Plan général

- Partie 1 : Modélisation orientée objet
  - Génie logiciel
  - UML
- Partie 2 : Programmation orientée objet
  - Le Langage C++
  - Interfaces Homme-Machine

POLYTECH
CLERMONT-FERRAND

Partie 2 : Programmation Orientée Objet

Le langage C++

Marinette Bouet & Christophe de Vaulx

Plan du cours

- Introduction
- Un C ANSI amélioré
- Classes et objets
- Polymorphisme (surcharge)
- Généricité
- Exceptions
- Héritage
- Polymorphisme (redéfinition)
- Classe abstraites
- Les fichiers
- La STL

POLYTECH
CLERMONT-FERRAND

Introduction

Historique

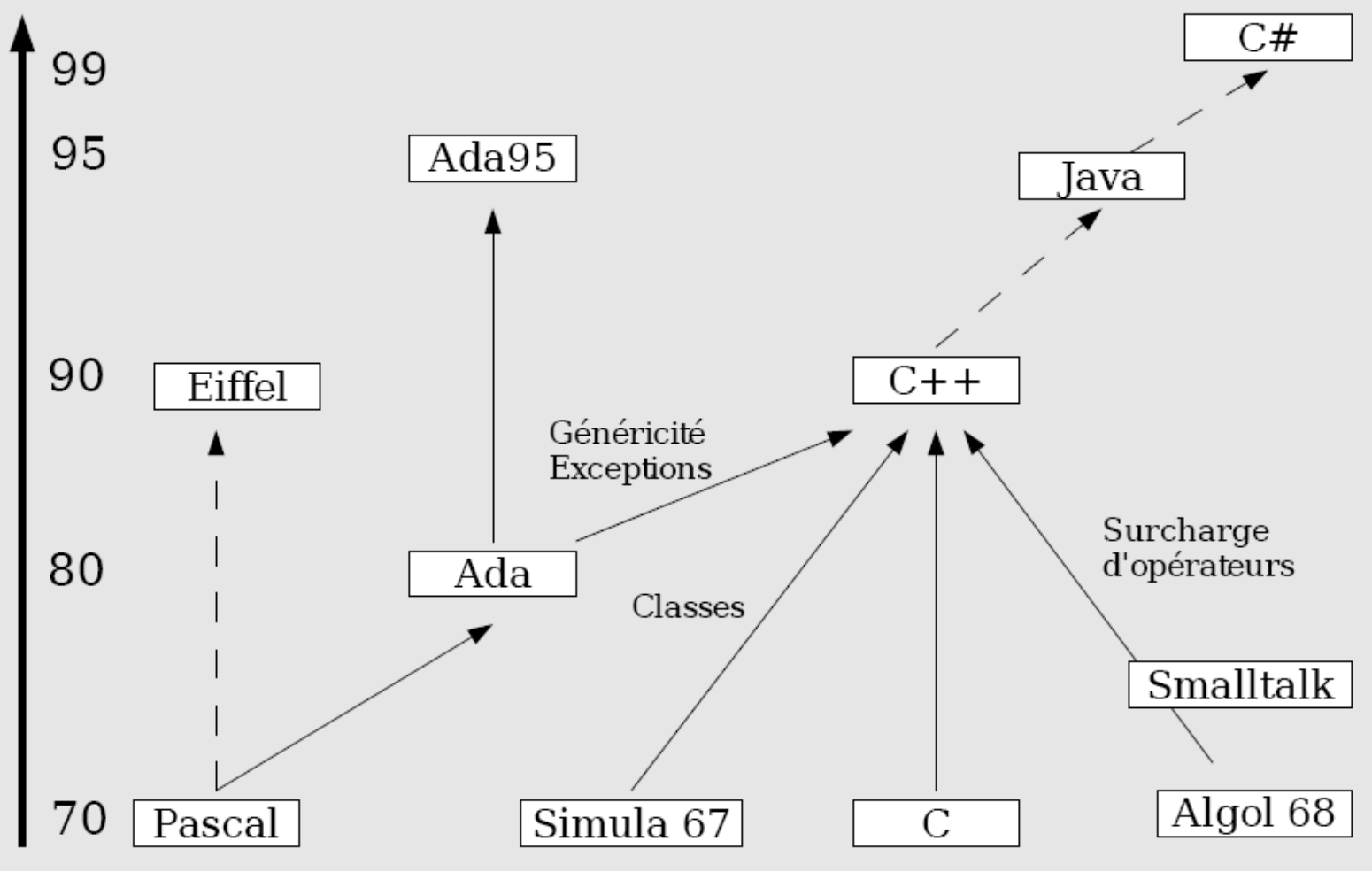
- Créé par B. Stroustrup (Bell Labs.) à partir de 1979 (“C with classes”).
- Initialement: code C++ pré compilé → code C
- Devient public en 1985 sous le nom de C++.
- La version normalisée (ANSI) paraît en 1996.

Historique

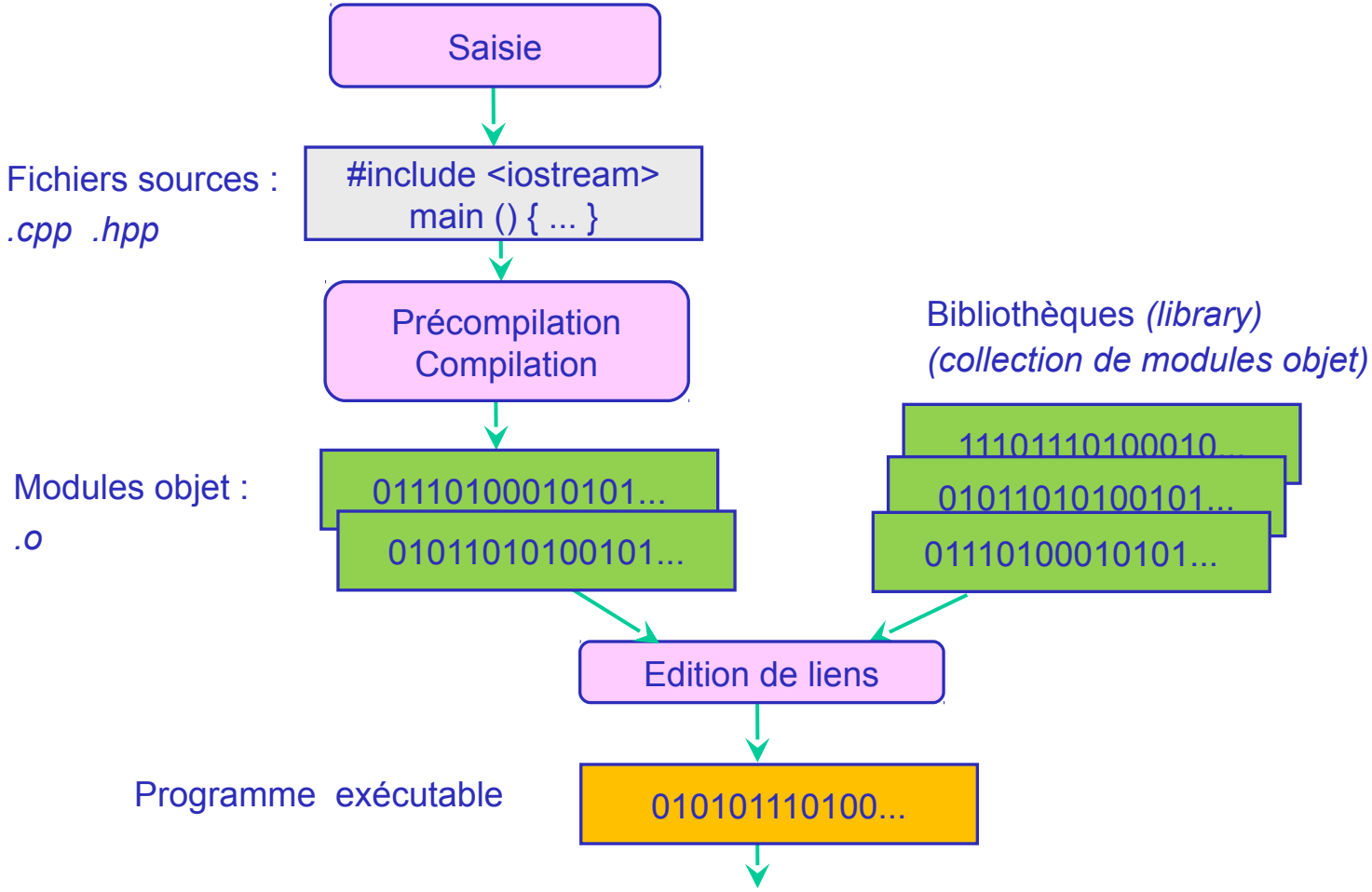
C++ = C +

- Vérifications de type + stricte
- Surcharge de fonctions
- Références
- Gestion mémoire + facile
- Entrées/sorties + facile
- Programmation Orientée Objet (classes, héritage, généricité,…)

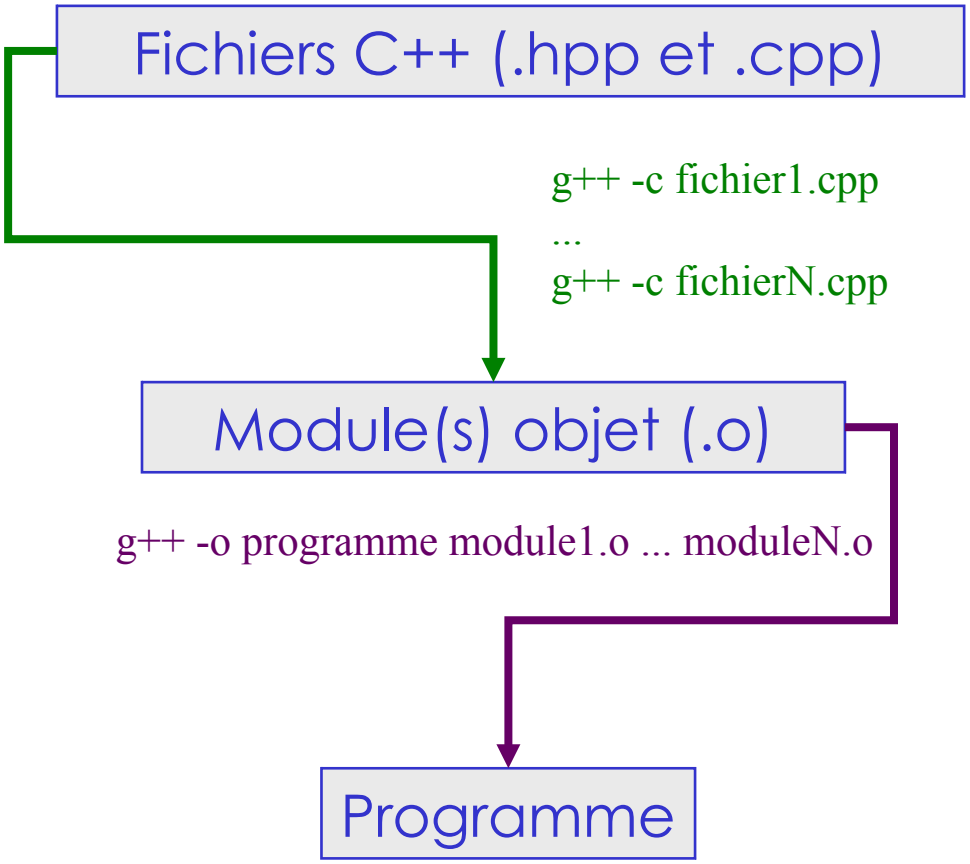
Historique


99
95
90
80
70
C#
Ada95
Java
Eiffel
C++
Généricité
Exceptions
Surcharge
d'opérateurs
Ada
Classes
Smalltalk
Pascal
Simula 67
C
Algol 68

Un langage compilé…



Un langage compilé…



POLYTECH
CLERMONT-FERRAND

Un C ANSI amélioré

Commentaires

- Il existe deux types de commentaires en C++ :
  - Les commentaires de type C (/* … */) .
  - Les commentaires de fin de ligne (//…) qui indiquent au compilateur de tout ignorer jusqu'à la fin de la ligne.

```
/* Paramètres de la fontion main :
 *  argv : tableau de chaînes de caractères contenant le
 *         nom du programme et la liste des paramètres qui
 *         sont passés au programme via la ligne de commande
 *  argc : nombre d'éléments du tableau argv
 */
int main(int argc, char** argv)
{
  // argv[0] = nom du programme
  // argv[1] → argv[argc-1] = paramètres du programme

  return 0;
}
```

Définition des constantes

- En C, les constantes peuvent être définies à l'aide de la directive du pré-processeur #define.

- En C++ cette façon de procéder n'est plus valable. Pour définir une constante, on utilise uniquement le mot clé **const** :

```
const int N = 10; // N est une constante entière.
```

Déclaration des variables

- En C++, les variables peuvent être déclarées n'importe où dans le code.
- Une variable peut être utilisée à partir de l'endroit où elle a été déclarée jusqu'à la fin du bloc courant.

```
int main()
{
    int a=3 ;
    a++;
    int b=5; // Déclaration d'une variable au milieu du code...
    b=a;
    //...
    return 0;
}
```

Variables de boucle

- En C++, il est possible de déclarer des variables dans l'instruction for elle-même. On appelle ces variables des variables de boucle.

```
int main()
{
  int tab[10];

  for (int i=0; i<10; i++) // Déclaration d'une variable de boucle
  {
      tab[i]=0;
  }
  // i n'est plus accessible en dehors de la boucle...

  return 0;
}
```

Les types composés

- Tout comme en C, il possible en C++, de définir des types composés à l'aide des mots clés struct, enum ou union.
- L'avantage est qu'en C++, il n'y a plus besoin d'utiliser le mot clé typedef.

```
enum booleen { FAUX, VRAI };
typedef enum booleen BOOLEEN;

//…

BOOLEEN trouve;
```

En C

En C++

```
enum BOOLEEN { FAUX, VRAI };

//…

BOOLEEN trouve;
```

Les types composés

```
struct FICHE {
  char *nom, *prenom;
  int  age;
};

//…

FICHE f;
```

En C++

En C

```
struct fiche {
  char *nom, *prenom;
  int  age;
};
typedef struct fiche FICHE;

//…

FICHE f;
```

Le type « booléen »

- Contrairement au langage C, le C++ possède un type de base qui permet de gérer les variables booléennes.

- Pour définir une variable booléenne en C++, on utilise le mot clé bool.

```
// Déclaration et initialisation de variables booléennes.
bool i = true;
bool j = false;

bool k = 1; // 1 <-> true
bool j = 0; // 0 <-> false ;
```

Les entrées/sorties

- En C++, les opérations d'écriture et de lecture s'effectuent à l'aide des flots d'entrée et de sortie (**stream**) définis dans la **STL** (Standard Template Library) du C++ :
  - Le flot correspondant à la sortie standard (écran) s'appelle **cout**.
  - Le flot correspondant à l'entrée standard (clavier) s'appelle **cin**.

Les entrées/sorties

- Pour interagir avec ces flots, on utilise deux opérateurs :
  - L'opérateur << qui permet d'envoyer des valeurs dans un flot de sortie.
  - L'opérateur >> permet d'extraire des valeurs d'un flot d'entrée.
- Attention : pour utiliser les flots, il ne faut pas oublier d'inclure le fichier **iostream**.

Les entrées/sorties

- Exemple :

```
#include <iostream>

int main() {
  int i;

  // Affichage d'un message à l'écran.
  std::cout << "Entrez un entier : " << endl;

  // Lecture de la valeur saisie au clavier.
  std::cin >> i;

  return 0;
}
```

Les entrées/sorties

- Pour ne pas avoir à spécifier que cin et cout appartiennent à l'espace de nom standard de la STL, il faut ajouter au début du programme l'instruction « using namespace std » :

```
#include <iostream>

using namespace std ;

int main() {
  int i;

  // Affichage d'un message à l'écran.
  cout << "Entrez un entier : " << endl;

  // Lecture de la valeur saisie au clavier.
  cin >> i;

  return 0;
}
```

Les chaînes de caractères

- La STL propose fournit également le type string qui permet de manipuler les chaînes de caractères beaucoup facilement qu'en C…

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

using namespace std;

int main() {
  string s = "Bonjour";
  s = s + " tout le monde !\n" // Concaténation de 2 chaines
  cout << s;  // Affichage de Bonjour tout le monde !
  return 0;
}
```

Allocation de mémoire

- En C++, l'allocation / la libération d'une variable simple s'effectue via les opérateurs new et delete.

- L'allocation / la libération d'un tableau s'effectue quant à elle à l'aide des opérateurs new[] et delete[].

Allocation de mémoire

- Exemples :

```
// Allocation dynamique d'un entier.
int *ptrEntier = new int;

*ptrEntier = 10 ;

// Libération d'un entier.
delete ptrEntier;

/*****************************************************/

// Allocation d'un tableau de 10 entiers.
int *tab = new int[10];

tab[0] = 20 ;

// Libération d'un tableau d'entiers.
delete[] tab;
```

Les références

- En plus des variables classiques et des pointeurs, le C++ permet de manipuler un autre type de variables : les références.

- Une référence peut être vue comme l'"**alias**" d'une variable. C'est une deuxième façon de désigner une variable.

Les références

- Remarques :
  - Toute modification du contenu de la référence affecte le contenu de la variable référencée ;
  - Une référence doit obligatoirement être initialisée ;
  - Le type d'une référence doit être le même que celui de la variable référencée.
- Pour déclarer une référence on utilise le caractère spécial &.

Les références

```
int i;
int& ir = i; // déclaration d'un alias de la variable i.


ir = 2; // initialisation  de i à 2.
```

- Les références sont souvent employées lors de la transmission d'objets en arguments aux fonctions ou aux méthodes…

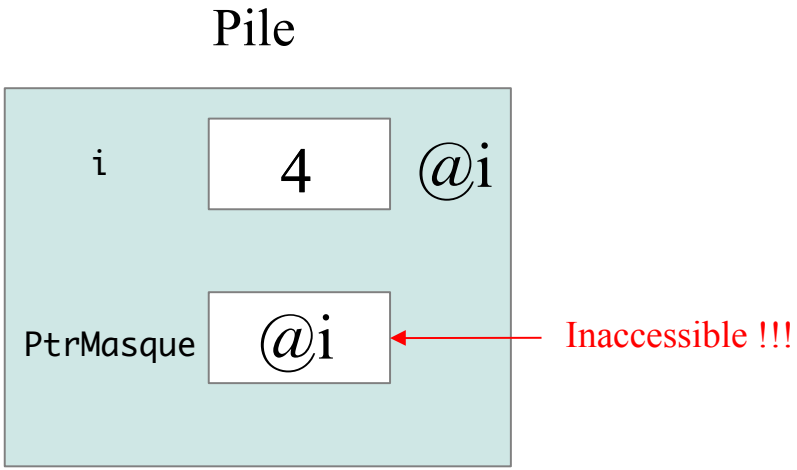
Les références

- Fonctionnement interne :
  - Lorsque l'on déclare une référence dans une fonction, le compilateur crée un pointeur caché sur la pile de la fonction. Ce pointeur est initialisé avec l'adresse de la variable référencée ;
  - Lors de la compilation du programme, toutes les occurrences d'une référence au sein des instructions sont remplacées par des occurrences au pointeur masqué ;
  - Tout est transparent pour l'utilisateur qui n'a pas connaissance et pas accès au pointeur masqué.

Les références

– Exemple :

