PROG2 : Programmation C++

F. Lamarche

Résumé du cours

- Rapide historique
- Syntaxe de C++
- Classes et structures
- Pointeurs et gestion mémoire
- Références
- Constructeur de copie et surcharge d'opérateurs
- Héritage
- Programmation générique
- Bibliothèque standard et conteneurs : la STL
- Gestion des exceptions
- Les lambdas fonctions ☺

Rapide historique

- Le langage C

  - ⇨ Né dans les laboratoires Bell en 1972.

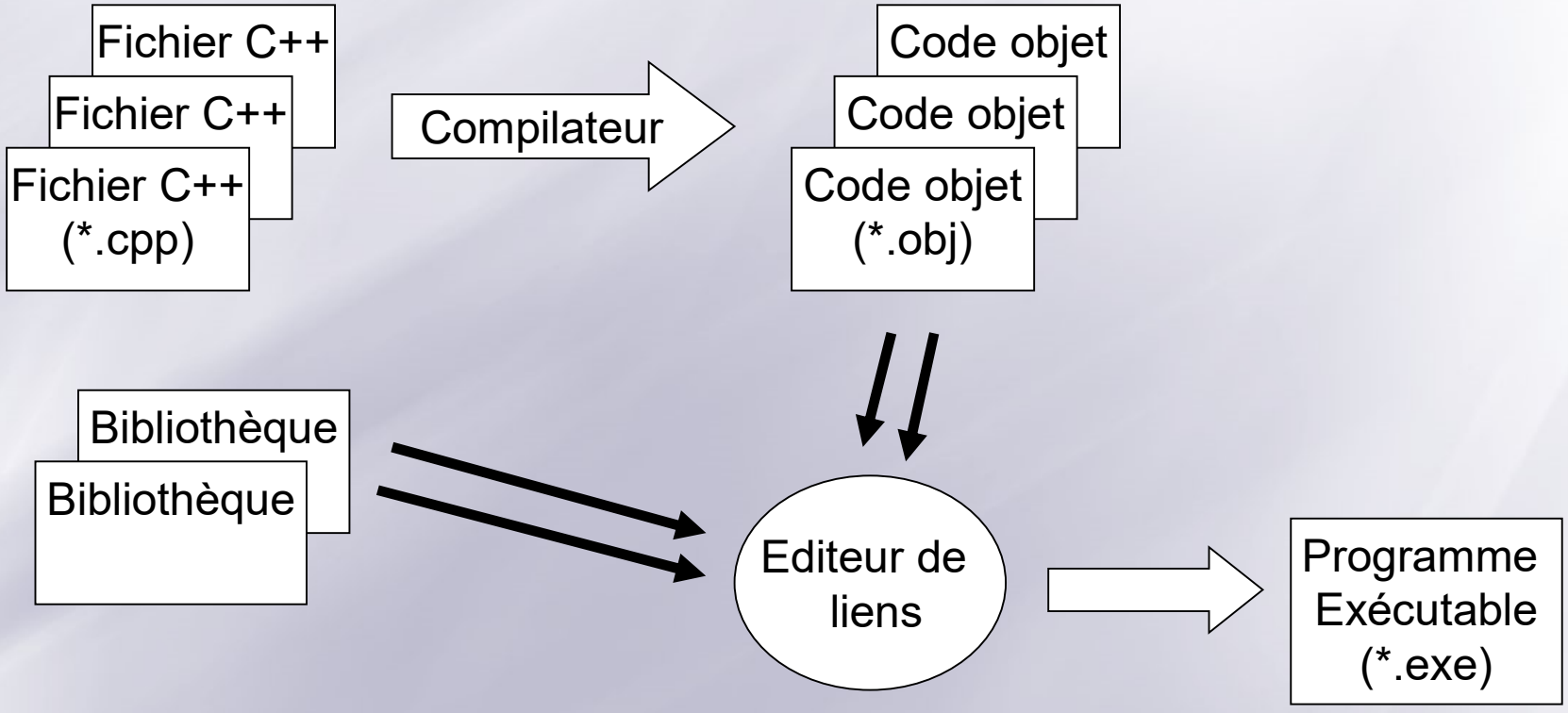
  - ⇨ Langage impératif.

  - ⇨ Utilisé dans de nombreux domaines
    - *Développement de systèmes, de logiciels système (réécriture d'Unix).*
    - *Ingénierie, Base de données, gestionnaire de communication, outils bureautiques.*

Rapide historique

- Le langage C++
  - ⇨ Créé en 1980
  - ⇨ Extension du langage C, **pour la programmation objet**
    - *Normalisé par la norme ANSI/ISO en 1998*
    - *Depuis : C++11, C++14, C++17, C++20 (en cours)*
  - ⇨ Structuré et modulaire
  - ⇨ Efficace
    - *Programmes compilés en langage machine*
    - *Optimisations effectuées par le compilateur*
  - ⇨ Portable
    - *Compilateurs disponibles sous tous les environnements*
    - *Attention : dépendance du code vis-à-vis du système d'exploitation*

Structure d'un programme C++



Structure d'un programme C++

- Deux « types » de fichiers source

  - ⇨ En-tête (extensions .h, .hpp)
    - *Déclarations de fonctions*
    - *Déclarations de classes*
    - ***Nécessaire** à l'utilisation de fonctions / méthodes*

  - ⇨ Implémentation (extensions .cxx, .cpp, .cc)
    - *Implémentation des fonctions*
    - *Implémentation des méthodes*

  - ⇨ Déclarations et implémentations faites dans des fichiers distincts

Exemple de programme

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

/*-----------------------------------------
Prototypes des fonctions
-----------------------------------------*/
int lireTableau (int valeurs[], int nmax);
float calculerMoyenne (const int valeurs[], int compteur);
void afficherValeurs (const int valeurs[], int compteur);
void afficherMoyenne (float moyenne);
int lireNombre (void);
```

- Prototypes de fonctions
  - ⇨ Déclaration pour utilisation
  - ⇨ Implémentation ultérieure

Exemple de programme

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

/*-----------------------------------------
Prototypes des fonctions
-----------------------------------------*/
int lireTableau (int valeurs[], int nmax);
float calculerMoyenne (const int valeurs[], int compteur);
void afficherValeurs (const int valeurs[], int compteur);
void afficherMoyenne (float moyenne);
int lireNombre (void);
```

- Inclusion de fichier
  - ⇨ Contient des prototypes

Exemple de programme

```
/*-------------------------------------------------------------
Calculer la moyenne d'une suite d'entiers positifs
-------------------------------------------------------------*/
int main()
{
  // Constante donnant la capacité du tableau
  const int capaciteTableau = 100;
  // Initialiser le tableau et déterminer le nombre de nombres lus !!
  int valeurs[capaciteTableau]; // tableau des nombres
  int compteur = lireTableau(valeurs, capaciteTableau);
  std::cout << "Nombre de valeurs lues: " << compteur << std::endl;
  if (compteur > 0)
  {
    float moyenne = calculerMoyenne(valeurs, compteur); // Calculer la moyenne
    afficherValeurs(valeurs, compteur); // Afficher les valeurs lues
    afficherMoyenne(moyenne); // Afficher le résultat
  }
  return 0;
}
```

Syntaxe

- Syntaxe proche de Java
  - ⇨ Attention : quelques différences cependant
- Sensible à la casse
  - ⇨ instruction ≠ INSTRUCTION
- Toute déclaration / instruction (ou presque) se termine par un point virgule
- Structure de bloc
  - ⇨ Les instructions sont dans des blocs { … }
  - ⇨ La portée d'une variable est le bloc dans lequel elle est définie

Les types élémentaires

- Les entiers (machine 32 bits)
  - ⇨ Type int, long
    - *Entiers signés, taille dépendante de la machine (32 bits)*
  - ⇨ Type short
    - *Entiers signés, taille dépendante de la machine (16 bits)*
  - ⇨ Pour des entiers non signés, ajouter le mot clé unsigned
    - *unsigned int, unsigned short, unsigned long*
  - ⇨ Constantes : -1, 1…
- Les flottants
  - ⇨ Type float : réel simple précision (4 octets)
  - ⇨ Type double : réel double précision (8 octets)
  - ⇨ Type long double : précision maximale, celle du FPU (10 octets)
  - ⇨ Constantes : 1.0, -1.0…

Les types élémentaires

- Les caractères
  - ⇨ Type char, unsigned char (1 octet)
  - ⇨ Aussi considéré comme un entier signé / non signé
  - ⇨ Constantes : ‘a’, ‘b’, ‘#’ ou 1, 2, -1, -2…

- Les booléens
  - ⇨ Type bool
  - ⇨ Constantes : true, false

Les variables

- Déclaration d'une variable
  - Syntaxe
    - *Type identifiant ;*
    - *Attention : variable non initialisée*
  - Syntaxe avec initialisation à la déclaration
    - *Type identifiant = valeur ;*
    - *Type identifiant(valeur) ;*
- Attention
  - C++ n'initialise pas les variables avec des valeurs par défaut
  - Une variable non initialisée possède une valeur indéterminée
    - *Source de nombreux « bugs »*
  - Règle : toujours initialiser les variables
  - Différence avec Java : *ici la variable n'est pas une « référence » mais bel et bien une* ***instance***

Exemple

```
// Variable de type entier
int min; // Attention : non initialisée

// Variable de type booléen
bool trouve = false ;
bool trouve(false) ;

// Variable de type réel
double pi = 3.1415926535;
double pi(3.1415926535) ;

// Variable de type caractère
char c = ‘a’ ;
char c(‘a’) ;

// Déclaration de plusieurs variables de même type
int min=0, max=100 ;
int min(0), max(100) ;
```

Les constantes

- Utilisation du mot clé const
- Syntaxe
  - ⇨ const type identifiant = valeur ;
  - ⇨ const type identifiant(valeur) ;
    - *Initialisation uniquement possible à la déclaration*
- Exemple

```
// Constante de type entier
const int v = 15 ;

// Constante de type réel
const double pi(3.1415926535) ;
```

Les opérateurs

⇨ Affectation: =

⇨ Opérateurs arithmétiques binaires (deux opérandes)

- *Addition: +*
- *Soustraction: -*
- *Multiplication: **
- *Division: / (division entière dans le cas de int)*
- *Modulo: % (reste de la division entière)*

⇨ Opérateurs arithmétiques unaires (un seul opérande)

- *Moins unaire: -*
- *Incrémentation: ++*
- *Décrémentation: --*

Les opérateurs

- Affectation avec opération
  - ⇨ Addition: +=, ex : a += b ⇔ a = a + b
  - ⇨ Soustraction: -=, ex : a -= b ⇔ a = a – b
  - ⇨ Multiplication: *=, ex : a *= b ⇔ a = a * b
  - ⇨ Division: /=, ex : a /= b ⇔ a = a / b
  - ⇨ Modulo: %=, ex : a %= b ⇔ a =a % b

- Opérateurs logiques (expressions booléennes)
  - ⇨ Négation: !
  - ⇨ ET logique: &&
  - ⇨ OU logique: ||

- Opérateurs de comparaison
  - ⇨ < , > , <= , >= , == , !=

Les chaines de caractères

- Type std::string
  - Déclaration d'une variable
    - *std::string ch ; // Crée une chaine vide*
    - *std::string bonjour = "hello " ;*

- Opérations
  - Concaténation : +
    - *ch = ch + "hello" ;*
    - *ch = ch + 'c' ;*
  - Comparaisons : ==, <, >, <=, >=
  - Longueur de la chaîne : méthode length
    - *int longueur = ch.length() ;*

- Pour utilisation, inclure le fichier <string>

```
#include <string>
using namespace std ; // instruction à éviter dans les fichiers en-tête
```

Les entrées / sorties

- Affichage à l'écran
  - ⇨ Utiliser std::cout
  - ⇨ L'opérateur << permet d'afficher à l'écran

```
std::cout<<"hello"<<10<<10.0<<'c'<<endl ;
```

- Lecture au clavier
  - ⇨ Utiliser cin
  - ⇨ L'opérateur >> permet de lire au clavier

```
std::string ch ;
int val ;
std::cin>>ch>>val ; // Lecture d'une chaîne suivie d'un entier
```

- Pour utilisation, inclure le fichier <iostream>

```
#include <iostream>
using namespace std ; // A éviter dans les fichiers en-tête
```

Structures de contrôle

- Conditionnelle

```
if(condition)
{
    // Instructions si
    // condition vraie
}
else
{
    // Instructions si
    // condition fausse
}
```

- Le bloc else est optionnel

Structures de contrôle

- Les cas

```
switch(expression)
{
  case valeur1 :
    // instructions exécutées
    // si
    // expression == valeur1
    break ;
  case valeur2 :
    // instructions
    break ;

  default :
    // instructions
}
```

- expression doit être de type
  - ⇨ Caractère
  - ⇨ Entier
  - ⇨ Enuméré
- Le mot clé break
  - ⇨ Provoque la sortie du switch
  - ⇨ Si absent, l'exécution continue séquentiellement
- Le bloc default
  - ⇨ Optionnel
  - ⇨ Exécuté si aucun des cas n'est valide

Structures de contrôle

- Boucle tant que

```
while(condition)
{
    // Instructions
}
```

- Ordre des traitements
  - ⇨ Test de la condition
    - *Sortie si condition fausse*
  - ⇨ Exécution des instructions
- Instructions par forcément exécutées

Structures de contrôle

- Boucle faire … tant que

```
do
{
    // Instructions
}
while(condition) ;
```

- Ordre des traitements
  - ⇨ Exécution des instructions
  - ⇨ Test de la condition
    - *Sortie si condition fausse*
- Instructions exécutées au moins une fois

Structures de contrôle

- Boucle pour…

```
for(initialisation ; condition ; progression)
{
    // Instructions
}
```

- Equivalent à

```
Initialisation
while(condition)
{
    // Instructions
    progression
}
```

- Partie *initialisation*
  - ⇨ Peut contenir une déclaration de variable
  - ⇨ Cette variable est **locale** à la boucle

Structures de contrôle

- Exemple de boucle for
  - *Affichage des nombre de 0 à 1000*

```
for(int i=0 ; i<1000 ; i++)
{
    cout<<i ;
}

cout<<i ;
```

Erreur car i n'est pas utilisable en dehors de la boucle

Rupture de séquence

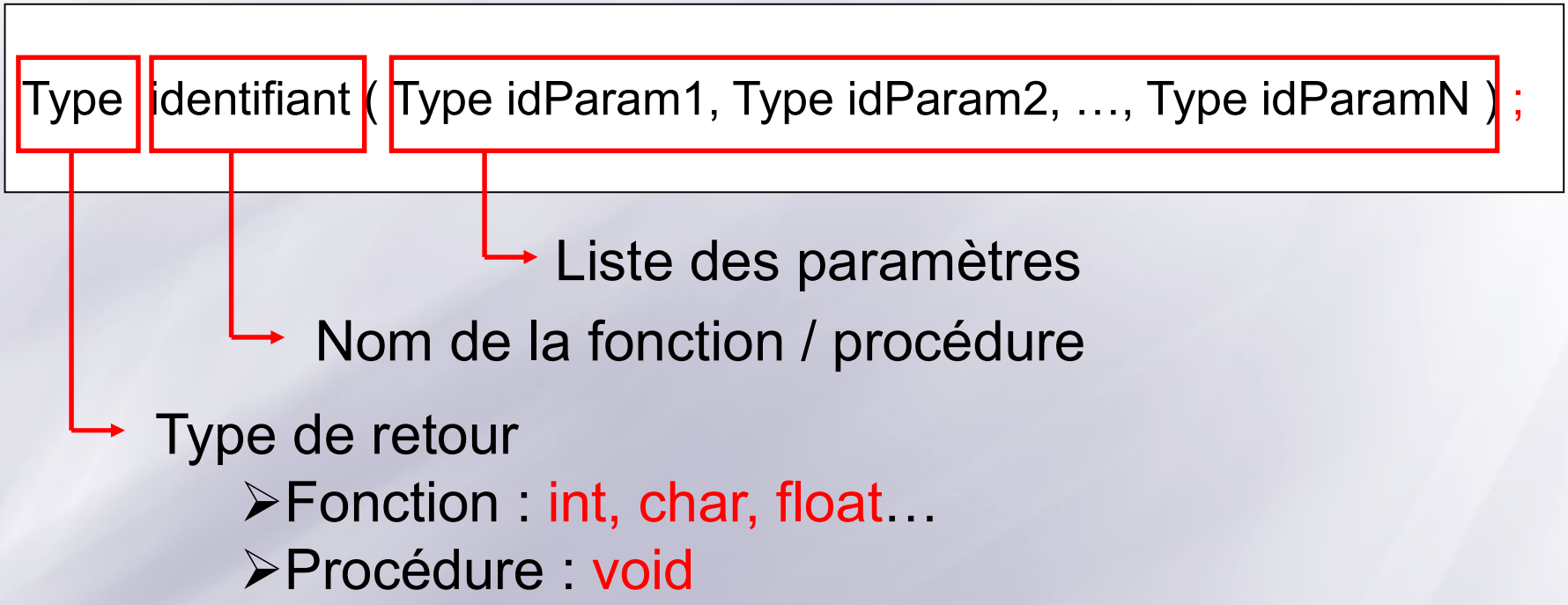
- Plusieurs mots clés permettent de rompre les séquences
  - ⇨ break
    - *Provoque la sortie immédiate d'une boucle*
      - for, while, do … while
    - *Interruption de séquence dans un switch*
  - ⇨ continue
    - *Saute l'itération en cours*
      - for, while, do … while
  - ⇨ return
    - *Dans une procédure : retour à la fonction/procédure appelante*
      - return
    - *Dans une fonction : renvoie immédiatement la valeur résultat*
      - return x

Procédures et fonctions

- Séparation déclaration / implémentation
  - ⇨ Prototypes
    - *Déclaration de la fonction*
      - Type de retour
      - Nom
      - Paramètres (types et identifiants)
    - *Pas d'implémentation*
    - *Nécessaires à l'utilisation des fonctions / procédures*
  - ⇨ Implémentation
    - *Association du code au prototype*
- Avantages
  - ⇨ Possibilité de compilation séparée
  - ⇨ Possibilité de « cacher » le code
  - ⇨ Bibliothèques : code compilé + prototypes

