Chapitre II

Encapsulation

Sommaire

Introduction

Encapsuler = placer dans des boîtes nommées.

Pourquoi encapsuler ?

- mettre ensemble les données qui représentent un objet abstrait afin d'être en mesure de les manipuler ensemble.
- intégrer à l'objet abstrait les fonctions qui vont permettre de le manipuler.
- verrouiller l'accès aux données interne d'un objet abstrait afin de n'autoriser la modification de l'objet qu'aux fonctions associées à l'objet.
  permet d'être assuré que l'objet sera toujours modifié de manière cohérente.
- mettre dans un seul container l'ensemble des types, classes et fonctions représentant une ou plusieurs fonctionnalités dans un module isolé du reste du code.
  évite la duplication et/ou les conflits de noms

1 `struct` du C

En C, l'encapsulation s'effectue grâce à `struct`.

On appelle un TDA (Type de Données Abstrait) un ensemble de données, qui, prises ensemble, représentent un objet complexe.
Exemple : matrice = nombre de lignes et de colonnes + éléments du vecteur.

Une structure est le moyen le plus simple de créer des données complexes en C/C++ :

- une structure est une boîte nommée contenant des données.
- chaque donnée (ou champ) de la boîte est typée et nommée (de façon unique).

Ainsi,

- une structure permet de définir un type.
  ce type peut être utilisé pour définir une variable.
- la variable représente une instance de cette structure.
  cette variable contient assez de place pour stocker tous les champs de la structure.
- le nom d'un champs permet d'accéder à ce champs particulier

Exemple :

Soit `Point2D` est une structure définissant les coordonnées `(x,y)` d'un point dans $\mathbb{R}^2$, sous forme d'un couple de flottant.

On voudrait pouvoir écrire quelque chose comme :

```
Point2D   P0 = { 1.f, -2.f };
float     norm = sqrtf( P0.x * P0.x + P0.y * P0.y );
```

`P0` est une instance (= une réalisation) d'un `Point2D`.
Les noms des champs permettent d'utiliser les composants stockés dans la structure.

Comment définir une telle structure ?

1.1 Définition

La déclaration du contenu d'une structure se fait sur la base du modèle suivant :

```
struct {
    Type1    Nom1;
    Type2    Nom2;
    ...
    TypeN    NomN;
}
```

Notes :

- Le nom d'un champs doit identifier le champ de manière unique dans la structure (*i.e.* pas d'autres champs avec le même nom).
- `TypeI` peut être n'importe quel type (simple, pointeur, tableau, autres structures, ...).
- Les tableaux statiques sont stockés **intégralement** dans la structure.
- les modificateurs `short`, `long`, `unsigned` sont autorisés. Tous les autres modificateurs sont interdits, ou leur sens sera explicité plus tard.
- Si plusieurs types consécutifs sont identiques, la notation "`Type Nom1, ... NomP;`" est équivalente à "`Type Nom1; ... Type NomP;`".

Cette définition pourrait être utilisé comme type, mais elle obligerait à la redonner en entier à chaque fois que l'on souhaiterai l'utiliser.

La déclaration d'un nouveau type (nommé par exemple ***NomStruct***) associé à une structure s'effectue

de la manière suivante :

En C :
```
typedef struct {
    /* contenu structure */
} NomStruct;
```

En C++ :
```
struct NomStruct {
    // contenu structure
};
```

L'utilisation de ce nouveau type se fait ensuite naturellement.

Exemple :

```
// déclaration en C
typedef struct { double x,y; } Point2D;
typedef struct { Point2D  a,b;  double  len; } Segment;
// (ou exclusif) déclaration en C++
struct Point2D { double x,y; };
struct Segment { Point2D  a,b;  double  len; };
// utilisation (pour les deux)
Point2D P;
Segment S;
```

Remarque : la syntaxe du C est évidemment autorisée en C++.
Important : Ne jamais oublier le ; à la fin de définition d'une structure (ou d'une classe) en C++.

1.2 Mémoire et alignement

a) Règles d'ordonnancement

- une structure est une boîte dont la taille est suffisante pour contenir l'ensemble des données qui la compose.
- les champs de la structure sont stockés dans la structure dans l'ordre **exact** de leurs déclarations.
- lors de la compilation, l'accès à un champs est remplacé par un décalage par rapport à l'adresse du début de la structure.

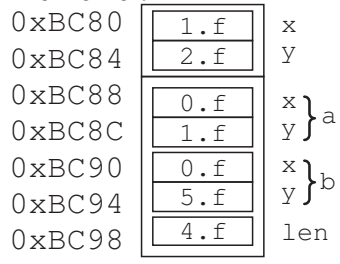
Exemple :

```
typedef struct { float x,y; } Point2D;
typedef struct { Point2D  a,b;  float  len; } Segment;
// utilisation (pour les deux)
Point2D P = {1.f,2.f};
Segment S = { {0.f,1.f}, {0.f,5.f}, 4.f };
```

Table des symboles :

nom	type	adresse
P	Point2D	0xBC80
S	Segment	0xBC88

Mémoire :



Rappel : Les variables de type élémentaire `T` sont alignés sur des adresses multiples de `sizeof(T)`. Il y a 4 types d'alignements possibles :

1 `char`
2 `short int`
4 `float`, `int`, pointeur 32bit
8 `double`, pointeur 64bit

b) Règles d'alignement

- chaque champs de la structure est aligné.
- chaque structure est alignée sur une adresse multiple de la taille de son champ élémentaire le plus grand.
  y compris pour une structure dans une structure.
- le nombre d'octets (blancs) nécessaire est ajouté entre les champs ou pour compléter la taille de la structure afin que ces règles soient respectées.

Conséquences :

- `sizeof` d'une structure n'est pas nécessairement égale à la somme des `sizeof`s de ses composants.
- l'ordre des champs peut changer de façon dramatique la taille de la structure et générer un nombre de blancs conséquent si on ne prend pas garde à les organiser correctement.
- pour certaines structures, les règles d'alignement peuvent faire qu'il soit possible d'ajouter des champs sans augmenter la taille de la structure.

Exemple 1 :

```
struct A { char a; double b; };
```

`A` aligné sur un multiple de 8. La structure contient un `double`, elle est alignée sur un multiple de 8.
Le `double` qui suit le `char` est aligné sur un multiple de 8.

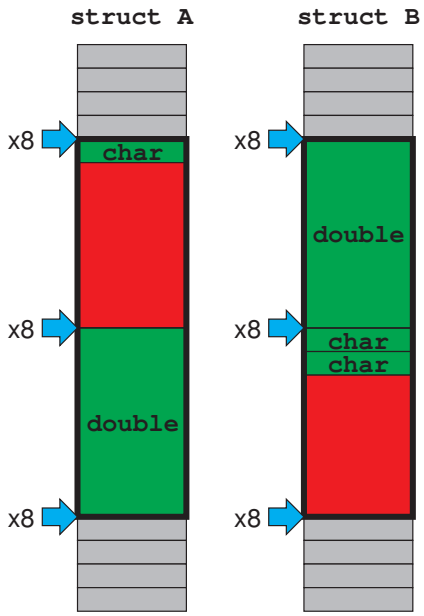
Donc `sizeof(A)` = 16
Données = 9 octets, perdus = 7 octets

Exemple 2 :

```
struct B { double b; char a,b; };
```

La structure contient un `double`, elle est alignée sur un multiple de 8.
La taille de `B` devant être aligné sur un multiple de 8, la structure se termine au multiple de 8 suivant le dernier `char`.

`sizeof(B)` = 16
Données = 16 octets, perdus = 6 octets

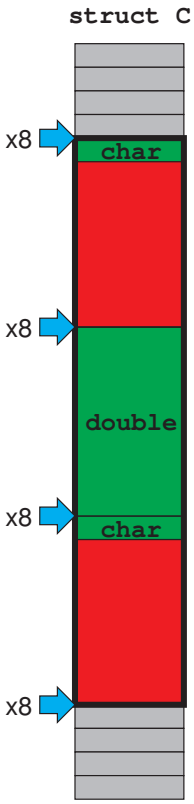


Exemple 3 :

```
struct C { char a; double b; char c; };
```

La structure contient un `double`, elle est alignée sur un multiple de 8.
Le `double` qui suit le `char` est aligné sur un multiple de 8.
La taille de `C` devant être aligné sur un multiple de 8, la structure se termine au multiple de 8 suivant le dernier `char`.

`sizeof(C)` = 24
Données = 10 octets, perdus = 14 octets



En conséquence,

- Ne pas arranger les champs dans une structure pour "faire joli".
- La quantité de mémoire perdue peut être **vraiment considérable**, d'autant plus si l'on fait des tableaux avec de telles structures.

En résumé :

- l'ordre des champs doit être soigneusement choisi afin d'éviter les trous.
- si cela n'est pas possible, ajouter des champs pour occuper les trous est gratuit (*i.e.* ne change pas la taille de la structure).

Attention : Le `sizeof` des pointeurs changeant entre une compilation 32bit et 64bit rendent ce problème non trivial.

Solution minimisant l'espace perdu :

- mettre les champs décroissant du multiple d'alignement (à savoir champs s'alignant sur un multiple de 8, puis de 4, puis de 2, puis les autres).
- pour les pointeurs, les placer après les champs s'alignant sur un multiple de 8, et avant les champs s'alignant sur un multiple de 4.
- La somme de la taille des champs est un multiple de la taille de la structure.

Remarque : Les deux premières règles ci-dessus permettent de faire en sorte qu'il n'y aura pas de trou **dans** la structure. La dernière qu'il n'y en a pas à la fin : les champs ajouté pour faire en sorte de respecter cette règle sont gratuits.

c) Pourquoi ces règles d'alignement ?

- L'alignement des types élémentaires (8bit, 16bit, 32bit, 64bit) est effectué par souci d'efficacité pour l'accès aux données
  une transaction mémoire de 256 bits = toujours 32x8bit, 16x16bit, 8x32bit, 4x64bit.
  s'il n'y avait pas d'alignement mémoire, le premier et le dernier paquet lus pourraient ne pas être complets.

- Il doit être possible de créer des tableaux de structures
  rappel : les accès au $i^{\text{ème}}$ aux éléments d'un tableau se font par décalage dans la mémoire de i fois la taille de la structure.
  - ◇ les types élémentaires dans la structure doivent eux aussi être aligné dans la mémoire.
  - ◇ le stockage est dans l'ordre des champs
    d'où les trous à l'intérieur de la structure,
  - ◇ si l'alignement de la structure n'est pas celle du plus grand type élémentaire, impossible de faire en sorte qu'en se décalant d'un multiple de la taille de la structure, ses champs élémentaires soient toujours alignés.

1.3 Opérateurs sur une structure

Soit une structure `S` :

```
struct S { ... TYPEi  NOMi; ... };
```

a) Accès aux éléments d'une structure

- pour une variable `x` de type `S`
  `x.NOMi` représente le champs `NOMi` et est de type `TYPEi`.
  Elle peut être utilisée en lecture (RHS) ou en écriture (LHS).
- pour un pointeur `x` de type `S*`
  `x->NOMi` représente le champs `NOMi` et est de type `TYPEi`.
  Elle peut être utilisée en lecture (RHS) ou en écriture (LHS).

Exemple :

```
struct Point2D { float x,y; };
// instance d'une structure
Point2D P;
P.x = 1.f;
P.y = 2.f * P.x;
// pointeur sur une structure
Point2D *Q = &P;
Q->x = 1.f;
Q->y = Q->x * Q->x;
```

b) Copie d'une structure

On utilise l'opérateur = entre deux variables de même type structure.
L'opérateur = revient à copier tous les champs (=`memcpy`).

Note : en C++, assuré par l'assignation par copie, ou idem si non définie.

Exemple :

```
struct Point2D { float x,y; };

// copie entre instances d'une structure
Point2D a = {1.f, 2.f}, b;
b = a;
// équivalent memcpy(&b,&a,sizeof(Point2D))
// ici a et b sont identiques

// pointeur sur une structure
Point2D *A = &a, *B = &b;
*B = *A;
// équivalent à memcpy(B,A,sizeof(Point2D))
// ici A et B sont identiques
```

Attention : on remarquera que dans le code ci-dessus, écrire `B=A` copie le pointeur et non les données pointées (*i.e.* `A` et `B` pointent tout deux vers `a`).

1.4 Initialisation

On donne les exemples en reprenant les structures déjà données :

```
struct Point2D { float x,y; };
struct Segment { Point2D  a,b;  float len; };
```

Plusieurs méthodes pour initialiser la structure :

- à la création de la variable, donner les champs dans l'ordre

```
Point2D P={ 1.f, 2.f };
Segment Q={ {1.f,2.f}, {3.f,4.f}, 0.f };
```

  Note : seulement valable à la création de la variable.
- initialisation à partir d'un objet par copie (opérateur =)

```
Point2D R=P;
Segment S={ P, R, 0.f };
```

1.5 Passage en paramètre et retour

a) Passage d'une structure par valeur

Lorsqu'une structure est passée par valeur à une fonction, alors, la structure est transmise par copie (=), i.e. la structure est copiée champs par champs (en C++, utilise le constructeur par copie s'il est défini).

Exemple :

```
float DistPoints(Point2D P1, Point2D P2) {
    float dx=P1.x-P2.x, dy=P1.y-P2.y;
    return sqrtf(dx*dx  + dy*dy);
}
```

Conséquence : potentiellement inefficace (autant de variable à copier que de champs dans la struc-

ture) et inutile (si tous les champs ne sont pas nécessaires).

