

# C++

---

Pier Donini ([Pier.Donini@heig-vd.ch](mailto:Pier.Donini@heig-vd.ch))

## Bibliographie sur C++

---

- **Le langage C++**  
Bjarne Stroustrup (Créateur du C++) Pearson
- **L'essentiel du C++**  
Stanley B. Lippman Addison-Wesley
- **Comment programmer en C++**  
Deitel & Deitel Eyrolles
- **Langage C++**  
Nino Silverio Eyrolles
- **Standards de programmation en C++**  
Herb Sutter et Andrei Alexandrescu Pearson (Addison-Wesley)

## Bibliographie: C/C++ sur le web

- Cours C/C++:
  - <http://casteyde.christian.free.fr>
  - <http://h.garreta.free.fr>
  - <http://cpp.developpez.com/cours>
  - <http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>
- Références C/C++:
  - <http://en.cppreference.com/w/cpp>
  - <http://www.cplusplus.com/reference>
- C++ FAQ:
  - <https://isocpp.org/faq>
  - <http://www.parashift.com/c++-faq>
- Tutoriel de Java à C++:
  - <http://www.infres.enst.fr/~elc/C++>

## Différences entre C++ et Java

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>■ Modularité: <code>.h</code> (déclaration) et <code>.cpp</code> (implémentation), pas de paquetages.</li><li>■ Pas de ramasse-miettes: mémoire explicitement allouée/désallouée (<code>new/delete</code>).</li><li>■ Héritage multiple, pas d'interfaces.</li><li>■ Polymorphisme explicite:<ul style="list-style-type: none"><li>• Par des pointeurs (flexibles mais pouvant être invalides) ou des références.</li><li>• Pas de liaison dynamique par défaut. Les méthodes pouvant en bénéficier doivent être déclarées virtuelles (<code>virtual</code>).</li></ul></li></ul> | <ul style="list-style-type: none"><li>■ Fonction, méthodes et classes <i>amies</i> (<code>friend</code>) pouvant accéder aux propriétés privées ou protégées d'une classe donnée.</li><li>■ Surcharge des opérateurs <code>=</code>, <code>+</code>, <code>+=</code>, <code>[]</code>, ... possible.</li><li>■ Généricité: fonctions et classes <i>template</i>.</li><li>■ C: pointeurs, types énumérés (<code>enum</code>), structures (<code>struct</code>)... avec plus de contrôle de type.</li><li>■ Pas de machine virtuelle: les programmes doivent être recompilés sur chaque plateforme.</li></ul> |
|---|---|

## En venant de Java...

- **Attention**, contrairement à Java,
  - En C++ les objets ne sont pas uniquement manipulés par des références!
- Soit une classe `Person`. Alors l'instruction...
  - `Person p("John", "Doe");`
  - ... construit **directement** l'objet `p` (qui est détruit à la sortie du bloc).
  - `cout << p.getName() << endl;`
- Les objets peuvent également être manipulés par des pointeurs:
  - `Person *ptr1 = &p;`
  - `Person *ptr2 = new Person("Jane", "Eod");` // allocation dynamique  
`cout << ptr2->getName() << endl;`
- Ou, comme en Java, par des références:
  - `Personne &ref = p;`  
`cout << ref.getName() << endl;`

## Affichage et saisie au clavier en C++

- Les **flux** d'entrée et de sortie standard (`cin` et `cout`) sont des **objets** (instances des classes `istream` et `ostream`)
- Opérateurs d'affichage `<<` et d'extraction `>>` d'un flux:
  - Opérande de gauche: le **flux**
  - Opérande de droite: l'**élément** (valeur ou objet) à afficher ou extraire ou un **manipulateur** de flux (`endl`, `setw()`, ...)
  - Chaque opération **rend** le flux modifié → chaînage des opérateurs  
`cout << a << " + " << b << " = " << (a + b) << endl;`
  - Opérateurs définis pour afficher et extraire des valeurs des types primitifs
  - Ils doivent être **surchargés** pour pouvoir être utilisés avec des **objets**
- Plus de détails dans le fichier [C++ \(iostream\)](#)

## Versions de C++

- Les versions 2 et 3 du C++ ont subi des refontes importantes.
- Entre autres:
  - Ajout de la librairie de classes utilitaires STL (*standard template library*).
  - Introduction des **espaces de nommage**, permettant de délimiter la recherche de noms des identificateurs (éviter les conflits).  
Syntaxe: `namespace nom '{' instructions '}'`
- Ainsi, le fichier d'en-tête `iostream.h` devient `iostream` et les objets `cout` et `cin` sont déclarés dans l'espace de nommage `std`
  - il faut préfixer ces objets du nom de leur espace de nommage:  
`std::cout << "hello";`
  - ou l'importer explicitement:  
`using namespace std;`  
`cout << "hello";`

## Encapsulation

- Principes de Parnas:
  - L'**utilisateur** d'un composant logiciel doit disposer de toute l'information nécessaire pour pouvoir utiliser ses services, et *ne pas avoir* d'informations sur son implémentation sujette à changement.
  - Le **programmeur** du composant doit disposer de toute l'information nécessaire pour réaliser les tâches assignées au composant, et *ne doit pas disposer* d'informations supplémentaires, ni publier des détails sujets à modifications.
- Exemple: les deux aspects d'un type `Pile`,

### Interface

```
estVide
estPleine
initialiser
empiler
desempiler
voirSommet
```

### Implémentation

```
const int limite = 40
int sommet
int pile[limite]
```

## Encapsulation (2)

- L'empaquetage (*bundling*) consiste à définir un ensemble de fonctions comme seul moyen de manipuler les propriétés d'un type. Ces fonctions définissent le **comportement** de l'abstraction.
- La protection des données (*information hiding*), limite l'accès aux données ainsi qu'aux fonctions et permet ainsi de contrôler les **états internes** de l'objet.
- En C++:
  - L'empaquetage est assuré par l'appartenance de méthodes à une classe.
  - La protection des données est gérée par des modalités d'accès (*access qualifiers*) de la classe qui peuvent être **private**, **protected** ou **public**.

<b>private:</b>	accessible depuis la classe.
<b>protected:</b>	accessible depuis la classe et ses descendants.
<b>public:</b>	accessible partout dans le programme.

## Définition d'une classe

- La déclaration doit précéder toute utilisation.
  - Généralement en début de fichier, ou dans un fichier **.h** à inclure.
  - Déclarer les attributs dans une section privée ou protégée.
  - Déclarer la signature des méthodes dans une section (en général) publique.
- *DéclarationClasse* = 

```
class NomClasse
{
    { Section }
};
```

  
*Section* = 

```
[ private: | protected: | public: ]
{ Déclaration }
```
- Prédéclaration d'une classe en cas d'interdépendance du type des attributs:

```
class B;
class A { B* ptrB; };
class B { A* ptrA; };
```

## Définition d'une classe (2)

- Définition de l'implémentation des méthodes.
  - Directement dans la classe:
    - méthodes implicitement `inline` (code recopié),
    - ne devraient comporter qu'une ou deux instructions.
  - En dehors de la classe, habituellement dans un fichier spécifique (`.cpp`):
    - méthodes non `inline` (sauf si indiqué explicitement),
    - le nom de la méthode doit être préfixé par le nom de sa classe,  
`TypeRetour NomClasse::methode(...) { /* ... */ }`
- Exemple:
  - Définition d'une classe `Date` contrôlant l'assignation de valeurs correctes pour ses attributs (e.g., le mois de février possède 28 ou 29 jours).
  - Principe POO: les objets sont responsables de la cohérence de leur état.

## Définition d'une classe (3)

```
class Date
{
private:
    int jour, mois, annee;
public:
    void definir(int, int, int);
    bool estBisextile(int a) const {
        return !(a % 4) && (a % 100 || !(a % 400));
    }
private:
    bool estValide(int jour, int mois, int annee);
};
-----
void Date::definir(int j, int m, int a) {
    if (estValide(j, m, a)) {
        jour = j; mois = m; annee = a;
    }
}
-----
Date d;
d.definir(31, 12, 2003);
```

Déclaration (`.h`)

// facultatif: accès par défaut privé

// méthodes de l'interface

// nom des paramètres facultatif

// méthode inline

// méthode privée (utilitaire)

// ne pas oublier le ;

Implémentation (`.cpp`)

Utilisation

## Méthodes constantes

- Méthodes permettant de manipuler les valeurs des attributs:
  - Ecriture: accesseurs en modification, `void setX(Type)`.
  - Lecture: accesseurs en consultation, `Type getX() const`. Elles peuvent (*doivent*) être déclarées constantes par le mot clef `const`.
- Une méthode déclarée constante indique et garantit qu'elle ne peut pas modifier l'objet (la valeur de ses attributs) sur lequel elle est invoquée.
  - Une erreur est générée à la compilation si une modification est effectuée.
  - Le mot clef `const` doit être spécifié à la déclaration et à l'implémentation (si celle-ci est fournie en dehors de la définition de la classe) de la méthode.

```
class Date {  
    /* ... */  
    void affiche() const;  
};  
  
void Date::affiche() const {  
    cout << jour << ": " << mois << ": " << annee;  
}
```

## Méthodes constantes (2)

- Avantages:
  - Améliorent la qualité du code en forçant une spécification rigoureuse.
  - Elles permettent d'appeler d'autres méthodes (constantes) d'un objet constant puisque celui-ci n'est pas modifié.
- Exemple:

```
class Date {  
    /* ... */  
    void affiche(); // non déclarée constante  
};
```

```
class Etudiant {  
public:  
    void affiche() const;  
private:  
    char* nom;  
    Date dateDiplome;  
};
```

```
void Etudiant::affiche() const {  
    if (nom)  
        cout << nom << " ";  
    dateDiplome.affiche();  
}
```

Ne compile pas: le compilateur ne sait pas que l'affichage ne modifie pas la date