Procédures et fonctions

- Syntaxe des prototypes



Procédures et fonctions

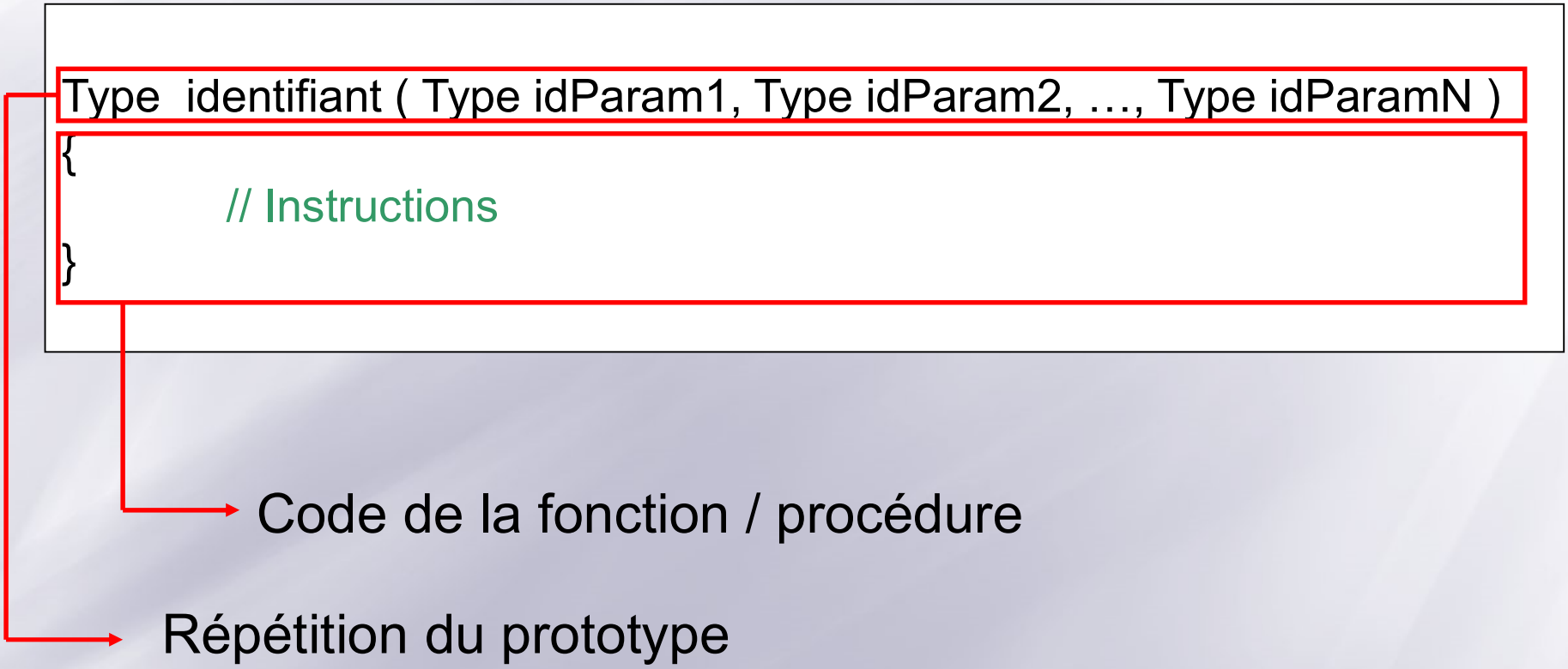
- Procédure / fonction sans paramètre

  ⇨ Deux syntaxes

```
Type identifiant(void) ;

Type identifiant() ;
```

Procédures et fonctions

- Implémentation



Procédures et fonctions

- Prototypes

```
// Prototype de la procédure d'affichage de moyenne
void afficherMoyenne (float moyenne);

// Prototype de la fonction permettant de lire un nombre
int lireNombre (void);
```

Procédures et fonctions

```
/*-----------------------------------
Lire un nombre; s'assurer qu'il est >= 0
Parame`tres donnés :
Parame`tres modifiés :
Résultat : un nombre entier >= 0
-----------------------------------*/
int lireNombre(void)
{
  int unnombre;
  cin >> unnombre;
  while (unnombre < 0)
  {
    cout << "Hé! On vous a dit un nombre entier positif ou nul\n";
    cin >> unnombre;
  } //--- unnombre >= 0 ---
  return unnombre;
}
```

Procédures et fonctions

- Par défaut, les paramètres sont passés **par valeur**

  ⇨ Dans une fonction / procédure, une modification de la valeur d'un paramètre n'est pas répercutée à l'appelant

  ⇨ La fonction / procédure travaille sur une **copie** du paramètre effectif

Procédures et fonctions

- Surcharge

  ⇨ Des fonctions/procédures peuvent porter le même nom

  ⇨ Différentiation sur
  - *Le nombre de paramètres*
  - *Le type des paramètres*
  - *Attention : pas de différenciation sur le type de retour*

  ⇨ A l'appel, le compilateur détermine quelle fonction / procédure appeler

Procédures et fonctions

- Valeurs de paramètres par défaut
  - ⇨ Possibilité offerte par C++
    - *Permet de simplifier la programmation*
  - ⇨ Valeurs précisées dans le **prototype**
  - ⇨ Syntaxe

```
Type identifiant(Type param, …,Type param = valeur);
```

  - ⇨ Importance de l'ordre des paramètres
    - *Attention : Les valeurs par défaut doivent être fournies du dernier paramètre vers le premier paramètre*

Exemple

- Prototype de procédure avec valeurs par défaut

```
// Fonction permettant de jouer une harmonique sur une carte son
// frequence : fréquence de l'harmonique en Hz
// amplitude : amplitude du signal généré
// frequenceRejeu : fréquence de rejeu sur la carte son
void jouerHarmonique( float frequence, float amplitude=1.0, int freqRejeu=44100);
```

- Exemple d'appels

```
int main(void)
{
  jouerHarmonique(10.0, 0.5, 22500) ;
  jouerHarmonique(10.0, 0.5) ;  // Equivalent à jouerHarmonique(10, 0.5, 44100)
  jouerHarmonique(10.0) ;       // Equivalent à jouerHarmonique(10, 1.0, 44100)
  return 0 ;
}
```

Les tableaux

- Déclaration
  - ⇨ Syntaxe

    ```
    Type identifiant[N] ;
    ```

  - ⇨ Déclare un tableau de N éléments de type Type nommé identifiant
  - ⇨ La capacité est forcément fournie à la déclaration
  - ⇨ Attention : dès la déclaration, le tableau est alloué (différence avec Java)
- Exemple

```
int tab[10] ; // Déclaration d'un tableau d'entiers de 10 éléments
```

Les tableaux

- Déclaration de tableaux initialisés
  - ⇨ Syntaxe

```
Type identifiant[] = {valeur1, valeur2,…,valeurN} ;
```

  - ⇨ La taille est déduite du nombre de valeurs fournies

- Exemple

```
// Déclaration d'un tableau de 4 éléments initialisé
int tab[] = {10, 11, -1, 5} ;              // tableau variable
const int tab[] = {10, 11, -1, 5} ;   // tableau constant
```

Les tableaux

- Utilisation
  - ⇨ Soit un tableau tab de capacité N
    - *Déclaration : Type tab[N]*
  - ⇨ tab[i] fournit le i-ème élément du tableau
  - ⇨ L'indice doit être dans l'intervalle [0;N-1]
    - *Attention : C++ ne vérifie pas la validité des indices*
      Source de nombreux « bugs »
  - ⇨ La capacité du tableau **doit toujours être connue**
    - *C++ n'offre pas de fonctionnalité permettant de connaître la capacité d'un tableau après sa déclaration*

Les tableaux

- Tableaux en paramètre de fonction / procédure
  - ⇨ Syntaxe au niveau du prototype

```
Type identifiantFnc(…,Type param[],…)
```

  - ⇨ param est un paramètre de type tableau de Type
  - ⇨ Le passage des tableaux se fait par « référence »
    - *La modification d'une case du tableau dans une fonction/procédure est répercutée à l'appelant*
  - ⇨ Pour assurer que le tableau ne peut être modifié, utiliser le mot clef const

```
Type identifiantFnc(…,const Type param[],…)
```

Les tableaux

- Passage de tableau en paramètre

  - ⇨ C++ ne fournit pas de système permettant de connaître la capacité d'un tableau

  - ⇨ Toute fonction prenant un tableau en paramètre doit aussi **prendre la capacité/taille de ce tableau en paramètre**

  - ⇨ Appel typique à une fonction prenant un tableau en paramètre

```
{
  type tab[N] ;
  fonction(tab, N) ;
}
```

Les tableaux

- Tableau comme résultat de fonction

  ⇨ Une fonction ne peut renvoyer un tableau en résultat

  ⇨ Pour obtenir le même effet

  - *Passer le tableau en paramètre*
  - *Les modifications effectuées dans la fonction / procédures seront répercutées à l'appelant*

Les tableaux

- Exemple de prototypes de fonctions / procédures utilisant des tableaux

```
// Fonction pouvant modifier le tableau
int lireTableau (int valeurs[], int nmax);

// Fonction ne modifiant pas le tableau
float calculerMoyenne (const int valeurs[], int compteur);

// Procédure ne modifiant pas le tableau
void afficherValeurs (const int valeurs[], int compteur);
```

Les tableaux

```
int main()
{
  // Constante donnant la capacité du tableau
  const int capaciteTableau = 100;
  // initialiser le tableau et déterminer le nombre de nombres lus !!
  int valeurs[capaciteTableau]; // tableau des nombres
  int compteur = lireTableau(valeurs, capaciteTableau);
  std::cout << "Nombre de valeurs lues: " << compteur << std::endl;
  if (compteur > 0)
  {
    float moyenne = calculerMoyenne(valeurs, compteur); // Calculer la moyenne
    afficherValeurs(valeurs, compteur); // Afficher les valeurs lues
    afficherMoyenne(moyenne); // Afficher le résultat
  }
  return 0;
}
```

Les tableaux

```
int lireTableau(int valeurs[], int nmax)
{
  int nombrelu, compteur = 0;
  cout << "Tapez une suite d'entiers > 0, terminée par 0\n";
  nombrelu = lireNombre(); // Lire un nombre >= 0
  while (nombrelu > 0 && compteur < nmax)
  {
    valeurs[compteur] = nombrelu;
    compteur = compteur + 1;
    nombrelu = lireNombre(); // Lire un nombre >= 0
  }
  return compteur;
}
```

- Fonction initialisant le tableau passé en paramètre
  - ⇨ Les modifications sont répercutées à l'appelant

Les tableaux

```
float calculerMoyenne(const int valeurs[], int compteur)
{
  int somme = 0;
  for (int i = 0; i < compteur; i++)
  {    somme = somme + valeurs[i]; }
  float moyenne;
  moyenne = (float) somme / (float) compteur;
  return moyenne;
}
```

- Fonction travaillant sur le tableau sans le modifier
  - ⇨ Utilisation du qualificateur const pour le paramètre

Classes et structures

Classes

- Déclaration de la classe
  - ⇨ Syntaxe

```
class NomClasse
{
  // Contrôle d'accès, attributs, prototypes de méthodes
} ;
```

  - *Contrôle d'accès par défaut : rien n'est accessible de l'extérieur*

```
struct NomClasse
{
  // Contrôle d'accès, attributs, prototypes de méthodes
} ;
```

  - *Contrôle d'accès par défaut : tout est accessible de l'extérieur*

Exemple

```
class Vecteur
{
public:
  // Un constructeur
 Vecteur(double v1 = 0, double v2 = 0, double v3 = 0);
  // Le destructeur
  ~Vecteur(void);
  // Affichage d'un Vecteur
  void    affiche(void)     const;
  // Produit scalaire de Vecteurs
  double  prodscal(Vecteur v) const;
  // Somme de 2 Vecteurs
  Vecteur somme(Vecteur v)    const;
  // Somme d'un Vecteur et d'un nombre
  Vecteur somme(double n)     const;
  // Calcul de l'homothétie du vecteur
  void homothetie(double valeur);
  // Somme de deux Vecteurs
  static Vecteur somme(Vecteur v1, Vecteur v2);
private:
  // Les attributs : les 3 coordonnées
  double x, y, z;
};
```

Exemple

```
class Vecteur
{
public:
  // Un constructeur
 Vecteur(double v1 = 0, double v2 = 0, double v3 = 0);
  // Le destructeur
  ~Vecteur(void);
  // Affichage d'un Vecteur
  void    affiche(void)     const;
  // Produit scalaire de Vecteurs
  double  prodscal(Vecteur v) const;
  // Somme de 2 Vecteurs
  Vecteur somme(Vecteur v)    const;
  // Somme d'un Vecteur et d'un nombre
  Vecteur somme(double n)     const;
  // Calcul de l'homothétie du vecteur
  void homothetie(double valeur);
  // Somme de deux Vecteurs
  static Vecteur somme(Vecteur v1, Vecteur v2);
private:
  // Les attributs : les 3 coordonnées
  double x, y, z;
};
```

- Déclaration d'une classe nommée **Vecteur**

Exemple

```
class Vecteur
{
public:
  // Un constructeur
 Vecteur(double v1 = 0, double v2 = 0, double v3 = 0);
  // Le destructeur
  ~Vecteur(void);
  // Affichage d'un Vecteur
  void    affiche(void)     const;
  // Produit scalaire de Vecteurs
  double  prodscal(Vecteur v) const;
  // Somme de 2 Vecteurs
  Vecteur somme(Vecteur v)    const;
  // Somme d'un Vecteur et d'un nombre
  Vecteur somme(double n)     const;
  // Calcul de l'homothétie du vecteur
  void homothetie(double valeur);
  // Somme de deux Vecteurs
  static Vecteur somme(Vecteur v1, Vecteur v2);
private:
  // Les attributs : les 3 coordonnées
  double x, y, z;
};
```

- Déclaration d'une classe nommée **Vecteur**
- Ensemble des prototypes de méthodes et des attributs

Classes

- Contrôle d'accès
  - public:
    Les attributs / méthodes accessibles depuis toute fonction ou méthode
    - *De préférence, uniquement des méthodes*
      - Assure l'encapsulation des données
  - protected:
    Les attributs / méthodes accessibles depuis une instance de la classe courante ou d'une classe dérivée
    - *Attributs*
    - *Méthodes*
  - private:
    Les attributs / méthodes uniquement accessibles depuis une instance de la classe courante
    - *Attributs « internes »*
    - *Méthodes « internes »*

Exemple

```
class Vecteur
{
public:
 // Un constructeur
Vecteur(double v1 = 0, double v2 = 0, double v3 = 0);
 // Le destructeur
 ~Vecteur(void);
 // Affichage d'un Vecteur
 void    affiche(void)     const;
 // Produit scalaire de Vecteurs
 double  prodscal(Vecteur v) const;
 // Somme de 2 Vecteurs
 Vecteur somme(Vecteur v)    const;
 // Somme d'un Vecteur et d'un nombre
 Vecteur somme(double n)     const;
 // Calcul de l'homothétie du vecteur
 void homothetie(double valeur);
 // Somme de deux Vecteurs
 static Vecteur somme(Vecteur v1, Vecteur v2);
private:
  // Les attributs : les 3 coordonnées
  double x, y, z;
};
```

- Ensemble de méthodes déclarées publiques
  - ⇨ Attention, différence avec Java
  - ⇨ Tout ce qui suit le label public: est considéré comme publique
  - ⇨ Il en est de même pour protected: et private:

Classes

- Déclaration des attributs
  - ⇨ Syntaxe

    Type indentifiant ;

  - ⇨ *Le type d'un attribut est un type élémentaire, une classe précédemment déclarée ou un tableau*
- Prototypes des méthodes
  - ⇨ Syntaxe pour une méthode modifiant les attributs

    TypeRetour identifiant(Type param1, …, Type paramN) ;

  - ⇨ Syntaxe pour une méthode ne modifiant pas les attributs

    TypeRetour identifiant(Type param, …, Type paramN) const ;

  - ⇨ Accepte la surcharge et les valeurs de paramètres par défaut

Classes

- Implémentation des méthodes
  - ⇨ Syntaxe

```
TypeRetour NomClasse::methode(Type param1,…)
{
    // Corps de la méthode
}
```

```
TypeRetour NomClasse::methode(Type param1,…) const
{ /* Corps de la méthode */ }
```

- Implémentation des méthodes de classe
  - ⇨ Syntaxe

```
TypeRetour NomClasse::methodeClasse(Type param1,…)
{ /* Corps de la méthode de classe*/  }
```

  - ⇨ *Pas de répétition du mot clé static*

Exemple d'implémentation

```
#include <iostream>
#include "Vecteur.h"
using std::cin;
using std::cout;

// Produit scalaire de 2 Vecteurs
double Vecteur::prodscal(Vecteur v) const
{
  return x * v.x + y * v.y + z * v.z;
}

// Homothétie
void Vecteur::homothetie(double valeur)
{
  x *= valeur; y *= valeur; z *= valeur;
}
```

- Inclusion du fichier contenant la déclaration de la classe
  - *Nécessaire*

Exemple d'implémentation

```
#include <iostream>
#include "Vecteur.h"
using std::cin;
using std::cout;

// Produit scalaire de 2 Vecteurs
double Vecteur::prodscal(Vecteur v) const
{
  return x * v.x + y * v.y + z * v.z;
}

// Homothétie
void Vecteur::homothetie(double valeur)
{
  x *= valeur; y *= valeur; z *= valeur;
}
```

- Inclusion du fichier contenant la déclaration de la classe
  - *Nécessaire*
- Implémentation de la méthode prodScal (ne modifiant pas les attribut : utilisation de const)
  - ⇨ Passage de paramètre par valeur
  - ⇨ Accès aux attributs du paramètre v : utilisation de '.'
  - ⇨ Accès aux attributs de l'instance courante : utilisation du nom de l'attribut

Exemple d'implémentation

```
#include <iostream>
#include "Vecteur.h"
using std::cin;
using std::cout;

// Produit scalaire de 2 Vecteurs
double Vecteur::prodscal(Vecteur v) const
{
  return x * v.x + y * v.y + z * v.z;
}

// Homothétie
void Vecteur::homothetie(double valeur)
{
  x *= valeur; y *= valeur; z *= valeur;
}
```

- Inclusion du fichier contenant la déclaration de la classe
  - *Nécessaire*
- Implémentation de la méthode prodScal (ne modifiant pas les attributs : utilisation de const)
- Implémentation de la méthode homothetie (modifiant les attributs)

Classes

- Constructeur
  - ⇨ Même nom que la classe mais pas de retour de résultat
  - ⇨ Syntaxe dans la déclaration

```
class NomClasse
{
    // Constructeur
    NomClasse(Type param1, …, Type paramN) ;
    // …
} ;
```

  - ⇨ Accepte la surcharge
    - *Plusieurs constructeurs avec des paramètres différents*
  - ⇨ Accepte les valeurs de paramètres par défaut
    - *Même syntaxe que pour les fonctions et procédures*

Classes

- Constructeur
  - ⇨Appelé automatiquement lors de la création d'une instance de classe
  - ⇨Rôle : initialiser les attributs de manière cohérente

  Exemple :

```
{
    // Déclaration de variable de type Vecteur
    // et appel du constructeur avec les trois paramètres explicités
    Vecteur v2(1.0,0.0,1.0) ;
    // Déclaration de variable de type Vecteur
    // et appel du constructeur avec paramètres par défaut
    Vecteur v1 ; // Equivalent à Vecteur v1(0.0, 0.0, 0.0)
}
```

Classes

- Constructeur
  - ⇨ Syntaxe dans l'implémentation

```
NomClasse::NomClasse(Type param1, …, Type paramN)
    : attribut1(valeur), attribut2(valeur)
{ /* Corps du constructeur */ }
```

  - ⇨ Initialisation des attributs
    - ***Explicite*** *: après les « : »*
      Obligatoire pour les attributs constants, les objets et les références

```
nomAttribut(paramètres du constructeur)
```

    - ***Implicite*** *: si non explicite, appel du constructeur sans paramètre*
      - Attention : les attributs instances d'objets doivent posséder un constructeur sans paramètres
      - Attention : les attributs de type élémentaire ne sont pas initialisés

Exemple d'implémentation

```
// constructeur : initialisation avec 0, 1, 2 ou 3 réels
Vecteur::Vecteur(double v1, double v2, double v3)
  :  x(v1), y(v2), z(v3)
{  // Affichage du vecteur construit
   cout << "construit " ; affiche();
}
```