Sauf lorsque ce comportement est recherché, une structure est (sinon) **toujours passée par référence ou par pointeur**, en utilisant le mot-clé `const` si besoin.

b) Retour d'une structure par valeur

Lorsqu'une structure est retournée par une fonction, alors :

- soit la structure est transmise par copie (=), i.e. la structure est copiée champs par champs (en C++, utilise le constructeur par copie s'il est défini).
- soit si l'écriture s'y prête, il y a une élision de copie (voir en annexe de ce chapitre).

Exemple :

```
Point2D InitPoints(float x, float y) {
  Point2D P={x,y};
  return P;
}
```

```
Point2D P1;
// copie
P1 = InitPoints(0.f,0.f);

// élision de copie (P=P2 dans la fonction).
Point2D P2 = InitPoints(1.f,0.f);
```

2 Encapsulation en C++

Dans un premier temps, nous ne décrivons que les TDAs C++ **élémentaires**, à savoir sans y inclure les concepts d'héritage, de virtualité et de polymorphisme.

En C++, le principe des TDAs est repris de la manière suivantes :

- les données sont stockées comme un struct du C.
- la définition de fonctions s'appliquant spécifiquement à une instance du TDA peuvent être encapsulée dans le corps du TDA.
  De telles fonctions sont appelées des **méthodes**.
- de restreindre les accès aux champs de la TDA,
- des méthodes spécifiques permettent d'automatiser le traitement des instances d'objet :
  - ◇ un constructeur est une méthode appelée à la création de l'instance d'un objet,
  - ◇ le destructeur est la méthode appelée à la destruction de l'instance d'un objet.
- les surcharges d'opérateur permettent de définir comme appliquer les opérateurs usuels du langages (+, *, =, ...) à des objets.

Par la suite, on appellera classe un TDA en C++, que ce TDA soit défini par le mot-clef `struct` ou `class`.

a) Notion de déplacement

Le C_{11}^{++} introduit la possibilité de différencier entre :

- un objet en cours de vie

- un objet en fin de vie (par exemple un objet temporaire)

à travers l'utilisation du passage par référence [1] ou une qualification de référence [2].

Quel est l'intérêt de savoir qu'un objet est en fin de vie ?

- en fin de vie = ses ressources privées sont sur le point d'être détruites.
- il est donc possible de "voler" ces ressources afin de les réutiliser, ce qui est toujours plus efficace de les copier, puis de les détruire.

Par ressource privée d'un objet, on entend : une zone de mémoire (donc pointée par un pointeur) possédée par un objet et qui peut être réaffectée à un autre objet, et non la mémoire utilisée par l'objet lui-même (= celle de sa structure), qui est déjà affectée et ne peut pas être déplacée.

Ce concept est précisé en annexe, à la fin de ce chapitre.

2.1 Méthodes

Une méthode permet d'appliquer une fonction à l'instance d'un objet sans le passer en paramètre d'une fonction.

L'écriture `X.Y(params)` signifie : "appeler la méthode `Y` avec les paramètres `params` sur l'objet `X`".

Exemple : Si `V` est une instance d'un objet de type `Vecteur`, alors on peut définir les méthodes suivantes :

- la méthode `length` pour calculer la longueur d'un vecteur est appelée avec `V.length()`
- la méthode `add` qui place dans `V` la somme des vecteurs `V1` et `V2` est appelée avec `V.add(V1,V2)`.

Dans ce cadre,

- une méthode est déclarée encapsulée dans l'objet,
  on l'appelle comme si on accédait à l'un de ses champs,
- néanmoins, la déclaration d'une méthode ne modifie la taille d'un objet (*i.e.* son `sizeof`) [3].
  Pour le compilateur, la fonction à appeler est déterminée statiquement à la compilation [4].

2.2 Déclaration

Les méthodes peuvent être définies de deux façons différentes.

a) Définition dans la structure

La méthode est incluse directement dans la définition de la classe (i.e. avec son code).
Ce type de déclaration est réservé aux fonctions `inline` de la classe.

1. **Rappel :** & pour une référence à une `lvalue`, && pour une référence à une `rvalue`
2. voir plus loin.
3. Attention, ce n'est pas le cas pour une classe donc au moins l'une des méthodes est virtuelle.
4. Même remarque si la méthode appelée est virtuelle.

Exemple:

```
struct Vector {
    ...
    // calcule la norme du vecteur
    inline float Norm(void) { /* code méthode */ };
};
```

NOTE :

- le mot-clef `inline` n'est pas nécessaire car toute définition de méthode dans celle de la classe est considérée comme inline.
- les méthodes `inline` ne génèrent le plus souvent pas d'appels de fonction lors d'une compilation en mode optimisée.

b) Déclaration dans l'entête, définition à l'extérieur

- le prototype de la méthode est déclaré dans la structure (typiquement dans le fichier d'en-tête `.h` associé à la classe)
- le définition de la méthode est dans le code (typiquement dans le `.cpp` associé à la classe)

```
// file: vector.h
struct Vector {
    ...
    // norme du vecteur
    float Norm(void);
};
```

```
// file: vector.cpp
#include "Vector.h"
// def. Norm() de Vector
float Vector::Norm(void) {
    ...
}
```

L'opérateur `::` s'appelle l'**opérateur de résolution de portée**.

- `A::B` permet d'indiquer que `B` est défini dans un container nommé `A`.
- Dans cette section, le container sera toujours une classe. Cette syntaxe est aussi utilisée pour les `namespace`s (voir cette section).

c) Définition d'une méthode

Dans une méthode, les champs de l'instance de la classe sont des variables locales qui ont le même nom que le champs.

```
struct Complex {
    float  r,i;
    float Norm(void);
};
```

```
#include "Complex.h"
float Complex::Norm(void) {
    return sqrtf(r*r+i*i);
}
```

Autrement dit, dans une méthode, le nom d'un champs contient la valeur du champs associé à l'instance de l'objet sur lequel il est appelé (= référence implicite à l'objet sur lequel la méthode appelée).

ATTENTION : ne pas donner à un paramètre de la méthode ou à une de ses variables locales le même nom que l'un des champs (shadowing possible sans Warnings).

```
struct Dummy {
    int a,b;
    int Shadow(int a);
};
```

```
int Dummy::Shadow(int a) {
    int b = 4;
    return a+b;
} // no warnings
```

d) Pointeur `this`

Un pointeur nommé `this` est également défini dans toute méthode. Ce pointeur représente l'adresse de l'instance sur laquelle la méthode est appelée.

Quel est l'intérêt de `this` ?

- permet d'identifier l'instance courante, avec pour application :
  - ◇ retourner sa propre adresse ou soi-même (retour d'une référence)
  - ◇ vérifier si l'instance courante est identique à une autre instance passée en paramètre.

```
void Dummy::swap(Dummy &c) {
    if (this == &c) return; // nothing to do
    else { /* real swap */ }
}
```

- permet de lever l'ambiguïté sur le fait qu'une variable locale est en fait un champ de la classe (convention de programmation, ou en cas de shadowing),

```
int Dummy::Shadow(int a) {
    int b = 4;
    return this->a + this->b;
} // a and b from instance
```

- peu constituer une règle de codage (à savoir un accès à tout membre ou méthode de la classe doit obligatoirement s'effectuer à travers `this`) afin de souligner qu'il s'agit de références à l'instance de l'objet courant.

3 Méthodes automatiques

On appelle méthode automatique une méthode appelée automatiquement (à savoir sans appel **explicite** par l'utilisateur) au cours de la vie d'un objet.

Le code suivant fait appel à ces fonctions automatiques :

```
struct A { ... };
A fun() { ... }
int main() {
  A   a1;        // appel au constructeur par défaut
  A   a2=a1;     // appel au constructeur par copie
  A   a3=fun();  // appel au constructeur par déplacement (*)
  a1 = a2;       // appel à l'assignation par copie
  a2 = fun();    // appel à l'assignation par déplacement
}                  // appel au destructeur sur a1, a2 et a3
```

Si ces méthodes ne sont pas définies par l'utilisateur, le compilateur en génère automatiquement une version pour l'utilisateur [5].

Remarque : dans le code ci-dessus, ce n'est pas le constructeur par déplacement qui est appelé, mais le constructeur par copie avec l'élision de copie (voir pourquoi en annexe de cette leçon).

5. Attention, ce n'est pas toujours le cas, voir les exceptions plus loin et dans la leçons sur l'héritage.

3.1 Constructeurs et assignation

a) Principe

Les constructeurs sont des méthodes qui sont appelées **automatiquement** à la création de **toute instance** d'une structure.

A savoir, après l'allocation de la mémoire nécessaire pour stocker la structure.

- Trois constructeurs sont définis par défaut[6] : les constructeurs par défaut, par copie, et par déplacement (pour ce dernier, à partir du C_{11}^{++}).
- Pour une structure nommé `A`, le constructeur par défaut est `A()`. Les autres constructeurs sont des surcharges (i.e. `A(...)`).
- Il est possible de définir autant de constructeurs que l'on souhaite (qui doivent cependant différer par leurs paramètres).
  Dans tous les cas, un constructeur ne renvoie pas de valeur.
- La construction d'une instance de `A` avec :
  - ◇ A a ; fait appel au constructeur par défaut `A()`.
  - ◇ A a(4) ; fait appel au constructeur `A(int)`.
  - ◇ A a(4,2.f) ; fait appel au constructeur `A(int,float)`.
  - ◇ ...

  Si le constructeur appelé n'existe pas, la compilation échoue.
- Les éléments d'un tableau de `A` sont tous alloués avec le constructeur par défaut.

Exemple : (à corriger avec les chaines d'initialisation vues ci-après)

```
// Complex.h
struct Complex {
   // champs
   float r,i;
   // constructeur
   inline Complex() { r=i=0.f; };
   inline Complex(float v) { r=v; i=0.f; };
   inline Complex(float u, float v) { r=u; i=v; };
};
```

```
main() {
   Complex z0;           // z0 = 0+0i
   Complex z1(1.f);      // z1 = 1+1i
   Complex z2(2.f,4.f);  // z2 = 2+4i
   Complex *Pz0 = new Complex;      // pointe sur 0+0i;
   Complex *Pz1 = new Complex(2.f); // pointe sur 2+0i;
   Complex *PTz = new Complex[10];  // pointe sur 10 Complex
                                    // initialisés avec 0+0i
   ...
}
```

6. = constructeur défini si aucun constructeur n'est défini.

b) Chaine d'initialisation

Le but des chaines d'initialisation est d'utiliser les constructeurs de champs d'une classe dans le constructeur de cette structure.

Problème : Soit `struct B { A a; }`. Comment utiliser le constructeur `A(int)` dans un constructeur de `B` pour initialiser du champs `a` dans `B` ?

Pour une structure `struct S { T1 c1; T2 c2; }`, un constructeur de `S` utilisant la chaine d'initialisation est la suivante :

```
S(int u, int v) : c1(u), c2(v,4) { ... }
```

Exemple:

```
struct Complex {
   float r,i;
   Complex(): r(0.f), i(0.f) {};
   Complex(float v): r(v), i(v) {};
   Complex(float u, float v): r(u), i(v) {};
};
```

Remarques :

- Tous les champs n'ont pas besoin d'être passés à la chaine.
- L'ordre des champs de la chaine d'initialisation est sans importance : l'initialisation des champs est toujours effectuée dans l'ordre de déclaration des champs de la structure.
  warning possible du compilateur si l'ordre des champs dans la chaine d'initialisation et dans la structure n'est pas le même.

Les chaines d'initialisation doivent être **absolument le moyen privilégié** pour construire un objet :

- Ce type d'initialisation est toujours plus rapide que d'initialiser manuellement les champs que dans le bloc de code du constructeur.
- A savoir que, dès l'on entre dans le bloc du code du constructeur (*i.e.* on commence à exécuter le code entre accolade), l'objet est **déjà construit**.
  l'exécution de ce bloc de code peut être compilé comme un appel de fonction.
- C'est le seul moyen de lancer les constructeurs des champs internes qui possèdent des constructeurs.
  par extension, on verra que c'est aussi la seule façon de lancer les constructeurs sur les parties héritées de l'objet (voir la leçon sur l'héritage).

En conséquence, les chaines d'initialisation doivent être l'outil central autour duquel se la conception des constructeurs doit être effectuée.

Deux outils supplémentaires peuvent faciliter cette tâche seront vues ultérieurement dans cette leçon :

- les constructeurs délégués, qui permettent d'appeler dans un constructeur de l'objet `A` un autre constructeur de l'objet `A` lui-même permettant ainsi de mutualiser les codes communs aux différents constructeurs.
- les initialisations par défaut des membres, qui permettent d'affecter une valeur par défaut à un membre (en général constante) pour tous les constructeurs.

Un constructeur par défaut ne peut être que trivial, à savoir :

- il ne peut être défini que si :
  - ◇ chaque membre possède un constructeur par défaut (cas des types abstraits) ou est un type élémentaire (dans ce cas, constructeur = pas d'initialisation).

  - ◇ et que l'on est dans aucun des cas suivants : aucun membre non statiques n'a d'initialisation par défaut (C_{11}^{++}), ses classes de base n'ont pas un constructeur par défaut trivial (cf héritage), et n'a ni méthode virtuelle, ni classe de base virtuelle (cf polymorphisme).
- le code du constructeur par défaut d'une classe `A` est `A() {}`
- il s'exécute en appelant le constructeur par défaut sur chacun de ses membres (non statiques).

Attention : le constructeur par défaut n'est automatiquement défini que si aucun autre constructeur n'est défini (i.e. si les constructeurs définis par l'utilisateur ne contiennent pas un constructeur par défaut, il n'y a pas de constructeur par défaut).