## Méthodes privées

- De façon générale, les méthodes sont publiques et les attributs privés.
- Mais il est fréquent d'avoir besoin de méthodes *internes* à la classe dont on ne souhaite pas un emploi public.
  - Elles sont alors déclarées privées (ou protégées), à disposition exclusive des autres méthodes de la classe.
  - Par exemple, pour contrôler la validité d'une Date.

```
class Date {
    /* ... */
private:
    bool estValide(int j, int m, int a) const; // pourrait être statique
};

bool Date::estValide(int j, int m, int a) const {
    static int jours[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    return j > 0 && (m == 2 && estBisextile(a) && j < 30 ||
                    m > 0 && m < 13 && j <= jours[m - 1]);
}
```

## Méthodes inline

- Optimisation des méthodes simples: les déclarer `inline`.
    - Implicitement lors de la déclaration de la classe,
    - Explicitement lors de l'implémentation de la classe par le mot clef `inline`.
  - Les fonctions peuvent également être déclarées `inline` (explicitement).
  - Le compilateur remplace l'invocation de la méthode par un accès direct à son corps → Possible optimisation et augmentation de la taille de l'exécutable.
  - Le corps d'une méthode ou une fonction `inline` ne devrait comporter que peu d'instructions.
  - Alternative nettement préférable aux macros (`#define`) car les fonctions `inline` sont plus sûres et sont débogables.
- P.ex., contrairement aux macros, chaque paramètre n'est évalué qu'une fois:

```
#define unsafe(i)  ( (i) >= 0 ? (i) : -(i) )
inline int safe(int i) { return i >= 0 ? i : -i; }
unsafe(x++) // x est incrémenté deux fois!
safe(x++) ; // x est incrémenté une seule fois
```



## Méthodes inline (2)

- Explicitement lors de l'implémentation de la classe. L'implémentation de la méthode (ou fonction) `inline` doit être visible pour chacune de ses invocations.

```
class A {  
public:  
    void m();  
    void n(); // non inline (a.cpp)  
};  
inline void A::m() { /* ... */ }
```

a.h

Pour plus de lisibilité les méthodes `inline` peuvent être placées dans un fichier (`a.hpp`), à inclure à la fin du fichier `a.h`.

```
#include "a.h"  
A a;  
a.m();
```

Doit être visible

- Implicitement lors de la déclaration de classe (moins recommandé):

```
class A {  
public:  
    void m() { /* ... */ }  
};
```

a.h

## Surcharge

- C++ permet la surcharge de méthodes définies pour une classe (ou de fonctions) pour peu que celles-ci possèdent des **signatures différentes**:
  - Même nom,
  - Nombre différent de paramètres ou types des paramètres différents,
  - Type de résultat quelconque.
- De plus,
  - Que des paramètres formels soient déclarés constants (`const`) ou non n'intervient pas dans la comparaison des signatures.
  - Une méthode constante (`const`) peut être surchargée par une méthode de mêmes paramètres pouvant modifier l'instance en cours (non-`const`).
  - Une méthode héritée d'une super-classe peut être surchargée dans une sous-classe.
- Deux méthodes (ou deux fonctions) possédant la même signature et possédant des types de résultat différents sont ambiguës.

## Valeurs par défaut des paramètres

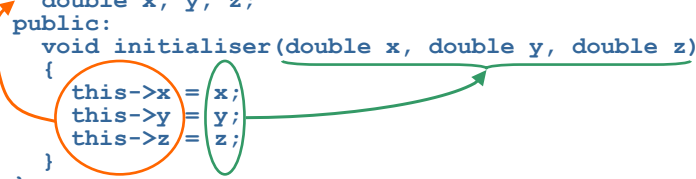
- Des valeurs par défaut peuvent être définies pour les paramètres des méthodes et des fonctions.
  - Si une valeur par défaut est définie pour un paramètre → tous les paramètres suivants doivent en posséder une,
  - A l'invocation une valeur peut ou non leur être fournie,
  - Si un paramètre est omis → la valeur de tous les suivants doit l'être aussi.
- → Une certaine concurrence avec la surcharge.
- Exemple:

```
void f(int a, double b = 3.14, char c = 'a') { /* ... */ }  
  
f(2, 5, 'b');  
f(3);  
f(1, 3.2);  
// f(1, 'b') incorrect!
```

## Le pointeur prédéfini `this`

- Le pointeur `this` est fourni dans toute méthode.
  - Il pointe sur l'instance en train d'être manipulée,
  - Il permet de préciser parfois l'objet concerné,
  - Il permet à l'objet de s'identifier vis-à-vis d'un autre (passé en paramètre).
- Exemple, distinguer les paramètres `x`, `y` et `z` des attributs `x`, `y` et `z`:

```
class Vecteur {  
    double x, y, z;  
public:  
    void initialiser(double x, double y, double z)  
    {  
        this->x = x;  
        this->y = y;  
        this->z = z;  
    }  
};  
  
Vecteur v; v.initialiser(1, 2, 3);
```



- Noter l'usage de `this->x` au lieu de `(*this).x`.

## Constructeurs

- Un constructeur est invoqué à la création de chaque instance:
  - Initialisation des valeurs des attributs de l'instance,
  - Autres actions à effectuer pour que l'instance soit dans un état cohérent.
- Un constructeur est une méthode possédant le **même nom que la classe**, **sans type de résultat** et d'un nombre variable de paramètres (ou aucun).
- Toute classe définit au moins un constructeur (surcharge des constructeurs).
- Un constructeur peut être déclaré privé ou protégé, mais il doit exister un *constructeur public* pour que la classe soit *instanciable* depuis l'extérieur.
- Si, et seulement si, aucun constructeur n'est spécifié pour une classe, le compilateur en définit implicitement un, sans paramètre :
  - Le **constructeur par défaut**, qui ne fait (*presque*) rien.
  - L'implémentation de ce constructeur par défaut invoque le constructeur sans paramètre de la classe parent (si elle existe).

## Invocation du constructeur

- Exemple:

```
class Date {
    int jour, mois, annee;
public:
    Date(int j, int m, int a);
};
Date::Date(int j, int m, int a) {
    jour = j; mois = m; annee = a;
}
Date d(1, 6, 2007);
Date* p = new Date(17, 7, 2007);
```

- Ici, il n'est plus possible d'écrire:

```
Date d; // attention: Date d(); est la déclaration d'une fonction d() !
Date* p = new Date();
```

- Il faudrait rajouter un constructeur sans paramètre ou définir des valeurs par défaut pour tous les paramètres d'un constructeur existant:

```
Date() { /* ... */ }
Date(int j = 1, int m = 1, int a = 2007);
```

## Destructeur

- Destruction d'un objet:
  - S'il est *global*, à la fin du programme,
  - S'il est *local*, lorsqu'on quitte son bloc d'instruction,
  - S'il est *dynamique* (alloué par l'opérateur `new`), par l'opérateur `delete` (ou `delete[]` pour les tableaux).
- Déclaration d'un **destructeur** (unique) pour la classe, pour libérer des ressources allouées par un objet lors de sa destruction.
- Propriétés du destructeur:
  - Ne possède pas de paramètres ni de type de résultat,
  - De même nom que la classe précédé du symbole `~`,
  - Nécessairement déclaré dans une section publique,
  - Ne doit (presque) jamais être invoqué explicitement, il est automatiquement invoqué à la destruction de l'objet.
  - Si aucun destructeur n'a été déclaré, le compilateur en fournit implicitement un, qui ne fait (presque) rien.

C++

23

## Destructeur (2)

- Déclarer un destructeur dès que des ressources ont été allouées dynamiquement (mémoire, fichiers, etc.) dans un constructeur.
  - P. ex., `new` dans un constructeur → `delete` dans le destructeur.
- Exemple:

```
class String
{
    char *value;
public:
    String(const char* s);
    ~String();
};

String::String(const char* s) {
    value = new char[strlen(s) + 1];
    strcpy(value, s);
}

String::~~String() {
    delete[] value;
}
```

```
void f()
{
    String s1("foo");
    String *s2 =
        new String("bar");
    // allocation dynamique

    /* ... */

    delete s2; // destruction
    s2 = 0;    // s2 indéfini
}
// s1 est détruit
```

C++

24

## Tableaux

- Déclaration d'un tableau d'objets:

```
Date array[10];
```

- Le constructeur sans paramètre est invoqué sur les objets du tableau,
- Un tel constructeur doit impérativement exister.

- Tableau dynamique:

```
Date *array = new Date[10];
```

- Construction des objets du tableau (constructeur sans paramètre).
- Appel d'un constructeur spécifique pour chacun des éléments:

```
Date *array = new Date[10](1, 1, 2000);
```

- Libération d'un tableau d'objets:

- Le destructeur est appelé pour chaque élément du tableau.
- Tableau dynamique: `delete[] array` lance la destruction.

- Tableau dynamique et types primitifs: `int *a = new int[10];` alloue un tableau d'entiers (libéré par `delete[] a;`) non initialisé. Ne pas confondre avec `int *p = new int;` qui alloue un entier (libéré par `delete p;`).

## La variable référence

- Le **type référence** est un type pointeur constant (caché) qui délègue au compilateur le soin d'effectuer les opérations `&`, puis `*`.

- A sa déclaration une référence (également appelée un **alias**) doit obligatoirement être initialisée pour se **référer** à une autre variable.

- Utilisation de l'opérateur `&` (= *ici* "référence à...")

```
double reel;  
double &refReel = reel;
```

- Ne peut plus être modifiée par la suite (= pointeur constant).

- Une fois initialisée s'utilise comme la variable qu'elle représente.

```
reel = -23.0;  
refReel *= 5;  
cout << reel;           // affiche -115.0  
cout << refReel;        // affiche la même chose
```

- Comportement idéal dans le cas d'un paramètre: `void f(Personne &p);`

- Pas de références universelles (sur le type `void`).

## Constructeur de copie

- Le compilateur fournit un **constructeur de copie** permettant de déclarer un objet et de l'initialiser au moyen d'un autre objet de la même classe.

- Analogue à l'initialisation d'une variable de type primitif par une autre (`int i = j;`):

```
String s("foo");  
String t(s); // utilisation du constructeur de copie
```

- Utilisé également lors du passage par **valeur** d'une instance.

```
void f(String x) { /* .. */ }  
f(s); // à l'exécution, déclaration String x(s);
```

- Comportement du constructeur de copie implicite:

- Il ne fait que **recopier les valeurs** de chacun des attributs de l'objet source dans le nouvel objet (copie superficielle).
- ➔ Pour les attributs de type objet, leur constructeur de copie est utilisé.
- ➔ Si la classe définit des attributs de type pointeur, leur valeur sera la même dans les deux objets: ils pointeront sur le même espace mémoire!