```
int i = 4 ;
int& refi = i ;
```



Instructions saisies par l'utilisateur		Instructions générées par le compilateur		
`refi = 20;`	⟺	`*PtrMasque = 20;`	⟺	`i = 20;`
`&refi`	⟺	`PtrMasque`	⟺	`@i`

Passages d'arguments aux fonctions

- En C++, il existe trois façons de passer les arguments aux fonctions :
  - Le passage par valeur ;
  - Le passage par adresse ;
  - Le passage par référence.

Le passage par valeur

- Identique au passage par valeur du langage C :

```
#include <iostream>

using namespace std;

void incremente(int i) { i++;} // Passage par valeur.

int main() {
  int j = 12;
  incremente(j);
  cout << j << endl ; // Affiche 12
  return 0;
}
```

Le passage par adresse

- Identique au passage par adresse du langage C :

```
#include <iostream>

using namespace std;

void incremente(int* i) { *i++;} // Passage par adresse.

int main() {
  int j = 12;
  incremente(&j);
  cout << j << endl ; // Affiche 13.
  return 0;
}
```

Le passage par référence

- Similaire à l'utilisation du mot clé var en Pascal :

```
#include <iostream>

using namespace std;

void incremente(int& i) { i++;} // Passage par référence.

int main() {
  int j = 12;
  incremente(j);
  cout << j << endl ; // Affiche 13.
  return 0;
}
```

Le passage par référence

- Le passage par référence est très utilisé en C++ pour deux raisons :
  - Les références sont plus simples à utiliser que les pointeurs ;
  - Les variables passées par références ne sont pas recopiées sur la pile de la fonction → gain de temps et de mémoire.

Passage par référence constante

- Lors d'un passage par référence, il est possible de forcer le compilateur à vérifier que la fonction ne modifie pas la variable référencée. Pour cela, on utilise le mot clé const :

```
void incremente(const int& i) // Passage par référence constante.
{
  i++; // Provoquera une erreur de compilation !!!
}
```

Retour d'une variable par référence

- En C++, une fonction peut retourner une variable par référence => cela permet de manipuler la variable référencée en dehors de la fonction...

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

int t[20];

int& nIeme(int i) { return t[i]; }  // Retourne une référence de t[i]

int main() {
  nIeme(0) = 123; // nIemme(0) est alias de t[0] => nIemme(0) = 123 <-> t[0] = 123
  nIeme(1) = 456; // nIemme(1) est alias de t[1] => nIemme(1) = 456 <-> t[1] = 456
  cout << t[0] << " " << t[1] <<endl; // Affichage de 123 456
  return 0;
}
```

Arguments par défaut

- En C++, il est possible d'associer des valeurs par défaut aux arguments d'une fonction :

```
#include <iostream>

using namespace std;

void affiche (int  i=0)
{
  cout<< i<<endl;
}

int main() {
  affiche(4); // Affiche 4.
  affiche();  // Affiche 0.
  return 0;
}
```

Arguments par défaut

- Attention : les arguments par défaut et les arguments normaux ne peuvent pas être mélangés. Il faut que les arguments par défaut soient placés après les arguments normaux !!!

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

void affiche (int  i=0, int j) // Erreur
{
  cout<< i << " " << j << endl;
}

int main() {
  affiche(1); // Erreur
  return 0;
}
```

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

void affiche (int  i, int j=0) // OK
{
  cout<< i << " " << j << endl;
}

int main() {
  affiche(1);
  return 0;
}
```

Fonctions inline

- Afin d'accélérer l'exécution des fonctions très courtes, le C++ permet de définir des fonctions inline.
- Lors de la compilation, tout appel à une fonction inline, est substitué par le code de celle ci.

```
#include <iostream>

using namespace std;

inline int carre(int n) {  return n * n; }

int main() {
  cout << carre(10) << endl;
  return 0;
}
```

Surcharge des fonctions

- En C++, on peut surcharger les fonctions, c.a.d. donner le même noms à plusieurs fonctions à condition que leurs arguments soient différents (nombre et/ou types) :

```
#include <iostream>

using namespace std;

int somme(int n1, int n2) { return n1 + n2; }
int somme(int n1, int n2, int n3) { return n1 + n2 + n3; }
float somme(float n1, float n2) { return n1 + n2; }

int main() {
  cout << "1 + 2 = " << somme(1, 2) << endl;
  cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
  cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
  return 0;
}
```

POLYTECH
CLERMONT-FERRAND

Les classes et les objets

Objets/Classe

- Un **objet** est une entité concrète ou abstraite qui regroupe des **attributs** et sur laquelle on peut appliquer un certain nombre d'**opérations** (qui en pratique sont implémentées à l'aide d'une ou plusieurs **méthodes**).

- L'ensemble formé par les valeurs des attributs d'un objet représente **l'état** de cet objet. Chaque objet possède un état qui lui est propre et qui évolue dans le temps.

Objets/Classe

- L'ensemble formé par les opérations d'un objet décrit **le comportement** de cet objet. Tous les objets appartenant à une même famille possède le même comportement.

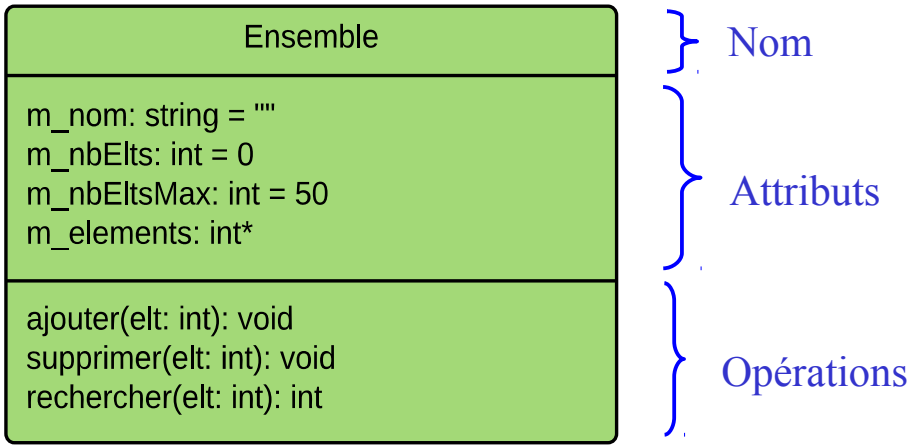
- Une **classe** est un modèle qui décrit la structure (attributs et opérations) d'une famille d'objets.

- Vocabulaire : un **objet** est une instance de sa classe.

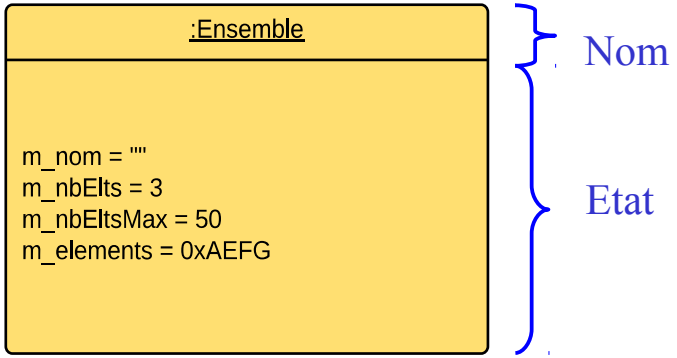
Notations UML

- Représentation d'une classe et d'un objet en UML :
- A Refaire (nom classe + noms attributs m_ !!!!!)

Une classe



Un objet



Déclaration d'une classe

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble // Déclaration d'un classe permettant de manipuler des ensembles d'entiers
{
  public:
    // Déclaration des attributs
    string m_nom;     // nom de l'ensemble
    int m_nbElts;     // nombre d'éléments présents dans l'ensemble
    int m_nbEltsMax;  // taille maximale de l'ensemble
    int* m_elements;  // tableau qui contient les éléments de l'ensemble

    // Déclaration des méthodes
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

Implémentation d'une classe

- Exemple :

```
#include "ensemble.hpp"

int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)&&(rechercher(elt)==-1))
  {
    m_elements[nbElts]=elt;
    m_nbElts++;
  }
}
```

```
void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

ensemble.cpp

- Remarque : il est possible de réunir la déclaration et l'implémentation d'une classe au sein d'un même fichier...

Instanciation d'une classe / accès aux membres d'un objet

- Exemple :

```
#include "ensemble.hpp"

int main()
{
  // Instanciation d'un objet à partir de la classe Ensemble
  Ensemble e;

  // Initialisation des attributs
  e.m_nbEltsMax = 50;
  e.m_elements = new int[50];
  e.m_nbElts = 0;
  e.m_nom = "Ensemble1";

  // Appel d'une méthode
  e.ajouter(20);

  //...


  // Libération de la mémoire occupée par le tableau
  delete[] e.m_elements;

  return 0;
}
```

prog.cpp

Instanciation dynamique / opérateur ->

- Exemple :

```
#include "ensemble.hpp"

int main()
{
  // Déclaration d'un pointeur
  Ensemble *e;

  //Instanciation dynamique d'un objet à partir de la classe Ensemble
  e = new Ensemble();

  // Initialisation des attributs
  e->m_nbEltsMax = 50;
  e->m_elements = new int[50];
  e->m_nbElts = 0;
  e->m_nom = "Ensemble1";

  // Appel d'une méthode
  e->ajouter(20);

  //...

  // Libération de la mémoire
  delete[]  e->m_selements;
  delete e;

  return 0;
}
```

prog.cpp

L'encapsulation

- Généralement les détails de la réalisation des objets sont masqués. Les objets peuvent donc être vus comme des boites noires que l'on ne peut manipuler qu'à l'aide des méthodes qu'ils proposent. C'est le principe de l'**encapsulation**.

- L'encapsulation est mise en oeuvre en réglant la **visibilité** des membres d'une classe.

Visibilité en UML

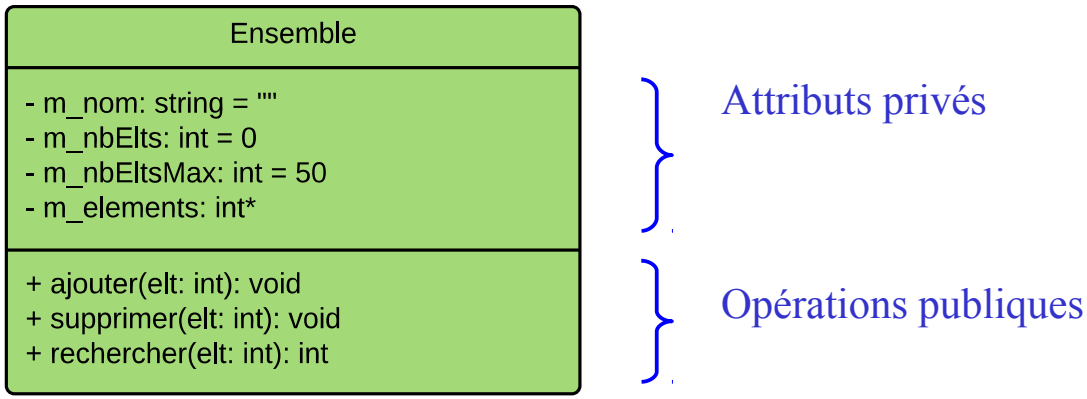
- Visibilité des membres d'une classe avec UML :

  + ↔ publique ;

  # ↔ protégé ;

  - ↔ privé.

- Retour exemple :



Visibilité des membres d'une classe

- Pour préciser la visibilité des attributs et des méthodes en C++, on utilise les modificateurs d'accès suivants :

```
private:   // les attributs et les méthodes placés après ce modificateur
           // sont utilisables uniquement par les méthodes de la classe

protected: // les attributs et les méthodes situés après ce modificateur
           // sont utilisables uniquement par les méthodes de la classe
           // et les méthodes de ses classes dérivées

public:    // les attributs et les méthodes situés après ce modificateur
           // sont utilisables par n'importe qu'elle méthode.
```

Visibilité des membres d'une classe

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    // Les attributs sont privés
    string m_nom;     // nom de l'ensemble
    int m_nbElts;     // nombre d'éléments présents dans l'ensemble
    int m_nbEltsMax;  // taille maximale de l'ensemble
    int* m_elements;  // tableau qui contient les éléments de l'ensemble

  public:
    // Les méthodes sont publiques
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

Visibilité des membres d'une classe

- Problème induit : il n'est plus possible d'initialiser les attributs des objets → les ensembles sont inutilisables !!!

```
#include "ensemble.hpp"

int main()
{
  // Instanciation d'un objet à partir de la classe Ensemble
  Ensemble e;

  // Initialisation des attributs impossible !!!
  // Ensemble inutilisable

  return 0;
}
```

prog.cpp

- Solution : les constructeurs...

Les constructeurs

- Les constructeurs sont des méthodes qui servent à initialiser les attributs des objets.
- Les constructeurs ne peuvent pas être invoqués manuellement !! Ils sont invoqués automatiquement par le système lors de la création des objets...
- En C++, un constructeur est en général une méthode publique. Il doit obligatoirement porter le nom de la classe dans laquelle il est déclarée. Il ne retourne pas de valeur.
- Une classe peut posséder plusieurs constructeurs (surcharge) mais deux sont particulièrement importants : le constructeur par défaut et le constructeur de recopie.

Le constructeur par défaut

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;

  public:
    // Constructeur par défaut
    Ensemble();
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

// Constructeur par défaut
Ensemble::Ensemble() {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = 50;
  m_elements = new int[50];
}


int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}
```

```
void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    elements[m_nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

ensemble.cpp

Le constructeur par défaut

```
#include "ensemble.hpp"

int main()
{
  // Appel du constructeur par défaut
  Ensemble e;

  e.ajouter(20);

  //...

  return 0;
}
```

prog.cpp

```
#include "ensemble.hpp"

int main()
{
  Ensemble *e;

  // Appel du constructeur par défaut
  e = new Ensemble();

  e->ajouter(20);

  //...

  return 0;
}
```

prog.cpp

Le constructeur de recopie

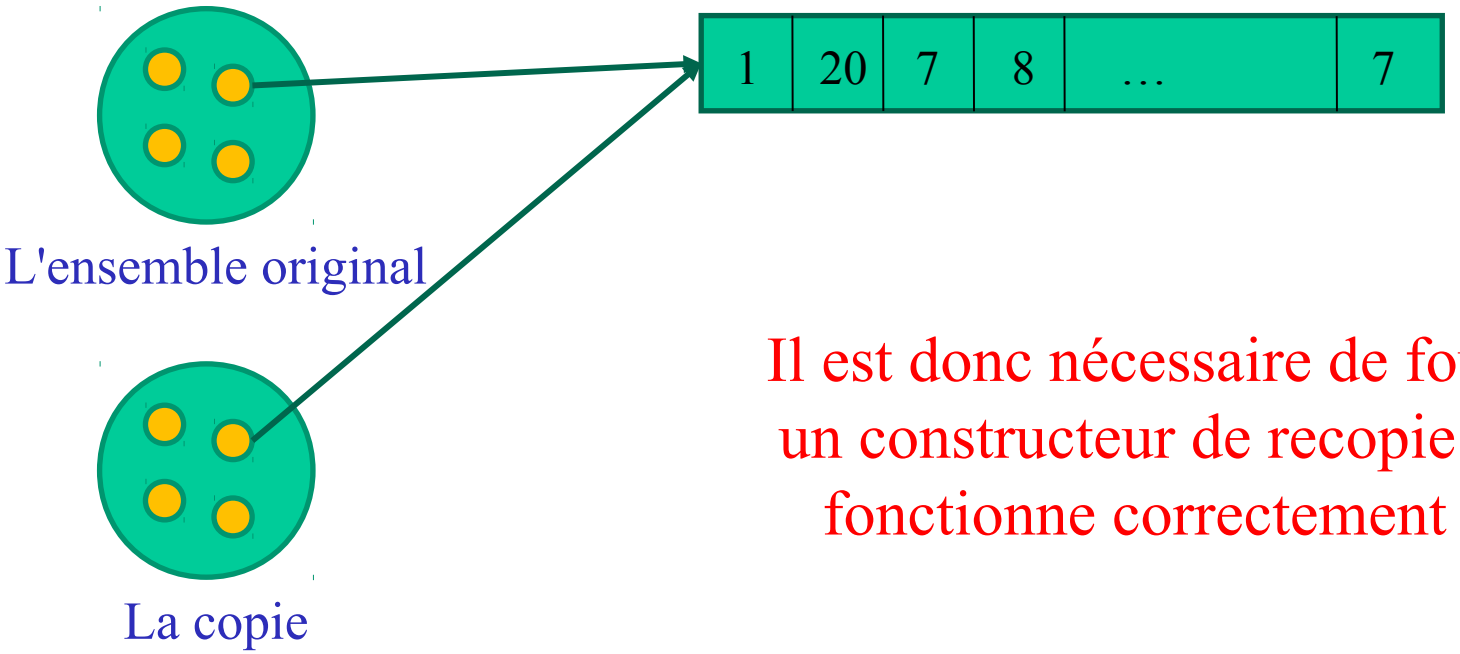
- Le constructeur de recopie permet de créer une copie d'un objet.
- Il est appelé automatiquement à chaque fois qu'une copie d'objet est réalisée en mémoire et notamment à chaque à fois que qu'un objet est transmis par valeur à une fonction, ou qu'une fonction retourne un objet de la même façon.
- Le système fourni automatiquement un constructeur de recopie. Ce constructeur par défaut recopie bit à bit les valeurs de tous les attributs de l'objet transmis en argument dans les attributs de la copie.

Le constructeur de recopie

- Attention : le constructeur de copie par défaut n'est pas toujours suffisant, il est quelquefois nécessaire de le fournir explicitement. En effet, dans le cas de références ou de pointeurs, ce n'est pas la valeur de la donnée membre elle-même qui sera recopiée mais la valeur du pointeur sur cette donnée...

Le constructeur de recopie

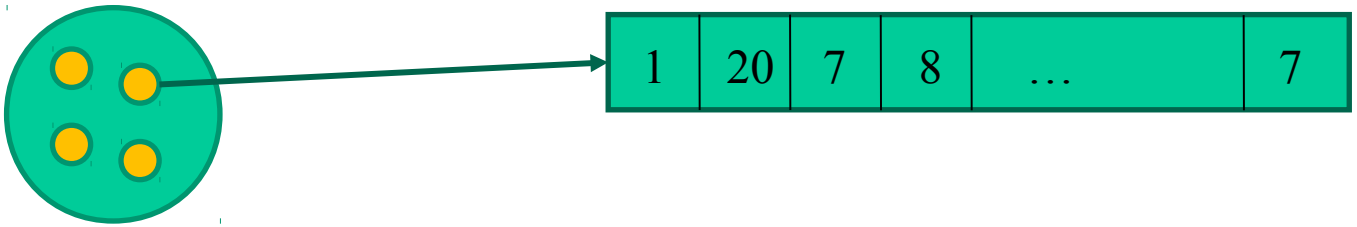
- Fonctionnement du constructeur de recopie par défaut sur notre classe Ensemble :



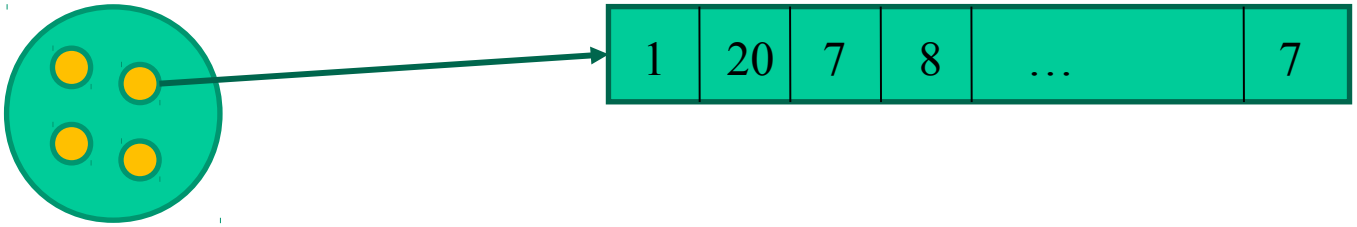
Il est donc nécessaire de fournir un constructeur de recopie qui fonctionne correctement !!!

Le constructeur de recopie

- Fonctionnement attendu du constructeur de recopie sur notre classe Ensemble :



L'ensemble original



La copie

Le constructeur de recopie

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;

  public:
    Ensemble();
    // Constructeur de recopie
    Ensemble(const Ensemble& e);
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

Ensemble::Ensemble() {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = 50;
  m_elements = new int[50];
}

// Constructeur de recopie
Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
}
```

```
int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[m_nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

ensemble.cpp

Constructeur supplémentaire

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;

  public:
    Ensemble();
    Ensemble(const Ensemble& e);
    // Constructeur supplémentaire
    Ensemble(int max);
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

Ensemble::Ensemble() {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = 50;
  m_elements = new int[50];
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
}

// Constructeur supplémentaire
Ensemble::Ensemble(int max)
{
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new int[max];
}
```

```
int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[m_nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

ensemble.cpp

Utilisation de ce constructeurs

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 50 entiers
  Ensemble e(50);

  e.ajouter(20);

  //...

  return 0;
}
```

prog.cpp

```
#include "ensemble.hpp"

int main()
{
  Ensemble *e;

  // Création d'un ensemble de 50 entiers
  e = new Ensemble(50);

  e->ajouter(20);

  //...

  return 0;
}
```

prog.cpp

Autre forme du constructeur par défaut

- Le constructeur par défaut peut aussi être défini à l'aide d'un constructeur dont tous ses arguments possèdent des valeurs par défaut :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;

  public:
    // Constructeur par défaut
    Ensemble(int max = 50);
    Ensemble(const Ensemble& e);
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