Donc, si l'utilisateur défini un constructeur, et qu'il souhaite avoir un constructeur par défaut, il doit le redéfinir (cf le code par défaut ci-dessus).

c) Construction et assignation par copie

un objet peut être copié de deux façons :

- **par initialisation :** (**copy constructor**) à savoir un objet est initialisé au moment où il est créé (i.e. associé au **constructeur par copie**).
- **par assignation :** (**copy assignation operator**) on veut copier un objet dans un autre objet qui existe déjà (i.e. associé à l'**opérateur=** avec comme argument une référence à une `lvalue`).

Il faut porter une attention particulière au **constructeur par copie** car il est utilisé dans de nombreux cas :

- lors de la construction à partir d'un autre objet : `A a0(a1)`
- lors de l'affectation d'un objet à un autre à la construction : `A a0 = a1`
  ce **n'est pas** une assignation mais équivalent à la construction `A a0(a1)`.
- lorsqu'un objet passé par valeur à une fonction : `void fun(A a)`
- lorsqu'un objet est retourné par valeur par une fonction : `A fun(...)`
  Attention : dans ce cas, une élision de copie peut avoir lieu.

Leurs prototypes doivent être :

- pour le constructeur par copie, `A::A(const A&)` éventuellement, avec des paramètres supplémentaires ayant tous des valeurs par défaut.
- pour l'assignation par copie, `A& A::operator=(const A& a)`
  Exception : cas de l'idiome copy-and-swap.

Attention, l'assignation par copie doit toujours :

- vérifier que l'on n'essaye pas d'affecter un objet à lui-même (*i.e.* tester si `this == &a`), car ce cas pose presque toujours problème.
- retourner l'objet courant afin de permettre les chaines d'assignation (exemple : $z1 = z2 = z3$).
- dans l'exemple ci-dessous, écrire `*this=z` (au lieu d'affecter champ à champ) lance un appel récursif, car lance une assignation par copie.

Exemple:

```
struct Complex {
   float x,y;
   Complex(const Complex &z) : x(z.x), y(z.y) {}
   Complex& operator=(const Complex &z) {
      if (this != &z) { x=z.x; y=z.y; }
      return *this;
   }
};
```

Le code ci-dessus reste néanmoins un code naïf au sens de la gestion des erreurs (exception). Voir le TD correspondant pour une gestion correcte.

Règles :

- toute structure qui contient un pointeur (y compris dans l'un de ses agrégats) **doit** définir le constructeur et l'assignation par copie.
  copier un pointeur fait partager l'objet pointé entre les deux objets (l'original et le copié) ce qui n'est en général pas le comportement attendu (objets liés) et potentiellement dangereux (libération mémoire) sauf à utiliser des pointeurs intelligents.
- si un constructeur par copie est défini, alors une assignation par copie **doit** également être défini.
  sinon, cela créé un probable défaut de cohérence : la construction par copie fait donc quelque chose de potentiellement différent de l'assignation par copie.

Remarques :

- S'il n'est pas défini, le compilateur définit un constructeur (resp. une assignation) par copie par défaut,
  attention : voir limitations plus loin
- le constructeur (resp. l'assignation) par copie par défaut utilise le constructeur (resp. l'assignation) par copie par défaut sur chacun des champs.
  pour tous les types triviaux, c'est une copie binaire.

Exemple :

```
const int VectorSize = 10;
struct Vector {
   float *v;
   Vector() : v(new float[VectorSize]) {}
   Vector(const Vector &z)
      : v(new float[VectorSize])
      { memcpy(v,z.v,VectorSize*sizeof(float)); }
   Vector& operator=(const Vector &z) {
      if (this != &z)
         memcpy(v,z.v,VectorSize*sizeof(float));
      return *this;
   }
   ...
};
```

```
main() {
   Vector V1;
   Vector V2(V1); // constructeur par copie
   ...
   V2.v[0]=0.f;
   V1 = V2;      // assignation par copie
   ...
};
```

d) Construction et assignation par déplacement

En C_{11}^{++}, en raison de la possibilité de faire référence à une rvalue, on peut maintenant savoir si le paramètre passé à un constructeur ou une assignation par copie est un objet temporaire.

Remarques :

- le déplacement n'a de sens que pour les classes dans lesquelles il y a quelque chose à déplacer, à savoir une zone de mémoire externe (le plus souvent privée) à l'instance pointée par un membre de la classe.
- un membre interne interne à l'instance (à savoir stocké dans l'espace obtenu par le `sizeof` de la classe) ne saurait être déplacé, car la destruction de l'objet temporaire détruira aussi ses membres.
- **autre formulation :** ne peut être potentiellement déplacé que les objets qui nécessitent des destructions spécifiques dans le code du destructeur. Il peuvent alors être déplacé plutôt que détruit.

Attention : certaines écritures qui sembleraient relever du constructeur par déplacement sont en fait traité par élision de copie (voir la section à ce sujet dans cette section).

Exemple : (élision de copie et non construction par déplacement)

```
A fun();
A a = fun(); // élision de copie
```

Un objet peut être déplacé de deux façons :

- **par initialisation :** à savoir un objet est initialisé à partir du déplacement d'un objet temporaire ou en fin de vie (i.e. associé au **constructeur par déplacement**).
- **par assignation :** on veut déplacer un objet temporaire ou en fin de vie dans un autre objet qui existe déjà (i.e. associé à `operator=` avec comme argument une référence à une `rvalue`).

Deux méthodes supplémentaires sont définies par défaut :

- **le constructeur par déplacement (move constructor) :** qui permet de construire le résultat d'une rvalue directement dans l'objet à construire.
  prototype : `T(T&&)`
- **l'opérateur d'assignation par déplacement (move assignment operator) :** qui permet d'assigner le résultat de la rvalue directement dans un objet déjà existant.
  prototype : `T& operator=(T&&)`

Des exemples sont explicités en annexe de la leçon.

Règles :

- toute structure qui contient un pointeur (y compris dans l'un de ses agrégats) **devrait** définir le constructeur et l'assignation par déplacement.
  optimisation : recycle des données plutôt que de les détruire.
  évite de copier des données qui doivent être détruite.
- si un constructeur par déplacement est défini, alors une assignation par déplacement **doit** également être défini.
  sinon, probable défaut de cohérence : la construction par déplacement est donc potentiellement différente de l'assignation par déplacement.
- notons que si un constructeur/assignation par copie est défini, alors un constructeur/assignation par déplacement permettrait l'optimisation.

Remarques :

- S'il n'est pas défini, le compilateur définit un constructeur (resp. une assignation) par déplacement par défaut,
  attention : voir limitations plus loin
- le constructeur (resp. l'assignation) par déplacement par défaut utilise le constructeur (resp. l'assignation) par déplacement par défaut sur chacun des champs.
  pour tous les types triviaux, c'est une copie binaire.
  donc, **déplacement par défaut = copie**.

Exemple :

```
const int VectorSize = 10;
struct Vector {
   float *v;
   Vector() : v(new float[VectorSize]) {}
   // constructeur par déplacement
   Vector(Vector &&z)
      : v(z.v)  // récupère le pointeur de z
      { z.v = nullptr; } // met le pointeur de z à null
                         // ainsi z peut être détruit
   Vector& operator=(Vector &&z) {
      if (this != &z) // si z n'est pas l'objet courant
        // échange les pointeurs entre l'objet courant et z
        std::swap(v,z.v);
      return *this;
   } ...
   ~Vector() { delete v; }
};
```

Attention : un constructeur par déplacement ou une assignation par déplacement doit toujours laisser l'objet dont on "vole" les ressources dans un état cohérent. Ce point est précisé dans l'annexe. Les ressources que l'on peut "voler" étant essentiellement des ressources allouées dynamiquement, il est aussi préférable de lire au préalable le chapitre sur la gestion mémoire.

3.2 Destructeur

Un destructeur est une méthode appelée automatiquement à chaque fois qu'une instance de la classe est libérée.

Donc, le destructeur est obligatoirement (et automatiquement) appelé :

- en fin de portée d'une instance de la structure (variable locale, paramètre passé par valeur).
- lors d'un `delete` vers une instance ou un tableau d'instances de la classe.

Remarques : pour une classe `A`

- le nom du destructeur est : `~A()` (par d'argument, pas de retour).
- il ne peut y avoir qu'un seul destructeur défini.
- si aucun destructeur est défini, le compilateur en définit un par défaut.
- le destructeur par défaut lance le destructeur sur chacun des membres de la classe (et son code est `~A() {}`).

Exemple :

```
struct Vector {
   int n; float *v;
   Vector(int N):n(N) {
      v = new float[N];
   }
   ~Vector() {
      delete [] v;
   }
};
```

```
main() {
   Vector V(4);
   Vector *pV = new Vector(5);
   ...
   delete pV; // pV->~Vector();
   ...
}; // V.~Vector()
```

3.3 Règle de génération des méthodes automatiques

Règles conduisant à l'absence de génération des méthodes automatiques par défaut :

- si n'importe quel constructeur est explicitement déclaré, alors le constructeur par défaut n'est pas généré,
- si un destructeur virtuel est déclaré, alors le destructeur par défaut n'est pas généré (voir leçon suivante pour la virtualité)
- si un constructeur ou une assignation par déplacement est explicitement déclaré, alors ni le constructeur par copie par défaut, ni l'assignation par copie par défaut n'est généré.
- si un constructeur ou une assignation par copie, un constructeur ou une assignation par déplacement, ou un destructeur est déclaré, alors ni le constructeur par déplacement par défaut, ni l'assignation par déplacement par défaut n'est généré.

En C_{11}^{++}, les règles suivantes sont destinées à renforcer la cohérence des classes :

- si un constructeur par copie ou un destructeur est explicitement déclaré, alors l'assignation par copie est obsolète.
- si une assignation par copie ou un destructeur est explicitement déclaré, alors le constructeur par copie est obsolète.

Note : obsolète = ces règles finiront par générer des warnings.

a) Membres par défaut

Problème avec la gestion par des membres par défaut en C++ :

- la définition des constructeurs est couplée : définir n'importe quel constructeur supprime le constructeur par défaut,
- le destructeur par défaut est inapproprié pour le polymorphisme de classe, et nécessite une définition explicite (raison du destructeur virtuel).
- une fois un défaut supprimé, impossible de le réutiliser.
- l'implémentation par défaut est souvent implémentée plus efficacement que si elle est définie manuellement,
- pas moyen d'empêcher une méthode par défaut ou un opérateur global sans déclarer un substitut (avant : créer une méthode private).

Le C^{++}_{11} permet une gestion effective des membres par défaut :

- L'écriture `=default` spécifie que l'implémentation par défaut doit être utilisée pour ce membre,
- L'écriture `=delete` spécifie que l'implémentation par défaut doit être retirée de la définition de la classe, utilisée aussi pour désactiver certaines conversions ou instanciations de templates non souhaitées.

Ecriture `=default` :

L'écriture `=default` spécifie que l'implémentation par défaut doit être utilisée pour ce membre.

Exemple:

```
struct type {
  type() = default; // conserve le défaut efficace
  virtual ~type(); // destructeur virtuel
  type(const type &); // declaration
};

// utilise le constructeur par défaut (version inline)
inline type::type(const type &) = default;
// utilise le destructeur par défaut
type::~type() = default;
```

Notes :

- une définition inline et par défaut est triviale ssi la définition implicite aurait été triviale.
- un destructeur virtuel peut utiliser le destructeur par défaut de la manière suivante (la définition du type n'est alors plus triviale)
  `virtual ~type() = default;`

Ecriture `=delete` :

- Utilisation de `=delete` pour spécifier qu'une implémentation par défaut doit être retirée de la définition de la classe.
  Exemple 1 : suppression de la copie par assignation et du constructeur par copie

```
struct type {
    type & operator =(const type &) = delete;
    type(const type &) = delete;
    type() = default;
};
```

- Utilisation de =delete pour éviter l'instanciation de certains types dans un template.
  Exemple 2 : suppression de l'instanciation d'un type particulier dans un template

```
class Widget {
public:
    template<typename T> void processPointer(T* ptr) { ... };
};
template<> void Widget::processPointer<void>(void*) = delete;
```

- Utilisation de =delete pour éviter certaines conversions automatiques.
  Exemple 3 : suppression de conversions automatiques non souhaitées

```
struct type {
    type(long long); // constructeur avec un long long
    type(long) = delete; // mais rien de moins
};
// idem avec une fonction
extern void bar(type, long long); // appel avec long long
void bar(type, long) = delete; // mais rien de moins
```

  Exemple 4 : suppression de conversions automatiques non souhaitées.

```
bool isLucky(int number); // fonction originale
bool isLucky(char) = delete; // rejette les chars
bool isLucky(bool) = delete; // rejette les bools
bool isLucky(double) = delete; // rejette les doubles/floats
```

b) Constructeurs délégués

En C++ classique, lors de l'écriture de constructeur, il arrive fréquemment que :

- le code de ces constructeurs se ressemble beaucoup,
- l'on finisse par écrire une fonction d'initialisation privée appelée par l'ensemble des constructeurs.

C_{11}^{++} permet l'utilisation d'un constructeur délégué, à savoir la possibilité d'appeler un constructeur dans un constructeur.

Exemple :

```
class A {
private: int x,y,z;
public:
    A(int u, int v, int w): x(u), y(v), z(w) {};
    A(int u, int v) : A(u,v,u+v) {};
    A(int u) : A(u,u,2*u) {};
    A() : A(0,0,0) {};
};
```

Note : lorsqu'un constructeur délégué est appelé, aucune construction de membre ou appel à un autre

constructeur délégué ne peut être ajouté dans la liste d'initialisation.

c) Qualification de référence

Dans une classe, la qualification de référence d'une méthode est un moyen d'indiquer :

- avec `const`, que la méthode laisse l'objet sur lequel elle s'exécute constant,
- avec `&` (resp. `&&`), que la méthode (non statique) s'exécute sur un objet sous forme d'une `lvalue` (resp. `rvalue`).