Syntaxe à préférer

OU

```
// constructeur : initialisation avec 0, 1, 2 ou 3 réels
Vecteur::Vecteur(double v1, double v2, double v3)
{
   x=v1 ;
   y=v2 ;
   z=v3 ;
   // Affichage du vecteur construit
   cout << "construit " ; affiche();
}
```

Classes

- Exemples d'initialisation d'un objet : trois implémentations de la méthode Vecteur::somme fournissant le même résultat

```
Vecteur Vecteur::somme(Vecteur v) const
{
    Vecteur somme ;          // Appel du constructeur sans paramètre
    somme.x = x+v.x ;        // Initialisation explicite des trois attributs
    somme.y = y+v.y ;
    somme.z = z+v.z ;
    return somme ;
}
```

```
Vecteur Vecteur::somme(Vecteur v) const
{
    Vecteur somme(x+v.x, y+v.y, z+v.z) ; // Initialisation du résultat avec le constructeur
    return somme ;
}
```

```
Vecteur Vecteur::somme(Vecteur v) const
{
    return Vecteur(x+v.x, y+v.y, z+v.z) ; // Utilisation d'un objet temporaire de type Vecteur
}
```

Classes

- Intérêt des deux dernières constructions

```
Vecteur Vecteur::somme(Vecteur v) const
{
    Vecteur somme(x+v.x, y+v.y, z+v.z) ; // Initialisation du résultat avec le constructeur
    return somme ;
}
```

```
Vecteur Vecteur::somme(Vecteur v) const
{
    return Vecteur(x+v.x, y+v.y, z+v.z) ; // Utilisation d'un objet temporaire de type Vecteur
}
```

  - ⇨ Le constructeur initialise d'emblée les attributs avec des valeurs correctes
    - *Évite une double initialisation*
  - ⇨ Au niveau des traitements
    - *Le constructeur peut effectuer des traitements supplémentaires*
    - *Par exemple : normaliser le vecteur*

Classes

- Destruction des objets
  - ⇨ Attention : pas de « ramasse miettes » en C++
    - *La gestion mémoire est laissée à la responsabilité du programmeur*
  - ⇨ Utilisation d'un destructeur
    - *Sert à « desinitialiser »l'objet*
    - *Principalement utile lors de l'utilisation d'allocation dynamique de mémoire*
  - ⇨ Appel automatique lors de la destruction de l'objet
    - *Pour les objets créés par déclaration, appel automatique en fin de vie de l'objet (fin du bloc)*
    - *Pour les objets alloués dynamiquement, appel lors de la demande de libération de la mémoire*

Classes

- Destructeur
  - ⇨ Syntaxe de la déclaration

```
class NomClasse
{
    // Destructeur
    ~NomClasse() ;
    // …
} ;
```

```
class Vecteur
{
    // Destructeur
    ~Vecteur() ;
    // …
} ;
```

  - ⇨ Syntaxe de l'implémentation

```
NomClasse::~NomClasse()
{ /* Corps du destructeur */ }
```

```
Vecteur::~Vecteur()
{ cout << « détruit » ; affiche() ; }
```

Classes

- Attributs de classe
  - ⇨ Communs à toutes les instances (variables partagées)
    - *Toute modification effectuée par une instance est visible d'une autre instance*
  - ⇨ Syntaxe dans la déclaration

```
static Type indentifiant ;
```

- Méthodes de classe
  - ⇨ Indépendantes de l'instance
    - *Ces méthodes peuvent uniquement accéder aux attributs de classe*
  - ⇨ Syntaxe dans la déclaration

```
static TypeRetour identifiant(Type param1, …, Type paramN) ;
```

  - ⇨ Accepte la surcharge et les valeurs de paramètres par défaut

Exemple

```
#ifndef __Vecteur_H
#define __Vecteur_H

class Vecteur
{
public:
  //…
  // Somme de deux Vecteurs
  static Vecteur somme(Vecteur v1, Vecteur v2);
  // …
};
#endif
```

```
// Somme de 2 Vecteurs (méthode de classe)
Vecteur Vecteur::somme(Vecteur v1, Vecteur v2)
{
  return Vecteur(v1.x + v2.x, v1.y + v2.y, v1.z + v2.z);
}
```

- Exemple de méthode de classe
  - ⇨ La méthode somme de la classe Vecteur
  - ⇨ Ne peut accéder aux attributs x, y et z

Classes : utilisation

```
#include <iostream>
// Inclusion de la déclaration de la classe Vecteur
#include "Vecteur.h"
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int     main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1      : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2      : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
```

Classes : utilisation

```
v2.homothetie(2);
cout << "v2 *= 2 : ";   v2.affiche();
// v2 *= 2 : [ 2 4 6 ]
Vecteur v3(3, 2, 1);
// construit [ 3 2 1 ]
cout << "v3      : ";   v3.affiche();
// v3      : [ 3 2 1 ]
 cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
// v2 x v3 : 20
// détruit   [ 3 2 1 ]    : paramètre de prodscal
Vecteur v4 = v2.somme(v3);
// construit [ 5 6 7 ]
// détruit   [ 3 2 1 ]    : paramètre de somme
cout << "v2 + v3 : ";   v4.affiche();
// v2 + v3 : [ 5 6 7 ]
```

Classes : utilisation

```
v4 = v4.somme(10);
// construit [ 15 16 17 ] : objet local de somme
// détruit   [ 15 16 17 ] : objet local de somme
cout << "v4 + 10 : " ; v4.affiche();
// v4 + 10 : [ 15 16 17 ] : v4
v4 = Vecteur::somme(v4, v3);
// construit  [ 18 18 18 ] : objet local de somme
// détruit    [ 18 18 18 ] : objet local de somme
// détruit    [ 15 16 17 ] : paramètre de somme
// détruit    [ 3 2 1 ]    : paramètre de somme
cout << "v4 + v3 : ";   v4.affiche();
// v4 + v3 :  [ 18 18 18 ] : v4
return 0;
/* Fin du programme : destruction des objets restants
  détruit   [ 18 18 18 ] : v4
  détruit   [ 3 2 1 ]    : v3
  détruit   [ 2 4 6 ]    : v2
  détruit   [ 0 0 0 ]    : v1 */
}
```

Classes : utilisation

```
#include <iostream>
// Inclusion de la déclaration de la classe Vecteur
#include "Vecteur.h"
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int     main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1      : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2      : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
  v2.homothetie(2);
  cout << "v2 *= 2 : ";   v2.affiche();
  // v2 *= 2 : [ 2 4 6 ]
  Vecteur v3(3, 2, 1);
  // construit [ 3 2 1 ]
  cout << "v3      : ";   v3.affiche();
  // v3      : [ 3 2 1 ]
```

- Inclusion de la déclaration de la classe Vecteur

Classes : utilisation

```
#include <iostream>
// Inclusion de la déclaration de la classe Vecteur
#include "Vecteur.h"
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int     main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1       : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2       : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
  v2.homothetie(2);
  cout << "v2 *= 2 : ";   v2.affiche();
  // v2 *= 2 : [ 2 4 6 ]
  Vecteur v3(3, 2, 1);
  // construit [ 3 2 1 ]
  cout << "v3       : ";   v3.affiche();
  // v3      : [ 3 2 1 ]
```

- Inclusion de la déclaration de la classe Vecteur
- Déclaration de deux variables locales de type Vecteur : v1 et v2

Classes : utilisation

```
#include <iostream>
// Inclusion de la déclaration de la classe Vecteur
#include "Vecteur.h"
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int     main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1      : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2      : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
  v2.homothetie(2);
  cout << "v2 *= 2 : ";   v2.affiche();
  // v2 *= 2 : [ 2 4 6 ]
  Vecteur v3(3, 2, 1);
  // construit [ 3 2 1 ]
  cout << "v3      : ";   v3.affiche();
  // v3      : [ 3 2 1 ]
```

- Inclusion de la déclaration de la classe Vecteur
- Déclaration de deux variables locales de type Vecteur : v1 et v2
- Affichage des valeurs des deux vecteurs, appel de la méthode *affiche*

Classes : utilisation

```
#include <iostream>
// Inclusion de la déclaration de la classe Vecteur
#include "Vecteur.h"
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int     main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1      : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2      : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
  v2.homothetie(2);
  cout << "v2 *= 2 : ";   v2.affiche();
  // v2 *= 2 : [ 2 4 6 ]
  Vecteur v3(3, 2, 1);
  // construit [ 3 2 1 ]
  cout << "v3      : ";   v3.affiche();
  // v3      : [ 3 2 1 ]
```

- Inclusion de la déclaration de la classe Vecteur
- Déclaration de deux variables locales de type Vecteur : v1 et v2
- Affichage des valeurs des deux vecteurs, appel de la méthode *affiche*
- Appel de la méthode *homothetie* qui change la valeur des attributs du vecteur

Classes : utilisation

```
#include <iostream>
// Inclusion de la déclaration de la classe Vecteur
#include "Vecteur.h"
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int     main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1      : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2      : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
  v2.homothetie(2);
  cout << "v2 *= 2 : ";   v2.affiche();
  // v2 *= 2 : [ 2 4 6 ]
  Vecteur v3(3, 2, 1);
  // construit [ 3 2 1 ]
  cout << "v3      : ";   v3.affiche();
  // v3      : [ 3 2 1 ]
```

- Inclusion de la déclaration de la classe Vecteur
- Déclaration de deux variables locales de type Vecteur : v1 et v2
- Affichage des valeurs des deux vecteurs, appel de la méthode *affiche*
- Appel de la méthode *homothetie* qui change la valeur des attributs du vecteur
- Déclaration d'une variable de type Vecteur : v3 et appel de la méthode *affiche*

Classes : utilisation

```
  cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
  // v2 x v3 : 20
  // détruit   [ 3 2 1 ]    : paramètre de prodscal
  Vecteur v4 = v2.somme(v3);
  // construit [ 5 6 7 ]
  // détruit   [ 3 2 1 ]    : paramètre de somme
  cout << "v2 + v3 : ";   v4.affiche();
  // v2 + v3 : [ 5 6 7 ]
  v4 = v4.somme(10);
  // construit [ 15 16 17 ] : objet local de somme
  // détruit   [ 15 16 17 ] : objet local de somme
  cout << "v4 + 10 : " ; v4.affiche();
  // v4 + 10 : [ 15 16 17 ] : v4
  v4 = Vecteur::somme(v4, v3);
  // construit  [ 18 18 18 ] : objet local de somme
  // détruit    [ 18 18 18 ] : objet local de somme
  // détruit    [ 15 16 17 ] : paramètre de somme
  // détruit    [ 3 2 1 ]    : paramètre de somme
  cout << "v4 + v3 : ";   v4.affiche();
  // v4 + v3 :  [ 18 18 18 ] : v4
  return 0;
  /* Fin du programme : destruction des objets restants
    détruit   [ 18 18 18 ] : v4
    détruit   [ 3 2 1 ]    : v3
    détruit   [ 2 4 6 ]    : v2
    détruit   [ 0 0 0 ]    : v1 */
}
```

- Calcul du produit scalaire entre les vecteurs v2 et v3
  - ⇨ Appel de la méthode *prodscal* sur v2
  - ⇨ Passage de paramètre (v3) par valeur
  - ⇨ Affichage du résultat
  - ⇨ Destruction de la copie de v3

Classes : utilisation

```
  cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
  // v2 x v3 : 20
  // détruit   [ 3 2 1 ]    : paramètre de prodscal
  Vecteur v4 = v2.somme(v3);
  // construit [ 5 6 7 ]
  // détruit   [ 3 2 1 ]    : paramètre de somme
  cout << "v2 + v3 : ";   v4.affiche();
  // v2 + v3 : [ 5 6 7 ]
  v4 = v4.somme(10);
  // construit [ 15 16 17 ] : objet local de somme
  // détruit   [ 15 16 17 ] : objet local de somme
  cout << "v4 + 10 : " ; v4.affiche();
  // v4 + 10 : [ 15 16 17 ] : v4
  v4 = Vecteur::somme(v4, v3);
  // construit  [ 18 18 18 ] : objet local de somme
  // détruit    [ 18 18 18 ] : objet local de somme
  // détruit    [ 15 16 17 ] : paramètre de somme
  // détruit    [ 3 2 1 ]    : paramètre de somme
  cout << "v4 + v3 : ";   v4.affiche();
  // v4 + v3 :  [ 18 18 18 ] : v4
  return 0;
  /* Fin du programme : destruction des objets restants
    détruit   [ 18 18 18 ] : v4
    détruit   [ 3 2 1 ]    : v3
    détruit   [ 2 4 6 ]    : v2
    détruit   [ 0 0 0 ]    : v1 */
}
```

- Calcul du produit scalaire entre les vecteurs v2 et v3
  - ⇨ Appel de la méthode *prodscal* sur v2
  - ⇨ Passage de paramètre (v3) par valeur
  - ⇨ Affichage du résultat
  - ⇨ Destruction de la copie de v3
- Déclaration d'une variable de type Vecteur : v4 et initialisation à partir du résultat de l'évaluation de v2.somme(v3)
  - ⇨ Destruction de la copie de v3 passée en paramètre de *somme*

Classes : utilisation

```
  cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
  // v2 x v3 : 20
  // détruit   [ 3 2 1 ]    : paramètre de prodscal
  Vecteur v4 = v2.somme(v3);
  // construit [ 5 6 7 ]
  // détruit   [ 3 2 1 ]    : paramètre de somme
  cout << "v2 + v3 : ";   v4.affiche();
  // v2 + v3 : [ 5 6 7 ]
  v4 = v4.somme(10);
  // construit [ 15 16 17 ] : objet local de somme
  // détruit   [ 15 16 17 ] : objet local de somme
  cout << "v4 + 10 : " ; v4.affiche();
  // v4 + 10 : [ 15 16 17 ] : v4
  v4 = Vecteur::somme(v4, v3);
  // construit  [ 18 18 18 ] : objet local de somme
  // détruit    [ 18 18 18 ] : objet local de somme
  // détruit    [ 15 16 17 ] : paramètre de somme
  // détruit    [ 3 2 1 ]    : paramètre de somme
  cout << "v4 + v3 : ";   v4.affiche();
  // v4 + v3 :  [ 18 18 18 ] : v4
  return 0;
  /* Fin du programme : destruction des objets restants
    détruit   [ 18 18 18 ] : v4
    détruit   [ 3 2 1 ]    : v3
    détruit   [ 2 4 6 ]    : v2
    détruit   [ 0 0 0 ]    : v1 */
}
```

- Calcul du produit scalaire entre les vecteurs v2 et v3
  - ⇨ Appel de la méthode *prodscal* sur v2
  - ⇨ Passage de paramètre (v3) par valeur
  - ⇨ Affichage du résultat
  - ⇨ Destruction de la copie de v3
- Déclaration d'une variable de type Vecteur : v4 et initialisation à partir du résultat de l'évaluation de v2.somme(v3)
  - ⇨ Destruction de la copie de v3 passée en paramètre de *somme*
- Appel de la méthode *somme* sur v4
  - ⇨ Construction de la variable locale résultat
  - ⇨ Destruction de la variable locale lors du retour de résultat

Classes : utilisation

```
  cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
  // v2 x v3 : 20
  // détruit   [ 3 2 1 ]    : paramètre de prodscal
  Vecteur v4 = v2.somme(v3);
  // construit [ 5 6 7 ]
  // détruit   [ 3 2 1 ]    : paramètre de somme
  cout << "v2 + v3 : ";   v4.affiche();
  // v2 + v3 : [ 5 6 7 ]
  v4 = v4.somme(10);
  // construit [ 15 16 17 ] : objet local de somme
  // détruit   [ 15 16 17 ] : objet local de somme
  cout << "v4 + 10 : " ; v4.affiche();
  // v4 + 10 : [ 15 16 17 ] : v4
  v4 = Vecteur::somme(v4, v3);
  // construit  [ 18 18 18 ] : objet local de somme
  // détruit    [ 18 18 18 ] : objet local de somme
  // détruit    [ 15 16 17 ] : paramètre de somme
  // détruit    [ 3 2 1 ]    : paramètre de somme
  cout << "v4 + v3 : ";   v4.affiche();
  // v4 + v3 :  [ 18 18 18 ] : v4
  return 0;
  /* Fin du programme : destruction des objets restants
    détruit   [ 18 18 18 ] : v4
    détruit   [ 3 2 1 ]    : v3
    détruit   [ 2 4 6 ]    : v2
    détruit   [ 0 0 0 ]    : v1 */
}
```

- Appel de la méthode de classe *somme*
  - ⇨ Construction de l'objet local représentant le résultat
  - ⇨ Destruction de l'objet local au retour de résultat
  - ⇨ Destruction de la copie du paramètre v4
  - ⇨ Destruction de la copie du paramètre v3

Classes : utilisation

```
cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
// v2 x v3 : 20
// détruit   [ 3 2 1 ]    : paramètre de prodscal
Vecteur v4 = v2.somme(v3);
// construit [ 5 6 7 ]
// détruit   [ 3 2 1 ]    : paramètre de somme
cout << "v2 + v3 : ";   v4.affiche();
// v2 + v3 : [ 5 6 7 ]
v4 = v4.somme(10);
// construit [ 15 16 17 ] : objet local de somme
// détruit   [ 15 16 17 ] : objet local de somme
cout << "v4 + 10 : " ; v4.affiche();
// v4 + 10 : [ 15 16 17 ] : v4
v4 = Vecteur::somme(v4, v3);
// construit  [ 18 18 18 ] : objet local de somme
// détruit    [ 18 18 18 ] : objet local de somme
// détruit    [ 15 16 17 ] : paramètre de somme
// détruit    [ 3 2 1 ]    : paramètre de somme
cout << "v4 + v3 : ";   v4.affiche();
// v4 + v3 :  [ 18 18 18 ] : v4
return 0;
/* Fin du programme : destruction des objets restants
  détruit   [ 18 18 18 ] : v4
  détruit   [ 3 2 1 ]    : v3
  détruit   [ 2 4 6 ]    : v2
  détruit   [ 0 0 0 ]    : v1 */
}
```

- Appel de la méthode de classe *somme*
  - ⇨ Construction de l'objet local représentant le résultat
  - ⇨ Destruction de l'objet local au retour de résultat
  - ⇨ Destruction de la copie du paramètre v4
  - ⇨ Destruction de la copie du paramètre v3
- Fin du programme
  - ⇨ Renvoie le code de retour
  - ⇨ Destruction des variables locales v4, v3, v2, v1
- *Les variables locales sont détruites dans l'ordre inverse de leur déclaration (Norme du C++)*

Pointeurs

Pointeurs

- Définition

  Un pointeur est une variable dont la valeur est l'adresse d'une variable ou d'un objet

  Un pointeur pointe sur une variable d'un type précis, indiqué dans sa déclaration

Pointeurs

- Variable / paramètre formel de type pointeur
  - ⇨ Syntaxe
    - *Pointeur non constant vers un objet non constant*

```
Type * identifiant
```

    - *Pointeur constant vers un objet non constant*

```
Type * const identifiant
```

    - *Pointeur non constant vers un objet constant*

```
const Type * identifiant
```

    - *Pointeur constant vers un objet constant*

```
const Type * const identifiant
```

Pointeurs

⇨ Exemple

```
{
    // Pointeur sur un entier
    int * ptr_int ;
    // Deux pointeurs sur des réels
    float * ptr_flt1, * ptr_flt2 ;
    // Pointeur vers une instance de vecteur
    Vecteur * ptr_vecteur ;
}
```

⇨ Attention : un pointeur est une variable de type élémentaire. Sa valeur n'est donc pas initialisée par défaut (source de très nombreux bugs).