// Constructeur par défaut
Ensemble::Ensemble(int max) {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new int[max];
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
}
```

```
int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[m_nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

ensemble.cpp

Le destructeur

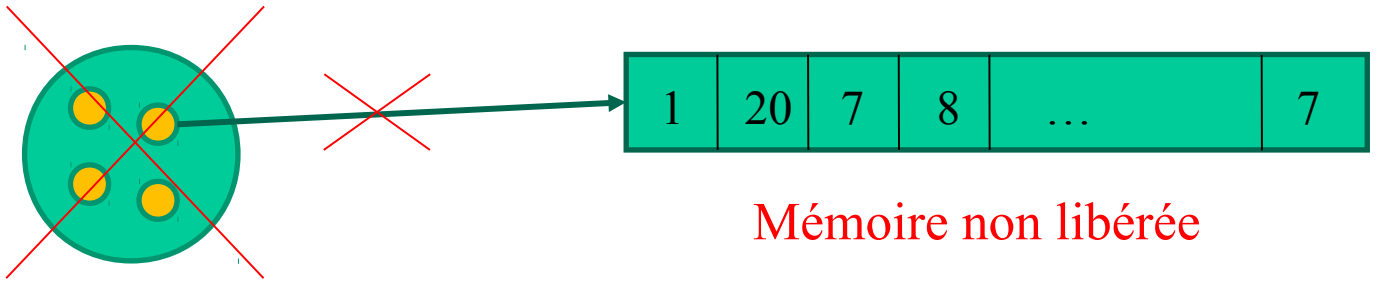
- Le destructeur sert à libérer les ressources qui ont été alloué dynamiquement par l'objet au cours de son existence.

- Le destructeur est appelé automatiquement avant la suppression de l'objet de la mémoire.

- Il n'est pas nécessaire de fournir un destructeur si la classe utilise aucune ressource allouée dynamiquement. Pour ces classes, le destructeur par défaut fourni par le C++ suffit.

Le destructeur

- Exemple : fonctionnement du destructeur par défaut sur notre classe Ensemble...



Mémoire non libérée

=> des risques de débordements de mémoire

Le destructeur

- Le destructeur est une méthode publique, sans arguments, qui ne retourne rien et qui porte le nom de la classe préfixé par ~ :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble& e);
    // Destructeur
    ~Ensemble();
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

Ensemble::Ensemble(int max) {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new int[max];
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
}

// Destructeur
Ensemble::~Ensemble() {
  delete[] m_elements;
}
```

```
int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

ensemble.cpp

Le mot clé this

- En C++, **le mot clé « this » désigne un pointeur qui pointe sur l'objet courant** (c.a.d. l'objet qui a appelé la méthode dans laquelle est utilisé ce pointeur).

- « this » sert entre autre :

  - aux méthodes pour retourner un pointeur sur l'objet courant ;

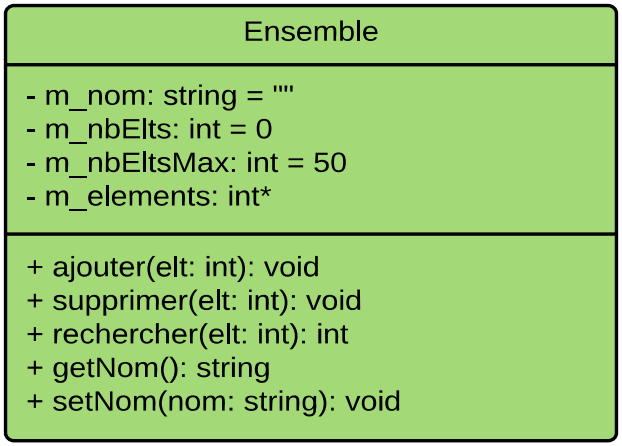
  - à faire la distinction entre les paramètres d'une méthode et les attributs de sa classe lorsqu'ils portent les mêmes noms.

Les accesseurs et mutateurs

- Un **accesseur** (getter) est une méthode qui permet d'obtenir la valeur d'un des attributs d'un objet.

- Un **mutateur** (setter) est une méthode qui permet de modifier la valeur d'un des attributs d'un objet.

- Conventions de nommage :
  - accesseurs : getNomAttribut
  - mutateurs : setNomAttribut



Les accesseurs et mutateurs

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int nbElts;
    int nbEltsMax;
    int* elements;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble& e);
    ~Ensemble();
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
    // Accesseur et mutateur
    string getNom();
    void setNom(string nom);
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

Ensemble::Ensemble(int max) {
  m_nom = "";
  nbElts = 0;
  nbEltsMax = max;
  elements = new int[max];
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  nbElts = e.nbElts;
  nbEltsMax = e.nbEltsMax;
  elements = new int[nbEltsMax];
  for (int i=0;i<nbElts;i++)
    elements[i]=e.elements[i];
}

Ensemble::~Ensemble() {
  delete[] elements;
}

// Accesseur et mutateur
string Ensemble::getNom() {
  return m_nom;
}

void Ensemble::setNom(string nom) {
  this->m_nom = nom;
}
```

```
int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<nbElts)&&(pos==-1))
  {
    if (elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((nbElts<nbEltsMax)
     &&(rechercher(elt)==-1))  {
    elements[nbElts]=elt;
    nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(nbElts-1);i++)
      elements[i]=elements[i+1];
    nbElts--;
  }
}
```

ensemble.cpp

Les accesseurs et mutateurs

- Exemple :

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 50 entiers
  Ensemble e(50);


  // Initialisation du nom
  e.setNom("Ensemble1");

  // Manipulation de l'ensemble
  e.ajouter(20);

  //...

  return 0;
}
```

prog.cpp

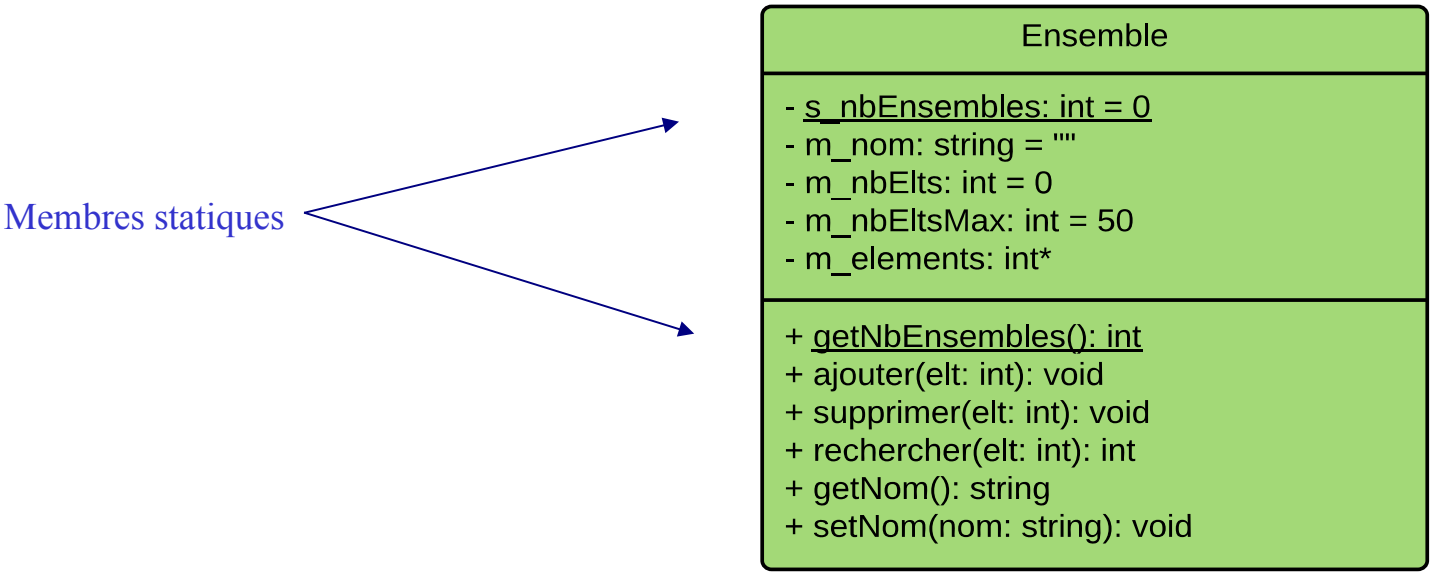
Les membres statiques

- Un attribut ou une méthode statique (ou de classe) est un membre qui est rattaché à sa classe et non pas à une instance de sa classe en particulier.
- Un attribut ou une méthode statique :
  - existe donc en mémoire même si aucune instance de sa classe n'a été crée ;
  - est accessible à partir de toutes les instances de sa classe mais aussi à partir de sa classe.
- Attention : la valeur d'un attribut statique est commune à toute les instances de sa classe.

Les membres statiques

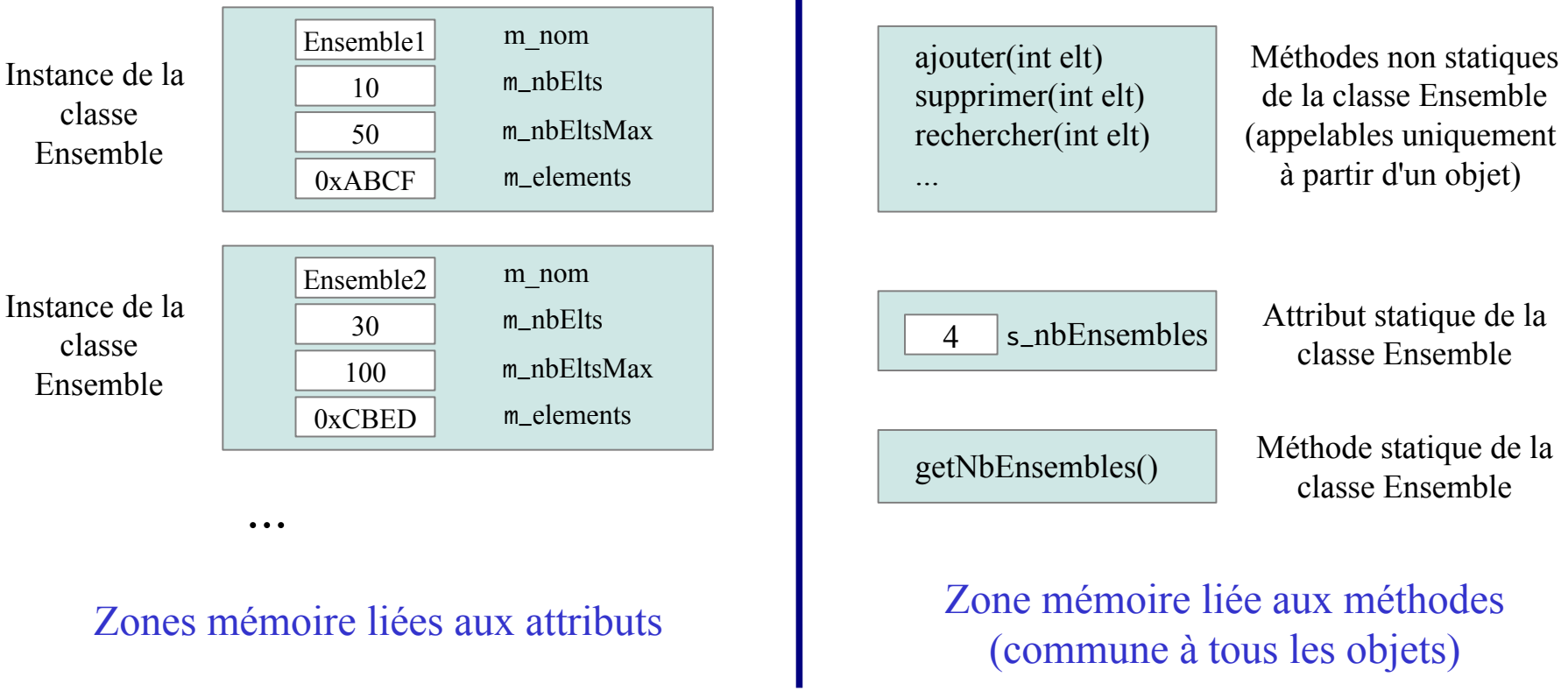
- Pour déclarer un membre statique on utilise en C++ le mot clé **static**.

- Notation UML :



Les membres statiques

- Fonctionnement en mémoire :



Les membres statiques

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    // Attribut statique
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble& e);
    ~Ensemble();
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt);
    string getNom();
    void setNom(string nom);
    // Méthode statique
    static int getNbEnsembles();
};

#endif
```

ensemble.hpp

```
#include "ensemble.hpp"

// Initialisation de l'attribut statique
int Ensemble::s_nbEnsembles=0;

Ensemble::Ensemble(int max) {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new int[max];
  s_nbEnsembles++;
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
  s_nbEnsembles++;
}

Ensemble::~Ensemble() {
  delete[] m_elements;
  s_nbEnsembles--;
}

string Ensemble::getNom() {
  return m_nom;
}
```

```
void Ensemble::setNom(string nom) {
  this->m_nom = nom;
}

int Ensemble::rechercher(int elt) {
  int i=0, pos=-1;
  while ((i<nm_bElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}

int Ensemble::getNbEnsembles() {
  return s_nbEnsembles;
}
```

ensemble.cpp

Les membres statiques

- Exemple :

```
#include "ensemble.hpp"

int main()
{
  // Appel méthode statique !
  cout << Ensemble::getNbEnsembles() << endl;

  // Création d'un ensemble de 50 entiers
  Ensemble e(50);

  // Appel méthode statique !
  cout << Ensemble::getNbEnsembles() << endl;


  e.ajouter(20);

  //...

  return 0;
}
```

prog.cpp

Les objets constants

- Lorsque l'on passe un objet par référence constante à une fonction, par défaut, dans le code de cette dernière, il n'est pas possible d'appeler les méthodes de l'objet !!

- En fait, comme le compilateur n'est pas capable d'identifier seul les méthodes qui modifient les attributs de l'objet, par défaut il bloque tout...

- Si l'on veut pouvoir appeler les méthodes qui ne modifient pas les attributs d'un objet, à partir d'une référence constante sur cet objet, il faut « donner un coup de main au compilateur » en les déclarant **const** dans la classe de l'objet !!!

Les objets constants

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int m
s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble& e);
    ~Ensemble();
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
};

#endif
```

ensemble.hpp

Les objets constants

- Exemple :

ensemble.cpp

```
#include "ensemble.hpp"

int Ensemble::s_nbEnsembles=0;

Ensemble::Ensemble(int max) {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new int[max];
  s_nbEnsembles++;
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
  s_nbEnsembles++;
}

Ensemble::~Ensemble() {
  delete[] m_elements;
  s_nbEnsembles--;
}
```

```
int Ensemble::rechercher(int elt) const
{
  int i=0, pos=-1;
  while ((i<nm_bElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

```
string Ensemble::getNom() const
{
  return m_nom;
}

void Ensemble::setNom(string nom)
{
  this->m_nom = nom;
}

int Ensemble::getNbEnsembles()
{
  return s_nbEnsembles;
}
```

Fonctions et classes amies

- Pour gagner en efficacité, il est possible en C++ d'autoriser certaines fonctions (classes) à accéder aux attributs et aux méthodes protégées et privées des objets.

- Pour se faire on utilise le mot clé friend .

Fonctions et classes amies

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble& e);
    ~Ensemble();
    void ajouter(int elt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();

    // Fonction amie
    friend void afficher(const Ensemble& e);


};

#endif
```

ensemble.hpp

Fonctions et classes amies

- Exemple :

ensemble.cpp

```
#include "ensemble.hpp"

int Ensemble::s_nbEnsembles=0;

Ensemble::Ensemble(int max) {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new int[max];
  s_nbEnsembles++;
}

Ensemble::Ensemble(const Ensemble& e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  nbEltsMax = e.m_nbEltsMax;
  m_elements = new int[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
  s_nbEnsembles++;
}

Ensemble::~Ensemble() {
  delete[] m_elements;
  s_nbEnsembles--;
}
```

```
int Ensemble::rechercher(int elt) const
{
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1))
  {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

void Ensemble::ajouter(int elt) {
  if ((m_nbElts<m_nbEltsMax)
     &&(rechercher(elt)==-1))  {
    m_elements[nbElts]=elt;
    m_nbElts++;
  }
}

void Ensemble::supprimer(int elt) {
  int pos = rechercher(elt);
  if (pos!=-1)
  {
    for(int i=pos;i<(m_nbElts-1);i++)
      m_elements[i]=m_elements[i+1];
    m_nbElts--;
  }
}
```

```
string Ensemble::getNom() const
{
  return m_nom;
}

void Ensemble::setNom(string nom)
{
  this->m_nom = nom;
}

int Ensemble::getNbEnsembles()
{
  return m_nbEnsembles;
}

void afficher(const Ensemble& e)
{
  if (e.nbElts>0)
  {
    cout << e.getNom() << " = {";
    for(int i=0;i<(e.nbElts-1);i++)
      cout << e.m_elements[i] << ",";
    cout <<e.m_elements[e.m_nbElts-1];
    cout << "}" << endl;
  }
}
```

Fonctions et classes amies

- Exemple :

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 5 entiers
  Ensemble e(5);

  // Initialisation du nom
  e.setNom("Ensemble1");

  // Manipulation de l'ensemble
  e.ajouter(10);
  e.ajouter(20);
  e.ajouter(30);
  e.ajouter(10);
  e.ajouter(20);
  e.ajouter(40);
  e.ajouter(50);
  e.supprimer(30);
  e.ajouter(60);

