = n; }  
        void toString(){  
            cout<<"Prenom"<<p_prenom<<"Nom"<<p_nom<<endl;  
        }  
  
    private:  
        string p_nom; string p_prenom;  
};
```

```
class couleur {  
    public:  
        couleur(string c){ clr = n; }  
        ~couleur(){cout<<"End couleur!"<<endl;}  
        void c_couleur(string c){ clr = n; }  
        void toString(){  
            cout<<"teint"<<clr<<endl;  
        }  
  
    private:  
        string clr;  
};
```

- La classe `toubab` sera alors déclarer comme suit:

```
class toubab : public personne, public couleur  
{  
    // traitement  
};
```

Héritage multiple

- Dans `toubab`, nous avons décidé de redéfinir la fonction `toString()` en se basant sur les fonctions `toString()` se trouvant successivement dans `personne` et dans `couleur`. **Rappelons que Lorsque plusieurs fonctions membres portent le même nom dans différentes classes, on peut lever l'ambiguïté en employant l'opérateur de résolution de portée.** Ainsi, nous avons:

```
void toubab::toString(){
    personne::toString();
    couleur::toString();
};
```

- Classiquement, un objet de type `toubab` pourra faire appel aux fonctions membres des classes de base `personne` et `couleur` (on pourra se servir de l'opérateur de portée pour lever les éventuelles ambiguïtés).
 - Si on a: `toubab p("Natacha", "Lacroix", "Blanc");`
 - Alors on pourra faire appel à la fonction `toString()` se situant dans `toubab` en faisant `p.toString()`; à celle se situant dans `personne` par `p.personne::toString()`; et puis à celle se situant dans `couleur` par `p.couleur::toString()`.

Héritage multiple

□ La suite de exemple:

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

...
class toubab : public personne, public couleur
{
    public :
        toubab(string p, string n, string c){
            personne::ppersonne(p, n);
            couleur::ccouleur(c);
        }
        ~toubab(){ cout<< "End toubab"<<endl; }
        void toString ();
};

void toubab::toString (){
    personne::toString();
    couleur::toString();
};
```

```
int main()
{
    toubab p("Natacha", "Lacroix", "Blanc");
    cout<< "toString dans toubab"<<endl;
    p.toString();
    cout<< "toString dans personne"<<endl;
    p.personne::toString();
    cout<< "toString dans couleur"<<endl;
    p.couleur::toString()
}
```


Héritage multiple

Problème de conflits

- Dans le cas d'un héritage multiple, le problème de doublon peut facilement se poser.
 - Supposons que nous avons une classe D qui hérite à la fois de la classe B et de la classe C et que ces dernières héritent toutes les deux de la classe A. leurs déclarations donnent la configuration suivante:

```
class A
{
    ....
};
class B : public A {....} ;
class C : public A {....} ;
class D : public B, public C
{
    ....
};
```

- Il y'aura une redondance des membres données de A dans tous les objets de type D;
- Mais maintenant, si l'on souhaite que cela arrive, on pourra toujours faire la distinction en utilisant l'opérateur de portée .

Héritage multiple

Problème de conflits

- Il y'a toujours un mécanisme permettant de travailler avec un seul motif de A dans la classe de D: il suffit de déclarer dans les classes de B et de C que la classe A est virtuelle avec le mot-clé *virtual*.

```
class B : public virtual A {.....} ;  
class C : public virtual A {.....} ;
```

- Déclarer la classe A comme virtuelle dans B et C n'a pas d'effet sur elles mais sur leur descendance (ici, D);

```
class D : public B, public C {.....} ;
```
- Si A est déclarée comme virtuelle dans B, alors A sera introduite qu'une seule fois dans les descendances de C.

Héritage multiple

Appels des constructeurs et des destructeurs

- Lorsque **A** a été déclarée virtuelle dans **B** et **C**, le choix des informations à fournir au constructeur de **A** a lieu dans **D** et non dans **B** ou **C**. On spécifie dans le constructeur de **D**, les informations destinées à **A**.
- Exemple:
`D(string p, string n, string c) : B(string p, string n, string c), A(string p, string n)`
- **A** doit absolument disposer d'un constructeur par défaut:
- Lorsqu'on crée un objet de type **D**, le constructeur de **A** est appelé en premier, puis celui de **B** ensuite celui de **C** et en fin celui de **D**.