➢ ***Il faut toujours initialiser un pointeur***

Pointeurs

- Initialisation d'un pointeur

  - ⇨ Pointeur ne pointant sur rien : NULL

```
{
    // Un pointeur ne pointant sur rien
    Type * ptr_variable = NULL ;
}
```

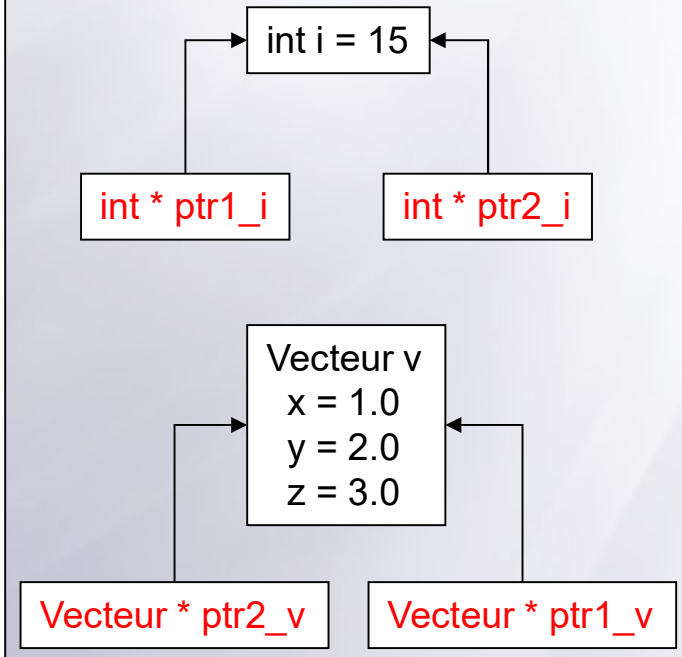
  - ⇨ Opérateur & : renvoie l'adresse d'une variable

```
{
    // Déclaration d'une variable
    Type identifiant ;
    // Initialisation d'un pointeur par l'adresse d'une variable
    Type * ptr_identifiant = & identifiant ;
}
```

Pointeurs

- Exemple d'initialisation d'un pointeur

```
{
    // Déclaration d'une variable de type entier (int)
    int i = 15 ;
    // Déclaration et initialisation de deux pointeurs
    // sur cette variable de type entier (int)
    int * ptr1_i(&i) ;
    int * ptr2_i = &i ;
    // Déclaration d'une variable de type Vecteur
    Vecteur v(1.0,2.0,3.0) ;
    // Déclaration et initialisation de deux pointeurs
    // sur cette variable de type Vecteur
    Vecteur * ptr1_v(&v) ;
    Vecteur * ptr2_v = &v ;
}
```



Pointeurs

- Indirection
  - ⇨ Opérateur * : récupération de la valeur pointée
    - *Soit ptr un pointeur*
    - *La valeur pointée est accessible (lecture/écriture) via l'expression *ptr*

Exemple

```
{
    // Variable de type entier
    int a = 10 ;
    // Pointeur sur cette variable
    int * ptr_a(&a) ;
    // Déclaration d'une seconde variable
    int b = *ptr_a ;  // b vaut 10 (initialisé à partir de la valeur de a, pointée par ptr_a)
    // Changement de la valeur de a, pointée par ptr_a
    (*ptr_a) = 5 ;    // a vaut 5
    // Incrementation de la valeur de a, pointée par ptr_a
    (*ptr_a)++ ;      // a vaut 6
}
```

Pointeurs

- Indirection

  - Accès à un attribut d'un objet pointé : opérateur ->
    - *Soit ptr un pointeur sur un objet*
    - *L'expression ptr->x renvoie la valeur de l'attribut x de l'objet pointé par ptr*
    - *Notation équivalente : (*ptr).x*

  - Appel de méthode sur un objet pointé: opérateur ->
    - *Soit ptr un pointeur sur un objet*
    - *L'expression ptr->identifiant() appelle la méthode identifiant sur l'objet pointé par ptr*
    - *Notation équivalente : (*ptr).identifiant()*

Exemple

```
{
    Vecteur v(1.0, 2.0, 3.0) ; // Variable de type vecteur
    Vecteur * ptr_v(&v) ;      // Pointeur sur la variable v
    ptr_v->affiche() ;         // Affichage du vecteur v pointé par ptr_v
    (*ptr_v).homothetie(2) ;   // Multiplie la valeur du vecteur v par 2
}
```

Pointeurs

- Passage de paramètre par pointeur
  - ⇨ Dans un but d'efficacité
    - *Passage par valeur d'un objet en paramètre*
      - Copie de l'objet
      - Pénalisant en temps et en mémoire
    - *Passage par pointeur*
      - Pas de copie de l'objet, simplement du pointeur
      - Gain en temps et en mémoire

Exemple de déclaration

```
// Passage de paramètre par valeur
double Vecteur::prodscal(Vecteur v) const
{ return x*v.x + y*v.y + z*v.z ; }
```

```
// Passage de paramètre par pointeur
double Vecteur::prodscal(Vecteur * v) const
{ return x*v->x + y*v->y + z*v->z ; }
```

Exemple d'utilisation

```
// Passage par valeur
 Vecteur v1, v2 ;
 v1.prodscal(v2) ;
```

```
// Passage par pointeur
Vecteur v1, v2 ;
v1.prodscal(&v2)
```

Pointeurs

- Passage de paramètres par pointeur
  - ⇨ Paramètres modifiables
    - *Pour modifier la valeur d'un paramètre dans une fonction, il faut transmettre l'adresse de ce paramètre*
    - *Paramètre effectif : l'adresse de la variable à modifier*
    - *Paramètre formel : pointeur*
- Exercice : fonction *permuter* échangeant la valeur de deux réels

Pointeurs

- Exercice : fonction *permuter* échangeant la valeur de deux réels

La fonction

```
void permuter(float * ptrV1, float * ptrV2)
{
    float tmp ;
    tmp = * ptrV1 ;
    *ptrV1 = *ptrV2 ;
    *ptrV2 = tmp ;
}
```

Appel avec &

```
float v1(10.0), v2(20.0) ;

permuter(&v1, &v2) ;
```

Appel avec pointeurs

```
float v1(10.0), v2(20.0) ;
float * ptr_v1(&v1), * ptr_v2(&v2) ;
permuter(ptr_v1, ptr_v2) ;
```

Pointeurs

- Passage de paramètres par pointeur
  - ⇨ Paramètre non modifiable
    - *Lorsqu'un paramètre est passé par pointeur mais n'est pas modifié, il faut le signaler*
    - *Utilisation du mot clé const*
    - *Syntaxe*

```
TypeRetour identifiant(…, const Type * param, …)
```

    - *param pointe vers une variable non modifiable de type Type*
    - *Attention : différence avec Java qui ne possède pas cette fonctionnalité*

Exemple

```
double Vecteur::prodscal(const Vecteur * v) const
{ return x*v->x + y*v->y + z*v->z ; }
```

- ⇨ Dans cet exemple, le vecteur pointé par v n'est pas modifiable
- ⇨ Toute tentative de modification génère une erreur

Pointeurs

- Fonction renvoyant un pointeur

  ⇨ Le pointeur renvoyé doit désigner :
  - *Une variable dont l'adresse est paramètre de la fonction*
  - *Une variable allouée dynamiquement*

  ⇨ Attention : le pointeur renvoyé ne doit pas désigner :
  - *Une variable locale*
  - *Un paramètre passé par valeur*
    - Le pointeur, au retour, désignera une variable détruite

- En C++, la manipulation de pointeurs requiert une grande vigilance

- Elle est la cause de très nombreux bugs parfois difficiles à corriger

Pointeurs

- Exemples
  - ⇨ Ce qu'il ne faut pas faire

```
float * pointeurMax(float v1, float v2)
{
    if(v1>v2)
    { return &v1 ; }
    return &v2 ;
}
```

Renvoie l'adresse d'un
Paramètre passé par valeur
(détruit après l'appel)

```
float * pointeurMax(float v1, float v2)
{
    float max ;
    if(v1>v2)
    { max = v1 ; }
    else
    { max = v2 ; }
    return &max ;
}
```

Renvoie l'adresse d'une
Variable locale
(détruite après l'appel)

Pointeurs

- Exemple

  - ⇨ Définition correcte d'une fonction renvoyant un pointeur

Déclaration de la fonction

```
const float * pointeurMax(const float * p1, const float * p2)
{
    if((*p1)>(*p2))
    { return p1 ; }
    return p2 ;
}
```

Utilisation de la fonction

```
{
    float v1(10.0), v2(20.0) ;
    const float * ptr_max ;
    ptr_max = pointeurMax(&v1, &v2) ;
}
```

Pointeurs et tableaux

- Tableaux à une dimension
  - ⇨ Déclaration

```
Type tab[NB] ;
```

  - *tab[i] est de type Type*
  - *tab est de type Type * const*
    (pointeur constant, zone pointée non constante)

  - ⇨ Déclaration d'un pointeur vers le premier élément

```
Type * ptr_tab (&tab[0]) ;
```

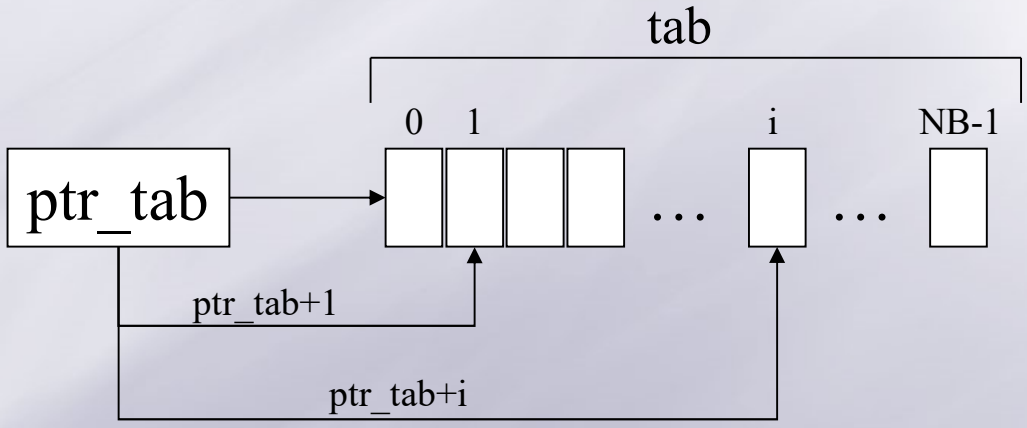
ou

```
Type * ptr_tab(tab) ;
```

  - ⇨ Il existe une arithmétique sur les pointeurs

Pointeurs et tableaux

- Arithmétique sur les pointeurs
  - ⇨ Relation 1 : *(ptr_tab + i) ⇔tab[i]



  - ⇨ Relation 2 : tab ⇔&tab[0]

  - ⇨ Relation 3 : tab[i] ⇔ *(tab+i) ⇔*(ptr_tab+i)⇔ptr_tab[i]

Tableau en paramètre

- Un tableau n'est jamais passé par valeur

  ⇨ Seule l'adresse du premier élément est transmise

  ⇨ Paramètre formel, deux possibilités :

  - *Paramètre formel de type tableau*

```
TypeRetour identifiant(…, T tableau[], …) ;
```

  - *Paramètre formel de type pointeur*

```
TypeRetour identifiant(…, T * tableau, …) ;
```

Tableau en paramètre

- Exemple :

  ⇨ Prototype de la fonction

```
int sommeTableau(int v[], int taille) ;
```

ou

```
int sommeTableau(int * v, int taille) ;
```

  ⇨ Appel

```
int tab[NB] ; // Déclaration du tableau
…
int somme = sommeTableau(tab, NB) ;
```

Allocation dynamique

- Permet de gérer des structures de données de taille variable
- Allocation de la mémoire gérée manuellement
- Libération de la mémoire gérée manuellement
- Attention : pas de « ramasse miettes »
  - *Pas de libération automatique de la mémoire*
    - Responsabilité du programmeur
  - *Grosse différence avec java*
- ***Règle : tout objet créé dynamiquement doit être désalloué***

Allocation dynamique

- Cas d'un objet
  - ⇨ Allocation : opérateur new
    - *Syntaxe*

```
T * p ;
p = new T (paramètres d'initialisation) ;
```

    - *Deux étapes*
      - Réservation de la mémoire dans le tas
      - Appel du constructeur de l'objet
  - ⇨ Désallocation : opérateur delete
    - *Syntaxe*

```
delete p ;
```

    - *Deux étapes*
      - Appel du destructeur de l'objet
      - Libération de la mémoire

Allocation dynamique

- Cas d'un tableau
  - ⇨ Allocation : opérateur new
    - *Syntaxe*

```
Type * tableau = new Type[NB] ;
```

    - *Deux étapes*
      - Allocation de la mémoire pour NB éléments de type Type
      - Appel du constructeur sans paramètre pour chaque instance
  - ⇨ Désallocation : opérateur delete[]
    - *Syntaxe*

```
delete[] tableau ;
```

    - *Deux étapes*
      - Appel du destructeur pour chaque instance
      - Libération de la mémoire allouée

Allocation dynamique

- Erreurs à éviter

⇨ Renvoyer une instance allouée dynamiquement par valeur

```
Vecteur Vecteur::sommeMAUVAIS(const Vecteur * const pv) const
{
    Vecteur * resultat = new Vecteur(x+pv->x, y+pv->y, z+pv->z) ;
    return *resultat ;
}
```

- *L'adresse du vecteur alloué est perdue*
- *La mémoire ne peut plus être libérée*

Allocation dynamique

⇨ Version correcte

```
Vecteur * Vecteur::somme(const Vecteur * const pv) const
{
    return new Vecteur(x+pv->x, y+pv->y, z+pv->z) ;
}
```

- *L'adresse du vecteur alloué est fournie en résultat*
- *La mémoire pourra être libérée*

Allocation dynamique

- Erreurs à éviter

⇨ Appel de l'opérateur delete et non delete[] sur un tableau

```
Vecteur * tabVecteurs = new Vecteur[1000] ;
…
delete tabVecteurs ;
```

- *Seul le destructeur du premier vecteur est appelé*
- *Dans certains cas, peut provoquer des problèmes dans le gestionnaire de mémoire*

Allocation dynamique

⇨ Version correcte

```
Vecteur * tabVecteurs = new Vecteur[1000] ;
…
delete[] tabVecteurs ;
```

- *Tous les destructeurs sont appelés*
- *La mémoire est correctement libérée*

Allocation dynamique

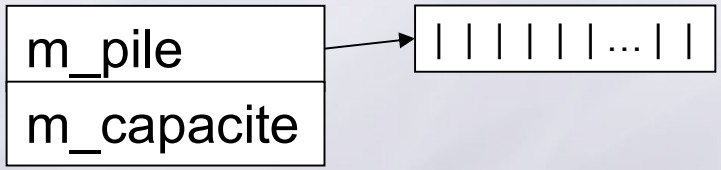
- Erreurs à éviter
  - ⇨ Oubli de programmation du destructeur

```
class PileNombres
{
public:
    PileNombres(int capacite) ;
    // Sans destructeur
private:
    float * m_pile ;
    int m_capacite ;
};
```

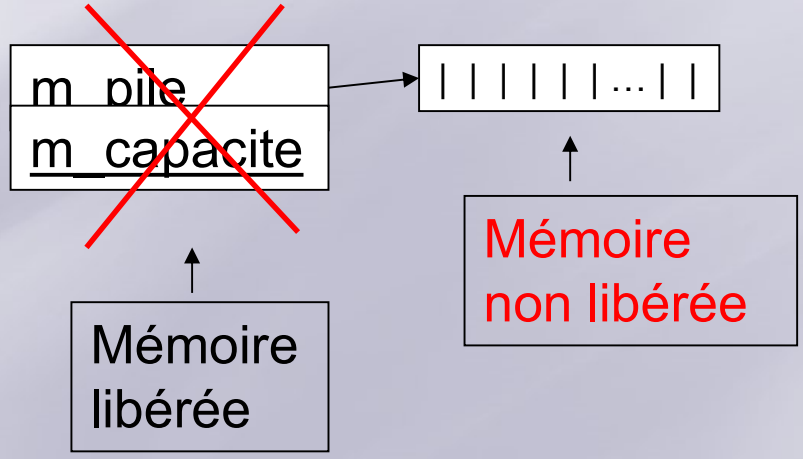
```
PileNombres::PileNombre(int capacite)
    : m_pile(new float[capacite]), m_capacite(capacite)
{}
```

Allocation dynamique

Après la construction d'une instance de PileNombres



Après la destruction d'une instance de PileNombres



Allocation dynamique

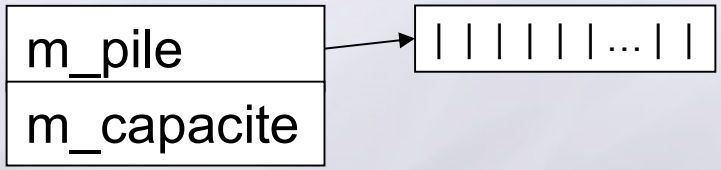
- Version correcte
  - ⇨ Ajout du destructeur et libération de la mémoire

```
class PileNombres
{
public:
    PileNombres(int capacite) ;
    ~PileNombres() ;
private:
    float * m_pile ;
    int m_capacite ;
};
```

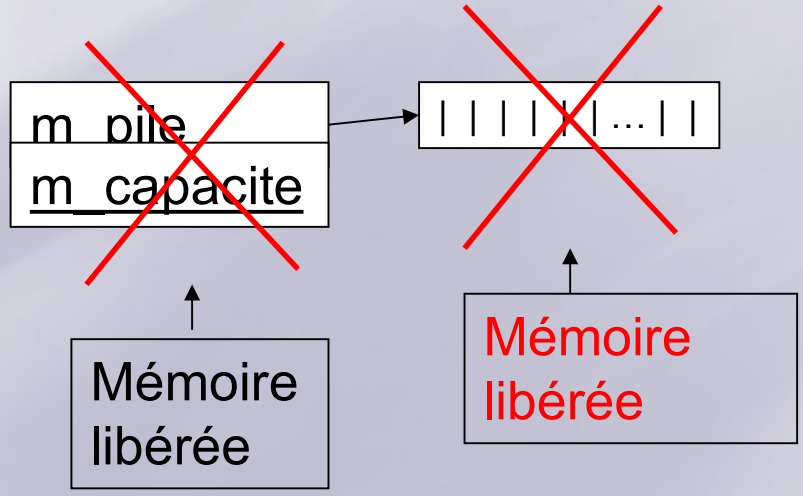
```
PileNombres::~PileNombre()
{ delete[] m_pile ; }
```