  // Affichage de l'ensemble
  afficher(e);

  return 0;
}
```

prog.cpp

POLYTECH
CLERMONT-FERRAND

Le polymorphisme : la surcharge des méthodes

Généralités

- Le nom de **polymorphisme** vient du grec « poly » qui signifie plusieurs et « morphos » qui signifie formes :

=> polymorphisme = plusieurs formes

- En POO, le polymorphisme traduit la capacité des objets à réagir différemment à un même message...

Généralités

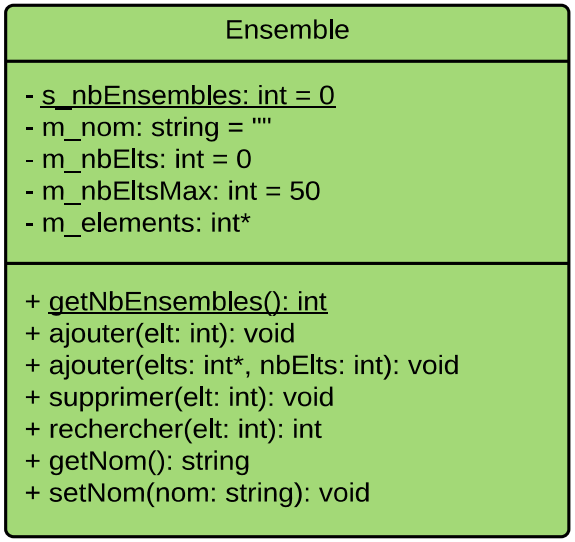
- On distingue deux formes de polymorphismes :
  - La forme faible :
    - **Surcharge des méthodes (overloading)**
    - Possibilité de définir des méthodes possédant le même nom à condition que leurs arguments soient différents
    - Détection de la bonne méthode lors de la compilation (statique)

Généralités

- La forme forte :
  - **Redéfinition (overriding)**
  - Possibilité pour une classe dérivée de redéfinir les méthodes de sa classe de base
  - Détection de la bonne méthode lors de l'exécution (dynamique)

Surcharge des méthodes

- Pour illustrer cette notion nous allons surcharger la méthode ajouter de la classe Ensemble :



Surcharge des méthodes

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein
{
  public:
    string message;
    ExceptionEnsemblePlein(string  m ="")
    {
      message=m;
    }
};

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;
  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    friend void afficher(const Ensemble & e);
};

#endif
```

ensemble.hpp

Surcharge des méthodes

- Exemple :

```
#include "ensemble.hpp"

int Ensemble::s_nbEnsembles=0;

Ensemble::Ensemble(int max) {//...}
Ensemble::Ensemble(const Ensemble &e) {//...}
Ensemble::~Ensemble() {//...}
int Ensemble::rechercher(int elt) const {//...}

void Ensemble::ajouter(int elt) {
  if (m_nbElts<m_nbEltsMax)
  {
    if (rechercher(elt)==-1)
    {
      m_elements[nbElts]=elt;
      m_nbElts++;
    }
  }
}
```

```
void Ensemble::ajouter(int* elts, int nbElts)
{
  for (int i=0; i<nbElts; i++)
    ajouter(elts[i]);
}

void Ensemble::supprimer(int elt) {//...}
string Ensemble::getNom() const {//...}
void Ensemble::setNom(string nom) {//...}
int Ensemble::getNbEnsembles() {//...}
void afficher(const Ensemble &e) {//...}
```

ensemble.cpp

Surcharge des méthodes

- Exemple :

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 5 entiers
  Ensemble e(5);
  int t[] = {10,20,30,40,50};
  // Initialisation du nom
  e.setNom("Ensemble1");

  // Ajout d'éléments dans l'ensemble
  e.ajouter(t,5);

  // Affichage de l'ensemble
  afficher(e);

  return 0;
}
```

prog.cpp

Surcharge des opérateurs

- Par défaut en C++, les opérateurs standards (+, -, *, /, <<, >>, etc.) sont définis de manière à pouvoir être utilisés avec tous les types de base du langage.

- Si l'on souhaite utiliser ces opérateurs avec des objets instanciés à partir de classes « utilisateurs », il faut au préalable les surcharger.

Surcharge des opérateurs

- En C++, les opérateurs peuvent être vu comme des fonctions amies :
  - a + b ≡ operator+(a,b)
  - cout << a ≡ operator<<(cout,a)
- Ou comme des méthodes membres :
  - a++ ≡ a.operator++()
- Pour surcharger un opérateur, il suffit donc de surcharger ces fonctions ou méthodes...

Surcharge des opérateurs

- En règle générale, pour surcharger les opérateurs unaires on utilise des méthodes membres et pour surcharger les opérateurs binaires on utilise des fonctions amies (sauf pour l'opérateur égal qui est toujours surcharger à l'aide d'une méthode membre).

- Attention : Pour pouvoir chaîner les opérateurs entre eux, il faut bien réfléchir aux valeurs retournées par ces derniers...

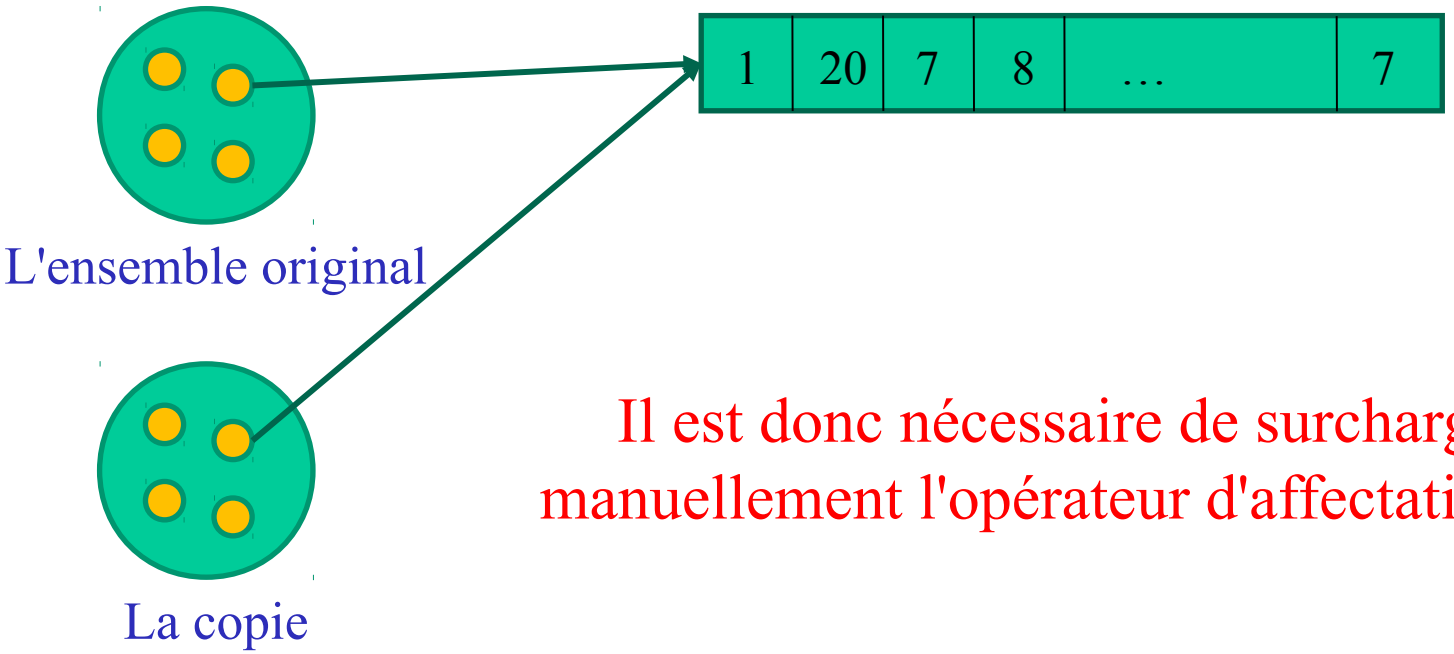
Surcharge de l'opérateur =

- Par défaut, le C++ surcharge automatiquement l'opérateur d'affectation pour toutes les classes utilisateurs. Cet opérateur par défaut recopie bit à bit les valeurs de tous les attributs de l'objet transmis en argument dans les attributs de la copie..

- Attention : cet opérateur par défaut n'est pas toujours suffisant, il est quelquefois nécessaire de le surcharger manuellement. En effet, dans le cas de références ou de pointeurs, ce n'est pas la valeur de la donnée membre elle-même qui sera recopiée mais la valeur du pointeur sur cette donnée...

Surcharge de l'opérateur =

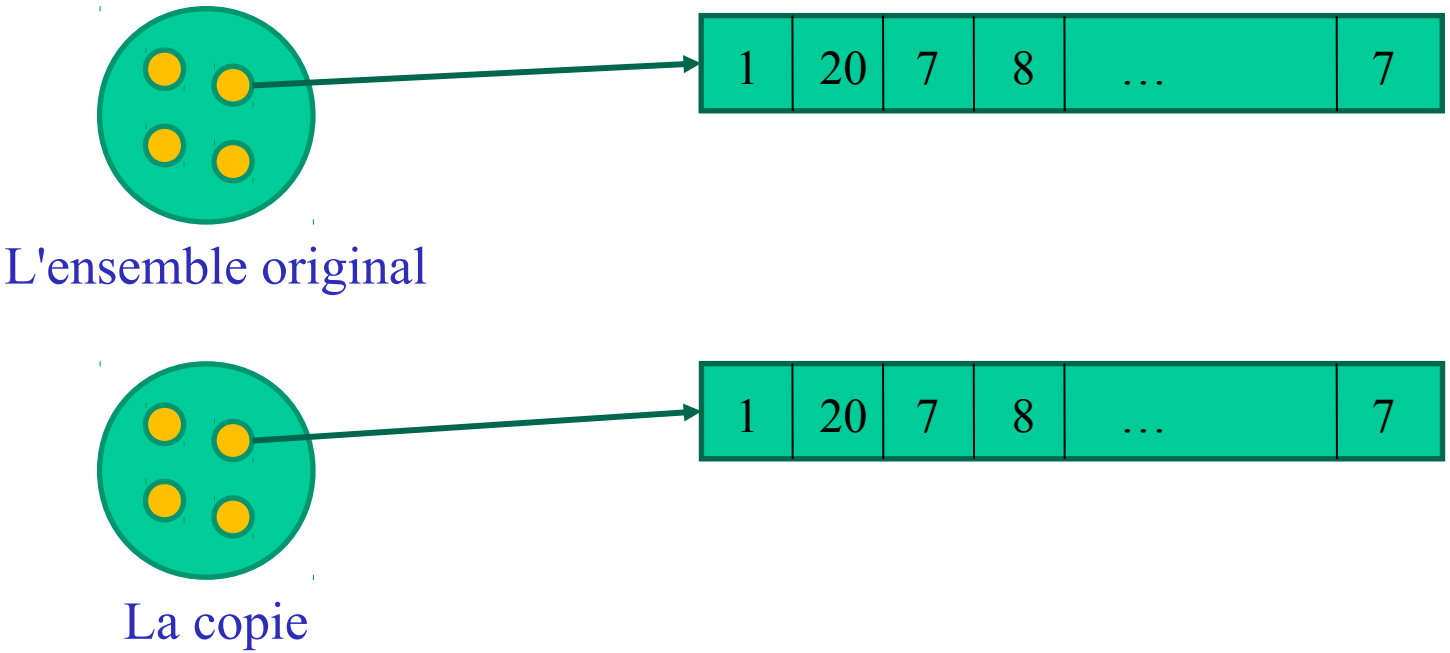
- Fonctionnement du l'opérateur d'affectation par défaut sur notre classe Ensemble :



Il est donc nécessaire de surcharger manuellement l'opérateur d'affectation !!!

Surcharge de l'opérateur =

- Fonctionnement attendu de l'opérateur d'affectation sur notre classe Ensemble :



Surcharge de l'opérateur =

- Attention :

L'opérateur d'affectation doit retourner une référence sur l'objet courant pour pouvoir chaîner les affectations :

$$a = b = c \leftrightarrow b = c \text{ puis } a = b$$

Surcharge de l'opérateur =

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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



class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;
```

```
  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();

    // Surcharge de l'opérateur =
    Ensemble& operator=(const Ensemble& e);

    friend void afficher(const Ensemble & e);
};


#endif
```

ensemble.hpp

Surcharge de l'opérateur =

- Exemple :

```
#include "ensemble.hpp"

//...

// Surcharge de l'opérateur =
Ensemble& Ensemble::operator=(const Ensemble& e)
{
  if (this != &e)
  {
    delete[] m_elements;
    m_nom = e.m_nom;
    m_nbElts = e.m_nbElts;
    m_nbEltsMax = e.m_nbEltsMax;
    m_elements = new int[m_nbEltsMax];
    for (int i=0;i<m_nbElts;i++)
      m_elements[i]=e.m_elements[i];
  }
  return *this;
}
```

ensemble.cpp

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 5 entiers
  Ensemble e(5), e2(10);
  int t[] = {10,20,30,40,50};

  // Initialisation de l'ensemble e
  e.setNom("Ensemble1");
  e.ajouter(t,5);

  // Initialisation de l'ensemble e2
  e2= e;
  e2.setNom("Ensemble2");
  e2.supprimer(30);

  // Affichage des ensembles
  afficher(e);
  afficher(e2);

  return 0;
}
```

prog.cpp

Surcharge de l'opérateur []

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein {//...};


class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;
```

```
  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    Ensemble& operator=(const Ensemble& e);

    // Surcharge de l'opérateur []
    int& operator[](int i);

    friend void afficher(const Ensemble & e);
};

#endif
```

ensemble.hpp

Surcharge de l'opérateur []

- Exemple :

```
#include "ensemble.hpp"

//...

// Surcharge de l'opérateur []
int& Ensemble::operator[](int i)
{
  return m_elements[i];
}
```

ensemble.cpp

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 5 entiers
  Ensemble e(5);
  int t[] = {10,20,30,40,50};

  // Initialisation de l'ensemble e
  e.setNom("Ensemble1");
  e.ajouter(t,5);
  e[0] = 5;
  cout << e[0] << endl;

  return 0;
}
```

prog.cpp

Surcharge des opérateurs == et !=

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein {//...};

class  ExceptionPbIndice {//...};

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    friend void afficher(const Ensemble & e);
    Ensemble& operator=(const Ensemble& e);
    int& operator[](int i);
    // Surcharge des opérateur == et !=
    friend bool operator==(const Ensemble& e1, const Ensemble& e2);
    friend bool operator!=(const Ensemble& e1, const Ensemble& e2);

    friend void afficher(const Ensemble & e);
};

#endif
```

ensemble.hpp

Surcharge des opérateurs == et !=

- Exemple :

```
#include "ensemble.hpp"

//...

// Surcharge des opérateurs == et !=
bool operator==(const Ensemble& e1, const Ensemble& e2 {
  int i=0;
  bool egaux = true;

  if (e1.m_nbElts != e2.m_nbElts)
    egaux = false;
  else {
    while ((i<e1.m_nbElts)&&(egaux==true)) {
      if (e2.rechercher(e1.elements[i])==-1)
        egaux = false;
      i++;
    }
  }
  return egaux;
}

bool operator!=(const Ensemble& e1, const Ensemble& e2) {
  return !(e1==e2);
}
```

ensemble.cpp

```
#include "ensemble.hpp"

int main()
{
  // Création de deux ensembles de 5 entiers
  Ensemble e1(5);
  Ensemble e2(5);
  int t1[] = {10,20,30,40,50};
  int t2[] = {40,30,20,10,50};


  // Initialisation des ensembles
  e1.setNom("Ensemble1");
  e2.setNom("Ensemble2");
  e1.ajouter(t1,5);
  e2.ajouter(t2,5);

  if (e1==e2)
    cout << "Ensembles égaux" << endl;
  else
    cout << "Ensembles différents" << endl;

  return 0;
}
```

prog.cpp

Surcharge des opérateurs << et >>

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein {//...};

class  ExceptionPbIndice {//...};

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;
```

```
  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    friend void afficher(const Ensemble & e);
    Ensemble& operator=(const Ensemble& e);
    int& operator[](int i);
    friend bool operator==(const Ensemble& e1, const Ensemble& e2);
    friend bool operator!=(const Ensemble& e1, const Ensemble& e2);
    // Surcharge des opérateur << et >>
    friend ostream& operator<<(ostream& f, const Ensemble& e);
    friend istream& operator>>(istream& f, Ensemble& e);
};

#endif
```

ensemble.hpp

Surcharge des opérateurs << et >>

- Exemple :

```
#include "ensemble.hpp"

//...

// Surcharge des opérateur << et >>
ostream& operator<<(ostream& f, const Ensemble& e)
{
  f << e.getNom() << " = {";
  for(int i=0;i<(e.m_nbElts-1);i++)
    f << e.m_elements[i] << ",";
  f <<e. elements[e.m_nbElts-1];
  f << "}";
  return f;
}

istream& operator>>(istream& f, Ensemble& e)
{
  int entier;
  f>>entier;
  e.ajouter(entier);
  return f;
}
```

ensemble.cpp

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 5 entiers
  Ensemble e(5);

  // Initialisation de l'ensemble
  e.setNom("Ensemble1");

  for (int i=0;i<5;i++)
    cin >> e;

  //Affichage de l'ensemble
  cout << e << endl;

  return 0;
}
```

prog.cpp

Surcharge des opérateurs + et -

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein {//...};

class  ExceptionPbIndice {//...};

class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    friend void afficher(const Ensemble & e);
    Ensemble& operator=(const Ensemble& e);
    int& operator[](int i);
    friend bool operator==(const Ensemble& e1, const Ensemble& e2);
    friend bool operator!=(const Ensemble& e1, const Ensemble& e2);
    friend ostream& operator<<(ostream& f, const Ensemble& e);
    friend istream& operator>>(istream& f, Ensemble& e);
    // Surcharge des opérateur + (union) et - (différence)
    friend Ensemble operator+(const Ensemble& e1, const Ensemble& e2);
    friend Ensemble operator-(const Ensemble& e1, const Ensemble& e2);
};

#endif
```

ensemble.hpp

Surcharge des opérateurs + et -

- Exemple :

```
#include "ensemble.hpp"

//...