## Constructeur de copie (2)

- Exemple:

```
class String {  
    char* value;  
    /* ... */  
};  
  
String s("foo"); // constructeur String(char*)  
String t(s);
```

- Par défaut, `s.value == t.value`.
- Si `s.value` est modifié, `t.value` l'est aussi.
- Si `s` est détruit, le destructeur `~String()` libère la mémoire pointée par `value` (par `delete[] value`). Mais lorsque `t` est détruit, le destructeur essaiera de libérer la même mémoire ➔ catastrophe!
- Solution: redéfinir le constructeur de copie chaque fois que les attributs sont de type pointeur (à moins d'être certain de ce que l'on fait...).
- Constructeur public de la forme `Classe(const Classe& modele)`; où les attributs de l'objet source sont dynamiquement recopiés.

## Constructeur de copie (3)

- Exemple:

```
class String {
    char *value;
public:
    /* ... */
    String(const String&);
};

String::String(const String& s) {
    value = new char[strlen(s.value) + 1];
    strcpy(value, s.value);
}

String s("foo"), t(s); // → t.value != s.value
```

- Interdiction de la copie d'objet

- Lors de l'initialisation: définir le constructeur de copie dans une section privée (avec une implémentation vide),
- Lors d'une affectation: surcharger l'opérateur = dans une section privée.

## Constructeur à un seul paramètre

- La déclaration de variables de types primitif s'écrit généralement:

```
double taille = 23.75, nbObjets = 25;
```

- La même syntaxe peut être utilisée pour déclarer des instances si un *constructeur à un seul paramètre* est invocable.

- Un constructeur à un seul paramètre définit par défaut une **conversion implicite** pour le type de son paramètre.
- Un constructeur à plusieurs paramètres peut être invoqué si les paramètres suivants possèdent des valeurs par défaut (p.ex. `Date(int j, int m = 1, int a = 2007);`).

- `Date d1 = uneDate`  
→ `Date d1(uneDate)` invoque `Date(const Date&)` (const. de copie)
- `Date d2 = 1;`  
→ `Date d2(Date(1))` invoque `Date(int)` et `Date(const Date&)`
- `Date d3;`  
`d3 = 1;`  
→ `d3 = Date(1)` invoque `Date(int)` et l'opérateur d'affectation = entre `Dates` (surchargeable)

## Constructeur à un seul paramètre (2)

- Ces conversions implicites peuvent être interdites en déclarant un constructeur comme `explicit`.
  - P.ex. si le constructeur de `Date` est: `explicit Date(int annee);`

```
Date d1(1);           // légal
d2 = 1;               // illégal
Date d3 = 1;          // illégal
Date getDate() { return 1; } // illégal
```
- La création d'objets temporaires utilisés pour construire un objet (comme dans `A a = 1` → `A a(A(1))`) est généralement optimisée par le compilateur pour construire directement l'objet (`A a = 1` → `A a(1)`).
  - P.ex. pour ne pas l'effectuer avec g++ (GNU) il est nécessaire de compiler avec l'option `fno-elide-constructors`.
- Pour être tout à fait cohérent, pour les types primitifs,

```
double taille(23.75), nbObjets(25);
```

... sont des déclarations de variables également acceptées en C++.

C++

31

## Exemple

- Soit `s` un `string`, attention à ne pas confondre...
    - `string s1 = s;`
    - `string s2 = "foo";`
    - `string s3;`  
`s3 = s;`  
`s3 = "bar";`
- qui se traduisent respectivement par:
- `string s1(s)` → Invoque `string(const String& other)`
  - `string s2("foo")` → Invoque `string(const char* arg)`  
(optimisation de `string s2(string("foo"))`)
  - `string s3` → Invoque `string()`  
`s3.operator=(s)` → Invocations de l'opérateur `=` (surchargeable)  
`s3.operator=(string("bar"))` ou `s3.operator=("bar")`

C++

32



## Constructeurs: liste d'initialisation

- Dans le corps du constructeur les attributs de l'instance sont déjà construits:

```
class Note
{
    double valeur;
    Date date;
    Note(double valeur, int jour, int mois, int annee) {
        // this->valeur vaut n'importe quoi
        // this->date initialisée par Date()
        this->valeur = valeur;
        date.setDate(jour, mois, annee);
    }
};
```

- → Utiliser une **liste d'initialisation** pour éviter ces initialisations par défaut:

```
Note(double valeur, int jour, int mois, int annee)
: valeur(valeur), date(jour, mois, annee)
{
    // Initialisation par Date(jour, mois, annee)
}
```

## Constructeurs: liste d'initialisation (2)

- La liste d'initialisation permet d'instancier un attribut de type objet au moyen de n'importe lequel des constructeurs de sa classe (y compris celui de copie).
  - Rend possible l'instanciation d'un attribut dont la classe n'offre pas de constructeur sans paramètre.

```
class Person
{
    String name;
    int age;
public:
    Person(const char* s, int i) : name(s), age(i) { /*...*/ }
    Person(const Person& p) : name(p.name), age(p.age) { /*...*/ }
};
```

String(const String& o)  
String(const char\* s)

- L'allocation dynamique est possible dans une liste d'initialisation.  
P.ex., attribut `Date *date` → `date(new Date(jour, mois, annee))`
- Une liste d'initialisation est également le seul endroit où peuvent être initialisés des attributs constants (`const`).

## Constantes

- Il est souhaitable d'utiliser à bon escient le mot clef `const` pour éviter des modifications de variables, pointeurs et références.
- Variable:
  - Déclarée non modifiable, doit être initialisée,  
`const int x = 3; const String s = "foo";`
- Pointeur:
  - Pointeur et données modifiables,  
`char* p = "foo"; p = "bar"; *p = 'B';`
  - Pointeur modifiable, données non modifiables,  
`const char* p = "foo"; p = 0; // *(p+1) = 'X' incorrect`
  - Pointeur non modifiable, données modifiables, doit être initialisé,  
`char* const p = "foo"; *(p+1) = 'X'; // p = "bar" incorrect`
  - Pointeur et données non modifiables, doit être initialisé,  
`const char* const p = "foo";`

## Constantes (2)

- Référence:
  - **Alias** d'une variable. Sémantiquement comparable à `x* const` (pointeur non modifiable, données modifiables).
  - ➔ une référence doit donc être initialisée (à une variable) lors de sa déclaration,
  - Données non modifiables: `const String& x = s;`
- Passage de paramètres:
  - Non modification des données originales,  
Par référence constante (préférable): `void f(const String& s);`  
Par pointeur constant (peut être `NULL`): `void f(const String* ptr);`  
Par valeur (copie / constructeur de copie): `void f(String s);`
  - Modification autorisée,  
Par référence: `void f(String& s);`  
Par pointeur: `void f(String* ptr);`

## Types résultat

- Valeur:
  - `x f() { /* ... */ }`  
`x x = f();` // le résultat de f est copié (cons. de copie pour les objets)
- Pointeur:
  - `x* f() { /* ... */ }`  
`x* x = f();` // copie de pointeur rendu
  - Ne pas rendre un pointeur sur une variable locale de `f()` non allouée dynamiquement (détruite en sortie)!
- Référence:
  - `x& f() { /* ... */ }`
  - `x& y = f();` // copie de référence rendue
  - `x x = f();` // copie de la valeur de la variable référencée rendue
  - `x z;`  
`f() = z;` // `f()` rend une référence → peut être manipulée
  - Ne pas rendre une référence sur une variable locale de `f()` non allouée dynamiquement!

## Visibilité des déclarations

- Rappel, indépendamment du concept d'objet,
- Une **fonction** déclarée dans un fichier est **globale**:
  - Elle peut être invoquée par toute autre fonction du fichier,
  - Elle peut être invoquée par toute fonction d'un autre fichier.
- Une **variable** déclarée **hors d'une fonction** est **globale**:
  - Elle est accessible par toutes les fonctions du fichier,
  - Elle est accessible par toute fonction d'un autre fichier.
  - Elle reste active du début à la fin du programme
- Une **variable** déclarée **dans une fonction** est **locale**:
  - Elle n'est accessible que par le code de cette fonction,
  - Elle n'existe, possède une valeur, que lors de l'exécution du code,
  - Elle est de nouveau déclarée à chaque invocation de la fonction.

## Modificateur `static`

- **Fonction**, `static void f(...) { /* ... */ }`
  - Devient **invisible** à partir d'un autre fichier (devient locale).
  - Reste accessible dans tout le fichier où elle est déclarée.
- **Variable**, `static int i = 0;`
  - N'est déclarée qu'une seule fois.
  - **Globale**:
    - Devient **invisible** à partir d'un autre fichier (devient locale).
    - Reste accessible dans tout le fichier où elle est déclarée.
    - Reste active du début à la fin du programme.
  - **Locale** (à un bloc `{ /* ... */ }` d'une fonction):
    - Conserve son existence et sa valeur entre les différentes invocations de la fonction.