Allocation dynamique

Après la construction d'une instance de PileNombres



Après la destruction d'une instance de PileNombres



Allocation dynamique

- Taille mémoire prise par une structure de données

  ⇨ Exprimée en octets

  ⇨ Objet / Structure

```
int tailleMemoire = sizeof(Type) ;
```

  ⇨ Tableau d'objets / structures / types élémentaire

  - *Tableau de NB élements*

```
int tailleMemoire = sizeof(Type)*NB ;
```

Les références

Les références

- Définition
  - ⇨ Une référence est une variable, invariable après initialisation
  - ⇨ Elle désigne de façon non modifiable un objet ou une autre variable
  - ⇨ Aucun opérateur ne s'applique directement sur une référence
    - *Il s'applique toujours sur la variable référencée*

Les références

- Utilisations principales
  - ⇨ Désignation non modifiable d'objets
  - ⇨ Paramètres (plus simple que pointeurs)
  - ⇨ Constructeur de copie
  - ⇨ Surcharge d'opérateurs

Les références

- Initialisation

  - ⇨ Référence déclarée comme variable
    - *Initialisation à la déclaration*

  - ⇨ Référence comme paramètre formel (fonction/méthode)
    - *Initialisation lors de l'appel*
    - *Désigne le paramètre effectif*

  - ⇨ Référence comme attribut d'un objet
    - *Initialisation dans le constructeur*
    - *Partie déclarative (après « : ») uniquement*

Les références

- Cas des variables
  - ⇨ Syntaxe
    - *Référence vers un objet / une variable modifiable*

```
Type & ref(variable) ;
```

    - *Référence vers un objet / une variable constant*

```
const Type & ref(variable) ;
```

Les références

- Exemples

```
int i1(10), i2(20) ;

int & ref_i1(i1) ; // Initialisation obligatoire à la déclaration
int & ref_i2 = i2 ;

ref_i1 = 10 ;              // Équivaut à i1 = 10
ref_i2 = ref_i1 ;          // Équivaut à i2 = i1
```

```
Vecteur v(10,10,10) ;

Vecteur & ref_v(v) ;

ref_v.afficher() ;         // Équivaut à v.afficher()
(&ref_v)->afficher() ;     // Équivaut à (&v)->afficher()
```

Les références

- Paramètre formel de méthode / fonction
  - ⇨ Syntaxe
    - *Référence sur un paramètre effectif non constant*

```
TypeRetour identifiant(…, Type & ref, …) ;
```

    - *Référence sur un paramètre effectif constant*

```
TypeRetour identifiant(…, const Type & ref, …) ;
```

Les références

- Exemple : permutation de deux valeurs

⇨ Avec pointeurs

```
void permuter(float * v1, float * v2)
{   float tmp = *v1 ;
    *v1 = *v2 ;
    *v2 = tmp ;
}
```

⇨ Appel

```
float v1(10), v2(20) ;
permuter(&v1, &v2) ;
```

⇨ Avec références

```
void permuter(float & rv1, float & rv2)
{  float tmp = rv1 ;
    rv1 = rv2 ;
    rv2 = tmp ;
}
```

⇨ Appel

```
float v1(10), v2(20) ;
permuter(v1, v2) ;
```

Les références

- Exemple : retour sur le produit scalaire

  ⇨ Passage par valeur

```
float Vecteur::prodScal(Vecteur v) const
{
    return x*v.x + y*v.y + z*v.z ;
}
```

  ⇨ Passage par pointeur

```
float Vecteur::prodScal(const Vecteur * const v) const
{
    return x*v->x + y*v->y + z*v->z ;
}
```

Les références

- Exemple : retour sur le produit scalaire

⇨ Passage par référence constante (Syntaxe à privilégier)

```
float Vecteur::prodScal(const Vecteur & v) const
{
    return x*v.x + y*v.y + z*v.z ;
}
```

Les références

- Attributs
  - ⇨ Dans la déclaration de la classe

```
class C
{
    // …
    Type & ref ;
    const Type & const_ref ;
    // …
} ;
```

  - ⇨ Initialisation dans le constructeur : partie déclarative

```
C::C(paramètres)
    : ref(objet1), const_ref(objet2)
{ /* … */ }
```

Les références

- Exemple de la classe cercle : déclaration

```
class Cercle
{
private:
        Vecteur &       m_centre ;
        float           m_rayon ;

public:
        Cercle(Vecteur & centre, float rayon=1.0) ; // Constructeur
        Vecteur & centre() ; // consultation / modification du centre
        const Vecteur & centre() const ;  // consultation du centre
} ;
```

Les références

- Exemple de la classe cercle : implémentation

```
Cercle::Cercle(Vecteur & centre, float rayon)
      : m_centre(centre), m_rayon(rayon)
{}

Vecteur & Cercle::centre()
{ return m_centre ; }

const Vecteur & Cercle::centre() const
{ return m_centre ; }
```

Les références

- Erreur à éviter
  - ⇨ Retour d'une référence sur une variable locale

```
const Vecteur & Vecteur::somme(const Vecteur & v) const
{
   Vecteur resultat(x+v.x, y+v.y, z+v.z) ;
   return resultat ;
}
```

  - *Le vecteur est détruit en fin d'appel*
  - *La référence désigne un vecteur détruit*

Les références

⇨ Version correcte

```
Vecteur Vecteur::somme(const Vecteur & v) const
{
   Vecteur resultat(x+v.x, y+v.y, z+v.z) ;
   return resultat ;
}
```

Les références

- Chose correcte mais à éviter

  ⇨ Retour d'une référence sur un objet alloué dynamiquement

```
Vecteur & Vecteur::somme(const Vecteur & v) const
{  Vecteur * resultat = new Vecteur(x+v.x, y+v.y, z+v.z) ;
   return *resultat ;
}
```

Les références

- Appel avec erreur

```
{   Vecteur v1(1,2,3), v2(4,5,6) ;
    Vecteur somme = v1.somme(v2) ;
}
```

  - *L'adresse de l'objet alloué est perdue*
  - *La mémoire ne pourra pas être libérée*

- Appel correct

```
{   Vecteur v1(1,2,3), v2(4,5,6) ;
    Vecteur &somme (v1.somme(v2)) ;
    // …
    delete &somme ;
}
```

Les références

- Conseils et erreurs courantes

  - ⇨ Ne renvoyez jamais de référence sur un objet temporaire ou une variable locale
    - *Bug assuré*

  - ⇨ Lorsque vous utilisez l'allocation dynamique, ne renvoyez pas une référence sur l'objet alloué (préférez la notation avec pointeur)

  - ⇨ Plutôt que de passer des paramètres par valeur, passez les par référence sur un objet constant
    - *Nécessaire dans certains cas (constructeur de copie)*

  - ⇨ Au même titre que pour les pointeurs, prenez soin de préciser si les objets passés par référence restent constants (utilisation de const Type &) ou non (utilisation de Type &)

Constructeur de copie et surcharge d'opérateurs

Constructeur de copie

- Rappels : appels des constructeurs et destructeurs

  - ⇨ Variable locale
    - *Construite à la déclaration*
    - *Détruite en fin de bloc*

  - ⇨ Allocation dynamique d'un objet
    - *Construit lors de l'appel à new*
    - *Détruit lors de l'appel à delete*

Constructeur de copie

- Rappels : appels des constructeurs et destructeurs

  - ⇨ Objet temporaire
    - *Construit en cours d'évaluation d'expression*
    - *Détruit à la fin de l'évaluation*

  - ⇨ Attribut
    - *Construit à la construction de l'objet composite*
    - *Détruit à la destruction de l'objet composite*

Constructeur de copie

- Appel du constructeur de copie
  - ⇨ Lorsqu'un objet est initialisé avec un objet de même classe
  - ⇨ Passage d'un objet par valeur
  - ⇨ Retour d'un objet par une fonction / méthode
  - ⇨ Gestion des exception
    - *Lors de la propagation d'une exception*
- Par défaut, C++ fournit un constructeur de copie
  - ⇨ Mais fait-il toujours les traitements adéquats ?

Constructeur de copie

- Fonctionnement du constructeur de copie définit par défaut

  ⇨ Copie de surface d'un objet

  ⇨ Les attributs sont copiés un à uns

  ⇨ Dans le cas d'allocations dynamiques

    - *Attribut de type pointeur*

    - *Seule la valeur du pointeur est copiée*

    - *Les deux instances se partagent donc la mémoire allouée*
      - Peut être cause de gros « bugs »

    - *Dans ce cas, il faut reprogrammer ce constructeur*

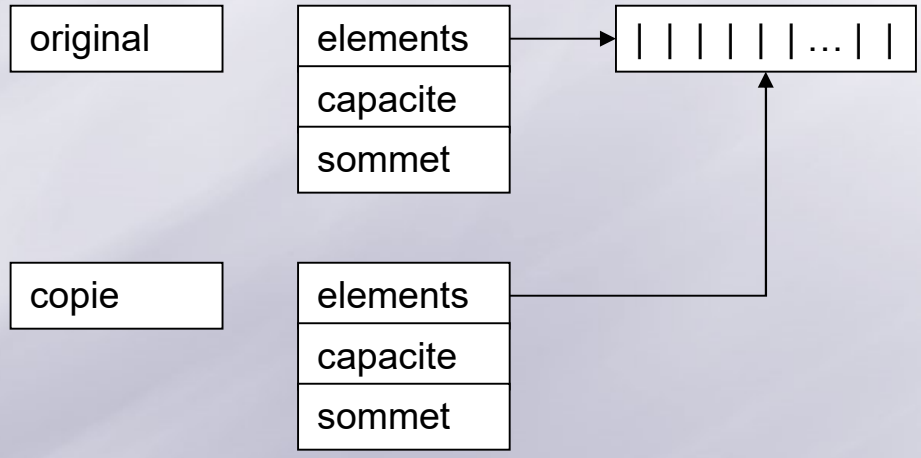
Constructeur de copie

- Exemple

```
class PileNombres
{
private:
    float * elements;
    int capacite ;
    int sommet ;

public:
    PileNombres(int capacite) ;
    ~PileNombres() ;
    // …
} ;
```

Constructeur de copie

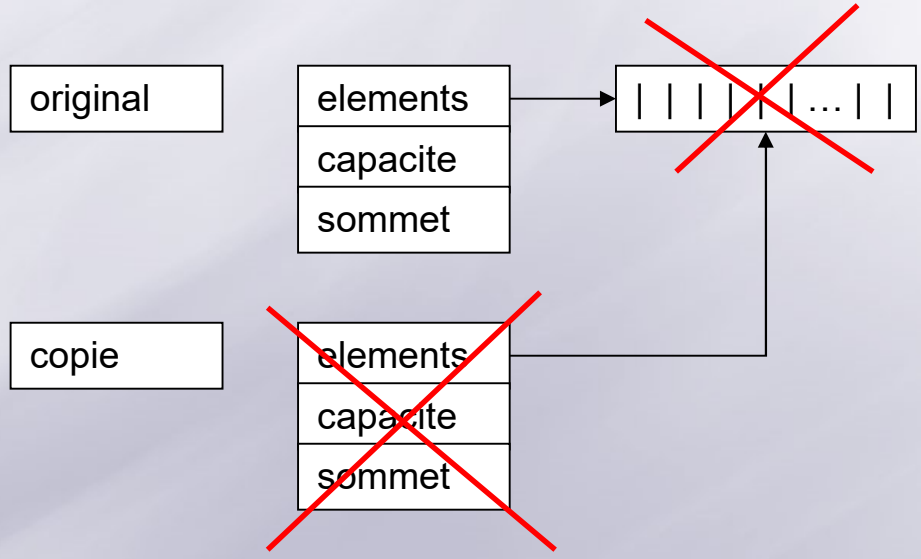
- Exemple de la classe PileNombres
  - ⇨ Comportement du constructeur de copie par défaut



  - ⇨ Les deux instances partagent la même zone mémoire
    - *Toute modification effectuée par l'un peut affecter l'autre*

Constructeur de copie

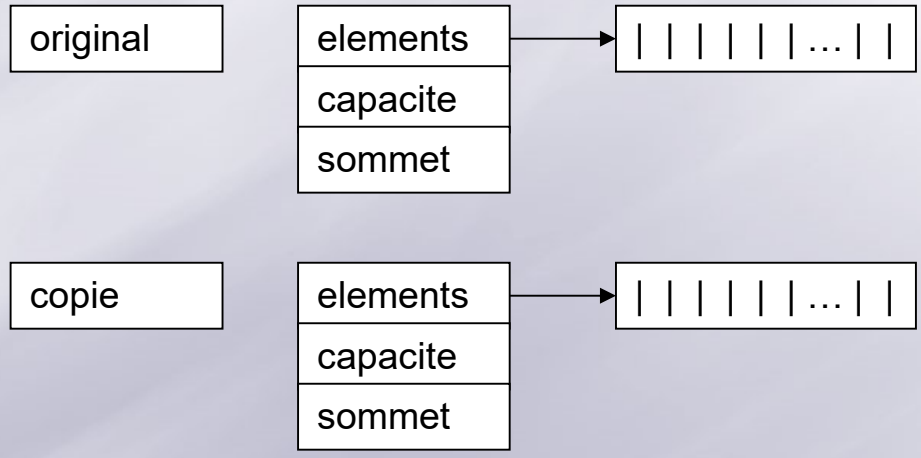
- Exemple de la classe PileNombres
  - ⇨ Et à la destruction ???



  - ⇨ La zone mémoire partagée par les deux instances est libérée
    - *Possibilité d'accès à une zone libérée…*

Constructeur de copie

- Exemple de la classe PileNombres
  - ⇨ Comportement souhaité



  - ⇨ Les deux instances on chacune leur zone mémoire

Constructeur de copie

- Syntaxe dans la déclaration

```
class C
{
    // …
    C(const C & o1) ;
    // …
} ;
```

⇨ Attention : le paramètre doit être une référence sur un objet non modifiable !

⇨ Dans le cas d'un passage par valeur
- *Appel récursif du constructeur de copie*
- *Plantage immédiat du programme après saturation de la pile…*

Constructeur de copie

- Traitement à effectuer

  - ⇨ Copier les attributs de o1 dans l'objet courant

    - *Sauf les pointeurs de données externes*

  - ⇨ Créer dynamiquement les données externes de l'objet courant

  - ⇨ Recopier le contenu des données externes de o1 dans les données externes de l'objet courant

Constructeur de copie

- Exemple : la classe PileNombes

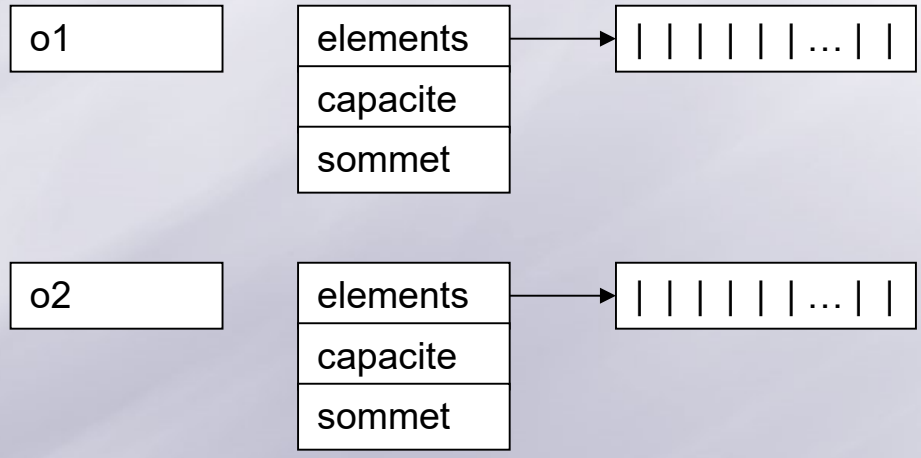
```
PileNombres::PileNombres(const PileNombres & p)
{       copierPile(p) ;
}

void PileNombres::copierPile(const PileNombres & p)
{       // Initialisation des attributs « non pointeurs »
        capacite = p.capacite ;
        sommet = p.sommet ;
        // Allocation de la mémoire et initialisation du pointeur
        elements = new float[capacite] ;
        // Copie le contenu de la pile p dans la zone allouée
        for(int i = 0 ; i<p.sommet ; i++)
        { elements[i] = p.elements[i] ; }
}
```

Opérateur d'affectation

- Même problème que pour le constructeur de copie
  - ⇨ Défini par défaut par C++
  - ⇨ Effectue une copie de surface
    - *Problème lors de l'utilisation d'allocation dynamique*
    - *Dans ce cas, il faut reprogrammer l'opérateur* **=**

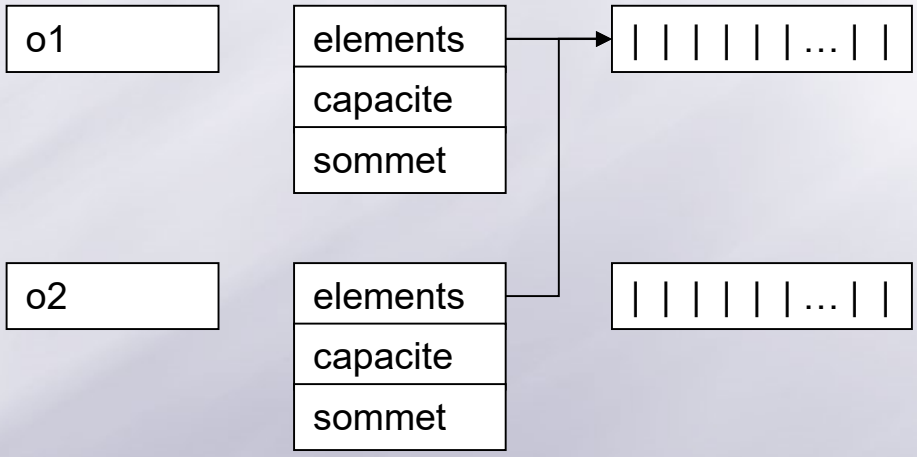
Opérateur d'affectation

- Deux instances de PileNombre : o1 et o2



Opérateur d'affectation

- Appel de o2 = o1



- Problèmes
  - ⇨ o1 et o2 partagent la même zone mémoire
    - *Cf. Problème du constructeur de copie*
  - ⇨ La mémoire préalablement allouée par o2 n'est pas libérée

Opérateur d'affectation

- Dans ce cas, il faut redéfinir l'opérateur =
- Traitement à effectuer
  - ⇨ Vérifier que l'opérande de gauche n'est pas la même que l'opérande de droite (ex : o2 = o2)
    - *Si même instance, ne rien faire*
  - ⇨ Si o2 désigne une zone de données externe
    - *Désallouer la mémoire associée*
  - ⇨ Copier les attributs de o1 dans o2
    - *Sauf les pointeurs de données externes*
  - ⇨ Créer dynamiquement les données externes pour l'objet courant
  - ⇨ Recopier le contenu des données externes de o1 dans celles de o2

Opérateur d'affectation

- Redéfinition de l'opérateur d'affectation

  ⇨ Syntaxe

```
class C
{
   // …
   C & operator = (const C & val) ;
   // …
} ;
```

  ⇨ Par convention
  - *L'opérateur = renvoie une référence sur l'instance courante*
  - *Permet d'enchaîner les affectations*
    - Ex : a = b = c (évaluation de droite à gauche)

Opérateur d'affectation

- Exemple

```
PileNombres & PileNombres::operator = (const PileNombre & p)
{
    if(this != &p) // Si le paramètre est différent de l'objet courant
    {
        // Désallouer la mémoire
        if(elements!=NULL) delete[] elements ;
        // Copier le contenu
        copierPile(p) ;
    }
    return (*this) ;
}
```