Exemple :

```
class A { ...
public:
   void View() const;
   void doWork() &;   // si *this lvalue
   void doWork() &&;  // si *this rvalue
};
A makeA();
```

```
A    a;
a.doWork();         // appel doWork()&
makeA().doWork();   // appel doWork()&&
```

Remarques :

- ces qualifications permettent notamment d'effectuer des moves dans le cas des `rvalues`,
- la surcharge de deux fonctions membres avec les mêmes types de paramètres requiert que les deux aient des qualificateurs de référence ou qu'elles en soient dépourvues,
- une qualification `const &` ou `const &&` est possible.
- un destructeur ne doit pas être déclaré avec un qualificateur de référence.

Exemple :

```
class Y {
    void h() &;
    void h() const &; // OK
    void h() && ;     // OK
    void i() &;
    void i() const;
// erreur: pas de qualif. ref.
};
```

d) Limites

Si les constructeurs permettent de faciliter et d'automatiser l'initialisation des instances d'une structure, néanmoins :

- on rappelle qu'une initialisation inutile est du temps perdu. L'aspect automatique des constructeurs permet d'en effectuer à très grande échelle.
  Il est souvent plus raisonnable que le constructeur par défaut n'initialise que ce qui est nécessaire.
  A savoir : `A() {};`

- La définition d'une constructeur `A(T t)` autorise l'écriture `A a=t`, interprété par le compilateur comme `A a=A(t)`.
  Ce type de construction est dite implicite.
  Règle : prendre l'habitude de désactiver la construction implicite pour les constructeurs à un argument de la façon suivante :
  `explicit A(T t)`
  dans la définition de la classe. Sinon, toute écriture qui pourra effectuer une conversion implicite avec l'un des constructeurs existant y fera appel automatiquement.
  Ne la réactiver que s'il s'agit d'un comportement essentiel pour la classe.

Truc 1 : Interdire une constructeur par défaut non explicite pour un type `A`

1. définir un type (par exemple DEFAULT) comme `struct DEFAULT ;`
2. définir comme constructeur :
3. définir dans le code une variable globale `Default` de type DEFAULT : `DEFAULT Default;` (un seul nécessaire pour toutes les structures).
4. lors de la définition des constructeurs :
   - ne pas définir de constructeur par défaut `A()`
   - définir le constructeur `A(DEFAULT);`
5. pour initialiser un objet de la structure `A` avec son constructeur par défaut, faire `A a(Default);`.

Exemple :

```
struct DEFAULT {};
DEFAULT Default;

struct A {
   // pas de A();
   A(DEFAULT) { ... };
   ..;
}
```

```
// attention: default est
// un mot-clé du langage

main() {
   A a(Default);
   ...
}
```

4 Initialisations

4.1 Initialisation par défaut

C_{11}^{++} permet également de définir des valeurs par défaut pour les membres des classes.

Exemple:

```
struct S {
   int Id() { static int id = 0; return ++id; };
   int n=7, p{n}, q=Id(), r{};
   std::string s1="abc", s2{ 'a', 'b', 'c' };
   S(): n(3) {};
   S(int v) : q{v} {};
};
S s1;    // n=3, p=3, q=1, r=0, s1="abc", s2="abc"
S s2(4); // n=7, p=7, q=4, r=0, s1="abc", s2="abc"
S s3;    // n=3, p=3, q=2, r=0, s1="abc", s2="abc"
```

Attention :

- L'initialisation par défaut d'un membre n'est utilisée que si ce membre ne fait pas partie de la liste d'initialisation.

- En conséquence, si un membre est initialisé par défaut, toujours faire en sorte que sa valeur ne soit modifiée pas dans le code du constructeur (sinon l'initialisation est double).
- Idée de cette fonctionnalité : "factoriser" les initialisations communes aux constructeurs (= réduire le code à écrire)
  mal utilisée : initialisations nombreuses inutiles et coûteuse.

4.2 Initialisation uniforme

Avant C_{11}^{++}, il y avait 3 façons d'initialiser un objet :

- avec `( ... )`, lors de l'appel d'un constructeur
- avec `{ ... }`, lors de l'initialisation d'un type agrégé (classe agrégée ou tableau)
- sans parenthèse (pour lancer le constructeur par défaut).

Exemple:

```
class A { ... };  // avec A::A(int) défini.
struct B { int b1; float b2; };   // type agrégé
A  ad;      // init. par défaut A::A()
A  a(2);    // init. par constructeur A::A(int)
B  b = {1,2.3f};          // init. type agrégé
int c[3] = { 1, 2, 3 };  // init. tableau
// init. tableau d'objets
A  aTab[3] = { A(1), A(2), A(3) };
// init. tableau de types agrégés
B  bTab[2] = { {1,2.3f}, {4,5.6f} };
```

Avec de nombreux cas où l'initialisation n'était pas possible :

- **avec les containers** :

```
vector<int> V = {1,2,3};    // non autorisé
```

- **en argument d'un fonction** :

```
int f(const std::vector<int> &v);
// int f(const int *v);
fun({ 1,2,3 }); // non autorisé
```

- **lors de l'allocation dynamique** :

```
// tableau dynamique de 3 entiers initialisés
// avec 1, 2 et 4
int *t = new int[3] { 1, 2, 4 }; // non autorisé
// tableau dynamique de 3 objets initialisés
// avec A(1), A(2) et  A(4)
A *ta = new A[3]{ 1,2,4 };        // non autorisé
```

- ...

Idée de l'initialisation uniforme (C_{11}^{++}) : permettre l'initialisation avec `{ ... }` dans tous les cas.

a) Initialisation uniforme pour les BITs ou les types agrégés

- Initialisation uniforme de la forme `T x{v}` (directe) ou `T x={v}` (par copie).
- L'initialisation uniforme est **sans perte de précision** (*i.e.* `int a{3.5f}` provoque une erreur de compilation).

- Introduction d'une initialisation uniforme par défaut avec `T x{}`.
  0 pour tous les BITs (int=0, float=0.f, bool=false), chaque champ à 0 pour les types agrégés.
- Pour les types agrégés, les champs non initialisés entre `{...}` sont initialisés à 0.

Exemple:
```
// 4 façons d'initialiser un BIT
int x=4, y(4), z = {4}, z{4};
// initialisation sans conversion restrictive
float a{2};     // ok: possible sans perte de précision
int   b{3.4f}; // erreur de compilation
float c{ 123456789 }; // erreur de compilation
float d{ x }; // erreur de compilation
// initialisation d'un type agrégé
struct Pos { int x,y; };
Pos P{ 1,2 }; // équivalent à P={1,2}
Pos Q{ 1 };   // équivalent à Q={1,0}
Pos R{};      // équivalent à R={0,0}
Pos S;        // S=(?,?) non initialisé
```

b) Initialisation uniforme pour les classes

- L'appel à un constructeur `A(x,y,z)` peut être toujours remplacé par un appel `A{x,y,z}`.
- L'appel `A{x,y,z}` doit se faire sans **perte de précision** sur chacun des arguments.
- L'appel `A{}` fait appel au constructeur par défaut
  Exception : si aucun constructeur défini = initialise tous les champs à 0.
- Un constructeur `explicit` n'accepte pas une construction avec =.

Exemple:
```
// déclaration
class A {  private: float x;
           public:  A() : x(1.f) {}
                    A(float X) : x(X) {}  };
// initialisations
A   x{};     // équivalent à x()
A   y{3};    // équivalent à y(3)
A   z{3.2f}; // échec compilation
             // car float->int avec perte de précision
```

Exemple:
```
// cas particulier: pas de constructeur défini
class B { private: float x,y; };
B   b{}; // initialise les champs à 0: b={0,0}
```

c) Initialisation uniforme pour les tableaux

- seul changement apparent pour les tableaux statiques de BITs (initialisation déjà uniforme) : la conversion restrictive.
  L'écriture `T a[]{ ... }` est aussi possible (*i.e.* sans le =).
- pour les tableaux statiques d'objet, même syntaxe que pour les tableaux statiques de BITs, sauf que chaque élément de la liste d'initialisation est passé au constructeur de chaque objet du tableau.
  Note : chaque élément de la liste peut faire appel à un constructeur différent.
- possible maintenant sur les tableaux dynamiques de BIT : ajouter la liste d'initialisation derrière l'argument du `new`.

- possible maintenant sur les tableaux dynamiques d'objet : idem.

Exemple:
```
struct Pos; // objet agrégé
Pos  as[3] = { {0}, {}, {4,5} };         // ok statique
Pos *ad = new Pos[3]{ {0}, {}, {4,5} }; // idem dynamique
class Couple; // classe
Couple bs[3] = { {0}, {}, {4,5} };             // ok statique
Couple *bd = new Couple[3]{ {0}, {}, {4,5} }; // idem dyn.
// note: un constructeur différent pour chaque élément
```

d) Formes et sens de l'initialisation uniforme

`T obj{Lst}`	D	init. directe de obj par `{Lst}`
`T obj={Lst}`	C	init. par copie de obj par `{Lst}`
`T{Lst}`	D	init. objet non nommé par `{Lst}`
`new T{Lst}`	D	init. objet dynamique par `{Lst}`
`class C { T m{Lst}; }`	D	init. directe du membre non statique m par `{Lst}`
`class C { T m={Lst}; }`	C	init. par copie du membre non statique m par `{Lst}`
`C::C():m{Lst} {...}`	D	init. du membre dans le const. par `{Lst}`
`fun({Lst})`	C	appel `fun` avec arg. init. par `{Lst}`
`return {Lst}`	C	type de retour initialisé avec `{Largs}`.
`obj[{Lst}]`	C	appel op. `[]` avec arg. init. par `{Lst}`
`obj={Lst}`	C	assignation de `obj` par l'opérateur = auquel `{Lst}` est passé en paramètre.
`T({Lst})`	C	construction de l'argument passé en paramètre du constructeur de `T` avec `Lst`.

D : considère les constructeurs explicites et non-explicites.
C : seulement les constructeurs par copie ou non explicites sont appelés.

e) Note sur les containers

Tous les containers std supportent aussi les "initialisations uniformes" :

Exemple :
```
std::vector<int> v{1,2,4,8};
```

Également utilisable dans les paramètres des fonctions :

Exemple :
```
void f(const std::vector<int>& v); // décl.
f({ val1, val2, 10, 20, 30 });     // appel
```

Cette fonctionnalité peut être implémentée dans tout container à travers le constructeur qui prend en paramètre une `initializer_list`.

Exemple : initialisation uniforme pour un container.

```
class A {
private: std::vector<int>  v;
public: A(std::initializer_list<int> list) {
          for(auto i : list) v.push_back(i);    }
};
A  a{ 1,2,3,4 };   // utilisation
```

Note : la liste d'initialisation de l'initialisation uniforme supportée par le compilateur est différente d'une `initializer_list<T>` (pour laquelle tous les éléments sont du même type `T`).

4.3 Liste d'initialisation

Le support d'une liste d'initialisation par un constructeur peut interférer avec les autres constructeurs.

Le comportement est alors le suivant :

- si le constructeur avec liste d'initialisation n'est pas défini, les constructeurs classiques sont lancés lors d'un appel avec `{...}`,
- si le constructeur par défaut et la liste d'initialisation existent :
  - ◇ `A a{}` appelle le constructeur par défaut,
  - ◇ `A a({})` ou `A a{{}}` appelle le constructeur avec liste d'initialisation.
- si le constructeur à 2 arguments et la liste d'initialisation existent :
  - ◇ `A a(x,y)` appelle le constructeur avec 2 arguments.
  - ◇ `A a{x,y}` ou `A a{{x,y}}` appelle le constructeur avec liste d'initialisation.

⇒ appel `{}` prioritairement redirigé vers le constr. avec liste d'initialisation.

Exemple:
```
class A { ...
public: A(int x, int y);                    // #1
        A(std::initializer_list<int> r); // #2
};
A   a1(u,v);    // appel #1
A   a2{u,v};    // appel #2
```

Attention au fonctionnement de certaines classes `std`.

Exemple:
```
// vecteur de 10 éléments initialisés à 20
std::vector<int> v1(10, 20);
// vecteur de deux éléments dont les
// deux valeurs respectives sont 10 et 20.
std::vector<int> v2{10, 20};
```

En général, il faut faire en sorte que les appels avec et sans accolade fassent appels à la même méthode.

Conséquence : deux types d'approches sont possibles

- **approche accolades par défaut :** utiliser des partout, sauf lorsque ce n'est le constructeur souhaité qui est appelé. Exige de connaître exactement les règles d'appel.
- **approche parenthèses par défaut :** utiliser des `()` partout, même lors d'appel des listes i.e. `({ ... })`, mais perte de fonctionnalité (ambiguïté de la définition `A a()` et appel sans perte de précision).

Il n'y a pas de consensus sur laquelle de ces approches est la meilleure, mais une seule approche doit être choisie et appliquée avec consistance (personnellement : préférence pour la seconde qui lève toute ambiguïté sur l'appel).

5 Opérateurs

En C++, il est possible de définir (ou redéfinir) des opérateurs sur des structures. Par opérateur, on entend (liste non exhaustive) :

- les opérateurs arithmétiques : =, +, -, *, /, %, +=, -=, *=, /=, ...
- les opérateurs de comparaison : ==, !=, >, <, >=, <=
- les opérateurs logiques : !, &&, ||
- les opérateurs sur les membres et les pointeurs : [], *, &, ->, .
- les opérateurs de redirection de flux (E/S) : <<, >>
- *etc* ...

La page wikipédia en anglais sur "Operators in C and C++ " permet de voir l'ensemble des opérateurs qui peuvent être surchargés, et donne les méthodes à définir afin de réaliser cette surcharge.