// Surcharge des opérateur + (union) et - (différence)
Ensemble operator+(const Ensemble& e1, const Ensemble& e2)
{
  Ensemble nv(e1.m_nbElts+e2.m_nbElts);
  nv.ajouter(e1.m_elements,e1.m_nbElts);
  for (int i=0; i<e2.m_nbElts; i++)
    nv.ajouter(e2.m_elements[i]);
  return nv;
}

Ensemble operator-(const Ensemble& e1, const Ensemble& e2)
{
  Ensemble nv(e1.m_nbElts);
  nv.ajouter(e1.m_elements,e1.m_nbElts);
  for (int i=0; i<e2.m_nbElts; i++)
    nv.supprimer(e2.m_elements[i]);
  return nv;
}
```

ensemble.cpp

```
#include "ensemble.hpp"

int main()
{
  // Création de deux ensembles
  Ensemble e1(5);
  Ensemble e2(4);
  int t1[] = {10,20,30,40,50};
  int t2[] = {10,60,40,90};

  e1.ajouter(t1,5);
  e2.ajouter(t2,4);
  Ensemble e3 = e1+e2;
  Ensemble e4 = e1-e2;

  cout << e3 << endl;
  cout << e4 << endl;

  return 0;
}
```

prog.cpp

POLYTECH
CLERMONT-FERRAND

La généricité

Généralités

- Une fonction ou une classe générique est une fonction ou une classe qui est paramétrée à l'aide d'un (ou plusieurs) type(s) générique(s).

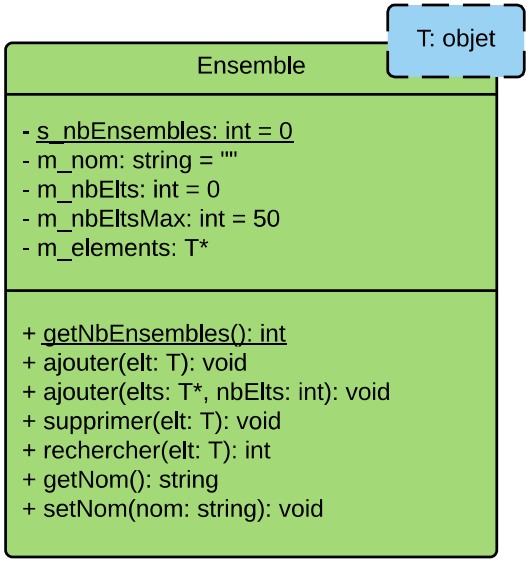
- Ce(s) dernier(s) ne sera (seront) vraiment connu(s) que lorsque la fonction ou la classe sera utilisée dans le programme principal.

- Lors de la phase de compilation, le(s) type(s) générique(s) est(sont) substitué(s) par le(s) type(s) réellement utilisé(s) afin d'obtenir le code définitif de la fonction ou de la classe classe.

Généralités

- Les **fonctions ou classes génériques** (paramétrables ou templates) peuvent donc être vues comme des modèles de fonctions ou de classes.

- Notation UML :



Syntaxe

- Déclaration d'une classe générique :

```
template<typename T1, typename T2 ...>
class nomClasse {
  //
};
```

```
template<class T1, class T2 ...>
class nomClasse {
  //
};
```

Déclaration d'une fonction générique :

```
template<typename T1, typename T2 ...>
typeRetour nomFonction(liste arguments)
{
  //
};
```

```
template<class T1, class T2 ...>
typeRetour nomFonction(liste arguments)
{
  //
};
```

- Attention : en C++, lorsque l'on crée une classe générique tout le code de cette classe doit être contenu dans le .hpp

Exemple

ensemble.hpp

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

#include <iostream>
#include <string>

using namespace std;

template <typename T>
class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    T* m_elements;
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const CEnsembe<T> &e);
    ~Ensemble();
    void ajouter(T elt);
    void ajouter(T* elts, int nbElt);
    void supprimer(T elt);
    int rechercher(T elt) const;
    string getNom() const;
    void setNom(string nom);
```

Exemple

ensemble.hpp

```
    static int getNbEnsembles();
    Ensemble& operator=(const Ensemble<T>& e);
    T& operator[](int i);
    template<typename V> friend bool operator==(const Ensemble<V>& e1, const Ensemble<V>& e2);
    template<typename V> friend bool operator!=(const Ensemble<V>& e1, const Ensemble<V>& e2);
    template<typename V> friend ostream& operator<<(ostream& f, const Ensemble<V>& e);
    template<typename V> friend istream& operator>>(istream& f, Ensemble<V>& e);
    template<typename V> friend Ensemble<V> operator+(const Ensemble<V>& e1, const Ensemble<V>& e2);
    template<typename V> friend Ensemble<V> operator-(const Ensemble<V>& e1, const Ensemble<V>& e2);
};

template<typename T>
int Ensemble<T>::s_nbEnsembles=0;

template<typename T>
Ensemble<T>::Ensemble(int max) {
  m_nom = "";
  m_nbElts = 0;
  m_nbEltsMax = max;
  m_elements = new T[max];
  s_nbEnsembles++;
}
```

Exemple

```
template<typename T>
Ensemble<T>::Ensemble(const Ensemble<T> &e)
{
  m_nom = e.m_nom;
  m_nbElts = e.m_nbElts;
  m_nbEltsMax = e.m_nbEltsMax;
  m_elements = new T[m_nbEltsMax];
  for (int i=0;i<m_nbElts;i++)
    m_elements[i]=e.m_elements[i];
  s_nbEnsembles++;
}

template<typename T>
Ensemble<T>::~Ensemble() {
  delete[] m_elements;
  s_nbEnsembles--;
}
```

```
template<typename T>
int Ensemble<T>::rechercher(T elt) const {
  int i=0, pos=-1;
  while ((i<m_nbElts)&&(pos==-1)) {
    if (m_elements[i]==elt)
      pos=i;
    i++;
  }
  return pos;
}

template<typename T>
void Ensemble<T>::ajouter(T elt) {
  if (m_nbElts<m_nbEltsMax) {
    if (rechercher(elt)==-1) {
      m_elements[m_nbElts]=elt;
      m_nbElts++;
    }
  }
}

template<typename T>
string Ensemble<T>::getNom() const {
  return m_nom;
}
```

ensemble.hpp

Exemple

```
template<typename T>
void Ensemble<T>::setNom(string nom) {
  this->m_nom = nom;
}

template<typename T>
int Ensemble<T>::getNbEnsembles() {
  return s_nbEnsembles;
}

template<typename T>
Ensemble<T>& Ensemble<T>::operator=(const
                          Ensemble<T>& e) {
  if (this != &e) {
    delete[] m_elements;
    m_nom = e.m_nom;
    m_nbElts = e.m_nbElts;
    m_nbEltsMax = e.m_nbEltsMax;
    m_elements = new T[m_nbEltsMax];
    for (int i=0;i<m_nbElts;i++)
      m_elements[i]=e.m_elements[i];
  }
  return *this;
}
```

```
template<typename T>
T& Ensemble<T>::operator[](int i){
   return m_elements[i];
}

template<typename T>
bool operator==(const Ensemble<T>& e1,
                const Ensemble<T>& e2) {
  int i=0;
  bool egaux = true;

  if (e1.m_nbElts != e2.m_nbElts)
           egaux = false;
  else {
    while ((i<e1.m_nbElts)&&(egaux==true)) {
      if (e2.rechercher(e1.m_elements[i])==-1)
        egaux = false;
      i++;
    }
  }
  return egaux;
}
```

ensemble.hpp

Exemple

```
template<typename T>
bool operator!=(const Ensemble<T>& e1,
                const Ensemble<T>& e2) {
  return !(e1==e2);
}

template<typename T>
ostream& operator<<(ostream& f, const Ensemble<T>& e) {
    f << e.getNom() << " = {";
    for(int i=0;i<(e.m_nbElts-1);i++)
      f << e.m_elements[i] << ",";
    f <<e. m_elements[e.m_nbElts-1];
    f << "}";
    return f;
}

template<typename T>
istream& operator>>(istream& f, Ensemble<T>& e) {
  T element;
  f>>element;
  e.ajouter(element);
  return f;
}
```

```
template<typename T>
Ensemble<T> operator+(const Ensemble<T>& e1,
                      const Ensemble<T>& e2) {
  Ensemble<T> nv(e1.nbElts+e2.nbElts);
  nv.ajouter(e1.m_elements,e1.m_nbElts);
  for (int i=0; i<e2.m_nbElts; i++)
    nv.ajouter(e2.m_elements[i]);
  return nv;
}

template<typename T>
Ensemble<T> operator-(const Ensemble<T>& e1,
                      const Ensemble<T>& e2) {
  Ensemble<T> nv(e1.nbElts);
  nv.ajouter(e1.m_elements,e1.m_nbElts);
  for (int i=0; i<e2.m_nbElts; i++)
    nv.supprimer(e2.m_elements[i]);
  return nv;
}

#endif
```

ensemble.hpp

Exemple

```
#include "ensemble.hpp"

int main() {
  // Création d'un ensemble borne de 5 entiers
  Ensemble<int> e1(5);
  Ensemble<float> e2(5);

  // Manipulation de l'ensemble e1
  e1.setNom("Ensemble1");
  e1.ajouter(10);
  e1.ajouter(20);
  e1.ajouter(30);
  e1.ajouter(40);
  cout<<e1<<endl;

  // Manipulation de l'ensemble e2
  e2.setNom("Ensemble2");
  e2.ajouter(10.5);
  e2.ajouter(20.5);
  e2.ajouter(30.5);
  e2.ajouter(40.5);
  cout<<e2<<endl;

  //...

  return 0;
}
```

prog.cpp

POLYTECH
CLERMONT-FERRAND

Les exceptions

Généralités

- En C, la gestion des erreurs pouvant survenir pendant l'exécution d'un programme (division par zéro, fichier inexistant...) est réalisée à l'aide de tests explicites sur des valeurs de retour des fonctions.

- Cette approche étant assez lourde à mettre en œuvre, un nouveau mécanisme a été proposé en C++ : les exceptions...

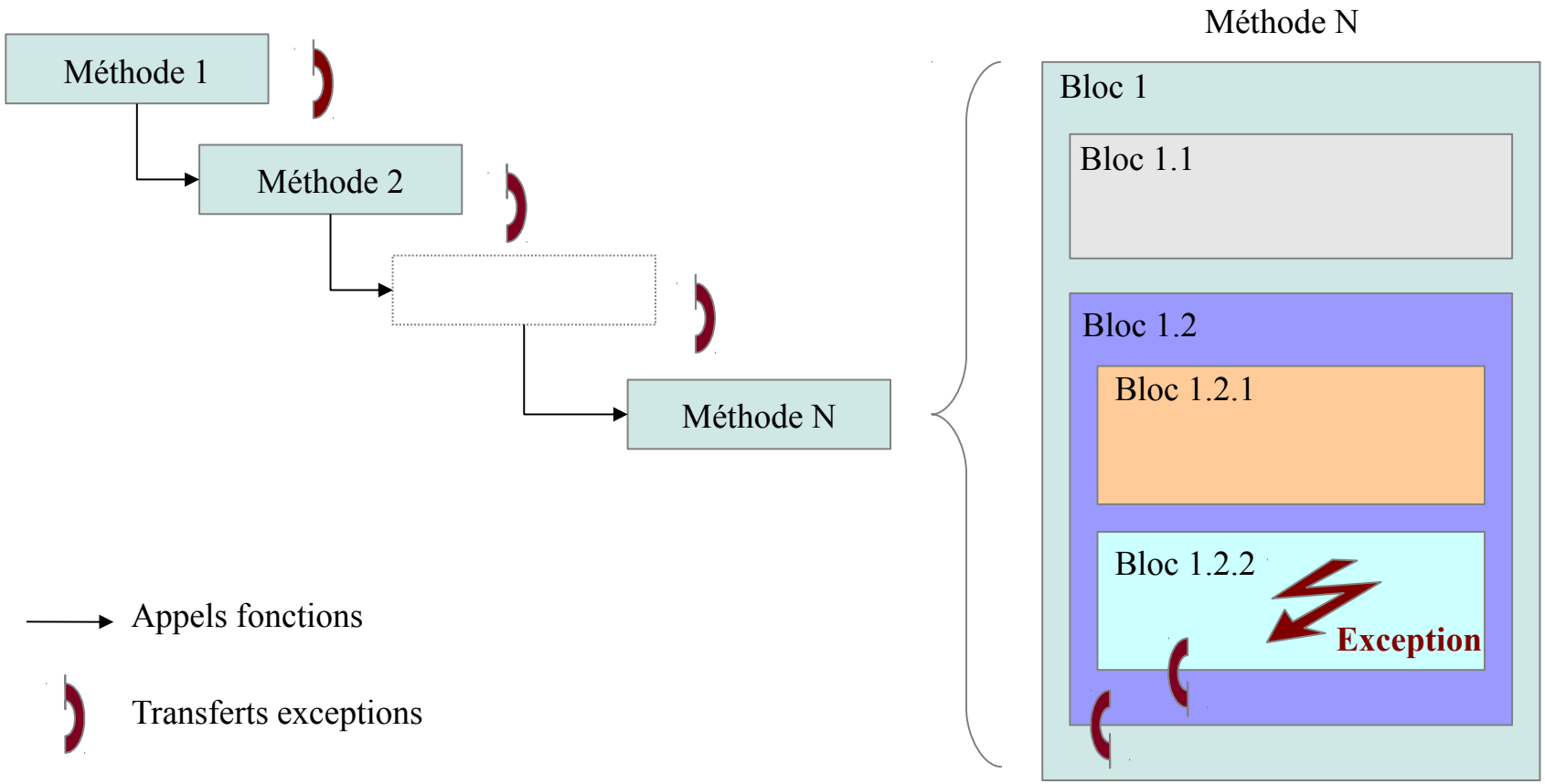
Généralités

- Une exception est un signal qui est déclenché lorsqu'une une situation anormale (une erreur) se produit lors de l'exécution d'un programme.

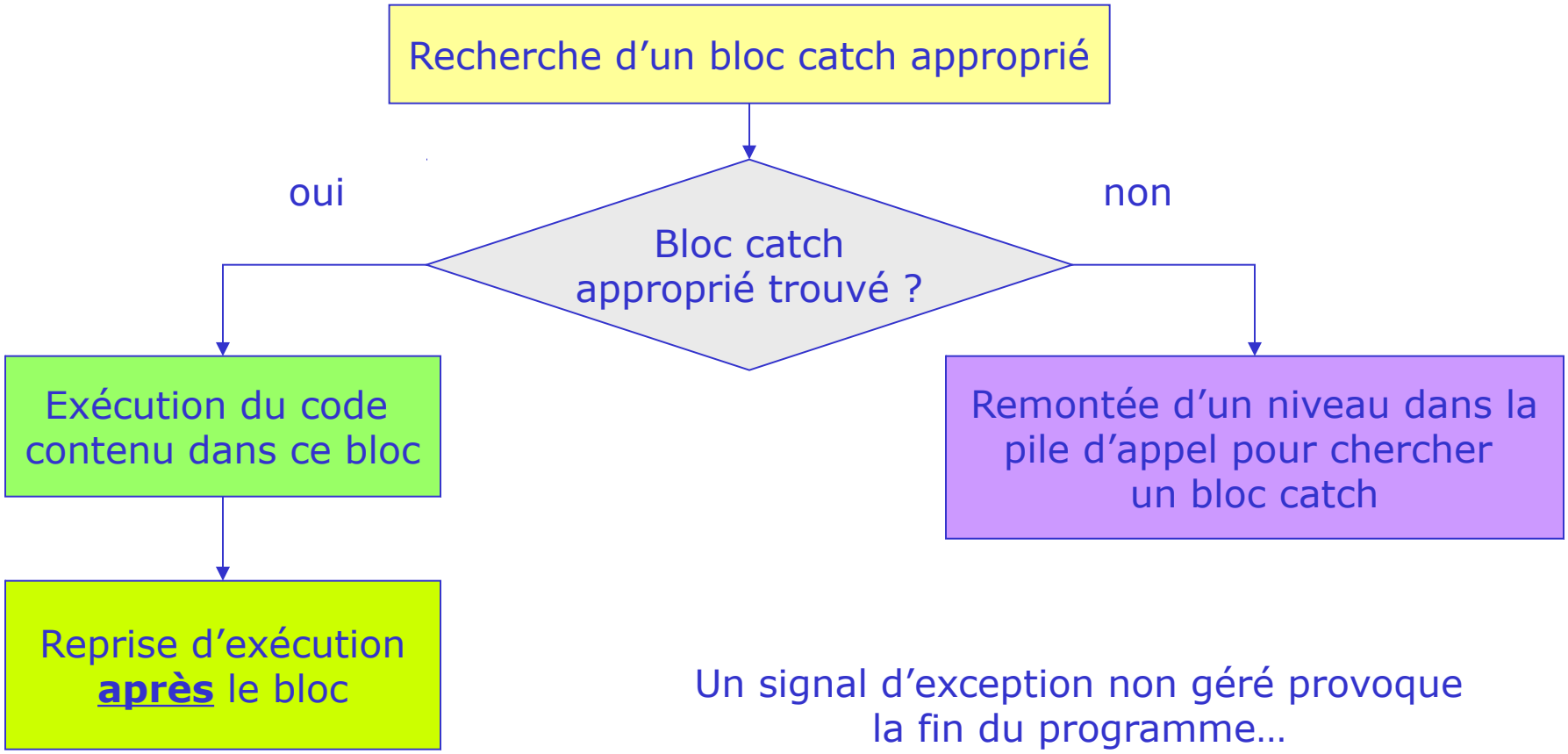
- Ce signal remonte ensuite la pile des appels du programme jusqu'à ce qu'il trouve un bloc destiné à le traiter.

- Si aucun bloc n'est trouvé le programme est arrêté.

Généralités



Généralités



Un signal d'exception non géré provoque la fin du programme…

Signal d'exception

- En C++ un signal d'exception est un objet !!!
- N'importe quel type peut servir de signal d'exception mais en général, pour faciliter la compréhension du code, on crée des classes d'exception spécifiques pour chaque type d'erreur pouvant survenir dans le code.
- Exemple :

```
class  ExceptionEnsemblePlein
{
  public:
    string m_message; // Message décrivant l'erreur
    ExceptionEnsemblePlein(string  m ="") {m_message=m;}
};
```

Lancement d'une exception

- Lorsque l'on détecte une situation anormale, on instancie un objet à partir de la classe destinée à traiter ce type de situation puis on le propage de l'instruction **throw** :

```
void Ensemble::ajouter(int elt)
{
  if (m_nbElts<m_nbEltsMax)
  {
    if (rechercher(elt)==-1)
    {
      m_elements[m_nbElts]=elt;
      m_nbElts++;
    }
  }
  else
  {
    throw ExceptionEnsemblePlein("Erreur méthode ajouter : ensemble plein");
  }
}
```

Capture des exceptions

- Pour capturer une exception, on utilise les blocs try et catch :

```
try
{
  // Code surveillé
}
catch (typeException1& ex)
{
  // Gestion des erreurs de type typeException1
}
//...
catch (typeExceptionN& ex)
{
  // Gestion des erreurs de type typeExceptionN
}
catch (...)
{
  // Autres types d'erreurs
}
```

Capture des exceptions

- Le bloc **try** contient le code qui effectue les instructions du programme risquant de rencontrer des erreurs.

- Les blocs **catch** contiennent les codes qui permettent de traiter les erreurs qui sont apparues lors de l'exécution des instructions du bloc try.