## Propriétés statiques

- Une **propriété statique** (ou *de classe*) appartient à la classe (et non à l'instance) et est disponible même s'il n'existe aucune instance de la classe.
  - Accès sur la classe `Classe::propriété`, sur un objet `objet.propriété`.
- **Attribut statique** (constantes générales, compteur d'instances, etc.):
  - Possède la **même valeur** pour tous les objets de la classe.
  - Doit être initialisé en dehors de la déclaration de la classe (sauf si `const`).
- **Méthode statique** (méthode utilitaire, etc.):
  - Ne peut accéder aux propriétés non statiques (pas de pointeur `this`).
- Exemple:

```
class Date {  
    static Date courant;  
public:  
    static void setCourant();  
    void set(long time);  
};  
  
Date::setCourant(); // ou Date d; d.setCourant();
```

```
Date Date::courant(1,1,2000);  
  
void Date::setCourant() {  
    courant.set(time(0));  
}  
// NB: pas de mot clef static
```

} **cpp**

} **.h**

## Exemple

```
■ class Cat
{
    static int nb;
    int id;
public:
    Cat() : id(nb++) {
        cout << "Nouveau chat (" << id << ")" << endl;
    }
    static void info() { // possible d'accéder à nb mais pas id
        cout << nb << " chats crees" << endl;
    }
    void mreow() {
        cout << "Le chat #" << id << " miaule..." << endl;
    }
};
int Cat::nb = 0; // initialisation

■ for (int i = 0; i < 3; i++)
    Cat(); // chat anonyme
Cat::info();
Cat c;
c.mreow();
c.info(); // préférer Cat::info()
```

Résultat:

Nouveau chat (0)  
Nouveau chat (1)  
Nouveau chat (2)  
3 chats crees  
Nouveau chat (3)  
Le chat #3 miaule...  
4 chats crees

## Types internes

- Définition de types (classes, énumérés...) dans une classe ou une structure, de visibilité donnée (selon la section où ils sont déclarés).
- Semblables aux classes statiques Java (sans lien avec l'objet englobant).
  - Pas de visibilité automatique de la classe englobante sur un type interne (ses propriétés doivent être publiques).
  - Hors classe englobante, le type interne est accédé en le préfixant du nom de la classe englobante (**Classe::Type**).

```
class A {
public:
    enum count { one, two, three };
    class B {
        int x;
    public:
        void f();
    };
    A() { B b; b.f(); // mais pas b.x = 2
    }
};
```

```
void A::B::f() {
    count c = one;
    /* ... */
}

int main() {
    A::count c = A::one;
    A::B b;
    b.f();
}
```

## Exercice: types énumérés sûrs

- En s'inspirant du langage Java, implémenter un type énuméré sûr au moyen d'une classe `Season` de manière à ce que le code,

```
for (int i = 0; i < Season::size(); i++)
{
    const Season& s = Season::get(i);
    cout << s.name() << " " << s.index() << endl;
}
```

- produise le résultat:

```
Spring 0
Summer 1
Autumn 2
Winter 3
```

## Opérateurs

- Evaluation d'une expression par le compilateur,
  - Analyse, selon la *priorité* et le *groupement* des opérateurs
  - Pour chaque opérateur rencontré:
    - Extraction la valeur des opérandes concernés (1 ou 2),
    - Invocation d'une *fonction* qui effectue les calculs requis.
  - P. ex. si `x` a et `y` b, l'évaluation de l'expression `a + b`, est effectuée au moyen d'une fonction `Z operator + (X x, Y y)`.
- Dans le cas de types simples (`int`, `char`, `float`, ...):

Opérandes	Fonction associée	Exemple
1	<code>Δ x → operator Δ (x)</code>	<code>++a → operator ++ (a)</code>
2	<code>x Δ y → operator Δ (x, y)</code>	<code>a - b → operator - (a, b)</code>

## Opérandes objets

- A priori, aucune fonction définie → erreur de compilation.
- Il est possible de surcharger chaque opérateur pour chaque classe définie.
- Particulièrement intéressant avec les nombres complexes, les fractions, les vecteurs, les matrices, les chaînes de caractères...
  - → Rend les expressions mathématiques plus facile à lire.
  - Des expressions avec des objets sont alors possibles:
 

```
Vecteur v1(1, 0), v2(2, 3);      String s = "foo";
Vecteur v = v1 * v2;             s += "bar";
```
- Le niveau de *priorité* et le *regroupement* restent fixés.
- Surcharge des opérateurs par une *fonction* ou une *méthode*.
- Attention:
  - Un des opérandes **au moins** doit être une **classe** ou un **type énuméré**.


## Opérateurs acceptant la surcharge

- Opérateurs pouvant être surchargés:

+	-	*	/	%	^	&		~	!
=	<	>	+=	-=	*=	/=	%=	^=	&=
=	<<	>>	<<=	>>=	==	!=	<=	>=	&&
	++	--	->	->*	[]	()	,	new	delete

- Opérateurs ne pouvant être surchargés: `.` `.*` `::` `?:` `sizeof`
- Impossible de définir de nouveaux opérateurs comme:
 

```
|x|, y := x, y = x**2
```
- Opérateurs prédéfinis pour les classes: `=` `&` `Classe o(Classe& i)`


const. copie
- Éviter de surcharger `&&`, `||`, `!` ou `,`

## Surcharge d'opérateur par une fonction

- Opérateur à une opérande: `Type operator ⊕ (Op x)`
  - `enum Jour { lun, mar, mer, jeu, ven, sam, dim };`
  - `Jour& operator ++(Jour& j) { // modifie le paramètre → référence`  
    `if (j == dim) j = lun;`  
    `else ++((int&) j); // référence sur int pour pouvoir utiliser ++`  
    `return j;`  
    `}`
  - `Jour j = mar;`  
    `++j; // ou operator++(j);`
- Opérateur à deux opérandes: `Type operator ⊕ (Op1 x, Op2 y)`
  - `Jour operator +(Jour j, int i) {`  
    `return (Jour) (((int) j) + i) % 7);`  
    `}`
  - `Jour k = j + 12; // ou Jour k = operator+(j, 12);`
- Si il faut accéder à des propriétés privées de la classe → **méthode** opérateur.

## Surcharge d'opérateur par une méthode

- L'**instance** sur laquelle est appliquée l'opérateur (**this**) est le **premier opérande** (de gauche).
- A utiliser de préférence quand cela est possible.
- Une fonction est nécessaire quand:
  - Le type du premier opérande n'est pas une classe.
  - Le type du premier opérande est une classe non modifiable,
  - Ou quand on désire une conversion de type du premier opérande.
  - P. ex. pour un `String s`, `s + "foo"` peut être évalué par une méthode  
    `String String::operator +(const char*) const`  
    alors que `"foo" + s` nécessite une fonction  
    `String operator +(const char*,const String&).`
- Opérateur à un opérande: `Type Classe::operator ⊕ ()`
- Opérateur à deux opérandes: `Type Classe::operator ⊕ (Op x)`



## Exemple

```
class Vector {
    int x, y;
public:
    Vector(int x, int y) : x(x), y(y) { }
    Vector operator -() const;
    Vector operator +(const Vector& v) const;
    Vector& operator +=(int i);
};

Vector Vector::operator -() const {
    return Vector(-x, -y);
}
Vector Vector::operator +(const Vector& v) const {
    return Vector(x + v.x, y + v.y);
}
Vector& Vector::operator +=(int i) { // rend une référence (chaînage)
    x += i; y += i;
    return *this;
}

Vector v = -v1 + v2;    // ou v = v1.operator-().operator+(v2);
(v += 42) += 24;        // ou v1.operator+=(42).operator+=(24);
```

C++

49

## Choix entre méthode et fonction

- En général un opérateur ne sera déclaré en tant que méthode que dans les cas où `this` joue un rôle privilégié par rapport aux autres arguments.
- Méthodes:
  - Affectations (`=`, `+=`, `*=` ...)
  - Opérateur « fonction » (`()`)
  - Opérateur « crochets » (`[]`)
  - Opérateurs d'indirection/appel (`*`, `->`, `->*`)
  - Opérateurs d'in(dé)crémentation (`++`, `--`)
  - Opérateurs de décalage (`<<`, `>>`)
  - Opérateurs `new` et `delete`
- Fonctions:
  - Opérateurs de lecture/écriture dans un flux (`<<`, `>>`)
  - Opérateurs arithmétiques (`+`, `*`, `/`, ...)

C++

50

## Opérateurs particuliers

- Définir un ensemble d'opérateurs cohérent:
  - `a += b` doit être  $\equiv a = a + b$
  - `a + (-b)` doit être  $\equiv a - b$
  - `-(-a)` doit être  $\equiv +a \equiv a$
- En C++, le compilateur ne transforme pas `v += w;` en `v = v + w;`  
→ il faut donc définir les opérateurs `+`, `=`, et `+=`.
- De même, `x++;` n'est pas nécessairement équivalent à `x = x + 1;`
- Les opérateurs `++` et `--` acceptent deux formes: post- et pré- in(dé)crément.
  - Pré-in(dé)crément (`++i`, `--i`):
    - Méthode: `Classe& Classe::operator ++()`
    - Fonction: `Classe& operator ++(Classe& v)`
  - Post-in(dé)crément (`i++`, `i--`), opérande muet de type `int`:
    - Méthode: `Classe Classe::operator ++(int)`
    - Fonction: `Classe operator ++(Classe& v, int);`

## Opérateurs particuliers (2)

- L'opérateur `=` par défaut n'est défini que sur des objets de même classe et ne copie que les valeurs des attributs.
  - A éventuellement surcharger dans une méthode si des attributs sont des pointeurs (c.f. constructeur de copie) ou pour accepter d'autres types.
  - `Classe& Classe::operator = (const Type& o);`  
// Ne pas modifier `o` (`const`) si `Classe = Type` → tester `&o != this`.
- Opérateur de conversion de type (*transtypage*), `(Type) obj`, surchargé par une méthode `Classe::operator Type()` (sans type résultat).
  - ```
Vector::operator double() const {  
    return sqrt(x * x + y * y);  
}
```

  
`Vector v(3, 3); double d = v; // invoque v.operator double();`
  - Conversion d'un `String` en un `const char*` (pour utiliser des bibliothèques):  

```
class String {  
    char* value;  
public:  
    operator const char*() const { return value; }  
};
```

  
Rend un pointeur interne à l'objet: ne devrait pas être stocké ni modifié!