Les opérateurs sont pratiques. Ils permettent :

- de donner un sens à certains opérateurs sur des objets.
- l'écriture d'expressions arithmétiques ou logiques.

et donc de faciliter l'utilisation de ces objets, en introduisant une écriture naturelle pour l'utilisateur sur tous les objets où ces opérateurs ont du sens.

Par exemple, si le TDA `Vector` représente un vecteur, on peut définir :

- l'opérateur == pour comparer deux vecteurs `a` et `b` (dont la définition signifie : même taille + même valeur pour chacune des composantes du vecteur.
- l'opérateur + pour ajouter deux vecteurs, composante par composante.
- l'opérateur `*` pour multiplier le vecteur par une constante (chaque composante est multiplié par la constante).
- l'opérateur `[]` permet d'accéder aux $i^{\text{ème}}$ élément du vecteur.
- l'opérateur = pour donner à = un sens différent du constructeur par copie (par exemple, affectation seulement si la taille est identique).
- l'opérateur << permet d'afficher l'objet dans un flux de sortie.

Typiquement, il existe deux façons de définir des opérateurs :

- soit comme méthode interne à la structure.
  par exemple : `R T::operator+(S b)` pour implémenter `a+b` où `a` est la structure courante et retourne une structure contenant la somme.
  L'écriture `c = a+b;` est compilée comme : `c = a.operator+(b);`
- soit comme fonction externe à la structure.
  par exemple : `R operator+(S a, T b)` pour implémenter `a+b` et retourne une structure contenant la somme.
  L'écriture `c = a+b;` est compilée comme : `c = operator+(a,b);`

Exemple :

```
struct Complex {
    float r,i;
    Complex(float u, float v) : r(u), i(v) {};
    Complex operator*(float b) { return Complex(r*b,i*b); };
};
Complex operator+(const Complex &a, const Complex &b)
    {  return Complex(a.r+b.r,a.i+b.i);  }
main() {
    Complex z1(1.f,2.f), z2(0.f,-1.f);
    Complex z=z1+z2*3.f;
}
```

Remarques :

- le type des paramètres d'un opérateur peut être adapté avec `&` et `const`.
  Dans l'exemple précédent : `Complex operator+(const Complex &a, const Complex &b)`
- faire très attention à ne pas modifier la structure affectée en même temps que le calcul est fait (utiliser `this` si besoin).
  Exemple : sur un calcul entre complexes, `a *= b` (peut conduire à l'écriture $x_a = x_a.x_b - y_a.y_b$ et $y_a = x_a.y_b + x_b.y_a$. Écrit ainsi, le calcul de x_a modifie sa propre valeur avant le calcul de y_a et rend ce dernier faux, d'où l'obligation d'utiliser une variable intermédiaire).
- pour une opération binaire entre deux objets `a` et `b` de types différents :
  - ◇ il faut alors définir deux surcharges d'opérateurs afin de pouvoir écrire, par exemple, `a+b` ou `b+a`
    Exemple : `Vector operator*(Vector,float)` et `Vector operator*(float,Vector)`.
  - ◇ ceci permet de rendre l'opération commutative si la définition est la même, ou non commutative si les définitions sont différentes ou si une seule est donnée.
- En général, toujours préférer la fonction externe lorsqu'elle existe : le prototype est plus intuitif.
  Néanmoins, la version interne possède l'énorme avantage de pouvoir être virtualisée, et donc de gérer des opérateurs polymorphes (cf leçon associée).
- si la classe `T` contient un type tableau de type `U`, et que l'on veut définir l'opérateur `[]` pour l'accès aux éléments, alors il est nécessaire de définir deux opérateurs :
  - ◇ `U& T::operator[](size_t)` qui permet de lire/écrire les éléments de tout objet mutable.
  - ◇ `const U& operator[](size_t) const` qui permet de lire les éléments de tout objet constant.

  observer dans ce cas l'utilisation du retour d'une référence qui éviter de copier l'objet stocké.
- **Rappel :** Lorsqu'un opérateur retourne un constructeur d'une classe, le compilateur peut construire le résultat directement dans l'objet résultat (cas du complexe `z` dans le transparent précédent, voir élision de copie).

Les opérateurs ont un côté obscur : la complexité de l'écriture correcte d'une algèbre avec l'ensemble des opérateurs arithmétiques, gérant la totalité des cas, et sans bogues est **très importante**.

- il faut être conscient que l'utilisation d'une suite d'opérateurs produit des appels successifs aux fonctions/méthodes qu'ils définissent en imbriquant les appels.

- ainsi l'écriture d'une expression arithmétique complexe peut engendrer de nombreux objets intermédiaires.
  Exemple : `Vector a=3*c+d*e;` génère au moins deux vecteurs intermédiaires (résultats de `3*c` et de `d*e`).
- les expressions complexes ont donc tendance à générer un grand nombre d'objets intermédiaires, possiblement inutiles, ce qui est un gaspillage de ressources et de temps si les objets sont de grande taille.
- on peut se mettre à écrire autre chose que ce que l'on croit écrire, ou devenir totalement inconscient de la complexité ce ce que l'on écrit.
- il faut compter sur la créativité d'un utilisateur des surcharges pour les utiliser dans un cadre ou d'une façon complètement non prévue par le concepteur, même s'il s'agit du même individu.

En conséquence,

- vous risquez de passer plus de temps à déboguer une erreur liée à l'utilisation d'un opérateur qu'à écrire explicitement les choses.
- cela nécessite **absolument** d'écrire une batterie de tests unitaires afin de tester le bon fonctionnement des opérateurs dans tous les cas imaginables.

Il est **fortement déconseillé** (dans un premier temps) d'utiliser de la surcharge d'opérateurs afin de gérer des calculs arithmétiques complexes.

Noter qu'avec le C^{++}_{11}, des optimisations utilisant le passage par référence à une rvalue peuvent être effectuée, ce qui peut réduire la charge associée aux objets temporaires, mais exige d'écrire les opérateurs prenant en paramètre des rvalues (donc 3 opérateurs supplémentaires par opérateur binaire).

Donc, dans un premier temps, limitez-vous aux opérateurs suivants :

- les opérateur de comparaison (qui généralement produisent un booléen et ne modifient pas les objets comparés),
- les opérateurs d'accès `[]`,
- les opérateurs avec affectation, qui généralement, ne produisent pas d'objet temporaire (par exemple, +=).
- les opérateurs de sortie de flux.

et n'implémentez les opérateurs algébriques qu'avec la plus grande parcimonie.

5.1 Conversions

Le C^{++} permet définir des conversions particulières :

- on rappelle que pour une classe `T`, un constructeur du type `T(int)` permet de convertir un entier en type `T`.
  Exemple :
  ```
  class T {
      T(int) {...}
  };
  T x;
  x = 4; // interprété comme x = T(4)
  ```
- inversement, la définition dans la classe `T` de la méthode `operator U()` (où `U` est un type) permet d'indiquer comme convertir le type `T` en type `U`.
  Cette méthode doit retourner un type `U`.

Exemple :

```
class T {
  operator int() {...}; // doit retourner un int
};
T x;
int v = x; // interprété comme v = x.operator_int()
```

Attention : penser à utiliser le mot-clef `explicit` afin de rendre ces conversions explicites et de ne pas laisser ces conversions à la décision du compilateur.

5.2 Littéraux

Rappel : un littéral permet de définir l'unité d'une valeur numérique à convertir dans un type particulier, et par exemple, d'initialiser un type distance avec la valeur littérale `12_km`.

La définition d'une forme littérale `n_x` (où `n` est la valeur, et `x` le suffixe) s'effectue en définissant la fonction `T operator"" _x(U)` où `T` est le type produit par ce littéral et `U` le type de la valeur.

Il y a plusieurs types de littéraux :

- les littéraux entiers : entier, octal (préfixe 0), hexadécimal (préfixe 0x), binaire (préfixe 0b),
  prototype : `T operator"" _x(unsigned long long)`
- les littéraux flottants : contient `.` ou `e`
  prototype : `T operator"" _x(long double)`
- les littéraux caractères :
  prototype : `T operator"" _x(U)` où `U` est un type caractère (`char`, `char16_t`, `char32_t` ou `wchar_t`).
- les littéraux chaines de caractères :
  prototype : `T operator"" _x(V,std::size_t)` où `V` est un type chaine de caractères (`const U*` où `U` est un type caractère) et `size_t` contient la longueur de la chaine.

Exemple :

```
// conversion en degré
long double operator"" _deg(long double deg) {
    return deg*3.141592/180; }
// type utilisateur
struct bitsize_t {
  size_t  value;
  operator size_t() const { return value; }
  explicit bitsize_t(unsigned long long n)
      : value(static_cast<size_t>(n)) {}
};
inline bitsize_t operator"" _bits(unsigned long long n)
  { return bitsize_t(n); }

// utilisation
int main(){
    double     x = 90.0_deg; // appel operator"" _deg
    bitsize_t  s = 24_bits;  // appel operator"" _bits
    size_t     n = s;        // appel bitsize_t::size_t()
}
```

Remarque : ne pas hésiter à définir ces fonctions comme `constexpr` afin de permettre les conver-

sions pendant la compilation.

6 Membres et fonctions statique

Le mot-clé `static` peut être utilisé pour définir :

- **un champ statique :** champ partagé entre toutes les structures.
  - ◇ Il est déclaré comme un champ normal avec le modificateur `static`.
  - ◇ On a accède comme tous les autres champs (*i.e.* par son nom).
  - ◇ Il ne compte pas dans le `sizeof` de la structure.
  - ◇ Il est unique et stocké à l'extérieur de la structure.
  - ◇ Il est initialisé à l'extérieur de la structure avec l'opérateur de résolution de portée.
- **une méthode statique :** méthode qui n'accède ou ne modifie que des champs statiques. ajouter le modificateur `static` devant la déclaration de la fonction.

Remarque : les champs/fonctions statiques sont partagés par TOUTES les instances de structure de ce type ou qui utilisent ce type. Attention à ce que cela soit bien le comportement souhaité.

Accès :

- soit à partir d'une instance de la structure.
- soit en utilisant l'opétateur de résolution directement à partir du nom de la structure.

Exemple :

```
// Déclarations
// dans Vector.h
struct Vector {
    static int nV;  // nombre de vecteurs alloués
    static int getnVectors() { return nV; };
    ...
};
// dans Vector.cpp (déclaration + valeur initiale)
int Vector::nV = 0;
```

```
// Utilisation
main() {
    // accès direct par opérateur de résolution de portée
    int a = Vector::nV;
    if (a == Vector::getnVectors()) ...
    // accès à travers une instance
    Vector  v;
    if (v.nV == v.getnVectors()) ...
}
```

7 Constance de classe

7.1 Champs constants

Propriété d'un champs constant :

- Le type déclaré d'un champ constant est modifié par le mot-clé `const`.
- Ce champs ne peut plus être modifié une fois que la structure a été créée.

- Un champs constant doit obligatoirement être initialisé dans la chaine d'initialisation.
- Ce champs est constant dans une instance de la structure, mais peut avoir des valeurs différentes entre les différentes instances.
- L'initialisation du champs n'est pas autorisés dans la déclaration du champ.

Exemple : vecteur dont la taille ne peut plus être changée après création.

```
struct Vector {
    const int n;
    float *v;
    Vector(int N) : n(N), v(new float[N]) {};
};
```

Attention : à part lorsque le constructeur par copie est utilisé (initialisation, passage en paramètre d'une fonction ou méthode), toute copie sur une instante déjà existante échoue (car a pour conséquence de modifier son champ constant).
Une constante de structure est une constante définie dans la structure et qui a une valeur unique pour toutes ses instances.

- Le type déclaré d'une constante de structure est modifié par les mots-clé `static const`.
- On a y accède comme tous les autres champs (*i.e.* par son nom).
- Il ne compte pas dans le `sizeof` de la structure (unique et donc stocké à l'extérieur de la structure).
- Il est déclaré à l'extérieur de la structure avec l'opérateur de résolution de portée (dans le `.cpp`).
- Sa valeur est initialisée :
  cas 1 soit avec la déclaration de la constante dans la structure (à savoir dans le `.h`) : valable seulement pour les types simples.
  cas 2 soit à l'extérieur, au moment où elle est déclarée à l'extérieur.

Attention : ne pas oublier le `static`, `const` seul fait seulement un champ dont la valeur est constante lors de la vie de l'instance, mais pouvant être différent entre deux instances.

Exemple : de deux façons différentes d'initialiser des constantes de structure.

cas 1 :

Définition et initialisation dans la structure, déclaration à l'extérieur de la structure. Cette approche est limitée aux types simples.

Exemple :

```
// stack.h
struct Stack {
  static const int Size=32;
  int data[Size];
};
// stack.cpp
const int Stack::Size;
```

cas 2 :

Définition dans la structure, déclaration et initialisation à l'extérieur de la structure.

Marche pour tous les types, mais ne peut pas être utilisées pour définir des tailles fixes comme dans le cas 1.

Exemple :

```
// stack.h
struct Stack {
  static const int MaxSize;
  int Size;
  bool HaveMaxSize(void) { return (Size==MaxSize); };
};
// stack.cpp
const int Stack::Size = 32;
```

Note :

Il est également possible de définir une constante entière comme membre d'un `enum` interne à la classe.

Exemple :

```
struct Stack {
    enum { size = 32 };
    int data[size];
}
```

Cette méthode doit être considérée comme obsolète, et était auparavant utilisée lorsque l'écriture `static const` n'était pas autorisée dans le standard.

7.2 Méthodes constantes

une méthode constante est une méthode qui ne modifie aucun des champs.