Exemple

```
#include "ensemble.hpp"

int main()
{
  // Création d'un ensemble de 5 entiers
  Ensemble e(5);

  // Initialisation du nom
  e.setNom("Ensemble1");

  // Ajout d'éléments dans l'ensemble
  try
  {
    for (int i=0; i<10; i++)
      e.ajouter(i);
  }
  catch (ExceptionEnsemblePlein& ex)
  {
    cerr << ex.message << endl;
  }

  // Affichage de l'ensemble
  afficher(e);

  return 0;
}
```

La procédure terminate

- Cette procédure est appelée pour mettre fin au programme lorsqu'une exception n'a pas été traitée.
- Il est possible de la remplacer de manière à fermer proprement certaines ressources avant la fermeture du programme à l'aide de la commande **set_terminate** :

```
void mon_terminate(void) {
  // Fermeture des ressources allouées dynamiquement
  exit(1);
}

//...

int main(void) {
  set_terminate(&mon_terminate);
  //...
  return 0;
}
```

POLYTECH
CLERMONT-FERRAND

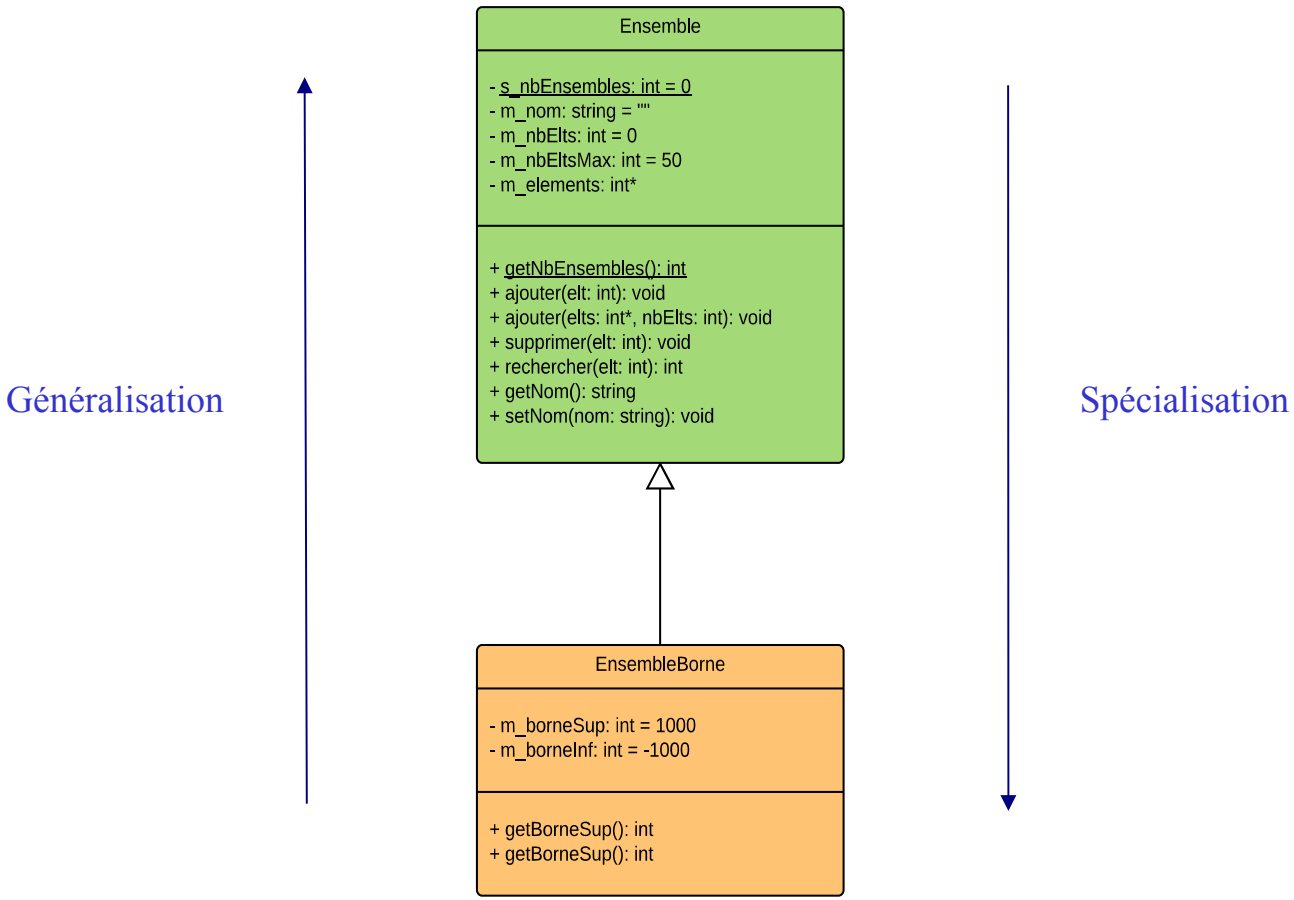
L'héritage

Rappels

- L'**héritage** est le nom du mécanisme qui permet de créer une nouvelle classe à partir d'une classe existante ;
- La classe nouvellement créée (la **classe dérivée**) hérite (contient) tous les attributs et toutes les méthodes de la classe à partir de laquelle elle a été créée (**classe de base**) ;
- La classe dérivée peut étendre la classe de base (en définissant de nouveaux attributs et de nouvelles méthodes) et/ou modifier le comportement de la classe de base (en redéfinissant certaines de ses méthodes).

Rappels

- Notation UML :



Mise en œuvre de l'héritage

- Syntaxe :

Types d'héritage

```
class NomClasseDerivee : [public|protected|private] NomClasseBase
{
  // déclaration des nouveaux attributs
  //...

  // définition des nouvelles méthodes
  //...
};
```

Les différents types d'héritage

Type d'héritage	Statut des membres de la classe de base	Statut des membres de la classe dérivée
Public	Public	Public
	Protected	Protected
	Private	Inaccessible
Protected	Public	Protected
	Protected	Protected
	Private	Inaccessible
Private	Public	Private
	Protected	Private
	Private	Inaccessible

Exemple

```
#ifndef _ENSEMBLE_BORNE_HPP_
#define _ENSEMBLE_BORNE_HPP_

#include <iostream>
#include <string>
#include "ensemble.hpp"

using namespace std;

class EnsembleBorne : public Ensemble
{
  private:
    // Attributs supplémentaires
    int m_borneInf;
    int m_borneSup;

  public:
    // Méthode supplémentaires
    int getBorneInf() const;
    int getBorneSup() const;
};

#endif
```

ensembleborne.hpp

```
#include "ensembleborne.h"

int EnsembleBorne::getBorneInf() const
{
  return m_borneInf;
}

int EnsembleBorne::getBorneSup() const
{
  return m_borneSup;
}
```

ensembleborne.cpp

Constructeurs et héritage

- En C++, une classe dérivée n'hérite jamais des constructeurs de sa classe de base...

=> il faut donc ajouter manuellement les constructeurs dans toutes les classes dérivées que l'on crée !!!

- Ordre d'appel des constructeurs : dans une hiérarchie de classe, le constructeur de la classe de base est toujours exécuté avant le constructeur de la classe dérivée.

- Par défaut c'est le constructeur par défaut de la classe de base qui est appelé mais lorsque la classe de base a plusieurs constructeurs, la classe dérivée peut décider du constructeur à appeler à l'aide d'une **liste d'initialisation**.

Constructeurs et héritage

- Exemple :

```
#ifndef _ENSEMBLE_BORNE_HPP_
#define _ENSEMBLE_BORNE_HPP_

#include <iostream>
#include <string>
#include "ensemble.hpp"

using namespace std;

class EnsembleBorne : public Ensemble
{
  private:
    // Attributs supplémentaires
    int m_borneInf;
    int m_borneSup;

  public:
    // Constructeurs
    EnsembleBorne(int max=50,int binf=-1000, int bsup=1000);
    EnsembleBorne(const EnsembleBorne & e);
    int getBorneInf() const;
    int getBorneSup() const;
};

#endif
```

ensembleborne.h

Constructeurs et héritage

- Exemple :

```
#include "ensembleborne.h"

EnsembleBorne::EnsembleBorne(int max, int binf, int bsup):Ensemble(max)
{
  m_borneInf=binf;
  m_borneSup=bsup;
}

EnsembleBorne::EnsembleBorne(const EnsembleBorne & e):Ensemble(e)
{
  m_borneInf=e.borneInf;
  m_borneSup=e.borneSup;
}

int EnsembleBorne::getBorneInf() const
{
  return m_borneInf;
}

int EnsembleBorne::getBorneSup() const
{
  return m_borneSup;
}
```

listes d'initialisations

ensembleborne.cpp

Destructeur et héritage

- En C++, une classe dérivée n'hérite jamais du destructeur de sa classe de base...

=> si nécessaire il faut donc ajouter manuellement le destructeur dans les classes dérivées que l'on crée !!!

- Ordre d'appel des destructeurs : le destructeur de la classe dérivé est exécuté avant le destructeur de la classe de base !

Destructeur et héritage

- Exemple :

```
#ifndef _ENSEMBLE_BORNE_HPP_
#define _ENSEMBLE_BORNE_HPP_

#include <iostream>
#include <string>
#include "ensemble.hpp"

using namespace std;

class EnsembleBorne : public Ensemble
{
  private:
    // Attributs supplémentaires
    int m_borneInf;
    int m_borneSup;

  public:
    EnsembleBorne(int max=50,int binf=-1000, int bsup=1000);
    EnsembleBorne(const EnsembleBorne & e);
    // Destructeur
    ~EnsembleBorne();
    int getBorneInf() const;
    int getBorneSup() const;
};

#endif
```

ensembleborne.h

Destructeur et héritage

- Exemple :

```
#include "ensembleborne.h"

EnsembleBorne::EnsembleBorne(int max, int binf, int bsup):Ensemble(max)
{
  m_borneInf=binf;
  m_borneSup=bsup;
}

EnsembleBorne::EnsembleBorne(const EnsembleBorne & e):Ensemble(e)
{
  m_borneInf=e.m_borneInf;
  m_borneSup=e.m_borneSup;
}

EnsembleBorne::~EnsembleBorne() {}

int EnsembleBorne::getBorneInf() const
{
  return m_borneInf;
}

int EnsembleBorne::getBorneSup() const
{
  return m_borneSup;
}
```

ensembleborne.cpp

Destructeur et héritage

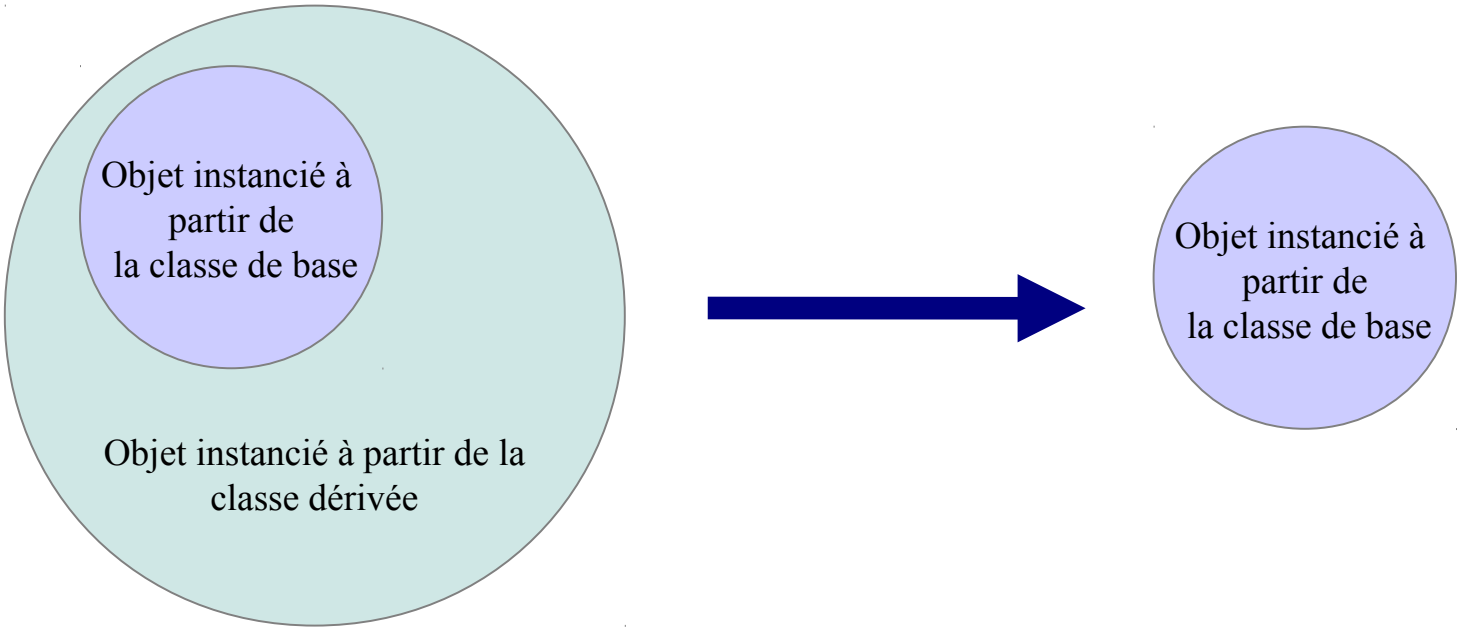
- Exemple :

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(5,-10,10);

  // Initialisation du nom
  e.setNom("EnsembleBorne1");

  // Manipulation de l'ensemble
  e.ajouter(10);
  e.ajouter(20);
  e.ajouter(30);
  e.ajouter(40);
  e.ajouter(50);

  //...

  return 0;
}
```

prog.cpp

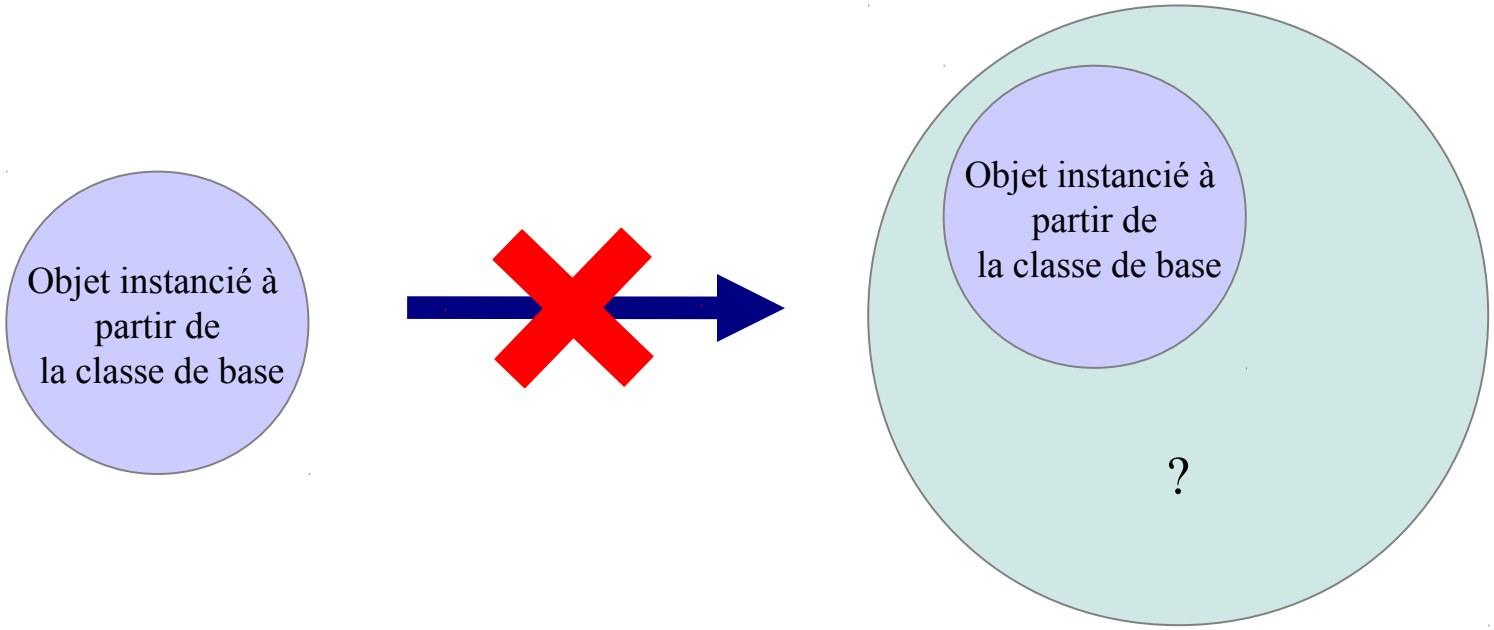
Conversions de type dans une hiérarchie de classes

- En C++, il est possible de convertir à l'aide d'un cast un objet instancié à partir d'une classe dérivée en un objet instancié à partir de sa classe de base.



Conversions de type dans une hiérarchie de classes

- Attention : l'inverse est faux !!!



Conversions de type dans une hiérarchie de classes

- De même, en C++, tout objet instancié à partir d'une classe dérivée peut être manipulé via une référence ou un pointeur sur sa classe de base.

- Attention : l'inverse est également faux !!!

Conversions de type dans une hiérarchie de classes

- Exemple :

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(5,-10,10);

  // Initialisation du nom
  e.setNom("EnsembleBorne1");

  // Manipulation de l'ensemble
  e.ajouter(10);
  e.ajouter(20);
  e.ajouter(30);
  e.ajouter(40);
  e.ajouter(50);

  // La fonction affiche va manipuler l'objet e de type EnsembleBorne
  // grâce à une référence sur sa classe de base (de type Ensemble)
  affiche(e)

  return 0;
}
```

Conversions de type dans une hiérarchie de classes

- Attention : les objets manipulés via un pointeur ou une référence sur leur classe de base perdent leur comportement issu de leur classe dérivé !!!

- Pour retrouver l'ensemble du comportement d'un objet manipulé à l'aide d'une référence ou d'un pointeur sur sa classe de base, il faut effectuer un transtypage de la référence ou du pointeur !!!

Conversions de type dans une hiérarchie de classes

- Exemples :

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(5,-10,10);
  Ensemble& refe = e

  refe.getNom() // OK car méthode de Ensemble
  refe.getBorneInf() // Erreur car méthode de EnsembleBorne

  return 0;
}
```

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(5,-10,10);
  Ensemble& refe = e
  EnsembleBorne& refeb = (EnsembleBorne&)refe; // Casting !!!

  refeb.getNom() // OK
  refeb.getBorneInf() // OK

  return 0;
}
```

POLYTECH
CLERMONT-FERRAND

Le polymorphisme : la redéfinition des méthodes

Rappels

- Redéfinition (overriding) des méthodes :
  - Possibilité pour une classe dérivée de redéfinir les méthodes de sa classe de base
  - Détection de la bonne méthode lors de l'exécution (dynamique)
  - Forme forte du polymorphisme

Redéfinition des méthodes

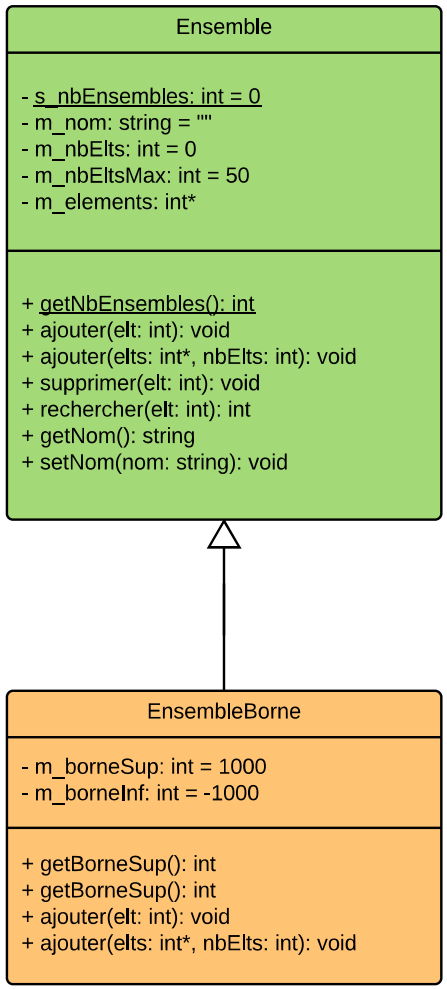
- Pour le moment, lorsqu'on ajoute des éléments à un objet du type EnsembleBorne, on utilise les méthodes "ajouter" de la classe Ensemble.

- Le problème est que ces méthodes ne vérifient pas que les éléments que l'on ajoute sont compris entre les bornes de l'ensemble...