Héritage multiple

Appels des constructeurs et des destructeurs - exemple

- Nous définissons d'abord trois classe: la classe `personne`, la classe `couleur` et la classe `ethnie`.

```
class personne {
public:
    personne();
    personne(string p, string n){p_prenom = p; p_nom = n; }
    void ppersonne(string p, string n){p_prenom = p; p_nom = n; }
    ~personne(){cout<<"End personne!"<<endl;}
    void toString(){
        cout<<"Prenom: "<<p_prenom<<"Nom: "<<p_nom<<endl;
    }

private:
    string p_nom;
    string p_prenom;
};
```

```
class ethnie{
public:
    ethnie();
    ~ethnie(){cout<<"End ethnie!"<<endl;}
    ethnie(string e){e_nom = e; }
    void p_ethnie(string e){e_nom = e; }
    void toString(){
        cout<<"Je suis de l'ethnie "<<e_nom<<endl;
    }

private:
    string e_nom;
};
```

```
class couleur {
public:
    couleur();
    couleur(string c){
        clr = c;
    }
    void ccouleur(string c){
        clr = c;
    }
    ~couleur(){
        cout<<"End couleur!"<<endl;
    }
    void toString(){
        cout<<" de teint "<<clr<<endl;
    }

private:
    string clr;
};
```

Héritage multiple

Appels des constructeurs et des destructeurs - exemple

- Ensuite, on définit une classe classe: une classe **toubab** fille de **personne** (virtuelle) et de **couleur** et une autre classe **negro** fille de **personne** (virtuelle) et de **ethnie**

```
class toubab : public virtual personne, public couleur
{
    public :
        toubab(string p, string n, string c) : couleur(c){}
        ~toubab(){
            cout<< "End toubab"<<endl;
        }
        void toString (){
            personne::toString();
            couleur::toString();
        };
};
```

```
class negro : public virtual personne, public ethnie
{
    public :
        negro(string p, string n, string e) : ethnie(e){}
        ~negro(){
            cout<< "End negro !"<<endl;
        }
        void toString (){
            personne::toString();
            ethnie::toString();
        };
};
```

Héritage multiple

Appels des constructeurs et des destructeurs - exemple

- Et enfin une classe métis fille de toubab et de negro.

```
class metis : public toubab, public negro
{
public:
    metis(string p, string n, string c, string e) : personne(p, n), toubab(p, n, c), negro(p, n, e){}
    void toString(){
        personne::toString();
        negro::toString();
        cout<<"d'une part, et ";
        toubab::toString();
        cout<<"d'autre part! "<<endl;
    };
};
```

```
int main(){
    toubab p("Natacha", "Lacroix", "Blanc");
    cout<< "toString dans toubab"<<endl;
    p.toString();

    negro n("Soundiata", "Keita", "Manding");
    cout<< "toString dans Nègro"<<endl;
    n.toString();

    metis m ("Natacha", "Keita", "Marron", "Manding");
    cout<< "toString dans metis"<<endl;
    m.toString();
}
```

Héritage multiple

EXERCICES D'APPLICATIONS

- Application 32:
- Application 33:

La POO

- ❑ Notion de Classe
- ❑ Fonctions membres
- ❑ Constructeurs, destructeurs et initialisation d'objet
- ❑ Héritage simple
- ❑ Héritage multiple
- ❑ **Fonctions virtuelles et polymorphisme**

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Considérons le programme suivant ou Carré dérive de rectangle:

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

class Rectangle{
public:
    void toString(){
        cout <<"je suis un rectangle"<<endl;
    }
};

class Carre : public Rectangle {
public:
    void toString(){
        cout <<"je suis un carre"<<endl;
    }
};
```

```
void main()
{
    Rectangle rect;
    Carre car;

    Rectangle *r = &rect;
    r->toString();
    r= &car;
    r->toString();

    return 0;
}
```

- ❑ La situation en est que lorsqu'on appelle la fonction `toString()` de `Carre` via l'objet pointé, c'est la fonction `toString()` de la classe `Rectangle` qui est appelée et non celle réellement pointée.
- ❑ Le problème en est que le compilateur ne connaît pas le type de l'objet réellement pointé, et se base uniquement sur le type du pointeur. **On parle donc de typage statique.**
- ❑ En C++, il est possible de faire face à cela, en permettant le **typage dynamique** de ces objets. Un tel mécanisme permettrait au compilateur de choisir à l'exécution, la fonction appropriée. **Il s'agit de la notion de Polymorphisme.**

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Supposons que l'on souhaite créer une collection d'objets de type Rectangle, en demandant `toString()` pour chacun de ces objets. Ce sera automatiquement la version correspondant à chaque forme qui sera appelée et exécutée : *on dit que `toString()` est polymorphe*.

Ce choix de la version adéquate de `toString()` sera réalisé au moment de l'exécution.

- ❑ **Règle générale**: Toute fonction-membre de la classe parente devant être redéfinie (*surchargée*) dans une classe dérivée doit être précédée du mot-clé *virtual*.

- ❑ Exemple:

```
class Rectangle{
    public:
        // fonction destinée à être surchargée
        virtual void toString();
};
```

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Exemple:

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

class Rectangle{
public:
    // fonction destinée à être surchargée
    virtual void toString();
};

void Rectangle::toString(){
    cout << "Je suis un rectangle !" <<endl;
}

class Carre : public Rectangle{
public:
    void toString();
};

void Carre::toString(){
    cout <<"Je suis un carré!"<<endl;
}
```

```
int main()
{
    Rectangle rect;
    Carre car;

    Rectangle *r= &rect;
    r->toString();
    r= &car;
    r->toString();

    return 0;
}
```

- ❑ Seule une fonction membre peut être virtuelle. Pas de fonction indépendante ou de fonction amie.
- ❑ Un constructeur ne peut pas être virtuel
- ❑ Un destructeur par contre, peut être virtuel

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

Reconsidérons la situation précédente en définissant que des constructeurs et destructeurs des classes concernées:

```
#include<iostream>
#include<string>
using namespace std;
class Rectangle{
public:
    Rectangle(int, int);
    ~Rectangle () {cout <<"Fin Rect"<<endl;}
protected:
    int longueur, largeur;
};
Rectangle::Rectangle(int lo, int la){
    longueur = lo;
    largeur = la;
}
class Carre : public Rectangle {
    int c_cote;
public:
    Carre(int cote);
    ~Carre() {cout <<"Fin Car"<<endl;}
};
Carre::Carre(int cote) : Rectangle(cote, cote) {
    c_cote=cote;
}
```

```
int main()
{
    Rectangle *r= new Carre(5);
    delete r;

    return 0;
}
```

- ❑ Dans ce scénario, c'est seulement le destructeur de Rectangle qui est appelé. Celui de Carre n'est pas appelé.
- ❑ Pour remédier à cela, le destructeur de Rectangle doit être déclaré comme virtuel :
 - ❑ `virtual ~Rectangle ();`

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

- Une fonction-membre virtuelle d'une classe est dite purement virtuelle lorsque sa déclaration est suivie de = 0.

Exemple:

```
class A {  
    public:  
        virtual void fonct() = 0;  
};
```

- Une fonction purement virtuelle n'a pas de définition dans la classe. Elle ne peut qu'être surchargée dans les classes dérivées.
- Une classe comportant au moins une fonction-membre purement virtuelle est appelée classe abstraite.
 - Aucune instance d'une classe abstraite ne peut être créée.
 - L'intérêt d'une classe abstraite est uniquement de servir de "canevas" à ses classes dérivées, en déclarant l'interface minimale commune à tous ses descendants.

Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

- Aucune instance d'une classe abstraite ne peut être créée. Exemple:

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

```
class Rectangle{
public:
    // fonction purement virtuelle destinée à être surchargée
    virtual void toString()=0;
```

```
};
```

```
class Carre : public Rectangle{
public:
    void toString();
};
```

```
void Carre::toString(){
    cout <<"Je suis un carré!"<<endl;
}
```

```
void main()
{
    Rectangle *r= new Rectangle();
    r->toString();

    Rectangle *c= new Carre();
    c->toString();

    return 0;
}
```

Error !



Fonctions virtuelles et polymorphisme

polymorphisme et classes abstraites

- Nous allons juste déclarer la fonctions `toString()` dans **A** et puis la redéfinir dans **B**. Exemple:

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

```
class Rectangle{
public:
    // fonction purement virtuelle destinée à être surchargée
    virtual void toString()=0;
};
```

```
class Carre : public Rectangle{
public:
    void toString();
};
```

```
void Carre::toString(){
    cout <<"Je suis un carré!"<<endl;
}
```

```
void main()
{
    Rectangle *c= new Carre();
    c->toString();

    return 0;
}
```

Correcte!