Autrement dit :

- la structure n'est pas changée par l'appel de cette méthode.
- une méthode constante peut être appelée sur un objet constant.

Déclaration : mettre `const` après le prototype de méthode.

Exemple :

```
struct Vector {
    int   n;
    float *v;
    ...
    float get(int i) const {
        return v[i];
    };
};
```

8 Droits d'accès

Inconvénient d'une structure : la boîte est ouverte.

A savoir, n'importe quel bout de code qui a accès à la structure (`struct`) peut :

- aller lire n'importe quel champ.
- et si l'accès est autrement qu'en `const`, écrire n'importe quelle valeur.

Autrement dit, la structure peut :

- être modifiée d'une façon qui la rende incohérente
  exemple : pour un `struct Vector { int n; float *v; }`, mettre une valeur de `n` plus grande sans modifier `v`.
- retourner des valeurs invalides
  exemple : demander l'accès à `v[i]`, pour `i` < 0 ou `i` $\geq n$.
- permettre l'accès à des champs "techniques", qui n'intéressent pas l'utilisateur de la structure.
  exemple : la valeur du pointeur `v` est sans importance.
- même idée pour les méthodes : certaines méthodes "techniques" ne doivent pouvoir être appelées que par les concepteurs de la structure.

8.1 Définition

On définit :

- une **classe** est une structure disposant d'un contrôle d'accès sur ses champs et ses méthodes.
- un **objet** est une instance d'une classe (= une réalisation).

Une classe est une structure :

- déclarée avec le mot-clé `class`
- et dont l'accès aux champs et aux méthodes peut être :
  - ◇ soit **privé** :
    - — un champs privé ne peut être utilisé que dans les méthodes de la classe.
    - — une méthode privée ne peut être appelée que depuis une méthode la classe.
  - ◇ soit **public** :
    - — un champs public peut être lu et modifié .
    - — une méthode publique peut être appelée sans restriction.

Remarque : ce comportement peut encore être restreint à la lecture seule avec le mot-clé `const` (voir plus loin).
On utilise le mot-clé :

`public:` les champs et les méthodes qui suivent sont publiques.

`private:` les champs et les méthodes qui suivent sont privées.

Dans une classe, sans spécification (par défaut), les champs et les méthodes sont privés.

Exemple:

```
class Vector {
private:
    int   n;
    float *v;
public:
    Vector(int N) : n(N), v(new float[n]) {};
    ~Vector() { delete [] v };
    float get(int i) { return v[i]; };
    void set(int i, float s) { v[i]=s; };
};
```

Remarque : les vérifications des droits d'accès aux champs ont lieu à la compilation.

8.2 Règles de construction

Règle de construction d'une classe :

Les champs de la classe ne peuvent être modifiés depuis l'extérieur de classe que par une méthode de la classe.

Ceci permet de faire en sorte qu'un champs ne soit modifié qu'en accord avec les règles de la classe.

Conséquences sur la conception d'une classe :

- tous les champs de la classe sont **privés**.
- les constructeurs et le destructeur sont **publics**.
- un champ est toujours lu par l'intermédiaire d'une méthode **publique** (nommée un **getter**, généralement `inline`).
- un champ est toujours écrit par l'intermédiaire d'une méthode **publique** (nommée un **setter**, généralement `inline`).
- toute méthode de classe qui modifie l'objet le laisse dans un état cohérent.

Voir l'exemple précédent pour lequel il faudrait en plus ajouter des `assert`s afin de faire en sorte que $0 \leq$ `i` $<$ `n`.

8.3 Relation d'amitié

Inconvénient de la protection privée : lorsqu'une fonction ou une autre classe ont besoin d'un accès direct à la classe, les champs et les méthodes privés deviennent des barrages.

D'où la nécessité d'avoir la possibilité de lever les restrictions d'accès.

Ceci s'effectue avec le mot-clé `friend` suivi du nom de la classe ou du prototype de la fonction à laquelle on ouvre l'accès à la classe :

Exemple:

```
class Complex {
  private: float r,i;
  public: Complex(float u, float v) : r(u), i(v) {};
          friend Complex operator+(Complex a, Complex b);
          friend class Polar;
};
Complex operator+(Complex a, Complex b) {
  return Complex(a.r+b.r,a.i+b.i);
};
class Polar { ... friend class Complex; };
```

Remarques :

- tout champs ou méthode privés devient directement accessibles à toutes les fonctions et classes amies.
- l'accès par une classe amie est complet.
- les surcharges d'opérateur sont généralement déclarés en `friend` (permet de voir les opérateurs qui sont définis pour cette classe à la lecture de la classe).
- une pré-déclaration peut être nécessaire
  - ◇ pour une classe : `class A;` (= une classe de nom `A` existe).
  - ◇ pour une fonction : son prototype.

Exemple:

```
// prédéclaration
class Polar;
class Complex;
Complex operator+(Complex a, Complex b);
// déclaration de la classe
class Complex {
  private: float r,i;
  public: Complex(float u, float v) : r(u), i(v) {};
          friend Complex operator+(Complex a, Complex b);
          friend class Polar;
};
```

8.4 Différence entre structure et classe

Différences entre classe et structure :

- il n'y a que les conditions d'accès `public`/`private`.
  un structure est une classe dont tous les champs et toutes les méthodes seraient publiques.
- tout le reste fonctionne de manière exactement identique.

Nous verrons dans la leçon suivante des notions plus avancées :

- assemblage d'objets : l'héritage (construire un objet par dérivation) et agrégation.
  exemple : un disque (`Disc`) ou un carré (`Square`) est un objet géométrique (dérivent de `Geom2D`)
- la virtualité = l'implémentation d'une méthode d'une classe dérivée dépend de la classe à laquelle il est intégré.
  exemple : la méthode `Surface` de `Geom2D` n'est pas implémentée de façon identique pour `Disc` ou `Square`.
- le polymorphisme = toute classe qui dérive d'une autre classe peut être également vu comme un objet cohérent issu d'une classe dont il dérive.
  exemple : un tableau de `Geom2D` peut contenir des `Disc`s ou des `Square`s.

8.5 Imbrications

Une définition de classe/structure peut inclure des définitions de nouveaux types (y compris classe/-structure).

En conséquence, une classe/structure "technique" (à savoir qui n'a d'intérêt que pour la classe courante) peut être définie dans la section privée de la classe, rendant ainsi l'implémentation interne

invisible depuis l'extérieur.

Exemple:

```
class ChainList {
private:
   // structure pour l'implémentation interne
   struct List {
      int  n;
      List *nxt;
      inline List() : n(0),nxt(nullptr);
      ...
   }
   List *head, *current;
public:
   ...
   inline void rewind() { current=head; };
   inline int next() { current=current->nxt; };
}
```

9 Coût d'utilisation

Par rapport à une structure C classique, une structure étendue :

- consomme exactement la même quantité de mémoire.
- accès aux champs de manière identique (par décalage dans la mémoire).
- une méthode et une fonction ont exactement le même coût d'appel.
- la présence de constructeurs et de destructeur peut considérablement réduire la performance de l'application si les appels sont nombreux et inutiles.
- la présence de constructeur à un argument peut provoquer des conversions implicites nombreuses et inattendues, et potentiellement coûteuse.
- l'utilisation d'opérateurs peut générer de nombreux objets intermédiaires.

Sinon, il n'y a aucune autre différence significative :

- Les méthodes spécifiques à la structure sont encapsulées dans la structure.
- Les erreurs courantes sont les mêmes que pour les structures classiques.

10 Espace de nommage

Motivation :

Les classes sont des outils intéressants pour encapsuler les éléments dont la classe a besoin pour être définie.

Il est d'ailleurs possible de définir une classe (privée) dans une autre classe, qui ne sera utilisée et définie que dans cette classe.

Mais que faire maintenant si l'on souhaite encapsuler n'importe quel type d'objet ?

La solution est d'utiliser les espaces de nommage.

Un espace denommage est un bloc nommé dans lequel tout type d'objet peut être placé (constantes, classes, structures, fonctions, variable globale, ...).

On accède à un objet dans un espace de nommage grâce à son nom et l'opérateur de résolution de portée.

10.1 Définition

Définition d'un espace de nommage nommé `Nom` :

```
namespace Nom {
   /* contenu du namespace */
};
```

Remarques :

- Le contenu de l'espace de nommage est utilisable à partir du moment où il est défini.
- un `namespace` est généralement défini dans un `.h`.
- les fonctions ou méthodes d'un `namespace` peuvent être déclarés dans un `.ccp` (de manière similaire à une classe).
  Utiliser l'opérateur de résolution de portée pour les définir.
- un `namespace` peut être défini par morceau (à savoir, par l'assemblage de plusieurs morceaux différents, chacun étant placé dans un `.h` différent).

Exemple :

```
// dans le Nom.h
namespace Nom {
   const int aMax=3;

   struct B {
      int a;
      void Calc(int);
   };

   class C {
   public:
      C(B);
      ~C();
      friend int fun(C);
   };

   int fun(C);
};
```

```
// dans le Nom.cpp
#include "Nom.h"

void Nom::B::Calc(int n) {
   ...
}

void Nom::C::C(Nom::B x) {
   ...
}

void Nom::C::~C() {
   ...
}

int Nom::fun(Nom::C a) {
   ...
}
```

10.2 Accès ùn espace de nommage

Méthode 1 : accès avec l'opérateur de résolution de portée `::`

Si un objet se nomme `B` dans l'espace de résolution `A`, alors son nom est `A::B`.

Exemple :

```
#include "Nom.h"
...
{
   Nom::B  b = { Nom::aMax };
   Nom::C  c(b);
   int n = Nom::fun(c);
   ...
}
```

Méthode 2 : avec `using namespace`

- `using namespace Nom` rend implicite la résolution de portée `Nom`.
  Le code précédent devient :

```
#include "Nom.h"
...
{
   using namespace Nom;
   B  b = { aMax };
   C  c(b);
   int n = fun(c);
   ...
}
```

- La portée d'un `using namespace` est celle du bloc dans lequel il est défini.
  - ◇ s'il est défini au début d'un bloc, le `using namespace` n'est défini que pour le bloc.
  - ◇ s'il est défini en dehors de tout bloc, il est valide pour tous les blocs qui suivent.
- un `using namespace` occulte et remplace le précédent (sur la portée où il est défini).

10.3 Imbrications

Un namespace peut contenir de très nombreuses définitions de classes, fonctions, ...

Il est possible de :

- déclarer un `namespace` dans un `namespace`,
  Permet une structuration en unités sémantiques plus fine du `namespace`.
- (C_{11}^{++}) déclarer un `inline namespace` dans un `namespace`,
  Dans ce cas, l'ensemble des déclarations du namespace inline est directement utilisable dans le namespace dans lequel il est inclus.
  Permet de structurer un namespace en bloc sans imposer une hiérarchie trop contraignante à l'utilisateur.

Exemple:

```
namespace A { ...
  namespace B { ...
    inline namespace C { ...
       int fun(int); ...
    }
    // fun aussi défini ici
  }
}
```

Conséquence : utiliser ces règles afin d'éviter qu'un namespace ne se transforme en un gigantesque

fourre-tout. Tout code doit clairement faire apparaître sa structuration.

Accès généraux dans les namespaces :

- l'accès à un objet se fait avec l'opérateur de résolution de portée (exemple : `A::B::C::fun(x)`).
- Lorsqu'il est souhaitable de ne pas intégrer la totalité du `namespace` dans le bloc courant (fonction, namespace), il est possible d'intégrer un objet particulier d'un `namespace` à partir de son nom avec `using`.

```
{  using A::B::C::fun;
   // seul fun est accessible
}
```

Attention :
  - ◇ l'ajout s'effectuant par nom, s'il s'agit d'une fonction, toutes les surcharges seront également ajoutées.
  - ◇ aucune définition d'un objet local ne doit venir interféré avec cette déclaration.
- possibilité de déclarer un alias (ex : `namespace myABC = A::B::C;`, `myABC::fun(x)` est une résolution équivalente à `A::B::C::fun(x)`) afin de simplifier les accès à l'un des namespaces de la structure.

10.4 Modules

Les namespaces permettent aussi de structurer l'application en modules :

- un module = un `namespace`.
- le nom du module est celui du `namespace`.
- l'accès aux membres du namespace peut rester simple grâce au `using namespace` ou aux alias.
- prenez l'habitude de placer vos propres objets dans un `namespace`.

Remarques :

- les objets de la bibliothèque standard sont tous inclus dans le `namespace std`.
- la bibliothèque `boost` contient de nombreux namespaces. Par exemple : `boost::geometry` est le namespace contenant les objets permettant de gérer de la géométrie.
  A noter que `boost` est (avec la bibliothèque standard) l'autre bibliothèque de référence du C++.
  Ses bibliothèques sont actuellement progressivement intégrées dans le standard.
  Son usage est conseillé.

10.5 Modules multiples

Attention, lors de l'utilisation consécutive de la directive `using namespace` sur plusieurs namespace différent dans un même contexte, alors :

- les définitions des objets des différents namespaces se cumulent naturellement dans le contexte ;
- ceci s'effectue indépendamment de l'ordre et de la localité de la déclaration (*i.e.* les objets du dernier `using namespace` n'occultent pas les objets des précédents) ;
- deux objets de même nom déclarés dans deux namespaces différents ne provoquent pas d'ambiguïtés sauf si ce nom est utilisé
  car alors, le compilateur se demande lequel des deux objets choisir.

Donc, tant que la question du choix ne se pose pas, le compilateur ne vérifie pas les ambiguïtés possibles entre namespaces.