Surcharge d'opérateurs

- Opérateurs binaires
  - ⇨ Deux façons de surcharger un opérateur binaire (noté @)

  - *Méthode*

```
class C
{
   // …
   TypeRetour operator @ (const Type & v) ;
   TypeRetour operator @ (const Type & v) const ;
} ;
```

  - *Fonction*

```
TypeRetour operator @ (Type & v1, const Type & v2) ;
TypeRetour operator @ (const Type & v1, const Type & v2) ;
```

Surcharge d'opérateurs

- Tous les opérateurs binaires peuvent être surchargés

  - ⇨ Opérateurs modifiant la valeur de la partie gauche
    - =, +=, -=, *=, /=, |=, &=, ^=…

  - ⇨ Opérateurs ne modifiant pas la partie gauche
    - +, -, *, /, &, &&, |, ||…

Surcharge d'opérateurs

- Exemple
  - ⇨ Méthode

```
Vecteur Vecteur::operator+ (const Vecteur & v) const
{
    return Vecteur(x+v.x, y+v.y, z+v.z) ;
}
double Vecteur::operator* (const Vecteur & v) const
{
    return x*v.x + y*v.y + z*v.z ;
}
```

  - ⇨ Fonction

```
Vecteur operator + (const Vecteur & v1, const Vecteur & v2)
{
    return Vecteur(v1.getX()+v2.getX(), v1.getY()+v2.getY(),
    v1.getZ()+v2.getZ()) ;
}
```

Surcharge d'opérateurs

- Opérateurs unaires
  - ⇨ Deux façons de surcharger un opérateur unaire (noté @)
    - *Méthode*

```
class Type
{
   // …
   TypeRetour operator @ () ;
   TypeRetour operator @ () const ;
   // …
} ;
```

    - *Fonction*

```
TypeRetour operator @ (Type & v1) ;
TypeRetour operator @ (const Type & v1) ;
```

- Opérateurs unaires pouvant être surchargés
  - ⇨ ++, --, -, *, &, !, ->

Surcharge d'opérateurs

- Cas particulier des opérateurs ++ et –-
  - ⇨ Méthode

```
class Type
{
   // Opérateur ++ préfixé
   TypeRetour & operator++() ;
   // opérateur ++ postfixé (ajout d'un paramètre bidon)
   TypeRetour & operator++(int foo) ;
} ;
```

  - ⇨ Fonction

```
// Opérateur ++ préfixé
TypeRetour & operator++(Type & val) ;
// opérateur ++ postfixé (ajout d'un paramètre bidon)
TypeRetour & operator++(Type & val, int foo) ;
```

Surcharge d'opérateurs

- Opérateurs d'E/S : prototypes pour la surcharge

```
// Insertion dans un flot (écriture)
std::ostream & operator<<(std::ostream & out, const Type & valeur) ;

// Extraction d'un flot (lecture)
std::istream & operator>>(std::istream & in, Type & valeur) ;
```

Surcharge d'opérateurs

- Exemple

```
std::ostream & operator<<(std::ostream & out,
                                    const Vecteur & v)
{
    out<<" [" <<v.getX()<<" "<<v.getY()<<" "<<v.getZ()<< "] " ;
    return out ;
}
```

```
sdt::istream & operator>>(std::istream & in, Vecteur & v)
{
    double x,y,z ;
    in>>x>>y>>z ;
    v.setX(x) ; v.setY(y) ; v.setZ(z) ;
    return in ;
}
```

Héritage

Héritage

- C++ : un langage orienté objet
  - ⇨ Héritage
    - *Simple / Multiple*
  - ⇨ Liaison dynamique
- Différences avec Java
  - ⇨ Pas de notion explicite d'interface
  - ⇨ Par défaut liaison dynamique non utilisée
  - ⇨ Notion d'héritage multiple

Exemple

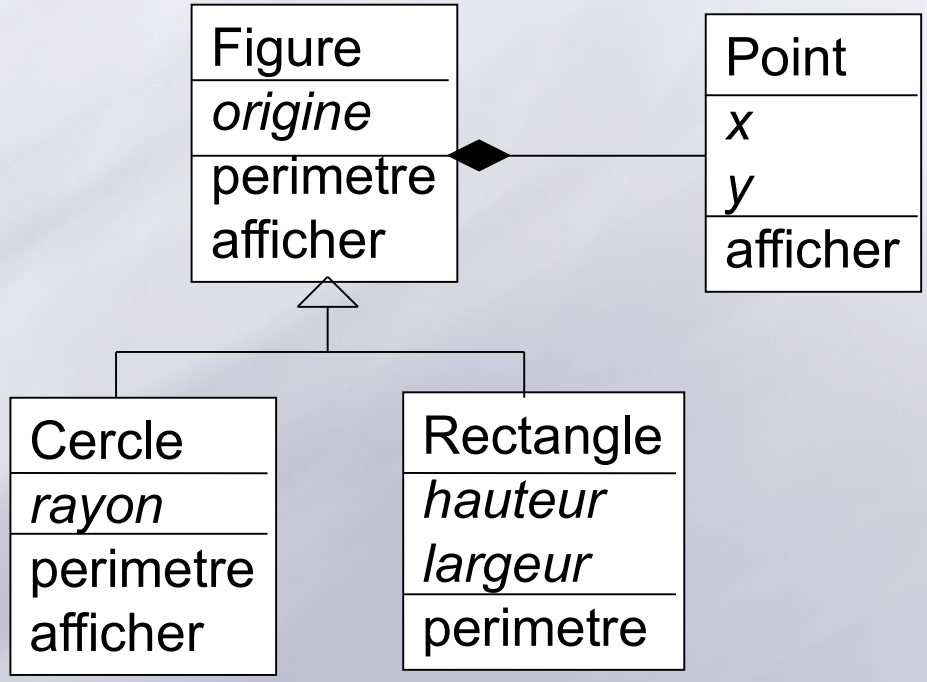


Figure
origine
perimetre
afficher
Point
x
y
afficher
Cercle
rayon
perimetre
afficher
Rectangle
hauteur
largeur
perimetre

Héritage et contrôle d'accès

- Un héritage peut-être
  - ⇨ Publique
    - *Aucun changement de contrôle d'accès sur les attributs / méthodes de la classe mère*
  - ⇨ Protégé
    - *Les attributs/méthodes publiques de la classe mère deviennent protégés*
    - *Les autres contrôles d'accès restent inchangés*
  - ⇨ Privé
    - *Les attributs/méthodes de la classe mère deviennent privés*

Héritage et contrôle d'accès

- Par défaut, en C++, l'héritage est privé
- Lorsqu'une classe B hérite d'une classe A, B peut accéder
  - ⇨ Aux attributs / méthodes publiques de A
  - ⇨ Aux attributs / méthodes protégés de A

Héritage et contrôle d'accès

⇨ Héritage publique (pas de changement de contrôle d'accès sur A)

```
class B : public A
{ // …
} ;
```

⇨ Héritage protégé (de l'extérieur, les attributs / méthodes de A sont protégés ou privés)

```
class B : protected A
{ // …
} ;
```

⇨ Héritage privé (les attributs / méthodes de A deviennent privés)

```
class B : private A
{ // …
} ;
```

```
class B : A
{   // …
} ;
```

Exemple

```
class Point
{
public:
  Point(double x=0, double y=0)
;
  virtual void afficher() const ;
private:
  double x,y ;
} ;
```

```
class Figure
{
public:
  Figure(Point o) ;
  virtual ~Figure() ;
  virtual double perimetre() const ;
  virtual void afficher() const ;
private:
  Point origine ;
} ;
```

Exemple

```
class Cercle : public Figure
{
public:
  Cercle(Point o, double r) ;
  virtual ~Cercle() ;
  virtual double perimetre() const ;
  virtual void afficher() const ;
private:
  double rayon ;
} ;
```

```
class Rectangle : public Figure
{
public:
  Rectangle(Point coin, double h,
                    double l) ;
  virtual ~Rectangle() ;
  virtual double perimetre() const ;
private:
  double hauteur, largeur ;
} ;
```

Constructeur de classe dérivée

- Pour initialiser les attributs hérités

  - ⇨ Il faut appeler le constructeur de la classe mère
  - ⇨ Appel **implicite** possible si la classe mère possède un constructeur sans paramètre
  - ⇨ Appel **explicite** dans la partie déclarative du constructeur (après « : »)

    - *Syntaxe*

```
class A : public B
{
   A(paramètres) ;
} ;
```

```
// Implémentation
A::A(paramètres)
  : B(paramètres), …
{ /*…*/ }
```

Constructeur de classe dérivée

- Exemple

```
Figure::Figure(Point o)
    : origine(o) // Initialisation de l'attribut
{}

Cercle::Cercle(Point o, double r)
    : Figure(o),    // Appel du constructeur de la classe mère
      rayon(r)      // Initialisation de l'attribut
{}

Rectangle::Rectangle(Point coin , double h, double l)
    : Figure(coin),             // Appel du constructeur de la classe mère
      hauteur(h), largeur(l) // Initialisation des attributs
{}
```

Constructeur de classe dérivée

- Exemple

  ⇨ Création par déclaration

```
Cercle c(Point(10, 10), 10) ;
Rectangle r(Point(50, 50), 20, 20) ;
```

  ⇨ Création par allocation

```
Cercle * c = new Cercle(Point(10, 10), 10) ;
Rectangle * r = new Rectangle(Point(50, 50), 20, 20) ;
```

Méthodes d'une classe dérivée

- Les méthodes définies dans la classe mère sont accessibles depuis une instance de la classe fille
  - ⇨ Comme en Java
    - *Même définition pour le contrôle d'accès*
- Possibilité de surcharge d'une méthode de la classe mère
  - ⇨ Paramètres différents
  - ⇨ L'appel de la méthode adéquate dépend des paramètres effectifs
- Possibilité de redéfinition d'une méthode de la classe mère
  - ⇨ Masque la première définition
  - ⇨ **Le type de l'objet auquel est appliqué la méthode détermine la méthode appelée**
  - ⇨ Pour accéder à la méthode d'une classe mère depuis une classe dérivée :

```
classeMere::methode(paramètres)
```

Méthodes d'une classe dérivée

```
void Figure::afficher() const
{
    std::cout<<" Figure : origine = " ;
    origine.afficher();
    std::cout<<" perimetre = “ <<perimetre()<<std::endl ;
}
```

```
Rectangle r(Point(50, 50), 20, 20) ;
r.afficher() ;
```

Méthodes appelées :

Figure::afficher : par héritage
    Point::afficher : par appel de origine.afficher()
    Rectangle::perimetre : car r est un rectangle

Méthodes d'une classe dérivée

- Exemple

```
void Cercle::afficher() const
{
    Figure::afficher() ;
    cout<<"Cercle : rayon = "<<rayon<<std::endl ;
}
```

```
Cercle c(Point(10,10), 10) ;
c.afficher() ;
```

Méthodes appelées

```
Cercle::afficher          : car c est un cercle
  Figure::afficher        : appel explicite
      Point::afficher     : appel explicite de origine.afficher()
      Cercle::perimetre   : car c est un cercle
```

Mélange de classes

- Utilisation
  - ⇨ Manipulation de collections
    - *Ex : collection de figures qui peuvent être*
      - Une figure, un cercle, un rectangle…
  - ⇨ Utilisation de la classe mère pour stockage
    - *Mais appels effectués sur les méthodes des classes dérivées*
    - *Permis par le mécanisme de liaison dynamique*
- Liaison dynamique
  - ⇨ Mécanisme permettant l'appel d'une méthode définie dans la classe effective (et non celle de la classe de l'identificateur, de la référence ou du pointeur)

Mélange de classes

- Règle de compatibilité
  - ⇨ Une instance d'une classe fille peut-être utilisée partout où une instance d'une classe mère publique peut-être utilisée
  - ⇨ Il en est de même pour les pointeurs et les références
- Règle valable
  - ⇨ Pour les références et les pointeurs
  - ⇨ Si passage par valeur : problème de troncature

Problème de troncature (slicing)

- Effet de la troncature
  - ⇨ Suppression des informations spécifiques des classes dérivées
  - ⇨ Conservation des informations de la classe de base
  - ⇨ Empêche le fonctionnement de la liaison dynamique
- Pour éviter de problème
  - ⇨ Manipulation par pointeur ou référence sur la classe de base
  - ⇨ Remarque : en Java, seules des références sont manipulées
    - *Évite de problème*

Problème de troncature (slicing)

- Il y a troncature lors des opérations suivantes

  ⇨ Initialisation de la classe de base par la classe dérivée

```
Cercle c(Point(10, 10), 10) ;
Figure f(c) ;
f.afficher() ;
```

```
Appel : Figure::afficher
Résultat : Figure : origine=(10,10) périmètre=-77
```

Problème de troncature (slicing)

- Il y a troncature lors des opérations suivantes

  ⇨ Affectation de la classe de base par la classe dérivée

```
Rectangle r(Point(10,10), 20,20)
Figure f = r ;
f.afficher() ;
```

```
Appel : Figure::afficher
Résultat : Figure : origine=(10,10) périmètre=-77
```

Problème de troncature (slicing)

- Il y a troncature lors des opérations suivantes

  ⇨ Passage de paramètre par valeur

  - *Initialisation du paramètre formel par paramètre effectif*

```
void afficher(Figure f)
{ f.afficher() ; }
```

```
Cercle c(Point (10,10), 10) ;
afficher(c) ;
```

```
Appel : Figure::afficher
Résultat : Figure : origine=(10,10) périmètre=-77
```

Problème de troncature

- Utilisation de pointeurs : pas de troncature

```
Cercle c(Point(10,10), 10) ;
Figure * ptr_figure(&c) ;
ptr_figure->afficher() ;
```

```
méthodes appelées
Cercle::afficher : liaison dynamique
  Figure::afficher : appel explicite
    Point::afficher : appel explicite
    Cercle::perimetre : liaison dynamique
```

- Utilisation de références : pas de troncature

```
Cercle c(Point(10,10), 10) ;
Figure & ptr_figure(c) ;
ptr_figure.afficher() ;
```

```
méthodes appelées
Cercle::afficher : liaison dynamique
  Figure::afficher : appel explicite
    Point::afficher : appel explicite
    Cercle::perimetre : liaison dynamique
```

Liaison dynamique

- Par défaut, liaison statique
  - *Choix de la méthode appelée fait à la compilation*
  - *Dépend uniquement du type déclaré de l'instance*
    - Que la désignation soit effectuée par identificateur, pointeur, référence
- Liaison dynamique
  - *Choix de la méthode appelée fait à l'exécution*
- Comportement inverse de celui de Java

Liaison dynamique

- Toute méthode précédée du mot clef virtual profite de la liaison dynamique
- Syntaxe

```
class A
{
    // Méthode virtuelle
    virtual TypeRetour methode(paramètres) ;
    // Destructeur virtuel
    virtual ~A() ;
} ;
```

Liaison dynamique

- Destructeur
  - ⇨ S'il existe une méthode virtuelle, le destructeur **DOIT** être virtuel dans la classe mère
  - ⇨ Permet d'appeler le destructeur de la classe dérivée à partir d'une référence ou un pointeur sur la classe de base
  - ⇨ Raison : les classes filles peuvent ajouter des attributs faisant référence à des données externes qui devront être désallouées
- Bonne habitude
  - ⇨ Déclarer tous les destructeurs comme étant virtuels
  - ⇨ Évite tout problème

Classe abstraite

- Il n'est pas toujours utile ou signifiant d'implémenter une méthode à un niveau abstrait d'une hiérarchie de classe
  - ⇨ Rappelez vous : Java et les interfaces…
  - ⇨ Ex : La méthode périmètre n'est pas signifiante dans la classe Figure
- Pour cela utiliser des méthodes abstraites
  - ⇨ Syntaxe

```
class A
{
    // Méthode abstraite
    virtual TypeRetour methode(paramètres) = 0 ;
} ;
```

  - ⇨ Les méthodes abstraites ne sont pas implémentées

Classe abstraite

- Une classe possédant au moins une méthode virtuelle pure est dite abstraite
  - ⇨ On ne peut créer d'instance d'une classe abstraite
  - ⇨ Toute classe héritant d'une classe abstraite et n'implémentant pas toutes les méthodes virtuelles pures est elle-même abstraite
  - ⇨ Cette classe peut être utilisée pour référencer ou pointer sur une instance d'une classe fille
- C++ n'offre pas la notion d'interface comme Java
  - ⇨ Mais : possibilité de définir des classes ne possédant que des méthodes virtuelles pures
  - ⇨ Dans ce cas, ces classes sont l'équivalent des interfaces Java

Classe abstraite

- Exemple

```
class Figure
{
public:
    Figure(Point o) ;
    virtual ~Figure() ;
    virtual void afficher() const ;
    // Méthode virtuelle pure devant être définie dans les classes filles
    virtual double perimetre() const = 0 ;
private:
    Point origine ;
} ;
```

Dans ce cas, on ne peut plus créer d'instance de Figure

Programmation générique

Programmation générique

- Programmation générique
  - ⇨ Possibilité de définir des fonctions et des classes
    - *Paramétrées par un type*
    - *Une classe*
    - *Une constante*

Fonctions et procédures génériques

- Syntaxe

```
template <class A, class B, …>
TypeRetour fonction(Type1 param1, Type2 param2, …)
{
    // Corps de la fonction / procédure
}
```

⇨ TypeRetour, Type1, Type2 peuvent être
- *Un type défini*
- *Un paramètre générique*

⇨ Note : le corps d'une fonction méthode générique doit être implémenté dans le fichier « xx.h » i.e. le fichier contenant les déclarations

Exemple

- Définition de la fonction : la fonction max générique

```
template <class T>
const T & max(const T & x, const T & y)
{
    if(x<y) { return y ; }
    return x ;
}
```

⇨ Contrainte sur T : ce type doit définir un opérateur <

⇨ Contrairement à Java, pas de mécanisme permettant de le dire…

Exemple

- Utilisation

```
int i(5), j(10), m(max(i,j)) ;
```

⇨ Ici, le compilateur instancie max<int>

```
Rationnel r1(1,2), r2(3,4), rm(max(r1,r2)) ;
```

⇨ Ici, le compilateur instancie max<Rationnel>

Exemple

- Cas de deux paramètres de type différents

```
int i(5) ;
double  j(19.0) ;

double m = max(i,j);  // Génération d'une erreur
```

  - *Le compilateur ne sait pas déterminer le type du paramètre T (int ou double ?)*

⇨ Dans ce cas, il faut instancier explicitement

```
double m = max<double>(i,j) ;
```

Fonctions et procédures génériques

- Dans certains cas, la fonction générique n'est pas correcte

  ⇨ Ex : l'opérateur < n'est pas défini sur les booléens

  ⇨ Dans ce cas, on peut décider d'un ordre et définir une fonction **spécialisée** pour un type donné

  ⇨ Exemple :

```
template <>
const bool & max<bool>(const bool & x, const bool & y)
{
    return x || y ;
}
```

Fonctions et procédures génériques

- Exemple de méta programming : paramètre générique sous forme de constante
  - ⇨ Calcul de factorielle à partir d'une constante

```
template <unsigned int value> // Paramètre de type valeur entière
int factorielle()
{ return value* factorielle<value-1>() ; }

template <>
int factorielle<0>() // Spécialisation pour la valeur 1
{ return 1 ; }
```