## Opérateurs particuliers (3)

- L'opérateur `[]` peut être surchargé par une méthode pour accéder aux éléments d'objets listes, vecteurs, chaînes de caractères...

```
char& String::operator[](int i) {  
    assert(i >= 0 && i < strlen(value));  
    return value[i];  
}  
String s = "toto";  
cout << s[3]; // imprime 'o'  
s[0] = 'm';    // donne "moto"
```

- L'opérateur `()` peut être surchargé avec un nombre quelconque de paramètres afin de donner à un objet le comportement d'une fonction. Par exemple pour accéder aux éléments de matrices:

```
class Matrice {  
    /* ... */  
    double& operator () (int i, int j) { /* ... */ }  
};  
Matrice m(10, 20);  
double d = m(2, 2); m(3, 3) = d + 1;
```

C++

53

## Opérateurs particuliers (4)

- L'opérateur de prise d'adresse `&` (unaire) peut être surchargé afin de rendre une autre adresse que celle de l'objet sur lequel il est appliqué.
- L'opérateur de déréférencement `*` (unaire) peut être surchargé afin de donner à un objet un comportement de pointeur et permet de rendre une référence sur un type donné.
  - En général `x == *&x` devrait être vrai.
- L'opérateur d'accès à une propriété `->` peut être surchargé par une méthode afin de donner à un objet un comportement de pointeur et permet d'accéder à une propriété donnée d'un type donné. Son type de retour doit être:
  - Un type pointeur sur lequel la propriété référencée est accédée, ou
  - Un type pour lequel l'opérateur `->` est défini et sur lequel cet opérateur va être utilisé (récursivement) jusqu'à ce qu'un type pointeur soit rendu.
  - P.ex. soient les méthodes `void C::m()`, `C* B::operator->()` et `B A::operator->()`, et soit `A a`, `a->m()` est valide et invoque `C::m()`.

C++

54

## Exemple

```
class A {
public:
    void m() { cout << "A::m()" << endl; }
    ~A()     { cout << "~A()" << endl; }
};

class APointer {
    A* _ptr;
public:
    APointer(A* ptr) : _ptr(ptr) { }
    A* operator&() const { return _ptr; }
    A& operator*() const { return *_ptr; }
    A* operator->() const { return _ptr; }
};

APointer ap(new A());
(*ap).m(); // (ap.operator*()).m(), affiche A::m()
ap->m();   // (ap.operator->())->m(), affiche A::m()
delete &ap; // delete (ap.operator&()), affiche ~A()
```

C++

55

## Accès aux membres privés d'une classe

- Seules les méthodes elles-mêmes d'une classe peuvent accéder à ses membres privés.
- Dans certains cas il est désirable d'étendre ce droit à différents éléments:
  - à une *fonction extérieure* (affichage, saisie...).
  - à une *méthode* d'une autre classe.
  - à *toutes les méthodes* d'une *autre classe* → à la classe.
- Cela est possible, en enregistrant comme "amie" (*friend*) la fonction, la méthode ou la classe privilégiée.
- Dans la classe qui veut autoriser l'accès à ses propriétés, les *éléments* privilégiés sont énumérés en les précédant du mot clef **friend**.
- Ceci permet de lever partiellement l'encapsulation des propriétés privées d'une classe.

C++

56

## Fonctions amies

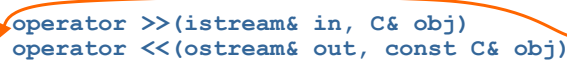
- Parfois une fonction convient pour réaliser une tâche (méthode inadéquate). P.ex. pour la surcharge d'opérateurs ne possédant pas une classe comme premier paramètre ou si cette classe n'est pas modifiable.
- Elle peut avoir besoin d'accéder aux membres privés des classes opérandes:  
→ La déclarer comme **amie** des classes opérandes.
- Dans la classe qui autorise une fonction à le faire reproduire l'en-tête de la fonction privilégiée en la précédant par le mot clef **friend**.
- P. ex., pour pouvoir évaluer "foo" + s, où s est un **String**, la fonction **String operator +(const char\* c, const String& s)** doit pouvoir accéder à l'attribut **value** de la classe **String**.

```
class String {  
    /* ... */  
    friend String operator +(const char* c, const String& s);  
};
```

## Exemple: opérateurs << et >>

- Pour effectuer l'affichage et la saisie d'un objet **obj** d'une classe **C** il est souhaitable de pouvoir écrire **cout << obj;** et **cin >> obj;**
- → Déclarer les fonctions suivantes (classe du 1<sup>er</sup> opérande non modifiable):  

```
istream& operator >>(istream& in, C& obj)  
ostream& operator <<(ostream& out, const C& obj)
```



Pour pouvoir chaîner les opérateurs:  
**cout << a << b;**
- Si ces fonctions doivent accéder à des membres privés de la classe **C** (souvent le cas) elles seront enregistrées comme amies dans la classe:

```
class C {  
    /* ... */  
    friend istream& operator >>(istream &, C&);  
    friend ostream& operator <<(ostream &, const C&);  
};
```

- Même principe pour la surcharge d'opérateurs d'entrée/sortie (**fstream**, **ifstream**, **ofstream**...) devant accéder aux propriétés d'une classe.

## Méthodes et classes amies

- De même, une classe peut donner accès à un ensemble de méthodes d'autres classes → **méthodes amies**.
  - Dans cette classe reproduire l'en-tête des méthodes privilégiées en les précédant par le mot clef **friend** et du nom de leurs classes.

```
class Vector {  
    /* ... */  
    friend Vector Matrix::getColumn(int i) const;  
};
```
- Ou à toutes les méthodes d'autre classes → **classes amies**.
  - Dans la classe qui l'autorise reproduire le nom des classes privilégiées en les précédant par le mot clef **friend**.

```
class Vector {  
    /* ... */  
    friend Matrix;  
};
```
- Remarque: la propriété d'amitié des classes n'est ni transitive, ni symétrique.

## Bonnes pratiques

- Une classe **T** est dite sous **forme canonique** si elle offre au moins les méthodes suivantes:

```
class T  
{  
    T(...);                // Constructeurs  
    T(const T&);            // Constructeur de copie  
    ~T();                  // Destructeur  
    T& operator=(const T&); // Operateur d'affectation  
};
```
- En général, s'il existe une allocation de ressources dans un constructeur (**new/new[]**)
  - Désallocation dans le destructeur (**delete/delete[]**),
  - Définition du constructeur de copie avec allocation,
  - Surcharge de l'opérateur **=** avec désallocation (courant) et réallocation.

## Héritage

- Déclaration d'une classe dérivée:
  - Indiquer de quelle(s) classe(s) elle hérite (dérive). Relation: *est-un*.
  - Ajouter des propriétés spécifiques (attributs & méthodes).
  - Redéfinir ou surcharger des méthodes héritées.

```
class Derivee : [accès] Parent { , [accès] Parent }
{
    // Déclarations, redéfinitions et surcharges
};
```

**accès:** redéclaration de l'accès aux propriétés héritées depuis la sous-classe.

  - **public:** toutes les propriétés conservent la même visibilité.
  - **protected:** **public** dans la sur-classe → **protected**.
  - **private:** toutes les propriétés deviennent **private**.- Remarque: l'héritage privé est une variante à la composition simple, *possède-un* (réalisée au moyen d'un attribut de type **Parent**).
- Il n'existe pas de super-classe racine en C++ (classe **Object** en Java).

C++

61

## Héritage: constructeurs et destructeur

- Pour initialiser une instance d'une sous-classe:
  - D'abord initialiser les attributs hérités,
  - Puis initialiser les attributs spécifiques.
- **Constructeurs:** appel au(x) constructeur(s) de(s) super-classe(s),
  - Par défaut invocation du constructeur sans paramètre de la super-classe (qui doit exister).
  - Le constructeur de copie par défaut invoque le constructeur de copie de la super-classe.
  - Invocation d'un constructeur spécifique dans la liste d'initialisation:

```
class Vector3D : public Vector2D {
    int z;
public:
    Vector3D(int x, int y, int z) : Vector2D(x, y), z(z) { }
};
```
- **Destructeur:** le destructeur d'une sous-classe invoque automatiquement le(s) destructeur(s) de sa (ses) super-classe(s).

C++

62

## Polymorphisme

- Principe de **substituabilité**:
  - Il est possible de fournir une instance d'un sous-type quand une instance d'un sur-type est attendue (affectations, passages de paramètres).
- En C++ cela est vrai pour les variables, pointeurs et références.
  - Pour les variables, le constructeur de copie du type attendu est utilisé et ne copie que la partie *connue* du super-type de l'instance.  
P.ex. `Vecteur2D v = unVecteur3D;`, `v` est *seulement* un `Vecteur2D`.
  - Pour les pointeurs et les références, l'instance référencée complète est disponible (pointeurs et références possédant la même taille quelque soit la variable référencée).  
P.ex. `Vecteur2D& v = unVecteur3D;`, `v` référence bien un `Vecteur3D`.  
→ il est possible de mettre en œuvre le mécanisme de liaison dynamique.

## Liaison dynamique

- Ne fonctionne qu'avec les **références** et les **pointeurs**.
- Les méthodes pouvant être redéfinies doivent explicitement le déclarer par le mot clef `virtual` (sinon, elles sont simplement surchargeables).  
→ La recherche de la méthode la plus spécialisée s'effectue dans les sous-classes (même si la méthode n'est plus définie comme virtuelle).
- Dans le corps de la méthode redéfinie il est possible d'invoquer une méthode originale (de n'importe quel niveau) en la préfixant du nom de sa classe:  
`Classe::methode(paramètres);`

- Exemple:

```
class A {  
public: virtual int m();  
};  
int A::m() { /* ... */ }
```

```
class B : public A {  
public: int m();  
};  
int B::m() { return A::m() * 2; }
```

```
A* a = new B(); cout << a->m() << endl;
```

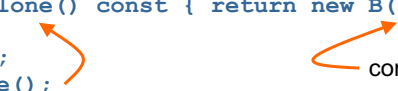
`virtual` optionnel ici



## Liaison dynamique (2)

- Une méthode redéfinie peut posséder un type de retour plus spécialisé que celui de la méthode originale si c'est un pointeur ou une référence:  
→ **covariance du type résultat** (partielle).
- Les constructeurs ne peuvent être virtuels.
  - Pourtant le constructeur de copie mériterait d'être virtuel.  
→ Définir une méthode virtuelle, utilisant le constructeur de copie.

```
class A {  
    /* ... */  
    virtual A* clone() const { return new A(*this); }  
};  
class B : public A {  
    /* ... */  
    virtual B* clone() const { return new B(*this); }  
};  
A* a = new B();  
A* b = a->clone();  
  
delete a; delete b; // Attention !
```



constructeur de copie

## Liaison dynamique (3)

- Le mécanisme de liaison dynamique n'opère pas dans un constructeur (contrairement à Java) ni dans un destructeur.
- Les destructeurs doivent être déclarés virtuels dès qu'un objet peut être détruit à partir d'un pointeur sur une super-classe.
  - Règle simple: déclarer un destructeur virtuel dès qu'une autre méthode est déclarée virtuelle dans la classe (→ le polymorphisme sera utilisé).
  - Dans l'exemple précédent il faut donc rajouter :  
`virtual ~A() { /* ... */ }` et `virtual ~B() { /* ... */ }`.
- **Méthode abstraite:**
  - Définie comme *virtuelle pure*, `virtual void m() = 0;` (sans corps).
- **Classe abstraite:**
  - Si et seulement si elle définit des méthodes abstraites.
  - Ne peut être instanciée directement.
- Une méthode d'une section privée n'a aucune raison d'être déclarée virtuelle.