Exemple :

```
namespace A { int a=1, b=2; }
namespace B { int a=2; }
```

```
// erreur d'ambiguité ici
using namespace A;
int main() {
    using namespace B;
    int v = a+2;
}
```

```
// mais pas ici
using namespace A;
int main() {
    using namespace B;
    int v = b+2;
}
```

En conséquence, pour limiter les conflits de nom :

- tout usage ponctuel devrait conduire à utiliser l'opérateur de résolution de portée.
  - ◇ Il a l'avantage de rendre l'origine de l'objet utilisé clair.
  - ◇ une résolution de portée n'induit jamais d'ambiguïté.
- sinon, toujours faire en sorte que l'impact de la la directive `using namespace` soit limité
  - ◇ si possible à la portée d'un bloc (section de code, fonction, autre namespace).
  - ◇ en particulier, jamais dans les fichiers d'en-tête ;
  - ◇ elle ne devrait être utilisée en portée globale que de manière très parcimonieuse (limitée aux unités de traduction qui utilisent intensivement un namespace).

10.6 `namespace` anonyme

L'intérêt d'un `namespace` anonyme (sans nom) est de cloisonner les déclarations qui y sont contenu dans l'unité de traduction dans laquelle il est déclaré en les rendant inaccessible depuis l'extérieur de cet environnement.

Utilisations : déclarer des variables globales dans une unité de traduction inaccessible depuis toute autre unité.

Ceci n'était pas possible dans les faits avec les variables globales (y compris déclarées en `static`) qui restaient accessibles depuis un autre module à partir du moment où celui-ci les déclaraient comme `extern`.

Exemple :

```
// unitA.cpp
namespace { // anonyme
    int a = 4;
} // a non static!
// a accessible ici
```

```
// unitB.cpp

// aucun moyen de
// faire référence
// à a ici.
```

Important : nouvelle façon de déclarer des variables globales locales à une unité de traduction.

Conclusion

Dans cette leçon, nous avons vu :

- comment les données sont stockées physiquement dans une structure,

- comment le C++ permet d'étendre les structures du C en y encapsulant des fonctions propres en plus des membres,
- les méthodes automatiques qui permettent de faciliter la construction, la copie et la destruction de la structure,
- les différentes méthodes pour initialiser ou construire les structures,
- comment définir des opérateurs pouvant s'appliquer aux structures,
- la signification de membres constants ou statiques,
- comment définir des classes en limitant l'accès aux membres et méthodes d'une structure,
- comment définir des espaces de nommages, qui sont des boites nommées dans lesquelles on peut placer tout type d'objets ou définition (structure, classe,

Cette leçon est complétée par une annexe sur le déplacement présent dans votre polycopié.

11 Annexe : complément sur le déplacement

11.1 Introduction au déplacement

Soit deux objets `src` et `dst`. On souhaite propager la valeur de l'objet `src` à l'objet `dst`.

Il y a deux façons d'effectuer cette propagation :

- **par copie :** propage la valeur de `src` vers `dst` sans modifier `src`.
- **par déplacement :** propage la valeur de `src` vers `dst` en modifiant possiblement `src`.

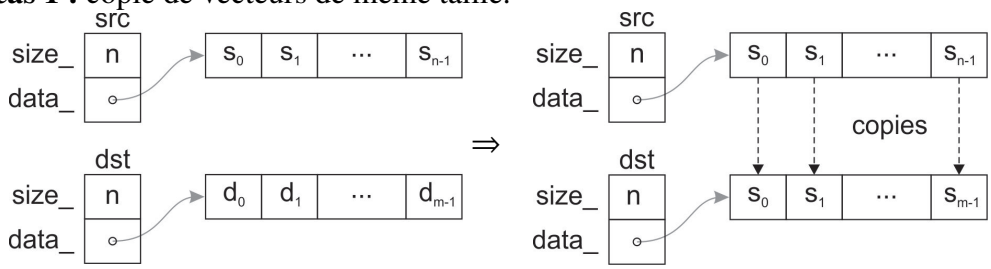
Exemple:

```
class Vector {
private:
  int    size_;  // taille du vecteur
  double *data_; // pointeur vers les données du vecteur
public: ...
};
```

Comment dans ce cas s'implémente la copie et le déplacement ?

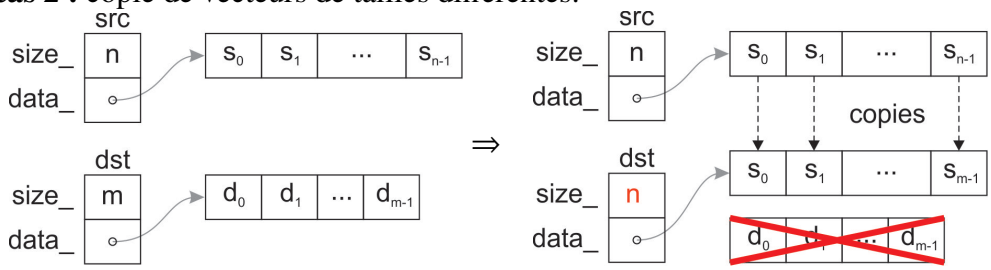
Exemple : (suite)

- **cas 1 :** copie de vecteurs de même taille.



La copie seule des données suffit.

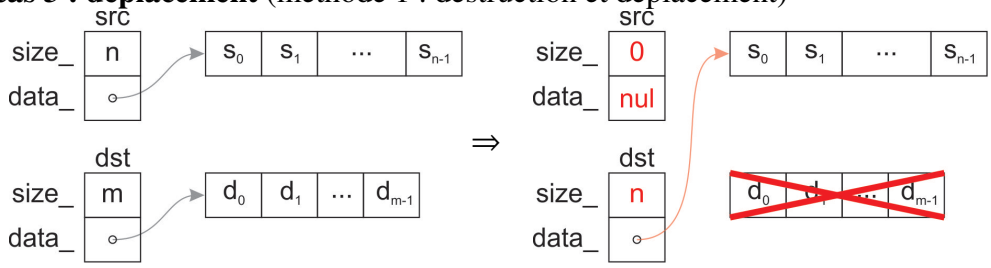
- **cas 2 :** copie de vecteurs de tailles différentes.



Dans ce cas, il faut pour `dst` :

◇ déallouer le tableau des données,
◇ allouer un nouveau tableau de taille `src.size_`,
◇ mettre à `dst.size_` avec `src.size_`

- **cas 3 : déplacement** (méthode 1 : destruction et déplacement)

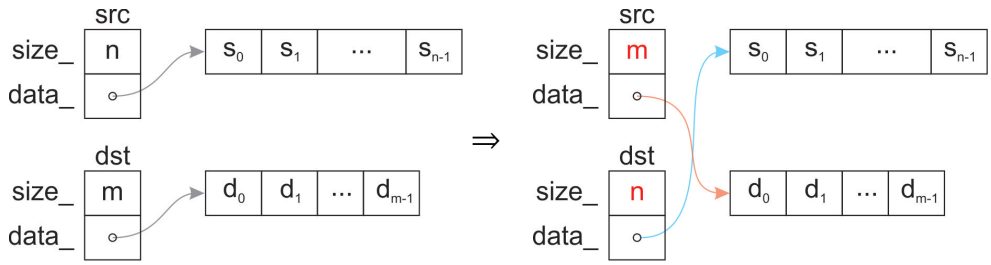


Il faut donc :

◇ déallouer le tableau des données de `dst`,
◇ copier `src` dans `dst`
◇ mettre `src` à 0 (*i.e.* `(size_,data_) = (0,nullptr)`)

Dans ce cas, le vecteur `src` est vide après le déplacement.

- **cas 3 : déplacement** (méthode 2 : permutation)



Il faut permuter `src` et `dst`, c'est-à-dire :

- ◇ permuter `src.size_` et `dst.size_`
- ◇ permuter `src.data_` et `dst.data_`

Dans ce cas, le vecteur `src` contient le vecteur `dst` (et inversement).

L'avantage de cette méthode, et que, si on manipule en général des vecteurs de même taille, la place mémoire peut être facilement recyclée.

La désallocation des anciennes données de `dst` est donc reportée à la libération de `src` (=déléguée au destructeur).

Conséquences :

- En général, un déplacement est plus efficace qu'une copie (souvent de beaucoup, dès qu'une partie des données est stockée à l'extérieur de l'objet).
- Si le code qui suit le déplacement ne dépend pas de la valeur de l'objet source, une copie peut être remplacée de manière sûre par un déplacement.

Comment fournir un moyen au langage tel qu'un déplacement soit effectué automatiquement chaque fois que le contexte garantit que ce déplacement sera sûr ?

Il serait également souhaitable de pouvoir ignorer le comportement par défaut et de forcer un déplacement dans les cas où la sureté du déplacement est garantie par le programmeur (du fait de sa connaissance du comportement du programme) et non par le contexte.

Les références sur les `rvalue`s sont le moyen de fournir ce mécanisme.

11.2 Typologie des valeurs

a) Terminologie

qualification cv (cv-qualified)

- un type qui utilise le qualificateur `const` et/ou `volatile` est qualifié cv.
- **Exemples :** `const int`, `volatile bool` sont qualifiés cv.

objet nommé (named/unnamed)

- un objet ou une fonction dont le nom est donné par un identificateur est dit nommé.
- un objet ou une fonction auquel on ne peut pas faire référence par un nom est dit anonyme (ou non nommé).
- **Exemple :**

```
Stack p;     // p nommé
p = Stack(10);  // Stack(10) non nommé
```

b) Typologie des valeurs

La sémantique de déplacement fournit un moyen de déplacer le contenu d'un objet entre objet, plutôt que de le copier.

A savoir, si la sémantique de déplacement est définie sur un objet, alors :

- le retour d'un objet par valeur déplace la valeur renvoyée dans l'objet accueillant le résultat (plutôt que de le copier).
- à l'appel d'une méthode, le passage en paramètre d'un objet temporaire peut le déplacer.
- l'appel explicite à la sémantique move permet de déplacer de déplacer un objet.

Exemple:
```
// construction par déplacement de l'objet retourné
Object v = MakeObject(1);
ObjectList L;
// déplacement de l'objet temporaire dans la liste
L.addObject( MakeObject(3) );
// déplacement de l'objet construit dans la liste
L.addObject( std::move(v) );
// ici, v est vide
```

Toute expression C++ peut être caractérisée par deux propriétés indépendantes : son type et la catégorie de sa valeur.

Les valeurs sont catégorisées suivant leurs propriétés de :

- **identifiabilité** : s'il est possible de déterminer si une expression fait référence à la même entité d'une autre expression (tel qu'en comparant les adresses des objets ou des fonctions qu'elles identifient, de manière directe ou indirecte).
- **déplaçabilité** : s'il est possible de déplacer la valeur dans le contexte (*i.e.* si le [constructeur par déplacement | assignation par déplacement | surcharge de fonction qui implémente la sémantique de déplacement] peut être attaché à l'expression).

On définit alors deux catégories primaires :

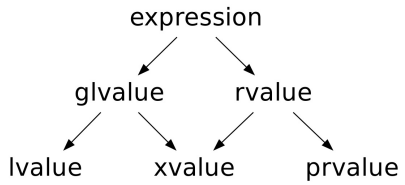
- une `glvalue` est une expression `identifiable` (=généralized lvalue ; auparavant nommée `lvalue`).
- une `rvalue` est une expression **déplaçable**.

Ces deux catégories primaires peuvent être à leur tour subdivisées en trois catégories :

- `lvalue` = expression identifiable mais non déplaçable.
- `xvalue` = expression identifiable et déplaçable
- `prvalue` = expression non identifiable mais déplaçable

Note : dernière catégorie possible non utilisée (ni identifiable et ni déplaçable).

Ces catégories se hiérarchisent donc de la manière suivante :



Note : la nouvelle définition de la `lvalue` ne se superpose pas avec l'ancienne (voir exemple plus loin).

c) Applications avec C++ ≤ C^{++}_{11}

Rappels : BIT = build-in type = `int`, `float`, `bool`, `char`, ...

Exemple:
```
int a = 5;          // a=I/ND=lvalue, 5=NI/D=prvalue
int &b = a;         // b=I/ND=lvalue, a=I/ND=lvalue
int c = (a+4)/2;    // c=I/ND=lvalue, (a+4)/2=NI/D=prvalue
// int &d = 5;      // erreur: 5 NI (non adressable)
// int &e = a/2;    // erreur: a/2 NI
```

Rappel du sens de &b :

- &b fait référence à un entier (= est un autre nom pour).
- adresse de b = adresse de a.
- une référence est en fait une référence à une `lvalue`.

Noter que :
```
const int &d = 5;       // valide
const int &e = (a+4)/2; // valide
```

car `const int&` signifie référence vers un entier constant (et non référence constante vers un entier).

Donc, une référence à une constante est le seul outil dont nous disposons qui permet de faire référence à un objet temporaire. Malheureusement, elle n'est pas déplaçable car constante.

Nous avons donc besoin d'un autre outil.

d) Référence à une `rvalue` ($\geq$ C++11)

Nouveau modificateur de type : &&

T&& = référence à une `rvalue` de type T.

Une référence à une `rvalue` fait référence au résultat d'une expression temporaire dont l'assignation ne persiste pas au-delà de l'expression qui la définit (une variable de ce type est une lvalue).

C'est un autre nom pour la place temporaire qu'occupera le résultat résultant de l'évaluation de l'expression.

Exemple:
```
int   funPR(), &funL(), &&funX(); // déclaration fonction
int a = funPR();  // funPR()=NI/D=prvalue, a=I/ND=lvalue
int &b = funL();  // funL()=I/ND=lvalue,   b=I/ND=lvalue
int &&c = funX(); // funX()=I/D=xvalue,    c=I/ND=lvalue
```

Une variable de type T&& a donc pour vocation d'être la référence à une variable de type T qui peut être déplacée.