  - ⇨ Appel correct

```
int f = factorielle<100>() ; // Le compilateur calcule la valeur
```

  - ⇨ Erreur :

```
int v = 100 ;
int fv = factorielle<v>() ; // Erreur : un paramètre template DOIT être une constante
```

Classes génériques

- Syntaxe

  ⇨ Déclaration

```
template <class A, class B, …>
class C
{
    // Déclaration d'attributs / méthodes utilisant
    // Des types connus ou types génériques (A,B…)
} ;
```

  ⇨ Implémentation de méthodes

```
template <class A, class B, …>
TypeRetour C<A,B,…>::methode(paramètres)
{
}
```

Exemple

```
// Une paire générique
template <class T1, class T2>
class Pair
{   T1 m_first ;  T2 m_second ;
public:
    Pair(const T1 & first, const T2 & second) ;
    const T1 & getFirst() const ;
    const T2 & getSecond() const ;
    void setFirst(const T1 & value) ;
    void setSecond(const T2 & value) ;
} ;
```

```
// Exemple d'implémentation du constructeur
template <class T1, class T2>
Pair<T1,T2>::Pair(const T1 & first, const T2 & second)
    : m_first(first), m_second(second)
{}
```

Exemple

- Opérateur d'affichage d'une paire générique

```
template <class T1, class T2>
std::ostream & operator<<(std::ostream& out,
                                  const Pair<T1,T2> & pair)
{
      out       << « ( » << pair.getFirst()
                << « , » << pair.getSecond() << « ) » ;
   return out ;
}
```

⇨ Contraintes sur les paramètres génériques :

- *Les types T1 et T2 doivent disposer d'un opérateur <<*

Programmation générique

- Pour chaque déclaration différente
  - ⇨ Le compilateur crée
    - *Un exemplaire de classe*
    - *Un exemplaire de méthode*
    - *Un exemplaire de fonction*
- Utilisation d'un paramètre générique
  - ⇨ Induit des contraintes au niveau du type lui-même
    - *Existence de méthodes spécifiques*
    - *Existence d'un constructeur par défaut*
    - *…*
  - ⇨ Il faut le spécifier dans les commentaires
    - *C++, à la différence de Java, n'autorise pas l'expression explicite de contraintes sur les paramètres génériques*

Exemple

```
template <class T> class Pile
{
public:
Pile(int n=20) ;
Pile(const Pile & p) ;
~Pile() ;
bool estPleine() ;
bool estVide() ;
void empiler(const T & e) ;
T getSommet() const ;
void depiler() ;
void afficher() const ;
Pile & operator=(const Pile<T> & p) ;
```

```
private:
    void copierPile(const Pile<T> & p) ;
    T * elements ;
    int capacite ;
    int nbelt ;
} ;
```

Exemple

```
// Constructeur
template <class T>
Pile<T>::Pile(int n)
    : elements(new T[n]), capacite(n), nbelt(0)
{}
```

⇨ Contrainte : new T[n] => constructeur sans paramètre pour T

```
// Empilement d'un élément
template <class T>
void Pile<T>::empiler(const T & e)
{
    elements[nbelt] = e ;
    nbelt++ ;
}
```

⇨ Constrainte : elements[nbelt]=e => opérateur = défini sur T

Exemple

- Opérateur d'affichage

```
template <class T>
std::ostream & operator << (std::ostream & out, const Pile<T> &
    p)
{   p.afficher() ;
    return out ;
}
```

Exemple

- Initialisation

```
template <class T>
void initialiser(Pile<T> & p)
{   T v ;
    while(!p.estPleine())
    {     cin>>v ;
          p.empiler(v) ;
    }
}
```

⇨ Contrainte : opérateur >> redéfini pour le type T

Exemple

- Utilisation

```
int main()
{
    Pile<float> pile ;
    initialiser(pile) ;
    std::cout<<pile<<std::endl ;
}
```

- Liste des contraintes
  - ⇨ Existence sur T
    - *D'un constructeur par défaut (constructeur de Pile)*
    - *D'un opérateur = (utilisé dans empiler)*
    - *D'un opérateur << (utilisé dans afficher)*
    - *D'un constructeur de copie (retour par valeur dans getSommet)*
    - *D'un opérateur >> (utilisé dans initialiser)*

La STL : Standard Template Library

STL

- Fournie avec tous les compilateurs C++
- Implémente
  - ⇨ Les structures de données classiques
    - *Tableaux, ensembles, tables de correspondance, files, piles…*
  - ⇨ Les algorithmes classiques
    - *Tri, calculs ensemblistes…*
    - *Tous les algorithmes sont indépendants des structures de données*
- **Exploitation intensive de la généricité**

STL : concepts

- Repose sur trois concepts
  - ⇨ Conteneur
    - *Structure de données de taille variable contenant des objets*
    - *Paramétrage par le type d'objet manipulé (généricité)*
  - ⇨ Itérateur
    - *Objet qui permet d'accéder aux éléments des conteneurs*
    - *Même utilisation pour TOUS les conteneurs*
    - *Utilisation proche du pointeur : opérateurs *, ->, …*
  - ⇨ Algorithmes
    - *Traitements sur les éléments d'un conteneur*
    - *Utilisent les iterateurs*
      - Indépendants du conteneur ☺

Deux typologies de conteneurs

- Conteneurs séquentiels
  - ⇨ Les éléments sont placés séquentiellement dans le conteneur
  - ⇨ Leur ordre dépend uniquement de l'ordre d'insertion
- Conteneurs associatifs
  - ⇨ Collection triée d'éléments
  - ⇨ Indépendance vis-à-vis de l'ordre d'insertion
  - ⇨ Ordre uniquement dépendant d'un critère de tri paramétrable
- Attention : le choix du conteneur doit être fait en fonction des utilisations algorithmes les utilisant
  - ⇨ Performances différentes en fonction des traitements effectués

Les conteneurs séquentiels

- Tableau : classe std::vector<Type>

  - ⇨ Gestion de tableaux dynamiques
    - *Agrandissement à la fin du tableau*

  - ⇨ Accès direct aux éléments (opérateur [])

  - ⇨ Ajout / retrait en **fin** rapide

  - ⇨ Ajout / retrait ailleurs, complexité linéaire (décalage d'une partie du tableau)

Les conteneurs séquentiels

- File à deux têtes : classe std::deque<Type>

  - ⇨ Tableau dynamique
    - *Agrandissement en fin et en tête*
  - ⇨ Accès direct aux éléments (opérateur [])
  - ⇨ Ajout / retrait en **début et fin** très rapide
  - ⇨ Ajout retrait ailleurs lent, complexité linéaire (décalage d'une partie du tableau)

Les conteneurs séquentiels

- Liste : classe std::list<Type>

  - ⇨ Liste doublement chaînée

  - ⇨ Pas d'accès direct aux éléments
    - *Accès au i ème élément : parcours des i-1 éléments*

  - ⇨ Accès successeur / prédécesseur immédiat

  - ⇨ Ajout / retrait à toute position très rapide

Les conteneurs séquentiels

- Quelques méthodes de vector / deque / list
  - ⇨ Ajout d'une valeur en tête de structure
    - *void push_front(const Type & v)*
  - ⇨ Ajout de valeur en fin de struture
    - *void push_back(const Type & v)*
  - ⇨ Récupération de la première valeur
    - *Type & front()*
    - *const Type & front() const*
  - ⇨ Récupération de la dernière valeur
    - *Type & back()*
    - *const Type & back() const*
  - ⇨ Nombre d'éléments contenus dans la structure
    - *size_t size() const*

Les conteneurs séquentiels

- Quelques méthodes de vector et deque

  ⇨ Récupération du ieme élément
  - *Type & operator[] (int i) ;*
  - *const Type & operator[] (int i) const ;*

- Exemple d'affichage d'un vecteur d'entiers

```
void afficher(const std::vector<int> & tab)
{
        for(int cpt=0 ; cpt<tab.size() ; cpt++)
        { std::cout<<tab[cpt]<<std::endl ; }
}
```

Conteneurs séquentiels

➢ *Efficacité des conteneurs séquentiels*

Structures de données	Typologies de traitement requis sur la structure de données			
	Ajout / retrait			Accès direct aux élements
	Début	Milieu	Fin	
std::vector	-	-	+	+
std::deque	+	-	+	+
std::list	+	+	+	-

Les conteneurs associatifs

- Un conteneur associatif trie ses éléments selon un ordre
  - ⇨ Par défaut, utilisation de l'opérateur <
  - ⇨ Possibilité de fournir une fonction de comparaison
- Accès aux éléments via itérateurs
- Implémentation en utilisant des arbres
  - *Temps d'ajout / insertion : $O(log_2(n))$*
  - *Algorithmes efficaces d'équilibrage d'arbres*

Les conteneurs associatifs

- Ensembles : std::set<Type, Comparateur>

  - ⇨ Gestion d'ensembles triés
    - *Assure l'unicité d'un élément dans la structure de données*

  - ⇨ Deux paramètres génériques
    - *Le type de la donnée manipulée*
    - *Le type du comparateur utilisé (optionnel)*
      - Par défaut, utilisation de l'opérateur <

  - ⇨ Insertion / suppression / recherche rapides : $O(\log_2(n))$

Les conteneurs associatifs

- Tables d'association :
  classe std::map<Clef, Valeur, Comparateur>

  ⇨ Association d'une valeur à une clef

  ⇨ Garantie d'unicité de la clef

  ⇨ Structure de donnée triée suivant la clef

  ⇨ Insertion / suppression / recherche rapides : $O(\log_2(n))$

  ⇨ Les valeurs manipulées en interne sont du type std::pair<Clef, Valeur>

Les conteneurs associatifs

- Suppression de la notion d'unicité de la valeur / clef

  ⇨ classe std::multiset<Type, Comparateur>

  - *Même chose que std::set*
  - *Mais possibilité de plusieurs occurrences d'une valeur*

  ⇨ classe std::multimap<Clef, Valeur, Comparateur>

  - *Même chose que std::multimap*
  - *Mais possibilité de plusieurs occurrences d'une clef*

Les itérateurs

- Itérateur
  - ⇨ Objet désignant la position d'un élément dans un conteneur
  - ⇨ Permet de manipuler cet élément
    - *consultation / modification*
  - ⇨ Permet de parcourir les éléments d'un conteneur
    - *Utilisation d'une interface normalisée*
  - ⇨ Plusieurs itérateurs peuvent parcourir différemment la même structure
- Manipulation similaire à celle des pointeurs
  - ⇨ Utilisation de la surcharge d'opérateurs

Les itérateurs

- Trois grandes catégories
  - ⇨ Différenciées sur les modes de parcours
  - ⇨ « Forward iterators »
    - *Accès à l'élément suivant*
  - ⇨ « Bidirectional iterators »
    - *Accès au prédécesseur et à l'élément suivant*
  - ⇨ « Random access iterators »
    - *Dotés d'une arithmétique autorisant l'accès direct aux autres éléments de la structure*

Les itérateurs

- Opérations des « Forward iterators »
  - ⇨ * : fournit la valeur de l'élément désigné
  - ⇨ -> : accès aux attributs / méthodes de l'élément désigné
  - ⇨ ++ : progression de l'itérateur sur l'élément suivant
  - ⇨ ==, != : Comparaison de deux itérateurs
  - ⇨ = : Affectation de deux itérateurs

- Opérations des « Bidirectional iterators »
  - ⇨ Celles des « Forward iterators »
  - ⇨ -- : progression de l'itérateur sur l'élément précédent

- Opérations des « Random access iterators »
  - ⇨ Celles des « Bidirectional iterators »
  - ⇨ Tous les opérateurs utilisables sur les pointeurs
    - [], +, -, +=, -=

Les itérateurs

- Les types iterator et const_iterator

  - ⇨ Les conteneurs séquentiels et associatifs définissent comme classe interne ces deux types d'itérateurs

  - ⇨ iterator : classe d'itérateur permettant de modifier la valeur désignée (sauf pour std::set)

  - ⇨ const_iterator : classe d'itérateur ne permettant que de consulter la valeur désignée

Récupération d'itérateurs

- Les containers séquentiels et associatifs disposent des méthodes suivantes pour récupérer des itérateurs

  ⇨ Itérateur sur le début de la structure

```
iterator begin() ;
const_iterator begin() const ;
```

  ⇨ Itérateur sur l'élément suivant le dernier élément

```
iterator end() ;
const_iterator end() const ;
```

Les itérateurs : exemples

- Exemple : affichage du contenu d'une « map » de chaînes de caractères et d'entiers

```
void afficher(const std::map<std::string, int> & dico)
{
    for(std::map<std::string, int>::const_iterator it=dico.begin() ;
        it!=dico.end() ; ++it)
    {
        std::cout << it->first << « , » << it->second <<std::endl ;
    }
}
```

Les algorithmes

- La STL propose des implémentations d'algorithmes standards
- Implémentés sous la forme de fonctions
- Utilisation des itérateurs pour les parcours
  - ⇨ Indépendants de la structure de données
  - ⇨ Attention : mais pas indépendant du type d'itérateur
    - *Ex : pour un tri, un itérateur de type « Random access iterator » est nécessaire*

Les algorithmes

- Algorithmes travaillant sur des valeurs

  ⇨ std::min(a,b) : renvoie le minimum entre a et b

  ⇨ std::max(a,b) : renvoie le maximum entre a et b

  ⇨ std::swap(a,b) : échange les valeurs de a et de b

Les algorithmes

- Algorithmes ne modifiant pas les structures et leurs valeurs

  ⇨ Comptage des occurrences d'un élément

```
template <class InputIterator, class EqualityComparable, class Size>
   void count(InputIterator first, InputIterator last,
                  const EqualityComparable & value, Size & n);
```

  ⇨ Recherche d'un élément

```
template<class InputIterator, class EqualityComparable >
   InputIterator find(InputIterator first, InputIterator last,
                        const EqualityComparable & value);
```

Les algorithmes

- Recherche de l'élément minimum

```
template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);
```

- Recherche de l'élément maximum

```
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);
```

Les algorithmes

- Algorithmes modifiant les éléments

  ⇨ Remplacement

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value)
```

  ⇨ Suppression

```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first,
                       ForwardIterator last,
                       const T& value);
```

Les algorithmes

- Algorithmes modifiant l'ordre des éléments

  ⇨ Tri (implémentation du tri rapide)

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

  ⇨ Fusion de deux structures triées

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                     InputIterator2 first2, InputIterator2 last2,
                     OutputIterator result);
```

Les algorithmes

⇨ Copie

```
template <class InputIterator, class OutputIterator >
OutputIterator copy(InputIterator first, InputIterator last,
                                    OutputIterator result);
```

Les algorithmes

- Algorithmes sur les ensembles
- Union ensembliste

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, InputIterator2 last2,
        OutputIterator result);
```

- Intersection ensembliste

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator set_intersection(InputIterator1 first1,
    InputIterator1 last1, InputIterator2 first2, InputIterator2 last2,
    OutputIterator result);
```

Les algorithmes

- Algorithmes sur les ensembles

  ⇨ Différence ensembliste

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator >
OutputIterator set_difference(InputIterator1 first1,
                              InputIterator1 last1,
                              InputIterator2 first2,
                              InputIterator2 last2,
                              OutputIterator result);
```

La STL : exemple

- Tri, copie et recherche

```
{   std::vector<Rationnel> vecteurRat ;
    std::list<Rationnel> listeRat ;
    // … Initialisation du vecteur
    // Tri du vecteur
    std::sort(vecteurRat.begin(), vecteurRat.end()) ;
    // Copie du vecteur trié dans une liste
    std::copy(vecteurRat.begin(), vecteurRat.end(),
    std::front_inserter(listeRat)) ;
    // Recherche d'un élément dans la liste
    std::list<Rationnel>::iterator f = find(listeRat.begin(), listeRat.end(),
                                            Rationnel(1,3)) ;
    if(f!=listeRat.end())
    { std::cout<<« Rationnel trouvé !!! »<<std::endl ; }
}
```

La STL : conclusion

- Collections de structures de données standards
  - ⇨ Utilisation intensive de la généricité
    - *Souplesse d'utilisation*
  - ⇨ Implémentations efficaces
    - *Complexité maîtrisée et spécifiée*
    - *Algorithmes et structures testés et fiables*
      - Une exception : Visual C 6.0 (Utiliser STL port)
  - ⇨ Attention au choix de la structure de données
    - *Lié à l'utilisation faite par vos algorithmes*

La STL : conclusion

- Collection d'algorithmes standards
  - ⇨ Tri, gestion de tas, min, max, échange, …
  - ⇨ Ici encore, implémentations efficaces
- Ne reprogrammez pas vos structures de données (sauf cas particulier)
  - ⇨ Utilisez la STL
- Documentation accessible à l'adresse suivante :
  - ⇨ http://en.cppreference.com/
  - ⇨ http://www.sgi.com/tech/stl/ (complexité algos fournies)
  - ⇨ Regardez la, je ne vous ai montré qu'une « toute petite » partie des capacités de cette bibliothèque !!!!!!

Les exceptions

Les exceptions

- Une *exception* est l'interruption de l'exécution du programme à la suite d'un événement particulier (assimilable à une erreur)

  - ⇨ Le but des exceptions est de réaliser des traitements spécifiques aux événements qui en sont la cause

  - ⇨ Ces traitements peuvent rétablir le programme dans son mode de fonctionnement normal, auquel cas son exécution reprend

  - ⇨ Il se peut aussi que le programme se termine, si aucun traitement n'est approprié

Les exceptions

- Lorsqu'une exception est levée
  - ⇨ L'exécution de la méthode levant l'exception est stoppée
  - ⇨ Le programme remonte la pile d'appels jusqu'à trouver une fonction / méthode dans laquelle on déclare attraper l'exception
    - *Si l'on ne rattrape pas l'exception, le programme s'arrête*
    - *Dans le cas contraire, le programme reprend un déroulement normal à partir du code attrapant l'exception*

Les exceptions

- Déclenchement

  ⇨ Syntaxe : `throw e ;`

  ⇨ *e est une expression d'un type quelconque dont la valeur sera l'exception levée*

  - Type prédéfini
    - throw int(15)
  - Type défini par l'utilisateur
    - class Toto {…} ;
    - throw Toto() ;

Les exceptions

- Traitement d'une exception

  - ⇨ Syntaxe

```
try
{
    // Code déclenchant potentiellement une exception
}
// Rattrapage de toute exception de type TypeException1
catch(TypeException1 e1)
{ /*  Traitement de e1 */ }
…
// Rattrapage de toute exception de type TypeExceptionN
catch(TypeExceptionN eN)
{ /* Traitement de eN */ }
```

Exemple

```
// Méthode levant une exception
Rationnel::Rationnel(int num, int den)
    : numerateur(num), denominateur(den)
{
    if(denominateur==0)
    {
        throw std::string(«  Dénominateur NUL ! ») ;
    }
}
```

Exemple