Fonctions virtuelles et polymorphisme

EXERCICES D'APPLICATIONS

- Application 34:
- Application 35:

La POO

- ❑ Notion de Classe
- ❑ Fonctions membres
- ❑ Constructeurs, destructeurs et initialisation d'objet
- ❑ Héritage simple
- ❑ Héritage multiple
- ❑ Fonctions virtuelles et le polymorphisme
- ❑ La gestion des exceptions
- ❑ Généralités sur la STL

La gestion des exceptions

Généralités

- ❑ Quelques fois, même si le programme est correct, il peut y survenir des risques pouvant compromettre son bon exécution.
- ❑ En C++, avec sa bibliothèque standard, les erreurs sont fournies par des codes qui constituent des valeurs de retour des différentes fonctions. En examinant ces valeurs de retour en de nombreuses points du programme, il est alors possible de capturer les éventuelles anomalies (exceptions) et puis de procéder à leurs traitements.
- ❑ **En production**, il est utile de séparer le code et la détection/correction des erreurs.

La gestion des exceptions

Généralités

- Une exception est une rupture de séquence qui peut être déclenchée par une instruction **throw**, comportant une expression quelconque dont le type (**classe** ou non) sert à identifier l'exception en tant que telle.
- Il est recommandable, pour chaque exception donnée, de proposer une classe afin de mieux représenter l'anomalie concernée.
- Exemple d'erreur avec "division par zéro":

Erreur

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

int main() {
    int x,a=5,b=0;
    x = a/b;
    cout << "x= " << x << endl;

    return 0;
}
```

La gestion des exceptions

Lever des exceptions avec « throw »

- En C++, la gestion des exceptions repose essentiellement sur *trois mots-clés* : **try**, **catch** et **throw**. Avec **try** qui définit le bloc de code test, **catch** qui définit le bloc de code à exécuter en cas de détection d'erreur et **throw** qui permet de lever une exception en cas de problème (on peut créer une erreur personnalisée à cet effet).
- **Lever des exceptions avec « throw »:**
en réutilisant l'exemple d'erreur précédent, il est possible d'identifier la nature de l'erreur. La syntaxe est la suivante: **throw expression;**

```
#include <iostream>
#include <string>

using namespace std;

int test_division(int a, int b)
{
    string x = " erreur: division par zero";
    if (b==0) {
        throw x;
    }
    return a/b;
}
```

```
int main() {
    int x,a=5,b=0;

    x = test_division(a, b);
    cout << "x= " << x << endl;

    return 0;
}

// Si une exception est levée et n'est
// interceptée nulle part, le programme se
// termine anormalement.
```

La gestion des exceptions

Interceptor/traiter une exception avec « try/catch »

- **Interceptor/traiter une exception avec « try/catch » :**

Dans catch, entre parenthèses, il est possible de spécifier le type d'exception à intercepter et puis de personnaliser le message d'erreur.

- **La syntaxe est la suivante:**

```
try {  
    // bloc d'instruction à protégé  
} catch( NomDeException e ) {  
    // bloc d'instruction pour la gestion de l'exception  
}
```

- Reprenons l'exemple précédent:

```
#include <iostream>  
#include <string>  
  
using namespace std;  
  
int test_division(int a, int b)  
{  
    string x = "erreur: division par zero";  
    if (b==0) {  
        throw x;  
    }  
    return a/b;  
}
```

```
int main() {  
    int x,a=5,b=0;  
    try{  
        x = test_division(a, b);  
        cout << "x= " << x << endl;  
    }catch(string s){  
        cout << s << endl;  
    }  
    return 0;  
}
```

La gestion des exceptions

Identification et interception d'une exception via une classe

□ Identification et interception d'une exception via une classe :

Il est possible de créer une classe pour caractériser une exception.

- En utilisant le mot clé **throw** puis l'objet correspondant à l'exception, il est possible d'identifier le type d'erreur!: (throw objet;)
- Puis de l'intercepter (**try**) et de le traiter (**catch**).