## Masquage de méthode

- Lorsqu'une méthode est définie dans une sous-classe, elle masque **toutes** les méthodes de même nom de ses super-classes.
- Le mot clef **using** permet d'importer un espace de nommage ou une partie de l'espace de nommage dans le contexte courant.
- Utile pour étendre la résolution de la méthode à invoquer aux super-classes.

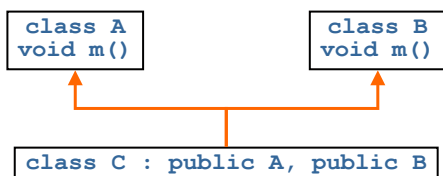
```
class A {
public:
    int  foo(int)      { cout << "A::foo(int)" << endl; }
    char foo(char)     { cout << "A::foo(char)" << endl; }
};
class B : public A {
    using A::foo;
    char foo(char)     { cout << "B::foo(char)" << endl; }
};
B b;
b.foo('a');           // B::foo(char)
b.foo(1);             // A::foo(int), sans la clause using: B::foo(char)
```

C++

67

## Héritage multiple

- Ambiguïtés:



`C c; // c.m();` est ambigu: erreur

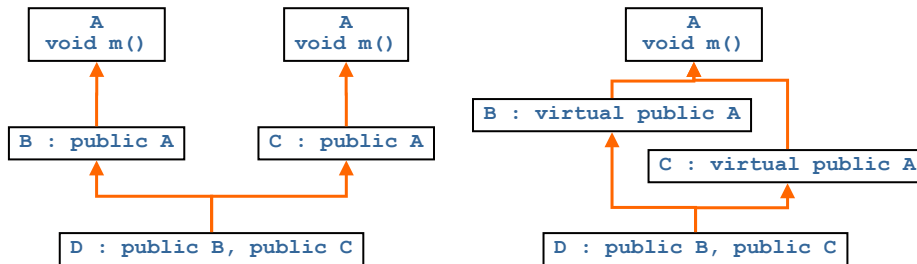
- Solution:
  - Soit spécifier le contexte de la méthode désirée. P.ex. `c.A::m()` ;.
  - Soit redéfinir dans `C` la méthode `m()` .  
P.ex. `void C::m() { A::m(); B::m(); }`

C++

68

## Héritage multiple en losange

- Classe de base (**A**) non virtuelle.
  - Chaque sous-classe hérite de sa *propre* version de **A**.
  - → Deux versions de **A** pour **D**. et donc de ses attributs!
  - → **D** **d**; **d.m()** est ambigu. Il est nécessaire de préciser le point de vue, p.ex. **d.B::m()**
- Classe de base (**A**) virtuelle.
  - Une version de **A** pour **D**.
  - Invocation du constructeur de **A** depuis **D**, implicite (cst. par défaut) ou explicite: les invocations depuis ceux de **B** et **C** sont ignorées!



C++

69

## Exemple

- ```
class A {
    int a;
public:
    A(int x) : a(x) {
        cout << "A(" << a << ")\n";
    }
};
```
- ```
class B : virtual public A {
    int b;
public:
    B(int x) : b(x), A(x) {
        cout << "B(" << x << ")\n";
    }
};
```
- ```
class C : virtual public A {
    int c;
public:
    C(int x) : c(x), A(x) {
        cout << "C(" << x << ")\n";
    }
};
```
- ```
class D : public B, public C
{
public:
    D(int x, int y)
        : A(x + y), B(x), C(y) {
    }
};
```
- Résultat de **D d(1, 2);**
  - A(3)
  - B(1)
  - C(2)

Les appels aux constructeurs **A()** sont ignorés depuis celui de **D()**!

Généralement la classe de base (**A**) n'offrira qu'un constructeur sans paramètre.

C++

70

## Transtypage dynamique


- Il n'est pas possible de fournir un objet d'une super-classe là où est attendu un objet d'une sous-classe.
- Pourtant, par polymorphisme, un pointeur ou une référence peut désigner un objet d'une sous-classe.
- Le **transtypage dynamique** permet de connaître le type de l'objet effectivement fourni.
  - Concept RTTI (*Run Time Type Identification*).
  - Effectuer un changement de type du résultat d'une expression rendant un *pointeur* ou une *référence* par l'opérateur:  
`dynamic_cast<type>(expression)`.
  - Le type rendu par l'*expression* doit être *polymorphique*:  
il doit disposer d'au moins une méthode virtuelle.
- Remarque: une utilisation fréquente du RTTI dénote souvent une mauvaise conception *objet*. Utiliser de préférence des méthodes virtuelles.

C++

71

## Transtypage dynamique (2)

- Pointeur:
  - `dynamic_cast<T*>(p)`
  - Si l'objet pointé par `p` est de type `T` ou a une sur-classe unique de type `T`, rend un pointeur de type `T*`, sinon renvoie `NULL` (0).
  - Soient `B` une sous-classe polymorphique de `A` et `A* pA; B* pB;`  

```
if ((pB = dynamic_cast<B*>(pA)) != NULL) {  
    // utiliser pB  
}
```
- Référence:
  - `dynamic_cast<T&>(r)`
  - Si l'objet référencé par `r` est de type `T` ou a une sur-classe unique de type `T`, rend une référence de type `T&`. Sinon une exception de type `bad_cast` (définie dans l'en-tête `typeinfo`) est levée.
  - ➔ `bloc try { /* dynamic_cast<...>(...) */ } catch (bad_cast) { }`  
Éventuellement spécifier le *namespace* par `std::bad_cast` 

C++

72

## Transtypage dynamique (3)

- L'opérateur `typeid(expression)` rend une référence sur un objet constant de classe `type_info` représentant le type de l'`expression`.
  - Pour un objet polymorphique (possédant au moins une méthode virtuelle) manipulé par une référence ou par un pointeur, l'objet `type_info` rendu représente la classe où est effectivement instancié l'objet.
  - Penser à déréférencer les pointeurs (`*ptr`) sinon l'information obtenue concerne le type du pointeur, non celui de l'objet pointé.
- L'opérateur `typeid(NomDeType)` rend une référence sur un objet constant de classe `type_info` représentant le type `NomDeType`.
- Méthodes principales de la classe `type_info`:
  - Nom du type: `const char *name() const;`
  - Egalité: `bool operator ==(const type_info &rhs) const;`
  - Inégalité: `bool operator !=(const type_info &rhs) const;`
- L'opérateur `typeid` et la classe `type_info` sont définies dans l'en-tête `<typeinfo>` de l'espace de nommage `std`.

C++

73

## Exemple

```
■ class Animal
{
public:
    virtual ~Animal() {}    // Au moins une méthode virtuelle
};

class Chat : public Animal { };
class Chien : public Animal { };

■ Chat c;
Animal& ref = c;

cout << typeid(ref).name() << endl;

cout << "Instance d'Animal: "
    << (typeid(ref) == typeid(Animal)) << endl;    // false

cout << "Instance de Chat: "
    << (typeid(ref) == typeid(Chat)) << endl;    // true

cout << "Membre d'Animal: "
    << (dynamic_cast<Animal*>(&ref) != NULL) << endl;    // true

cout << "Membre de Chat: "
    << (dynamic_cast<Chat*>(&ref) != NULL) << endl;    // true
```

C++

74

## Généricité

- C++ permet de définir des fonctions, des méthodes et des classes génériques au moyen de **templates** (modèles).
- Les templates sont paramétrés par un ou plusieurs identificateurs représentant des valeurs et des types:
  - Valeurs: représentées par leurs types,
  - Types: représentés par le mot clef **typename** (ou, **class**, qui est moins heureux - n'importe quel type étant acceptable)
  - **template<typename T, int I>** précède la déclaration d'une classe ou d'une fonction paramétrée par un type et un entier constant.
- La spécialisation d'un template (appelée *instanciation*) est effectuée à la compilation en fonction des paramètres (types et valeurs) fournis.
  - Contrairement à Java, il n'existe pas de code source propre au template, ce sont les différentes spécialisations qui sont effectivement compilées.

## Généricité: fonctions et méthodes

- Syntaxe: **template<param<sub>1</sub>, ... param<sub>i</sub>> type nom(arg<sub>1</sub>, ... arg<sub>j</sub>) { /\*...\*/ }**
  - Invocation: **nom<val<sub>1</sub>, ... val<sub>i</sub>>(eff<sub>1</sub>, ... eff<sub>j</sub>)**
    - Les valeurs des derniers paramètres (1..i), voire toutes, peuvent être omises si elles sont déduites des types des arguments effectifs (1..j).
  - Exemple:
    - ```
template<typename T> const T& max(const T& a, const T& b)
{
    return a > b ? a : b;
}
```
    - ```
int i = max<int>(42, rand());
int i = max(42, rand()); // par déduction automatique du type int
```
    - ```
Integer i(42);
Integer j = max(i, Integer(rand()));
```
- Remarque: l'opérateur > doit être défini pour le type **Integer**.

## Généricité: classes

- Syntaxe: `template<param1, ... parami> class nom déclaration`
- Spécialisation: `nom<val1, ... vali>`
  - Les valeurs des paramètres doivent être fournies explicitement.

- Exemple:

- ```
template<typename T> class Wrapper
{
    T _value;
public:
    Wrapper(T value) : _value(value) { }
    T get() { return _value; }
};
```
- ```
Wrapper<int> w1(42);
int i = w1.get();
```
- ```
Wrapper<double> w2(3.14);
double d = w2.get();
```

## Généricité: itérateurs

- En C++ les itérateurs sont obtenus sur une collection par les méthodes:
  - `Iterator begin()` // placé au début de la collection
  - `Iterator end()` // placé à la fin de la collection
  - Remarque: ces méthodes rendent un nouvel objet, cela peut être coûteux dans un `for (Iterator i = x.begin(); i != x.end(); ++i) /*...*/`
- Et définissent généralement les méthodes:
  - `Iterator& operator++()`
  - `bool operator==(const Iterator& o) const`
  - `bool operator!=(const Iterator& o) const`
  - `T& operator*() // déréférencement en l'élément courant`
- Par exemple, affichage des éléments d'une liste de strings `l`:

```
for (list<string>::iterator i = l.begin(); i != l.end(); ++i)
    cout << *i << endl;
```

## Généricité: exercice

- Définir une classe générique `Array`, permettant de définir un tableau d'un type donné et d'une taille donnée et une classe interne `Iterator` permettant de parcourir ses éléments.
- Méthodes de la classe `Array`:
  - `Array(const unsigned short size)`
  - `Array(const Array& o)`
  - `~Array()`
  - `Array& operator=(const Array& o)`
  - `T& operator[](const unsigned short index)`
  - `const short int size() const`
  - `Iterator begin()`
  - `Iterator end()`
- Tester cette classe avec un tableau de `strings` et un tableau contenant des personnes et des étudiants et en mettant en œuvre la liaison dynamique.

C++

79

## Liste d'initialiseurs

11

- Afin d'éviter de devoir insérer les valeurs d'une collection élément par élément, C++11 introduit le concept de liste d'initialiseurs par le template de classe `std::initializer_list`.
- Il permet d'utiliser la même syntaxe qu'en C pour l'initialisation des tableaux et des structures (`int[] array = {1, 2, 3};`).
- Ce type peut être utilisé non seulement dans des constructeurs mais également dans les méthodes (p.ex. l'opérateur `=`) ou les fonctions.
- Les collections de la STL utilisent les listes d'initialiseurs. Par exemple:
  - `list<int> l = {1, 2, 3, 4}; // constructeur`
  - `list<int> l;`  
`l = {4, 5, 6}; // opérateur =`
  - `map<int, string> m = { {1, "a"},`  
`{2, {'a', 'b', 'c'} },`  
`{3, string("foobar")} };`

C++

80



## Liste d'initialiseurs (2)

11

- Si **A** est un type, **a1, a2...** des valeurs de ce type et **T** est une classe définissant un constructeur **T(std::initializer\_list<A>)**, alors les initialisations suivantes sont légales:

- `T o({a1, a2, ...});`  
`T o{a1, a2, ...};`  
`T o = {a1, a2, ...};` // Variable nommée
- `T({a1, a2, ...});`  
`T{a1, a2, ...};` // Variable temporaire
- `new T({a1, a2, ...});`  
`new T{a1, a2, ...};` // Allocation dynamique
- `T fn() {`  
    `return {a1, a2, ...};` // Variable temporaire  
`}`
- `void fn(std::initializer_list<A> args) { /*...*/ }`  
    `fn({a1, a2, ...});`

## Liste d'initialiseurs (3)

11

- Méthodes du template de classe `std::initializer_list<T>`
  - `size_t size() const` // nombre d'éléments
  - `const T* begin() const` // pointeur sur le premier élément
  - `const T* end() const` // pointeur après le dernier élément
- L'arithmétique des pointeurs permet de passer à l'élément suivant ou précédent (`++` et `--`).
- Par exemple, pour pouvoir écrire `Array<int> array = {1, 2, 3};`, il faut définir le constructeur de la classe `Array` acceptant une liste d'initialiseurs:

```
Array(std::initializer_list<T> args)
: _size(args.size()), _data(new T[args.size()]) {
    int i = 0;
    for (const T* val = args.begin(); val != args.end(); ++val)
        _data[i++] = *val;
}
```

## Boucle for

11

- La boucle for a été étendue en C++11 pour permettre d'itérer facilement sur un ensemble d'éléments.
- Elle s'applique sur les tableaux, les listes d'initialiseurs et tout type définissant les fonctions `begin()` et `end()` rendant des itérateurs (dont le type surcharge les opérateurs `*`, `++` et `!=`),

```
• int array[] = {1, 2, 3, 4, 5};  
  for (int& x : array) x *= 2;  
  
• for (Person p : {Person("John"), Person("Paul")})  
  // initializer_list<Person>  
  cout << p.name() << endl;  
  
• Array<Person*> array = {  
  new Person("Paul"), new Student("John", "IL") };  
  for (Person* ptr : array) // Array<Person*>::iterator  
    ptr->display();        // liaison dynamique
```

## Boucle for\_each

- La fonction générique `for_each`, définie dans l'espace de nommage `std`, permet de parcourir un ensemble d'éléments et d'invoquer sur chacun d'entre eux une fonction donnée.
- Elle prend en paramètre deux itérateurs (dont le type surcharge les opérateurs `*`, `++` et `!=`) indiquant les positions initiales et finales de la collection et une fonction unaire acceptant un élément de la collection. Elle rend la fonction elle-même (pour l'utiliser dans une éventuelle affectation).

- Exemple:

```
void show(int i) { std::cout << i << std::endl; }  
Array<int> array = { 1, 2, 3, 4, 5 };  
std::for_each(array.begin(), array.end(), show);  
// affiche 1 2 3 4 5
```

- Exercice: proposer une implémentation pour la boucle `for_each`.

## Foncteurs

- Un foncteur (*functor* ou *function object*) est un objet qui se comporte comme une fonction.
- Sa classe définit une surcharge de l'opérateur fonctionnel `()`.
  - Un objet foncteur peut ainsi être utilisé à la place d'une fonction.
- Il possède donc un **état** (la valeur de ses attributs).
  - Ceci rend un foncteur plus puissant qu'une fonction en permettant de conserver des données entre les différentes invocations de la *fonction*.
- Les foncteurs sont utilisés dans les nombreux algorithmes de la STL et également pour l'écriture de fonctions *callback*.
- L'en-tête `<functional>` définit de nombreux templates foncteurs.

## Foncteurs (2)

- ```
class Sum
{
    int sum = 0; // C++11
public:
    void operator()(int n) {
        sum += n;
    }
    int value() const {
        return sum;
    }
};
```
- ```
Array<int> array = { 1, 2, 3, 4, 5};
Sum s = for_each(array.begin(), array.end(), Sum());
// Sum() crée un nouvel objet temporaire, passé au for_each
// qui le rend et est copié (constructeur de copie)
cout << s.value() << endl;
```

## Problématique des pointeurs

- Lors d'allocation dynamique de mémoire il faut la désallouer lorsqu'elle n'est plus utilisée → Risques de « fuite de mémoire » (*memory leaks*) et d'erreurs.

```
int* p1 = new int(42), p2 = p1;  
someFunction(p1);  
delete p1;
```

- Si `someFunction` lève une exception (non traitée localement), `delete p1` ne sera pas invoqué et la mémoire ne sera pas désallouée.
- Si `someFunction` désalloue le pointeur, `delete p1` engendrera une erreur (désallocation d'une zone non allouée).

```
*p2 = 24;
```

- Erreur: déréférencement d'un pointeur invalide (idem si le `new` échoue).

- Sémantique peu explicite des pointeurs:

```
int* someFunction(...);
```

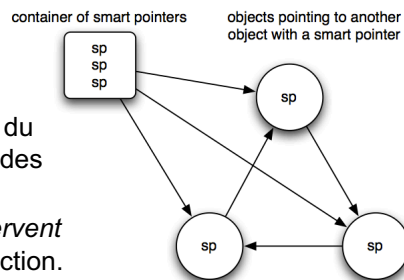
- Faudra-t-il désallouer le pointeur rendu par cette fonction ?

## Pointeurs *intelligents*

- Les pointeurs intelligents (*smart pointers*) se comportent comme des pointeurs bruts (*raw*) mais qui *gèrent* aussi les entités créées, sans que l'utilisateur n'ait à se soucier de quand et comment il est faut les supprimer.
- Ils sont définis de manière à ce qu'ils puissent être utilisés syntaxiquement pratiquement comme des pointeurs bruts.
- Un pointeur intelligent contient un pointeur brut et est défini par une classe *template* dont le type est celui de la donnée référencée.
- Un pointeur intelligent *possède* la donnée allouée dynamiquement et en gère le cycle de vie (copie, destruction, ...). Il est l'unique responsable de la désallocation de la donnée allouée.
- C++98 a proposé une première version de pointeur intelligent, `auto_ptr`, qui n'est pas sans problèmes. Sous l'impulsion du groupe Boost ([boost.org](http://boost.org)) de nouvelles versions ont été définies: `shared_ptr`, `unique_ptr` et `weak_ptr` (définis dans l'ent-tête `<memory>`).

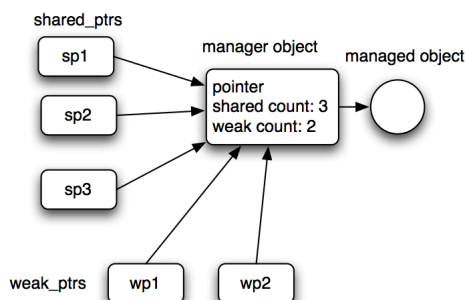
## Pointeur partagé

- Un pointeur partagé (`shared_ptr`) est un pointeur intelligent permettant le partage d'une donnée entre différents pointeurs intelligents.
  - N'importe quel pointeur partagé référençant une donnée garantit qu'elle sera conservée.
  - Cette donnée est supprimée lorsque le dernier pointeur partagé cesse de la référencer.
- Limitation:
  - La formation d'un cycle empêche la destruction des données allouées.
  - P.ex., supprimer les pointeurs partagés du conteneur n'entraîne pas la destruction des trois objets qui se référencent en cycle.
  - Solution: utiliser des `weak_ptr` qui *observent* les données sans empêcher leur destruction.



## Pointeur partagé (2)

- Un pointeur partagé utilise un compteur de références:
  - Celui-ci tient à jour le nombre de pointeurs intelligents référençant la donnée allouée dynamiquement.
  - Quand ce compteur tombe à 0 (p.ex. lors de la destruction du dernier pointeur intelligent) la donnée est automatiquement détruite.
- Implémentation (simplifiée)
  - Lorsque la donnée est allouée et le premier pointeur partagé est créé, un objet *gestionnaire* est alloué, référençant cette donnée.
  - Les nouveaux pointeurs partagés doivent être créés à partir de ceux existants et référencent ainsi le même gestionnaire.



## Pointeurs *partagés*: exercice

- Implémenter une classe `SharedPointer` définissant un pointeur partagé simplifié (sans la gestion de pointeurs faibles, `weak_ptr`). Méthodes:
  - Constructeur à partir d'un pointeur brut,
  - Constructeur de copie,
  - Destructeur,
  - Surcharge de l'opérateur `=`,
  - Surcharge de l'opérateur `*` (déréférencement de la donnée),
  - Surcharge de l'opérateur `->` (accès à une propriété de la donnée),
  - Surcharges des opérateurs `==` et `!=` (comparaison des pointeurs bruts),
  - `get()` rendant le pointeur brut (utilisation déconseillée),
  - `int useCount()` rendant le nombre d'objets partageant la donnée.
- Pour plus de sécurité, interdire la construction implicite d'objets temporaires à partir d'un pointeur brut (p.ex.dans `sp = new int(4);`).

## Pointeurs *partagés*: exercice (2)

- Pour bien comprendre le fonctionnement (et déboguer...) afficher pour chaque opération les éventuelles allocations, désallocations ainsi que l'état de l'objet gérant le pointeur brut.
- Exemple de programme de test:

```
SharedPointer<int> sp1(new int(1));
SharedPointer<int> sp2(sp1);
*sp2 = 2;
cout << *sp1 << endl;
SharedPointer<int> sp3(new int(3));
sp2 = sp3;
sp1 = SharedPointer<int>(new int(4));
```
- Le compléter en utilisant une hiérarchie de classes `Person/Student` mettant en œuvre la liaison dynamique.

## Opérations principales de `shared_ptr`

- Constructeur à partir d'un pointeur brut,
- Constructeur de copie (explicite),
- Destructeur,
- Surcharge de l'opérateur `=` : affectation (explicite) entre `shared_ptr`
- Surcharge de l'opérateur `*` : déréférencement de la donnée,
- Surcharge de l'opérateur `->` : accès à une propriété de la donnée,
- `T get()` : rendant le pointeur brut (utilisation déconseillée),
- `int useCount()` : rendant le nombre d'objets partageant la donnée,
- `bool unique`, rend vrai si aucun autre pointeur ne partage la donnée,
- `reset()` et `reset(p)` : réinitialisation (à vide ou sur un autre pointeur),
- Surcharge de l'opérateur de transtypage en `bool`: `true` si une donnée existe,
- Surcharge des opérateurs `==` et `!=` : test de l'égalité de contenu de pointeurs,
- Fonction `swap(p, q)` : échange du contenu de deux pointeurs.

## Pointeurs *partagés*: `make_shared`

- Quand un pointeur partagé est créé, deux allocations sont effectuées: une pour la donnée et l'autre pour l'objet gestionnaire.
- Pour palier à la lenteur induite par ces allocations, C++11 offre la fonction générique `make_shared` qui effectue une seule allocation de taille suffisante pour contenir les deux objets.
  - `shared_ptr<Person> sp1(new Student(...)); // deux allocations`
  - `shared_ptr<Person> sp2(make_shared<Student>(...)); // une seule allocation`
- Fonctionne également avec des types primitifs:
  - `shared_ptr<int> sp3 = make_shared<int>(42);`
- L'utilisation systématique de `make_shared` est recommandée car elle évite d'exposer des pointeurs bruts dans le code (pas de `new` explicite, pas de risque d'initialiser différents pointeurs partagés avec le même pointeur brut).

## Pointeurs faibles

- Un pointeur faible (`weak_ptr`) est un pointeur intelligent qui référence une donnée sans empêcher sa suppression.
- Implémentation
  - L'objet gestionnaire possède deux compteurs indiquant le nombre de pointeurs intelligents référençant la donnée: un pour les pointeurs partagés et un pour les pointeurs faibles.
  - La donnée et son gestionnaire ne sont supprimés que lorsque ces deux compteurs valent 0.
  - Par contre, si seul le compteur de pointeurs partagés vaut 0, seule la donnée est supprimée, le gestionnaire existe toujours.
  - ➔ Un pointeur faible n'a pas l'assurance de référencer une donnée.
- La donnée (éventuellement) référencée par un pointeur faible ne s'obtient pas directement, il est nécessaire de passer d'abord par un pointeur partagé (constructeur ou méthode `lock`).

## Pointeurs faibles: exemples

- Construction

```
shared_ptr<Person> sp = make_shared<Person>("John");
weak_ptr<Person> wp1(sp);           // wp1 pointe sur John
weak_ptr<Person> wp2;                // wp2 est vide
wp2 = sp;                           // wp2 pointe sur John
weak_ptr<Person> wp3(wp1)            // wp3 pointe sur John
weak_ptr<Person> wp4;                // wp4 est vide
wp4 = wp1;                          // wp4 pointe sur John
wp4.reset();                        // wp4 est vide
```
- Obtention d'un `shared_ptr` depuis un `weak_ptr`:
  - `shared_ptr<Person> sp = wp.lock();`  
if (sp) // opérateur bool  
cout << sp->name() << endl;
  - `shared_ptr<Person> sp(wp);` // lève `std::bad_weak_ptr` si vide
- Tester si un `weak_ptr` est vide par `wp.expired()`.

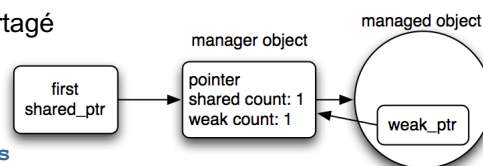


## Pointeur partagé sur `this`

- Avec des pointeurs bruts dans le corps d'une méthode d'une classe il est possible de passer l'objet courant en paramètre à une fonction ou une méthode (`o.m(*this)`), ou de le rendre comme résultat (`return *this`).
- En utilisant des pointeurs intelligents, créer un pointeur partagé sur `this` (`shared_ptr<Type>(this)`) est dangereux:
  - Le pointeur partagé va désallouer l'objet courant alors que celui-ci n'a pas forcément été alloué dynamiquement.
  - Si cet objet est déjà géré par un pointeur partagé, il sera libéré deux fois.
- Solution: faire hériter la classe devant obtenir un pointeur sur l'objet courant de la classe générique `enable_shared_from_this<T>`, définie dans l'en-tête `std::enable_shared_from_this`.
  - Cette classe offre la méthode `shared_from_this()` rendant un pointeur partagé sur l'objet courant.

## Pointeur partagé sur `this` (2)

- La classe `enable_shared_from_this<T>` possède un attribut de type pointeur faible (`weak_ptr`) permettant de référencer l'objet courant et n'empêchant donc pas sa suppression.
- A la création du premier pointeur partagé sur un objet, le constructeur de `shared_ptr` détecte (par *magie template*) que la classe de l'objet hérite de `enable_shared_from_this` et initialise le pointeur faible à partir du pointeur partagé (en réutilisant le gestionnaire existant).
- Attention, l'objet doit obligatoirement être créé dans un pointeur partagé pour que cette initialisation soit effectuée.
- La méthode `shared_from_this()` rend un pointeur partagé construit à partir du pointeur faible stocké dans l'objet.



## Pointeur partagé sur this (3)

- Dans le code d'un constructeur l'attribut pointeur faible n'est pas encore associé au gestionnaire. Il n'est donc pas encore possible d'utiliser la méthode `shared_from_this()`.
- Pour obtenir un pointeur partagé sur l'objet lors de sa création il est possible d'encapsuler le processus de création dans un MCR de type fabrique.

```
void m(shared_ptr<A> sp) { /* ... */ }  
class A  
{  
    A(...) { /* ... */ } // constructeur privé  
public:  
    static shared_ptr<A> create(...) {  
        shared_ptr<A> sp = make_shared<A>(...);  
        m(sp); // post-traitement  
        return sp;  
    }  
}
```

## Exercice

- Définir la classe `Musician` et la classe `Band` afin que le code,

```
shared_ptr<Musician>  
john = make_shared<Musician>("John");  
paul = make_shared<Musician>("Paul"),  
george = make_shared<Musician>("George"),  
ringo = make_shared<Musician>("Ringo");  
  
shared_ptr<Band> beatles = make_shared<Band>("The Beatles");  
beatles->setMembers({ john, paul, george, ringo});  
cout << beatles->toString() << endl;  
cout << john->toString() << endl;  
  
shared_ptr<Band> wings = make_shared<Band>("Wings");  
wings->setMembers({paul});  
beatles.reset();  
cout << paul->toString() << endl;  
wings->setMembers({paul});  
cout << paul->toString() << endl;
```

## Exercice (2)

- Produise le résultat...

```
The Beatles: John Paul George Ringo
John, band: The Beatles
Paul is already in The Beatles
~Band(): The Beatles
Paul, band: <none>
Paul, band: Wings
~Band(): Wings
~Musician(): Ringo
~Musician(): George
~Musician(): Paul
~Musician(): John
```

## Anneau Pointeur *unique*

- Un pointeur unique (`unique_ptr`) est un pointeur intelligent qui est le seul à gérer une donnée et qui la supprime lorsqu'il n'est plus utilisé.
- Contrairement aux pointeurs partagés et faibles, il n'utilise pas de gestionnaire (il possède directement un pointeur sur la donnée). Le coût d'un pointeur unique est donc plus faible.
- Son destructeur supprime directement la donnée si elle existe (rappel, l'opérateur `delete` ne fait rien sur un pointeur `nullptr`).
- L'unicité de `unique_ptr` est garantie par la suppression (`= delete`) du constructeur de copie et de l'opérateur d'affectation.
  - ➔ Les pointeurs uniques sont uniquement passés par référence.
  - La donnée possédée par un pointeur unique peut être transférée à un autre pointeur unique, implicitement (variable locale `unique_ptr` rendue par une fonction) ou explicitement en utilisant `std::move`.  
P.ex.: `p2 = std::move(p1);` // `p2` possède la donnée, plus `p1`