- Pour corriger ce problème, il faut les redéfinir dans la classe EnsembleBorne !!!

Redéfinition des méthodes

- Notation UML :



Redéfinition des méthodes

- Exemple :

```
#ifndef _ENSEMBLE_BORNE_HPP_
#define _ENSEMBLE_BORNE_HPP_

#include <iostream>
#include <string>
#include "ensemble.hpp"

using namespace std;

class  ExceptionHorsBornes
{
  public:
    string m_message;
    ExceptionHorsBornes(string  m ="") {m_message=m;}
};

class EnsembleBorne : public Ensemble
{
  private:
    int m_borneInf;
    int m_borneSup;
  public:
    EnsembleBorne(int max=50,int binf=-1000, int bsup=1000);
    EnsembleBorne(const EnsembleBorne & e);
    ~EnsembleBorne();
    int getBorneInf() const;
    int getBorneSup() const;

    // Redéfinition des méthodes "ajouter" de la classe
    // Ensemble
    void ajouter(int elt);
    void ajouter(int* elts, int nbElt);
};

#endif
```

ensembleborne.hpp

Redéfinition des méthodes

- Exemple :

```
#include "ensembleborne.h"

EnsembleBorne::EnsembleBorne(int max, int binf,
 int bsup):Ensemble(max){
  m_borneInf=binf;
  m_borneSup=bsup;
}

EnsembleBorne::EnsembleBorne(const EnsembleBorne & e)
 :Ensemble(e) {
  m_borneInf=e.borneInf;
  m_borneSup=e.borneSup;
}

EnsembleBorne::~EnsembleBorne() {}

int EnsembleBorne::getBorneInf() const {
  return m_borneInf;
}

int EnsembleBorne::getBorneSup() const {
  return m_borneSup;
}
```

```
// Redéfinition des méthodes "ajouter" de la classe
// Ensemble
void EnsembleBorne::ajouter(int elt) {
  if ((m_borneInf<=elt)&&(elt<=m_borneSup))
  {
    Ensemble::ajouter(elt);
  }
  else
  {
    throw ExceptionHorsBornes("Erreur méthode ajouter :
           element en dehors des bornes");
  }
}

void EnsembleBorne::ajouter(int* elts, int nbElts) {
  for (int i=0; i<nbElts; i++)
    ajouter(elts[i]);
}
```

ensembleborne.cpp

Redéfinition des méthodes

- Exemple :

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(10,-100,100);
  int t[] = {10,20,1000,40,50};

  // Initialisation du nom
  e.setNom("EnsembleBorne1");

  // Manipulation de l'ensemble
  try {
    e.ajouter(t,5);
  }
  catch (ExceptionHorsBornes& ex)
  {
            cout << ex.message << endl;
  }
  catch (ExceptionEnsemblePlein& ex)
  {
            cout << ex.message << endl;
  }

  // Affichage de l'ensemble
  cout << e << endl;

  return 0;
}
```

prog.cpp

Redéfinition statique par défaut

- Par défaut, le langage C++ gère l'appel de toutes les méthodes redéfinies de manière **statique**. C'est à dire que le choix de la méthode à exécuter est réalisé lors de la compilation du programme.

- Inconvénient : lors de la compilation du programme, on connait pas le type exact des objets manipulés via des références ou des pointeurs sur leurs classes de base => impossible de déterminer quelle méthode appeler => c'est la méthode de la classe de base qui est appelée.

- Avantage : ce mécanisme n'entraine pas de pertes de performances.

Redéfinition statique par défaut

- Exemple :

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(10,-100,100);
  Ensemble& refe = e;
  int t[] = {10,20,1000,40,50};

  // Initialisation du nom
  refe.setNom("EnsembleBorne1");

  // Manipulation de l'ensemble
  try {
    refe.ajouter(t,5); // Pb appel de la méthode de la classe Ensemble
                       // au lieu de celle de la classe EnsembleBorne
  }
  catch (ExceptionHorsBornes& ex) {
            cout << ex.message << endl;
  }
  catch (ExceptionEnsemblePlein& ex) {
            cout << ex.message << endl;
  }

  // Affichage de l'ensemble
  cout << e << endl;

  return 0;
}
```

prog.cpp

Le mot clé virtual

- Lorsque cela est nécessaire, il est possible de forcer le C++ à gérer l'appel des méthodes redéfinies de manière dynamique en utilisant des **méthodes virtuelles**.

- Pour cela, il faut préfixer les méthodes dont on veut que l'appel soit gérer de manière dynamique avec le mot clé **virtual**.

Le mot clé virtual

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein {//...};


class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    ~Ensemble();
    // Méthodes virtuelles
    virtual void ajouter(int elt);
    virtual void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    friend void afficher(const Ensemble & e);
    Ensemble& operator=(const Ensemble& e);
    int& operator[](int i);
    friend bool operator==(const Ensemble& e1, const Ensemble& e2);
    friend bool operator!=(const Ensemble& e1, const Ensemble& e2);
    friend ostream& operator<<(ostream& f, const Ensemble& e);
    friend istream& operator>>(istream& f, Ensemble& e);
    friend Ensemble operator+(const Ensemble& e1, const Ensemble& e2);
    friend Ensemble operator-(const Ensemble& e1, const Ensemble& e2);
};

#endif
```

ensemble.hpp

Le mot clé virtual

- Exemple :

```
#include "ensembleborne.h"

int main()
{
  // Création d'un ensemble borne de 5 entiers
  EnsembleBorne e(10,-100,100);
  Ensemble& refe = e;
  int t[] = {10,20,1000,40,50};

  // Initialisation du nom
  refe.setNom("EnsembleBorne1");

  // Manipulation de l'ensemble
  try {
    refe.ajouter(t,5); // Appel de la méthode de la classe
                       // EnsembleBorne
  }
  catch (ExceptionHorsBornes& ex) {
            cout << ex.message << endl;
  }
  catch (ExceptionEnsemblePlein& ex) {
            cout << ex.message << endl;
  }

  // Affichage de l'ensemble
  cout << e << endl;

  return 0;
}
```

prog.cpp

Les destructeurs virtuels

- Lorsque des objets sont manipulés via des pointeurs ou des références sur leur classe de base, leur destruction peut présenter quelques difficultés.

- Il est alors prudent de rendre virtuels les destructeurs de façon à ce que le destructeur de l'objet réellement manipulé soit également appelé.

- Ici encore, on utilise encore le mot clé virtual.

Le mot clé virtual

- Exemple :

```
#ifndef _ENSEMBLE_HPP_
#define _ENSEMBLE_HPP_

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

class  ExceptionEnsemblePlein {//...};



class Ensemble
{
  private:
    string m_nom;
    int m_nbElts;
    int m_nbEltsMax;
    int* m_elements;
    static int s_nbEnsembles;

  public:
    Ensemble(int max = 50);
    Ensemble(const Ensemble &e);
    // Destructeur virtuel
    virtual ~Ensemble();
    virtual void ajouter(int elt);
    virtual void ajouter(int* elts, int nbElt);
    void supprimer(int elt);
    int rechercher(int elt) const;
    string getNom() const;
    void setNom(string nom);
    static int getNbEnsembles();
    friend void afficher(const Ensemble & e);
    Ensemble& operator=(const Ensemble& e);
    int& operator[](int i);
    friend bool operator==(const Ensemble& e1, const Ensemble& e2);
    friend bool operator!=(const Ensemble& e1, const Ensemble& e2);
    friend ostream& operator<<(ostream& f, const Ensemble& e);
    friend istream& operator>>(istream& f, Ensemble& e);
    friend Ensemble operator+(const Ensemble& e1, const Ensemble& e2);
    friend Ensemble operator-(const Ensemble& e1, const Ensemble& e2);
};

#endif
```

ensemble.hpp

Méthodes et classes abstraites

- Une **classe abstraite** est une classe qui possède au moins une **méthode abstraite** c.a.d une méthode qui ne possède pas d'implémentation.

- Etant donnée qu'une classe abstraite est incomplète, il n'est pas possible de l'instancier ;

- A contrario, il est possible de créer des références et des pointeurs sur une classe abstraite.

Méthodes et classes abstraites

- Les classes abstraites sont utilisées principalement dans les hiérarchies de classes.

- L'intérêt est que toute classe qui hérite d'une classe abstraite est obligée de redéfinir les méthodes abstraites de cette dernière pour pouvoir être instanciée...

=> les classes abstraites permettent donc de définir à l'avance une partie du comportement de leurs classes dérivées.

Méthodes et classes abstraites

- Intérêt : Les classes abstraites permettent de définir les interfaces dont certains objets ont besoins pour fonctionner.

- Exemple :

  On veut créer un logiciel de dessin qui va manipuler des objets graphiques. On veut pouvoir afficher ces objets, leur faire subir des rotations, des translations...

  Pour cela, on va créer une classe abstraite ObjetGraphique qui liste toutes ces fonctionnalités via des méthodes abstraites. Le logiciel de dessin manipulera les objets graphiques uniquement via des pointeurs sur cette classe.

Méthodes et classes abstraites

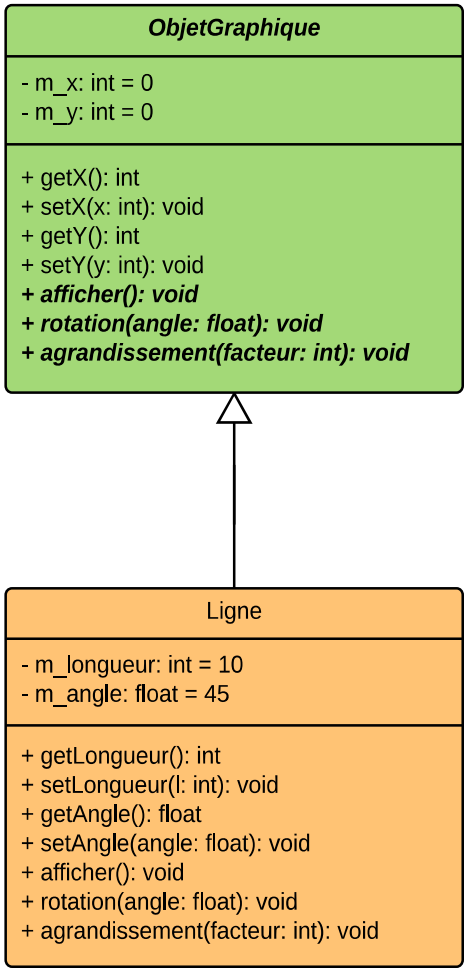
Les personnes qui vont développer les objets graphiques seront obligés d'implémenter les fonctions définies dans la classe abstraite s'il veulent que leurs classes soient utilisables avec notre logiciel de dessin...

=> moins de problème de compatibilité !

Notre logiciel sera également évolutif. Grâce au fait qu'il manipule les objets graphiques via des pointeurs sur la classe abstraite et au polymorphisme, si l'on veut étendre ses fonctionnalité en ajoutant de nouveaux objets graphiques, une grande partie du code sera réutilisable sans modifications.

Méthodes et classes abstraites

- Notations UML :



Méthodes virtuelles pures

- En C++, les méthodes abstraites sont implémentée à l'aide de **méthodes virtuelles pures** :

```
virtual typeRetour NomMethode(liste argumets) = 0;
```

Méthodes virtuelles pures

- Exemple :

```
#ifndef _OBJET_GRAPHIQUE_HPP_
#define _OBJET_GRAPHIQUE_HPP_

#include <iostream>
using namespace std;


// Classe abstraite
class ObjetGraphique
{
  private:
    int m_x,m_y;

  public:
    ObjetGraphique(int x=0, int y=0);
    int getX();
    void setX(int x);
    int getY();
    void setY(int x);

    // Méthodes virtuelles pures
    virtual void afficher() = 0;
    virtual void rotation(float angle) = 0;
    virtual void agrandissement(int facteur) = 0;
};

#endif
```

objetgraphique.hpp

```
#include "objetgraphique.hpp"

ObjetGraphique::ObjetGraphique(int x, int y) {
  m_x=x;
  m_y=y;
}

int ObjetGraphique::getX() {
  return m_x;
}

void ObjetGraphique::setX(int x) {
  m_x=x;
}

int ObjetGraphique::getY()
{
  return m_y;
}

void ObjetGraphique::setY(int x)
{
  m_y=y;
}
```

objetgraphique.cpp

Méthodes virtuelles pures

- Exemple :

```
#ifndef _LIGNE_HPP_
#define _LIGNE_HPP_

#include "objetgraphique.hpp"

class Ligne : public ObjetGraphique
{
  private:
    int m_longueur;
    float m_angle;

  public:
    Ligne(int x=0, int y=0, int l=10, float angle=45.0);
    int getLongueur();
    void setLongeur(int l);
    float getAngle();
    void setAngle(float angle);
    // Redéfinition des méthodes virtuelles pures
    void afficher();
    void rotation(float angle);
    void agrandissement(int facteur);
};

#endif
```

ligne.hpp

Méthodes virtuelles pures

- Exemple :

```
#include "ligne.hpp"

Ligne::Ligne(int x, int y, int l, float angle)
 :ObjetGraphique(x,y) {
  m_longueur=l;
  m_angle=angle;
}

int Ligne::getLongueur() {
  return m_longueur;
}

void Ligne::setLongeur(int l) {
  m_longueur = l;
}

float Ligne::getAngle() {
  return m_angle;
}

void Ligne::setAngle(float angle) {
  m_angle=angle;
}
```

```
void Ligne::setAngle(float angle) {
  m_angle=angle;
}

void Ligne::afficher() {
  cout << "Caractéristiques ligne :" << endl;
  cout << " x =" << getX() << endl;
  cout << " y =" << getY() << endl;
  cout << " longueur =" << m_longueur << endl;
  cout << " angle =" << m_angle << endl;
}

void CLigne::rotation(float angle) {
  this->angle+=angle;
}

void CLigne::agrandissement(int facteur) {
  longueur*=facteur;
}
```

ligne.cpp

Méthodes virtuelles pures

- Exemple :

```
#ifndef _LOGICIEL_DESSIN_HPP_
#define _LOGICIEL_DESSIN__HPP_

#include "objetgraphique.hpp"

// Classe abstraite
class LogicielDessin
{
  private:
    ObjetGraphique** m_liste;
    int m_nbObjetsMax;
    int m_nbObjets;

  public:
    LogicielDessin(int max=50);
    ~LogicielDessin();
    void ajouter(ObjetGraphique* obj);
    void afficher();
    void rotation(int i, float angle);
    void agrandissement(int i, int facteur);
    //...
};

#endif
```

logicieldessin.hpp

Méthodes virtuelles pures

- Exemple :

```
#include "logicieldessin.hpp"

LogicielDessin::LogicielDessin(int max) {
  m_nbObjetsMax = max;
  m_nbObjets = 0;
  m_liste = new ObjetGraphique*[max];
}

LogicielDessin::~LogicielDessin()
{
            delete[] m_liste;
}

void LogicielDessin::ajouter(ObjetGraphique* obj)
{
  if (m_nbObjets<m_nbObjetsMax)
  {
    m_liste[nbObjets]=obj;
    m_nbObjets++;
  }
}
```

```
void LogicielDessin::afficher()
{
  for (int i=0; i<m_nbObjets; i++)
  {
    m_liste[i]->afficher(); // Polymorphisme
  }
}

void LogicielDessin::rotation(int i, float angle)
{
  m_liste[i]->rotation(angle); // Polymorphisme
}

void LogicielDessin::agrandissement(int i, int facteur)
{
  m_liste[i]->agrandissement(facteur); // Polymorphisme
}
```

logicieldessin.cpp

Méthodes virtuelles pures

- Exemple :

```
#include "logicieldessin.hpp"
#include "ligne.hpp"

int main()
{
  LogicielDessin log(50);

  log.ajouter(new Ligne(10,10,20,45.0));
  log.rotation(0,90.0);
  log.afficher();


  return 0;
}
```

prog.cpp

L'héritage multiple

- On appel **héritage multiple**, le fait de pouvoir créer des classes dérivées à partir de plusieurs classes de base :

```
class A { //. . . };
class B { //. . . };

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

- L'héritage multiple pose de nombreux problèmes :
  - les appels des méthodes polymorphes sont plus compliqués à gérer ;
  - dans certains cas une classe peut hériter plusieurs fois d'une méthode ou d'un attribut de ses ancêtres.

Héritage virtuel

- Exemple :

```
class Vehicule
{
  public:
    string m_modele;
};

class Voiture : public Vehicule
{
  // Voiture hérite de m_modele
};

class Bateau : public Vehicule
{
  // Bateau hérite de m_modele
};

class VoitureAmphibie : public Voiture, public Bateau
{
  // Problème : VoitureAmphibie hérite donc deux fois de m_modele :
  // - une fois de Voiture
  // - une fois de Bateau
};
```

Héritage virtuel

- Pour solutionner ce problème on utilise l'héritage virtuel :

```
class Vehicule
{
  public:
    string m_modele;
};

class Voiture : public virtual Vehicule
{
  // Voiture hérite de m_modele
};

class Bateau : public virtual Vehicule
{
  // Bateau hérite de m_modele
};


class VoitureAmphibie : public Voiture, public Bateau
{
  // VoitureAmphibie n'hérite plus qu'une fois de m_modele
};
```

POLYTECH
CLERMONT-FERRAND

Les fichiers

Ecriture dans un fichier texte

- Exemple :

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

using namespace std;

struct personne {
  string nom;
  string prenom;
  int age;
};

void ecritureFichier(string nomFichier)
{
  personne p;
  int nbPersonnes=0;

  // Ouverture en écriture avec effacement
  // du fichier ouvert
  ofstream fichier(nomFichier.c_str(),ios::out|ios::trunc);

  if(!fichier.fail())
  {
    cin >> nbPersonnes;
    for (int i=0;i<nbPersonnes;i++)
    {
      cin >> p.nom >> p.prenom >> p.age;
      fichier << p.nom << endl;
      fichier << p.prenom << endl;
      fichier << p.age;
      if (i!=(nbPersonnes-1)) fichier << endl;
    }
    fichier.close();
  }
  else
    cerr << "Impossible d'ouvrir le fichier !" << endl;
}
```

Lecture d'un fichier texte

- Exemple :