```
// Exemple de programme
{
    Rationnel r, inv ;
    try {
            inv = r.inverse();
            // Traitement suivant le calcul d'inverse
    }
    // Attrape toutes les exceptions de type std::string
    catch(const std::string & erreur)        {
            std::cerr << erreur << std::endl ;
    }
}
```

Exceptions

- Remarque sur l'exemple précédent

  ⇨ Le type chaîne de caractères permet difficilement de différencier les exceptions

  - *Il est donc difficile d'appliquer un traitement différencié*

- Préférer l'utilisation de classes et d'héritage pour spécialiser les exceptions et effectuer un meilleur filtrage

- Ce mécanisme est peu coûteux en vérifications

  ⇨ Utilisez le tant que faire se peut

Foncteurs et lambda fonctions

Foncteur : définition

- *Un foncteur est un objet qui se comporte comme une fonction*

Notion de foncteur

- En C++, l'opérateur () est surchargeable
  - ⇨ Opérateur fournissant la syntaxe d'un appel de fonction sur une instance de classe

```
template <class T> class Compare
{       // Définition d'un opérateur () comparant deux instances de T avec <
        bool operator() (T const & v1, T const & v2) const
        { return v1<v2; }
} ;

void exemple()
{
        Compare<int> comparateur; // Création d'une instance de comparateur
        bool test = comparateur(10,20); // Utilisation de l'opérateur ()
}
```

Les possibilités offertes

- Un foncteur est une instance de classe.
- Il peut posséder :
  - ⇨ des attributs
    - *Permet de capter un contexte (variables locales, attributs…)*
    - *Permettent de configurer le comportement de l'instance*
  - ⇨ un constructeur
  - ⇨ *son comportement est paramétrable tout comme celui de n'importe quelle classe*
- Il doit posséder
  - ⇨ Un opérateur () redéfinit
    - *La signature de cet opérateur donne la signature de la fonction simulée*
- Il est utilisable partout où un appel de fonction (bien typé) peut être effectué

Foncteur : utilisation d'attributs

```
template <class T> class CompareInfSup
{
    bool m_inf ; // Ordre associé (true : utilisation de '<', false : utilisation de '>')
public:
    CompareInfSup(bool inf) : m_inf(inf) {}

    bool operator() (const T & v1, const T & v2) const   {
        if(m_inf) { return v1<v2 ; }
        else { return v1>v2 ; }
    }
};

void exemple()
{
    CompareInfSup<int> comparateur(true); // Création du comparateur (avec l'ordre en
paramètre)
    bool test = comparateur(10,20); // Utilisation de l'opérateur ()
}
```

Foncteurs et STL

- La STL fournit des foncteurs
  - ⇨ Opérations arithmétiques
    - *plus, minus, multiplies, divides, modulus, negate*
  - ⇨ Opérations de comparaison
    - *equal_to, not_equal_to, greater, less, greater_equal, less_equal*
  - ⇨ Opérations logiques
    - *logical_or, logical_and, logical_not*

Foncteurs et STL

- Exemples (implémentations de la STL)

```
template <class T> struct plus : binary_function <T,T,T> {
  T operator() (const T& x, const T& y) const {return x+y;}
};
```

```
template <class T> struct less : binary_function <T,T,bool> {
  bool operator() (const T& x, const T& y) const {return x<y;}
};
```

```
template <class T> struct logical_and : binary_function <T,T,bool> {
  bool operator() (const T& x, const T& y) const {return x&&y;}
};
```

Foncteurs et STL

- Les foncteurs sont utilisés par certaines structures de données pour fournir des comparateurs
  - std::set<Key, Compare>
  - std::map<Key, Value,Compare>
  - std::multiset<Key, Compare>
  - std::multimap<Key,Value,Compare>
  - Attention : seul le type est fourni, pas l'instance.
    Le foncteur **doit** donc posséder un constructeur par défaut.

- Compare : type générique devant implémenter un operateur () binaire permettant de comparer des instances de Key
  - Signature : bool operator() (const Key &, const Key &) const
  - std::less<Key> et std::greater<Key> peuvent être utilisés
  - Par defaut : std::less<Key>

Foncteurs et STL

- La STL utilise les foncteurs dans différents algorithmes

  ⇨ Algorithme de tri paramétré par un ordre

```
template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp);
```

  ⇨ Application d'une fonction sur chaque élément d'un conteneur

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

  ⇨ Et bien d'autres… (allez voir ce que propose la STL ☺)

Foncteurs et STL : exemple

- Foncteur mettant au carré son paramètre

```
template <class T>
class SquareValue
{
public:
        void operator() (T & value) const
        {
                value = value * value ;
        }
};
```

Foncteurs et STL : exemple

```
void exempleFoncteur2()
{
  ::std::vector<int> tab(10) ;
  // Remplissage du tableau avec des nombres aléatoires
  for(unsigned int cpt=0 ; cpt<tab.size() ; ++cpt)
  { tab[cpt] = rand() ; }
  // Tri dans l'ordre decroissant
  ::std::sort(tab.begin(), tab.end(), ::std::greater<int>()) ;
  // Tri dans l'ordre croissant (eq. CompareInf<int>() eq. à std::less<int>())
  ::std::sort(tab.begin(), tab.end(), CompareInfSup<int>(true)) ;
  // monte au carré tous les éléments du tableau
  ::std::for_each(tab.begin(), tab.end(), SquareValue<int>()) ;
}
```

Les lambdas fonctions

- Description de fonctions anonymes
  - ⇨ Peuvent être directement passées en paramètre d'algorithmes
  - ⇨ Syntaxe « simple » permettant d'écrire des foncteurs rapidement
- Syntaxe:

  [capture] (paramètres) -> typeRetour { corps fonction }

  - ⇨ capture : ensemble des variables dépendantes du contexte courant utilisées / utilisables dans la lambda fonction
  - ⇨ paramètres : paramètre(s) de la fonction sous la forme type identifiant classique
  - ⇨ typeRetour : le type de retour de la lambda fonction
  - ⇨ corps fonction : le code associé à la lambda fonction

Lambda fonctions : capture

[]	Ne capture rien
[&]	Capture les variables référencées par référence
[=]	Capture les variables référencées par copie
[&bar]	Capture la variable bar par référence
[bar]	Capture la variable bar par copie
[=, &foo]	Capture les variables référencées par copie, mais capture la variable foo par référence
[this]	Capture le pointeur this, la lambda fonction se comporte comme une méthode de la classe

Exemple

```
void exempleLambdaFonction()
{
  unsigned int multiplicateur = 10 ;
  ::std::vector<int> tab(10) ;
  // Remplissage du tableau avec des valeurs aléatoires en utilisant une lambda fonction
  ::std::for_each(tab.begin(), tab.end(), [](int & value) { value = rand() ; }) ;

  ::std::for_each(tab.begin(), tab.end(), [](const int & value) {::std::cout<<value<<" " ; }) ;

  ::std::cout<<::std::endl ;
  // Multiplication par multiplicateur de la valeur des éléments contenus dans le tableau
  ::std::for_each(tab.begin(), tab.end(),
                                  [&multiplicateur](int & value) { value = value*multiplicateur ; }) ;
 ::std::for_each(tab.begin(), tab.end(), [](const int & value) {::std::cout<<value<<" " ; }) ;

  ::std::cout<<::std::endl ;
}
```

Exemple

- La partie de code suivante :

```
::std::for_each(tab.begin(), tab.end(),
                [&multiplicateur](int & value) { value = value*multiplicateur ; }) ;
```

- Aurait pu être programmée comme suit :

```
// foncteur ayant un comportement équivalent à la lambda fonction
class Anonymous {
   unsigned int & multiplicateur ;
public:
   Anonymous(unsigned int & pMultiplicateur)
     : multiplicateur(pMultiplicateur) {}
   void operator() (int & val) const
   { val = val * multiplicateur ; }
};
```

```
// Utilisation du foncteur à la place de la lambda fonction
::std::for_each(tab.begin(), tab.end(), Anonymous(multiplicateur)) ;
```

Lambda fonctions

- Une lambda fonction est
  - ⇨ Une fonction anonyme
  - ⇨ Compilée sous la forme d'un foncteur
  - ⇨ Nous ne connaissons pas le nom de son type
- Une variable peut-elle désigner une lambda fonction ?
  - *Oui*
- Les types à utiliser
  - ⇨ auto : laisse le compilateur déduire le type de la variable en fonction de l'expression qui y est associée.
  - ⇨ ::std::function<R (Args…)> : classe générique permettant de désigner une fonction qui a R comme type de retour et Args comme type d'arguments.
    La syntaxe d'appel est la même que celle de la fonction

Exemple

```
void exempleLambdaFonction2()
{
    unsigned int multiplicateur = 10 ;
    ::std::vector<int> tab(10) ;
    // Déduction automatique du type de la variable
    auto random = [](int & value) { value = rand() ; } ;
    auto print = [](const int & value) {::std::cout<<value<<" " ; } ;
    // Utilisation de la classe std::function
    ::std::function<void (int &)> mult = [&multiplicateur](int & val) { val = val*multiplicateur ; } ;
    // Remplissage du tableau avec des valeurs aléatoires en utilisant une lambda fonction
    ::std::for_each(tab.begin(), tab.end(), random) ;
    ::std::for_each(tab.begin(), tab.end(), print) ;
    // Multiplication par multiplicateur de la valeur des éléments contenus dans le tableau
    ::std::for_each(tab.begin(), tab.end(), mult) ;
    ::std::for_each(tab.begin(), tab.end(), print) ;
}
```

Retour sur std::function<R (Args…)>

- Une instance de std::function<R (Args…)> peut désigner :
  - ⇨ Toute fonction ayant la signature *R (Args…)*
  - ⇨ Toute lambda fonction ayant la signature *R (Args…)*
  - ⇨ Tout foncteur possédant un operateur () ayant la signature *R (Args…)*

Exemple

```
// Fonction C++ comparant deux entiers
bool compare_fonction(int v1, int v2)
{ return v1<v2 ; }

// Foncteur C++ comparant deux entiers
struct Compare_foncteur {
    bool operator()(int v1, int v2) const
    { return v1<v2 ; }
};

void testFunction() {
    // L'instance de std::function désigne une fonction C++
    ::std::function<bool (int, int)> cmp_fnc = compare_fonction ;
    // L'instance de std::function désigne une instance de foncteur
    ::std::function<bool (int, int)> cmp_foncteur = Compare_foncteur() ;
    // L'instance de std::function désigne une lambda fonction comparant deux entiers
    ::std::function<bool (int, int)> cmp_lambda = [](int v1, int v2) -> bool { return v1<v2 ; } ;
}
```

Fonctions / Foncteurs en paramètre

- Passage d'une fonction / d'un foncteur en paramètre :
  - ⇨ Utilisation de la généricité
  - ⇨ Utilisation d'une instance de std::function<R (Args…)>

- Généricité
  - ⇨ *Solution utilisée dans la STL*
  - ⇨ Avantage : performance car pas d'indirection
  - ⇨ Inconvénient : pas de compilation séparée

- Utilisation de std::function<R (Arg…)>
  - ⇨ Avantage : possibilité de compilation séparée
  - ⇨ Inconvénient : impact sur les performances car implique des indirections

Exemple

- Utilisation de la généricité (STL like)

```
// Teste si un tableau est trié en utilisant un comparateur
template <class Comparator>
bool is_sorted(const ::std::vector<int> & tab, Comparator comp)
{
    for(unsigned int cpt=1 ; cpt<tab.size() ; ++cpt)
    {
        if(!comp(tab[cpt-1], tab[cpt])) { return false ; }
    }
    return true ;
}
```

Exemple

- Utilisation de std::function<R (Args…)>

```
bool is_sorted(::std::vector<int> const & tab, std::function<bool (int, int)> comp)
{
    for(unsigned int cpt=1 ; cpt<tab.size() ; ++cpt)
    {
        if(!comp(tab[cpt-1], tab[cpt])) { return false ; }
    }
    return true ;
}
```

Exemple

- Fonction de test compatible avec les deux implémentations

```
void test()
{
    ::std::vector<int> tab ;
    // Utilisation de la fonction de comparaison
    bool s = is_sorted(tab, compare_fonction) ;
    // Utilisation du foncteur de comparaison
    bool s2 = is_sorted(tab, Compare_foncteur()) ;
    // Utilisation d'une lambda fonction pour la comparaison
    bool s3 = is_sorted(tab, [](int v1, int v2) -> bool { return v1<v2 ; }) ;
}
```

Et une fonction qui renvoie un traitement ???

- Mais bien sûr !

```
// Fonction retournant une fonction calculant la somme des éléments de tab
::std::function<int ()> createSum(const ::std::vector<int> & tab)
{
    // Création d'une lambda fonction calculant la somme des éléments de tab
    auto func = [&tab]() -> int
    {
        int result = 0 ;
        for(unsigned int cpt=0 ; cpt<tab.size() ; ++cpt)
        { result += tab[cpt] ; }
        return result ;
    } ;
    return func ; // Retourne la lambda fonction créée
}
```

Et une fonction qui renvoie un traitement???

- Exemple d'utilisation

```
void testCreateSum()  {
    ::std::vector<int> toto ; // Un tableau qui contiendra deux valeurs
    toto.push_back(10) ;
    toto.push_back(11) ;
    // Création de la fonction de calcul de somme
    ::std::function<int ()> sum = createSum(toto) ;
    int s1 = sum() ; // s1 = 10+11 = 21
    toto.push_back(21) ; // Modifie le tableau en ajoutant une valeur
    int s2 = sum() ; // s2 = 10+11+21 = 42 (évalue la somme)
}
```

Conclusion

- Les foncteurs et les lambda fonctions offrent des mécanismes permettant
  - ⇨ Une grande généricité des traitements
    - Un traitement / une classe peut être paramétré par un traitement
  - ⇨ Utilisés par beaucoup d'algorithmes de la STL
    - Structures de données triées : *std::map, std::set*
    - Fonctions de parcours, de tri etc : *std::for_each, std::sort*
- La classe std::function<R (Args…)>
  - ⇨ Généralisation des pointeurs sur fonction (non présentés dans ce cours)
  - ⇨ Grande flexibilité : peut désigner une lambda fonction, une fonction, un foncteur…
- Jouer avec tout cela : demande de bien réfléchir…

Conclusion globale

- Le C++
  - ⇨ Avantage majeur : la rapidité du code à l'exécution
  - ⇨ Inconvénient majeur : un certain manque de sécurité
- Alors C++ ou Java ? ☺
  - ⇨ Réponse de Normand : tout dépend…
- Ai-je abordé l'intégralité du C++ ?
  - ⇨ Non, il vous reste encore quelques notions à acquérir
    - *C++ 11, C++ 14, C++ 20 apportent leur lot de nouvelles fonctionnalités !*
- *Mais surtout : de l'expérience !!!!*

Exemple complet (work)

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

/*-------------------------------------------------------------
Prototypes des fonctions
-------------------------------------------------------------*/
int lireTableau (int valeurs[], int nmax);
float calculerMoyenne (const int valeurs[], int compteur);
void afficherValeurs (const int valeurs[], int compteur);
void afficherMoyenne (float moyenne);
int lireNombre (void);

/*-------------------------------------------------------------
Calculer la moyenne d'une suite d'entiers positifs
-------------------------------------------------------------*/
int main(void)
{
  // saisir la taille du tableau
  int tailleTableau; // taille du tableau
  cout << "Donnez la taille du tableau, svp : ";
  cin >> tailleTableau;
  // initialiser le tableau et déterminer le nombre de nombres lus !!
  int valeurs[tailleTableau]; // tableau des nombres
  int compteur = lireTableau(valeurs, tailleTableau);
  if (compteur > 0)
  {
    float moyenne = calculerMoyenne(valeurs, compteur); // Calculer la moyenne
    afficherValeurs(valeurs, compteur); // Afficher les valeurs lues
    afficherMoyenne(moyenne); // Afficher le résultat
  }
  return 0;
}

/*-------------------------------------------------------------
Lire une suite d'entiers positifs terminée par zéro et les mettre dans un tableau
Parame`tres donnés : int nmax : nombre max de valeurs a` lire
Parame`tres modifiés : int valeurs[]: tableau des valeurs
```

Exemple Vecteur

```
#ifndef __Vecteur_H
#define __Vecteur_H

class Vecteur
{
public:
  // Un constructeur ; initialisation avec 0, 1, 2 ou 3 réels
  Vecteur(double v1 = 0, double v2 = 0, double v3 = 0);
  // Le destructeur
  ~Vecteur(void);
  // Affichage d'un Vecteur
  void    affiche(void)     const;
  // Produit scalaire de Vecteurs
  double  prodscal(Vecteur v) const;
  // Somme de 2 Vecteurs
  Vecteur somme(Vecteur v)    const;
  // Somme d'un Vecteur et d'un nombre
  Vecteur somme(double n)     const;
  // Calcul de l'homothétie du vecteur
  void homothetie(double valeur);
  // Somme de deux Vecteurs
  static Vecteur somme(Vecteur, Vecteur);
private:
  // Les attributs : les 3 coordonnées
  double x, y, z;
};

#endif
```

Exemple vecteur (2)

```
#include <iostream>
#include "Vecteur.h"
using std::cin;
using std::cout;

// constructeur : initialisation avec 0, 1, 2 ou 3 réels
Vecteur::Vecteur(double v1, double v2, double v3)
  :  x(v1), y(v2), z(v3)
{
  cout << "construit " ; affiche();                    // pour info
}

// destructeur
Vecteur::~Vecteur()
{
  cout << "détruit   " ; affiche();                    // pour info
}

// Affichage d'un Vecteur
void Vecteur::affiche(void) const
{
  cout << " " << "[ " << x << " " << y << " " << z << " ]\n";
}

// Produit scalaire de 2 Vecteurs
double Vecteur::prodscal(Vecteur v) const
{
  return x * v.x + y * v.y + z * v.z;
}

// Somme de 2 Vecteurs
Vecteur              Vecteur::somme(Vecteur v) const
{
  return Vecteur(x + v.x, y + v.y, z + v.z);
}
```

Exemple Vecteur (3)

```
#include <iostream>
#include "Vecteur.h"          // classe Vecteur
using std::cin;
using std::cout;

// petit programme de test de la classe
// Tous les commentaires à l'intérieur de la fonction
// main correspondent aux messages affichés à l'écran
int    main(void) {

  Vecteur v1, v2(1, 2, 3);
  // construit [ 0 0 0 ]    : v1
  // construit [ 1 2 3 ]    : v2
  cout << "v1      : ";   v1.affiche();
  // v1      : [ 0 0 0 ]
  cout << "v2      : ";   v2.affiche();
  // v2      : [ 1 2 3 ]
  v2.homothetie(2);
  cout << "v2 *= 2 : ";   v2.affiche();
  // v2 *= 2 : [ 2 4 6 ]
  Vecteur v3(3, 2, 1);
  // construit [ 3 2 1 ]
  cout << "v3      : ";   v3.affiche();
  // v3      : [ 3 2 1 ]
  cout << "v2 x v3 : " << v2.prodscal(v3) << "\n";
  // v2 x v3 : 20
  // détruit   [ 3 2 1 ]    : paramètre de prodscal
  Vecteur v4 = v2.somme(v3);
  // construit [ 5 6 7 ]
  // détruit   [ 3 2 1 ]    : paramètre de somme
  cout << "v2 + v3 : ";   v4.affiche();
  // v2 + v3 : [ 5 6 7 ]
  v4 = v4.somme(10);
  // construit [ 15 16 17 ] : objet local de somme
  // détruit   [ 15 16 17 ] : objet local de somme
  cout << "v4 + 10 : " ; v4.affiche();
```