```
try{  
    // bloc d'instruction à protégé  
}catch (classe &o){  
    // Traitement de l'exception associée à la classe  
}
```

- Exemple de classe d'exception:

```
class erreur {  
    public:  
        string cause;  
        erreur(string s) : cause(s) { }  
        // Le constructeur de copie (nécessaire pour le catch):  
        erreur(const erreur &e) : cause(e.cause) { }  
};
```

La gestion des exceptions

Identification et interception d'une exception via une classe

□ Utilisation dans un main

```
#include <iostream>
#include <string>
using namespace std;
// class erreur

int test_division(int a, int b)
{
    erreur x ("division par zero !");
    if (b==0) {
        throw x;
    }
    return a/b;
}

int main(){
    int a,b;
    cout<<"a=";
    cin>>a;
    cout<<"b=";
    cin>>b;
    // ...
```

// suite

```
try{
    cout << "resultat= " << test_division(a, b) << endl;
}
catch (erreur &e)
{
    // récupération de la cause.
    cout << "Erreur, cause: " << e.cause << endl;
}
catch (...)
{
    // le gestionnaire d'exception universel
    // pour toutes autres erreurs
    cout << "Erreur inattendue !" << endl;
}

return 0;
}
```

La gestion des exceptions

standard

- ❑ C++ nous fournit un certains nombres d'exceptions standard qui sont définies dans la classe **exception** qui est la classe de base de toutes les exceptions lancées par la bibliothèque standard.
- ❑ Elle est défini comme suite:

```
class exception
{
    public:
        //Constructeur.
        exception() throw() { }
        //Destructeur.
        virtual ~exception() throw();
        // what envoie une chaîne contenant des infos sur l'erreur.
        virtual const char* what() const throw();
};
```

- ❑ **throw** indique que ces méthodes ne vont pas lancer d'exceptions

La gestion des exceptions

standard

- ❑ Il devient possible de créer sa propre classe d'exception en héritant de la classe **exception**.
- ❑ Réécrivons la classe erreur:

```
class erreur: public exception
{
    private:
        int num;
        string msg;

    public:
        erreur(int n=0, string const& m=" ") throw() : num(n),msg(m) { }
        virtual ~erreur() throw() { }
        virtual const char* what() const throw() {
            return msg.c_str();
        }
};
```

La gestion des exceptions

standard

□ Utilisation dans un main

```
#include <iostream>
#include <string>
#include <exception>
using namespace std;
// class erreur

int test_division(int a, int b)
{
    if (b==0) {
        throw erreur(1,"division par zero !");
    }
    return a/b;
}

int main(){
    int a,b;
    cout<<"a=";
    cin>>a;
    cout<<"b=";
    cin>>b;
    // ...
```

```
// suite
```

```
try{
    cout << "resultat= " << test_division(a, b) << endl;
}
catch (exception const& e)
{
    // récupération de la cause.
    cout << "Erreur, cause: " << e.what() << endl;
}
catch (...)
{
    cout << "Erreur inattendue !" << endl;
}

return 0;
}
```

La gestion des exceptions

standard

- ❑ La bibliothèque standard est capable de lancer 5 types d'exceptions que sont:
 - ❑ **std::bad_alloc** – erreur lancée par new (en mémoire).
 - ❑ **std::bad_cast** - erreur lancée lors d'un `dynamic_cast`
 - ❑ **std::bad_typeid** – erreur lancée lors d'un `typeid`
 - ❑ **std::bad_exception** – erreur lancée si aucun catch ne correspond à un objet lancé
 - ❑ **ios_base::failure** – erreur lancée en manipulant un flux
- ❑ Si on ne souhaite pas créer une classe pour gérer les exceptions, on pourra faire appel au fichier standard **stdexcept** contenant des classes d'exceptions pour les erreurs les plus courants.
- ❑ Il contient exactement 9 classes subdivisées en 2 catégories: les **logic errors** et les **runtime errors**.
- ❑ les **logic errors** sont: `domain_error`, `invalid_argument`, `length_error`, `out_of_range` et `logic_error` (toutes autres erreurs logiques);
- ❑ les **runtime errors** sont: `range_error`, `overflow_error`, `underflow_error`, `runtime_error` (toutes autres erreurs d'exécution).

La gestion des exceptions

exemple avec `domain_error`

□ exemple avec `domain_error`

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
using namespace std;
int test_division(int a,int b) {
    if(b==0)
        throw domain_error("division par zero !");
    else
        return a/b;
}
int main(){
    int a=4, b=0;
    try{
        cout << "resultat= " << test_division(a, b) << endl;
    } catch (exception const& e) {
        // récupération de la cause.
        cout << "Erreur, cause: " << e.what()<< endl;
        throw; //relance de l'expection reçue pour la traiter une deuxième fois, plus loin dans le code.
    }
    return 0;
}
```

À suivre ...

Feedback sur:
pape.abdoulaye.barro@gmail.com

FIN

Feedback

pape.abdoulaye.barro@gmail.com