```
void lectureFichier(string nomFichier)
{
  personne p;

  ifstream fichier(nomFichier.c_str(), ios::in);

  if(!fichier.fail())
  {
    while (fichier.eof() == false)
    {
      fichier >> p.nom >> p.prenom >> p.age;
      cout << p.nom << ", " << p.prenom << ", " << p.age << endl;
    }
    fichier.close();
  }
  else
    cerr << "Impossible d'ouvrir le fichier !" << endl;
}


int main()
{
  string monFichier = "test.txt";
  ecritureFichier(monFichier);
  lectureFichier(monFichier);
  return 0;
}
```

POLYTECH
CLERMONT-FERRAND

La STL

Introduction

- La librairie standard du C++ (STL : Standard Template Library) est née de la volonté d'apporter aux programmeurs utilisant ce langage un canevas de programmation efficace, générique et simple à utiliser.
- Cette librairie comprend notamment :
  - Des classes (appelées **conteneurs**) qui implémentent les structures de données les plus utilisées.

Introduction

- Des **algorithmes** génériques permettant de travailler avec les différents conteneurs.
- La classe **string** qui permet de manipuler les chaînes de caractères comme un type normal (affectation, comparaisons, etc.).
- Les flux (**iostream)** qui permettent de gérer facilement les E/S.

Pourquoi utiliser la STL

- **Fiabilité :** les classes de la STL sont sûres, et, dès qu'une erreur est découverte, elle est rapidement corrigée.

- **Réutilisabilité :** la STL est totalement intégrée au standard du C++. Son utilisation est donc sensée garantir la portabilité du logiciel.

Pourquoi utiliser la STL

- **Compréhensibilité et maintenabilité :** la standardisation de la STL garantit que le fait que tout programmeur C++ est capable d'appréhender du code reposant sur elle.

- **Efficacité :** les composants de la STL sont optimisés à l'extrême.

Les conteneurs

- Les conteneurs sont des structures de données qui permettent :
  - D'organiser des données de même type en séquence.
  - De parcourir ces données
- Exemples :
  - Les vecteurs (vector)
  - Les listes (list)
  - Les ensembles (set)
  - Les dictionnaires (map)

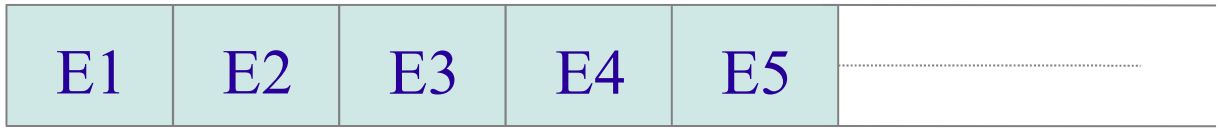
Les conteneurs

- Les conteneurs de la STL sont génériques (c.a.d. indépendants du type des données qu'ils contiennent).

- D'autre part, ils gèrent automatiquement l'allocation dynamique de mémoire.

Les vecteurs

- Les vecteurs sont des tableaux dynamiques pouvant contenir un nombre quelconque d'éléments :



Les vecteurs

- #include <vector>
- Opérateurs : =, ==, !=, <, <=, >, >=, []
- Quelques méthodes :
  - size : retourne le nombre de valeurs stockées dans le vecteur
  - empty : vérifie si le vecteur est vide
  - clear : vide le vecteur
  - push_back : ajoute un élément à, la fin du vecteur

Les vecteurs

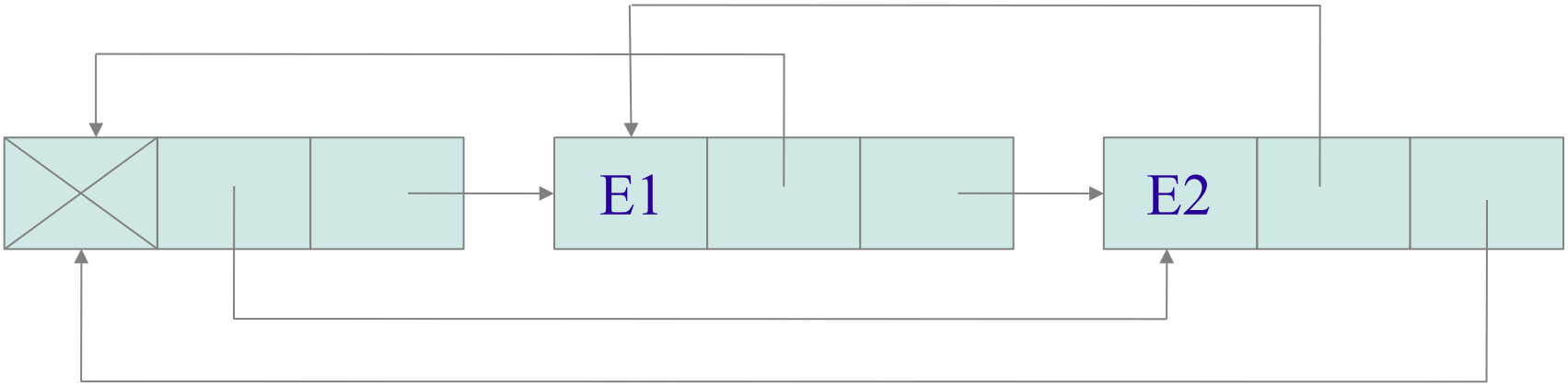
- Avantages:
  - accès aux éléments: O(1)
  - ajout / suppression d'éléments à la fin : O(1)
- Inconvénients:
  - ajout / suppression au début et au milieu : O(n) (nécessite de réallouer tout le tableau).

Les vecteurs

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> v1;
  v1.push_back(1); v1.push_back(2); v1.push_back(3);

  vector<int> v2(3); // Taille initiale : 3
  v2[0] = 1; v2[1] = 2; v2[2] = 3;

  cout << (v1 == v2) ? “Ok” : “Pas Ok” << endl;

  return 0;
}
```

Les listes

- Les listes proposées par la STL sont des listes doublement chaînées :



Les listes

- #include <list>
- Opérateurs : =, ==, !=, <, <=, >, >=
- Quelques méthodes :
  - size : nombre de valeurs stockées dans la liste
  - empty : vérifie si la liste est vide
  - clear : vide la liste
  - front, back : accès au premier ou au dernier élément de la liste

Les listes

- push_front, push_back : ajout d'un élément au début ou à la fin de la liste
- pop_front, pop_back : suppression du premier ou du dernier élément de la liste

- Quelques opérations :
  - sort : trie les éléments de la liste dans l'ordre croissant
  - reverse : inverse l'ordre des éléments de la liste

Les listes

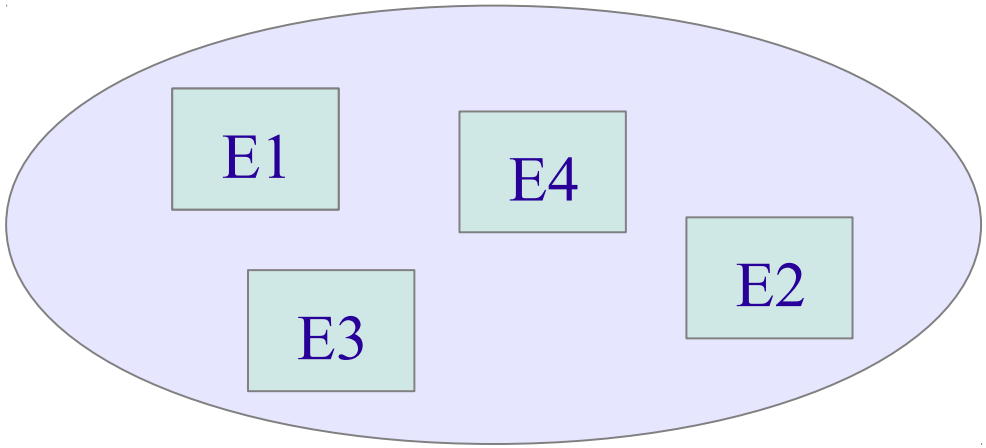
- Avantages:
  - ajout / suppression en tête ou fin: O(1).
- Inconvénients:
  - accès aux éléments: O(n)
  - emploi d'opérations spécialisées à la place d'algorithmes STL (sort, unique).
  - Insertions / suppressions au milieu: nécessite d'avoir vu les itérateurs ...

Les listes

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
  list<int> l;

  l.push_back(2);  l.push_back(1);  l.push_back(3);
  l.sort();
  l.reverse();

  // Affiche : 3 1
  cout << l.front() << ' ' << l.back() << endl;

return 0;
}
```

Les ensembles

- Les ensembles sont des collection d'éléments non ordonnés. Chacun de ces éléments est unique :



Les ensembles

- #include <set>
- Opérateurs : =, ==, !=, <, <=, >, >=
- Quelques méthodes :
  - size : nombre d'éléments de l'ensemble
  - empty : vérifie si l'ensemble est vide
  - clear : vide l'ensemble
  - insert : insertion d'un élément dans l'ensemble
  - erase : suppression d'un élément de l'ensemble
  - find : recherche d'un élément dans l'ensemble (renvoie un itérateur)

Les ensembles

- Les opérations sont toutes en O(n) (ajout, suppression, accès).

- En interne, les éléments sont toujours triés par ordre croissant.

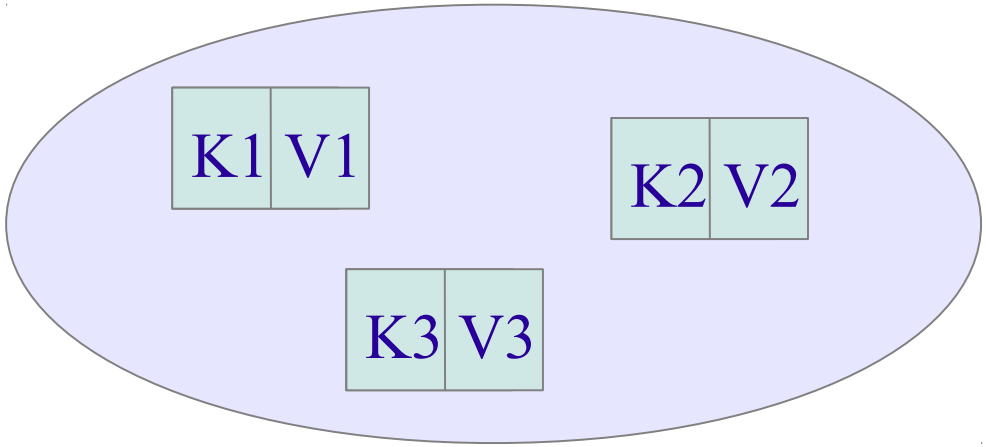
Les ensembles

```
#include <iostream>
#include <set>

using namespace std;

int main()
{
  set<int> ens;

  ens.insert(1); ens.insert(1);
  ens.insert(2); ens.insert(2);
  ens.erase(2);

  // Affiche : 1
  cout << ens.size() << endl;

  return 0;
}
```

Les dictionnaires

- Les dictionnaires sont des tableaux associatifs qui permettent d'associer un ensemble de valeurs et un ensemble de clefs.

- L'intérêt que les dictionnaires permettent de retrouver les valeurs à partir leurs clés.



Les dictionnaires

- Exemples d'utilisation :
  - le dictionnaire (clés = mots, valeurs = définitions) ;
  - l'annuaire (clés = noms, valeurs = coordonnées).

Les dictionnaires

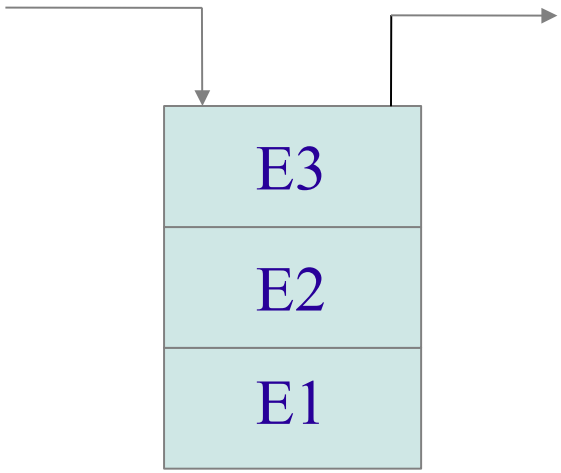
- #include <map>
- Opérateurs : =, ==, !=, <, <=, >, ≥, [] via la clé
- Quelques méthodes :
  - size : nombre d'éléments du dictionnaire
  - empty : vérifie si le dictionnaire est vide
  - clear : vide le dictionnaire
  - insert : insertion d'un élément dans le dictionnaire
  - erase : suppression d'un élément de le dictionnaire
  - find : recherche d'un élément dans le dictionnaire (renvoie un itérateur)

Les dictionnaires

```
#include <iostream>
#include <map>

using namespace std;

int main()
{
  // Paramétre 1: type clé, paramètre 2: type valeur
  map<string, string> agenda;

  agenda[“Aristide”] = “06-12-34-56-78”;
  agenda[“Paul”] = “03-86-38-12-34”;

  cout << “Numéro de Paul: “ << agenda[“Paul”] << endl;

  return 0;
}
```

D'autres types de conteneurs

- Les listes chaînées simples slist :
  - #include<slist>.
- Les piles (LIFO) stack :
  - #include<stack>
  - push : ajout
  - pop : lecture et retrait du sommet
  - top : lecture du sommet.



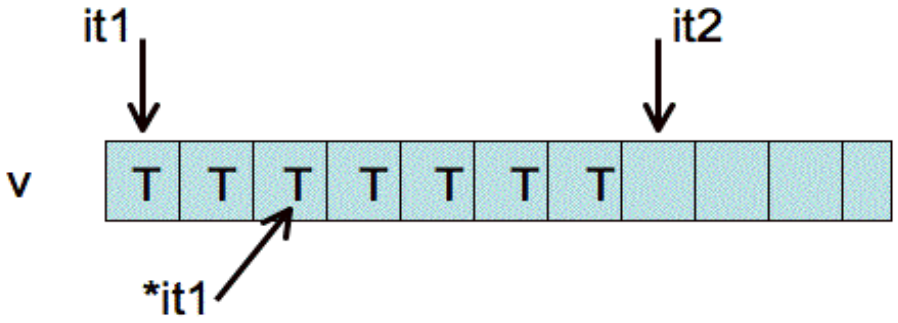
D’autres types de conteneurs

- Les files (FIFO) queue :
  - #include <queue>
  - push : ajout
  - pop : lecture et retrait du prochain élément.

Les itérateurs

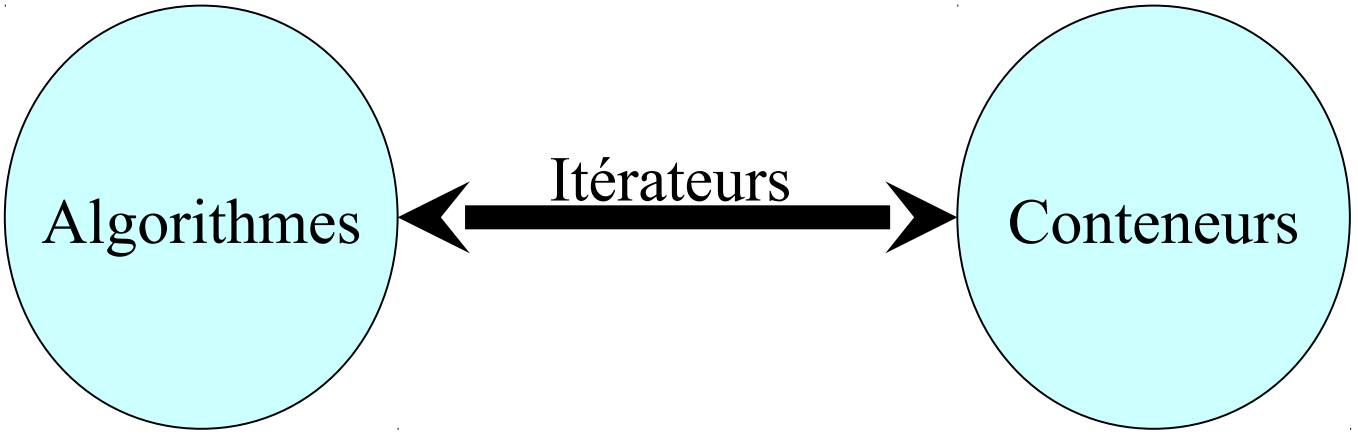
- Généralisation de la notion de pointeurs.
- Ils permettent de parcourir en séquence les éléments d'un conteneur, sans avoir à se préoccuper du type du conteneur

```
#include <vector>
{
  std::vector<int> v;
  //
 std::vector::iterator it1 = v.begin();
 std::vector::iterator it2 = v.end();
}
```



Les itérateurs

- Ils servent de liens entre les conteneurs et les algorithmes.



Les itérateurs

- Chaque conteneur fournit un type d'itérateur. Ex: vector<int>::iterator

- Chaque conteneur fournit deux itérateurs, accessibles par les méthodes:

  - begin(): itérateur sur le premier élément du conteneur.

  - end(): itérateur sur le premier emplacement vide du conteneur.

Les itérateurs

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
  list<int> l;

  l.push_back(1); (...) l.push_back(100);

  list<int>::iterator it;
  for (it = l.begin(); it != l.end(); ++it)
  {
    (*it) = (*it) + 1;
  }

  return 0;
}
```

Les algorithmes

- Les algorithmes sont des briques logicielles génériques (templates) capables d'opérer sur différents conteneurs de la STL.
- Ils sont fiables et optimisés (réduction drastique du code écrit).
- La plupart utilisent un itérateur de début et un itérateur de fin pour parcourir le conteneur.
- #include <algorithm>

count

- Compte le nombre d'éléments égaux à une valeur donnée.

```
vector<int> tab;

for (int i = 0; i < 3; ++i)
  for (int j = 0; j < 100; ++j)
    tab.push_back(j);

// Affiche : 3
cout << count(tab.begin(), tab.end(), 50) << endl;
```

equal

- Comparaison élément à élément de deux conteneurs (renvoie un booléen).
- Remarque : Les conteneurs peuvent être différents

```
list<int> l;
vector<int> v;

//...
bool res = equal(l.begin(), l.end(), v.begin(),
v.end());
```

find

- Recherche un élément.
- Renvoie un itérateur sur le premier élément égal à la valeur recherchée.

```
vector<double> vals;

vals.push_back(1); vals.push_back(2);
vals.push_back(3); vals.push_back(4);

vector<double>::iterator it1, it2;

it1 = find(vals.begin(), vals.end(), 3); // it1 -> vals[2]
it2 = find(vals.begin(), vals.end(), 5); // it2 ->
vals.end();
```

min_element / max_element

- Renvoient respectivement la valeur min et max d'un conteneur.

```
vector<int> v(3);

v[0] = 1; v[1] = 2; v[2] = 3;

int i = max_element(v.begin(), v.end());
int j = min_element(v.begin(), v.end());
```

sort / is_sorted

- 'sort' effectue un tri (opérateur <).
- 'is_sorted' renvoie un booléen pour dire si un conteneur est trié.

```
vector<int> v(3);

v[0] = 3; v[1] = 2; v[2] = 1;

cout << is_sorted(v.begin(), v.end()); // Affiche : false
sort(v.begin(), v.end());
cout << is_sorted(v.begin(), v.end()); // Affiche : true
```

copy

- Effectue une copie vers un autre conteneur.
- Attention : Il faut que le conteneur de destination soit de taille suffisante !

```
vector<int> v1(2);
vector<int> v2;

v1[0] = 1; v1[1] = 2;

v2.resize(v1.size());
copy(v1.begin(), v1.end(), v2.begin());
```

merge

- Fusionne 2 ensembles triés en 1 seul ensemble trié.

```
// Deux tableaux ...
int t1[] = {1, 3, 5, 7};
int t2[] = {2, 4, 6, 8};

// Fusion dans une liste ...
list<int> l;
merge(t1, t1 + 4, t2, t2 + 4,
back_insert_iterator<list<int> >(l));

// Fusion à l'écran ...
merge(t1, t1 + 4, t2, t2 + 4,
ostream_iterator<int>(cout, " "));
```

replace

- Remplace les occurrences d'une valeur par une autre.

```
// Création (avec une erreur)
list<string> tokens;
tokens.push_back(“Schumacher”);
tokens.push_back(“est”);
tokens.push_back(“le meilleur.”);

// Rectification de l'erreur
replace(tokens.begin(), tokens.end(), “Schumacher”,
“Alonso”);

// Affichage
copy(tokens.begin(), tokens.end(),
ostream_iterator<string>(cout, “ “));
```

set_union, set_intersection et set_difference.

- Opérations ensemblistes.

```
set<double> s1;
set<double> s2;

//...

set<double> res;
insert_iterator<set<double> > ii(res, res.begin());
set_intersection(s1.begin(), s1.end(), s2.begin(),
s2.end(), ii);
```