11.3 Propriété des valeurs

a) `glvalue`

- peut être implicitement convertie en une `prvalue` (conversion implicite `lvalue`→`rvalue`, `tableau`→`pointeur`, `fonction`→`pointeur`).
- peut être polymorphique (le type dynamique de l'objet n'est pas nécessairement le type de l'expression),
- peut avoir un type incomplet.

b) `rvalue`

- l'adresse d'une `rvalue` ne peut pas être prise (pas identifiable),
- ne peut pas être utilisé comme opérande de gauche dans une assignation (même raison),
- peut être utilisée pour initialiser une référence à une `rvalue` ou à une référence constante à une `lvalue` (la durée de vie de la rvalue utilisée pour l'initialisation est alors prolongée jusqu'à la fin de portée de la référence).
- (pour plus tard) si une fonction `F` a deux surcharges `F(T&&)` et `F(const T&)`, alors l'appel `F(rvalue)` appelle `F(T&&)`.

c) `lvalue`

Ce nom a été gardé pour des raisons historiques mais le sens ne recoupe pas l'ancien sens (à savoir, expression à gauche d'un =).

Rappel définition : expression identifiable et non déplaçable.

Comment reconnaître une `lvalue` ? Si elle peut être utilisée (ou est utilisée) pour stocker une valeur. Donc, une expression est une `lvalue` si :

- il est possible de prendre l'adresse de l'expression,
- c'est une référence sur une `lvalue`.

Exemples de `lvalue` :

- tout nom de variable ou de fonction dans la portée est une lvalue (y compris si son type est référence à une `rvalue`, cf plus tard).
- les chaines de caractères littérales (exemple : `"Toto"`),
- si `p` est un pointeur, `*p` est une `lvalue`, `p[n]` est une `lvalue`,
- `a.m` ou `p->m` où `m` est un membre mais pas un membre énuméré (exemple : `v` dans `struct S { enum{ v = 0 } };`) ou une méthode.
- les assignations (exemple : `a=b` dans `a=b=c`),
- un appel de fonction (surcharge d'opérateur, cast, incrémentation préfixes de BITS inclus) qui retourne un type qui est une référence à une lvalue (exemple : `static_cast<int&>(x)`).
- un appel de fonction (surcharge d'opérateur, cast inclus) qui retourne un type fonctionnel dont le type de retour est une référence sur une rvalue (exemple : `static_cast<void (&&)(int)>(x)`).
- un cast vers un type qui est une référence à une lvalue ou une rvalue.

Exemples : `lvalue`

```
// définitions
int fun();
const int i = 1;
int j = 0;
char buffer[] = "Hello"; // "Hello" lvalue
char * s = buffer;
char &Car(char *v, int i) { return v[i]; }
int &&k = j+3;
```

```
// objets et fonctions nommés
j = i+1;          // i,j lvalue
fun();            // fun lvalue (pas fun())
// pointeur déréférencé
*s = 'a';         // *s lvalue
*(s+1) = 'b';     // *(s+1) lvalue
// retour d'une reférence
j=Car(s,3);       // Car(s,3) lvalue
// référence à rvalue nommée
j=k;              // k lvalue
```

Donc **attention**, avec cette définition, une `lvalue` peut apparaître à droite d'un =.

d) `prvalue`

`prvalue` pour "pure rvalue", fait référence à un objet, en général proche de sa fin de vie (tel que sa ressource peut être déplacé) et est le résultat de certains types d'expression impliquant la référence à une `rvalue`.

Rappel définition : expression non identifiable mais déplaçable.

Exemples de `prvalue` :

- tout littéral (32, `false`, `nullptr`, ...) sauf les chaînes de caractères,
- un appel de fonction (surcharge d'opérateur inclus) qui retourne un type qui n'est pas une référence (exemple : `sin(x)`).
- toute expression arithmétique ou logique utilisant des types et opérateurs définis par défaut (exemple : `a+b` ou `u!=4` sur des BITs).
- l'incrémentation ou la décrémentation postfixe,
- `a.m` ou `p->m` où `m` est un membre statique énuméré ou une méthode non statique.
- `this`,
- un cast vers un type qui n'est pas une référence,
- une lambda-expression

Le type d'une `prvalue` est toujours celui de l'expression (donc n'utilise pas le polymorphisme, et son type ne peut pas être incomplet).

Une `prvalue` ne peut pas être qualifié cv.

Exemples : `prvalue`

```
// définitions
Stack    s;
int incr(int k) { return k++};
```

```
// objets temporaires
s = Stack(10);    // Stack(10) prvalue
// littéraux
int i = 42;       // 42 prvalue
// fonction ne retournant pas de référence
i = incr(i);      // incr(i) prvalue
// résultat d'expression arithmétique
i = 4*i+8;        // 4*i+8 prvalue
// résultat d'expression logique
bool v = (i==5); // (i==5) prvalue
// incrémentation postfixe
// rappel: renvoie la valeur incrémentée
i++;              // i++ prvalue
```

e) `xvalue`

`xvalue` pour "expiring value", fait référence à un objet, en général proche de sa fin de vie (tel que sa ressource peut être déplacé) et est le résultat de certains types d'expression impliquant la référence à une `rvalue`.

Rappel définition : expression identifiable et déplaçable.

Exemples de `xvalue` :

- un appel de fonction (surcharge d'opérateur inclus) qui retourne une référence à une `rvalue` (exemple : `T&& move(T&&)`).
- un cast vers un type qui est une référence vers une `rvalue`.
- `a[n]` où `a` est un tableau `rvalue`,
- `a.m` où `a` est un objet `rvalue`, et `m` un membre non statique qui n'est pas une référence,

une `xvalue` correspond donc :

- soit à une `lvalue` convertie explicitement en une référence à une `rvalue` avec une fonction appropriée (typiquement `std::move`), façon d'indiquer explicitement qu'elle est en fin de vie.
- soit à un objet qui fait référence à une `rvalue`

Une `xvalue` peut être polymorphique et être qualifié cv.

Avec ces catégories, un déplacement sur :

- une `lvalue` n'a pas la garantie d'être sûre, puisque le code peut accéder à l'objet associé à travers son nom, un pointeur ou une référence.
  donc, une `lvalue` ne doit jamais être automatiquement déplacée.
- une `prvalue` a toujours la garantie d'être sûre (puisqu'une prvalue n'a pas d'identité).
- une `xvalue` est considérée comme être sûre car elle est soit déjà une référence à une `rvalue`, soit a été explicitement convertie en une `xvalue` (et donc l'utilisateur considère que le déplacement est sûr).

Donc,

- si la source est une `rvalue` (une `prvalue` ou une `xvalue`), utiliser un déplacement à la place d'une copie est sûre,

- si la source est une `lvalue`, utiliser un déplacement à la place d'une copie n'a pas la garantie d'être sûre.

On veut donc que le langage utilise automatiquement le déplacement pour une `rvalue`, et la copie pour une `lvalue`.

11.4 Sémantique de déplacement

- une variable de type `T&&` (qualifiée cv ou non) est utilisée pour stocker une `rvalue` jusqu'à la fin de la portée de la variable,
- un paramètre de type `T&&` (qualifiée cv ou non) est utilisée pour indiquer que le paramètre est une `rvalue` et peut être déplacé au cours de l'appel de la fonction.

Le déplacement est typiquement implémenté de trois manières différentes :

- dans une classe T,
  - ◇ avec le constructeur par déplacement (typiquement `T(T&&)`) : permet de construire un objet en déplaçant une `rvalue`,
  - ◇ avec l'assignation par déplacement (typiquement `T& operator=(T&&)`) : permet d'assigner un objet en déplaçant une `rvalue`.
- explicitement, avec `T&& std::move(T&&)` (cette fonction indique seulement que l'objet passé en argument peut être déplacé, mais n'effectue pas le déplacement).
- mais également, dans toute fonction qui souhaite implémenter le déplacement de l'un de ses paramètres.

Exemple:

```
// on reprend l'exemple de la classe vector
class Vector {
private: size_t size_;  float *data_;
public: ...
    // constructeur par déplacement
    T( T &&v) :
        size_(v.size_),
        data_(v.data_) {
        v.size_ = 0;
        v.data_ = nullptr;
    }
    // assignation par déplacement
    T& operator=(T &&v) {
        std::swap<size_t>(size_,v.size_);
        std::swap<float*>(data_,v.data_);
        return *this;
    }
};
```

11.5 Élision de copie

a) Return value optimization

Définition : Return value optimization (RVO ou élision de copie) Technique d'optimisation du compilateur qui élimine la copie de l'objet local retournée **par valeur** par une fonction vers l'objet de la

fonction qui l'a appelée.

A savoir, le code suivant devrait produire une RVO :

```
A fun() { ... return A(k); }
void call() { ... A y=fun(); }
```

sans RVO :

- à la fin de la fonction `fun`, un objet temporaire est construit et retourné,
- la valeur du nouvel `y` est construit par copie à partir de l'objet temporaire, puis l'objet temporaire est détruit.

avec RVO :

- l'objet retourné doit être un temporaire fourni en argument du `return` (=construction de l'objet dans le return).
- le retour de la fonction est construit directement dans `y`.
- gain : une construction par copie + une destruction.

b) Named Return value optimization

Définition : Named Return value optimization (NRVO) Même idée que le RVO, mais améliorée de manière à ce que l'on puisse retourner le nom d'un objet local.

A savoir, le code suivant devrait produire une NRVO :

```
A fun() { ... A  a; ... return a; }
void call() { ... A y=fun(); }
```

sans NRVO : idem RVO/sans RVO.

avec NRVO :

- un objet local nommé est construit, éventuellement modifié, puis retourné par la fonction.
- l'objet local est construit directement à l'emplacement de l'endroit où il est retourné (*i.e.* y dans le cas ci-dessus).
- limitation : l'objet local ne doit pas être un paramètre ou volatile, paramètre d'un catch, et être de même type que le type de retour.

Remarque : si l'élision de copie (=RVO ou NRVO) n'est pas possible, le `return` essaye une construction par déplacement, et si elle n'est pas définie, une construction par copie.

c) Effet de bord de l'élision de copie

Attention : l'élision de copie est la seule forme d'optimisation autorisée par le standard qui peut avoir des effets de bord visibles :

- Certains compilateurs n'effectuent pas l'élision dans tous les cas où elle est possible et/ou autorisée (exemple : compilation en mode debug),
- Existence de cas où elle n'est pas effectuée (cf NRVO) + argument d'un throw/catch dépend de la version du C++ (oui si C++ $\geq$C++$_{11}$, non avant).
- Ne pas se baser sur cet effet de bord, et construire attentivement le constructeur par copie/déplacement (sinon code non portable).
  A savoir, faire en sorte que construction + copie = construction directe
- L'ignorance de cette règle relève de l'erreur de conception.

Exemple :

```
struct Foo {
public: int _a{}; // défaut=0
    Foo(int a) : _a{a} {}
    Foo(const Foo &) {}
};
```

```
Foo a{10};
Foo bar = 10; // équivalent = Foo{10}
// b._a = 0  // sans élision de copie
// b._a = 10 // avec élision de copie
```

d) Déplacement et élision de copie

Exemples : avec sémantique copie + déplacement et élision de copie

```
struct A { ...
 A();            // Cd
 A(int x);       // Ci
 A(const A& x); // Cc
 A(A&& x);       // Cm
 ~A();           // D
 A& operator=(const A&); // O=c
 A& operator=(A&&);      // O=m
};
A operator+(const A&, const A&); // O+
A fun(int);
// avec élision [E]
A fun(char);
// sans élision
```

```
A a1;         // Cd
A a2(2);      // Ci
A a3(a1);     // Cc
A a4 = a1;    // Cc
A a5(A(1));   // Ci
```

```
A f2();     // A f2(void)
A f1(A()); // A f1(A(*)(void))
A   a1(std::move(A())); // Cd|Cm|D
A   a2(a2 + a3);        // 0+=>Ci[E]
```

```
a1 = a2;    // O=c
a1 = A(2); // Ci|O=m|D
a1 = 1;     // Ci|O=m|D
a1 = a2 + a3;
    // O+=>Ci[E]|O=m|D
```

```
A a1 = fun(1);  // Ci[E]
A a2 = fun('a'); // Cd|Cm|D
a1 = fun(2);    // Ci|O=m|D
a2 = fun('b');  // Cd|Cm|D|Om=|D
```

11.6 Constructeur par déplacement et exceptions

Si une exception est lancée, le CPD ne sera pas en mesure de libérer la mémoire associé à l'objet temporaire passé en paramètre (puisqu'il est sensé être déplacé après l'appel à ce constructeur).

Conséquence : Un CPD ne doit pas lancer d'exception.

Pour l'assurer :

1. utiliser le mot-clef `noexcept` à la déclaration pour indiquer que le constructeur ne lance pas d'exception (i.e. `T(T&&) noexcept`),
2. utiliser `std::move_if_noexcept` à la place de `std::move`

   `move_if_noexcept<T>(x)` = `static_cast<T&&>(x)` si `T(T&&)` est `noexcept`

   et `static_cast<T&>(x)` sinon.

   Donc, le CPD est lancé s'il ne lance pas d'exception, et le CPC sinon.

Exemple :

```
struct T { ...
    T(T&& t) noexcept { ... }
    T(T& t) noexcept { ... }
};
```

```
struct U { ...
    U(U&& u) { ... }
    U(U& u) { ... }
};
```

```
T a, b = std::move_if_noexcept(a);      // lance le CPD
U c, d = std::move_if_noexcept(c);      // lance le CPC
```

11.7 Sémantique de déplacement

Comme nous l'avons déjà vu, si un paramètre d'une fonction est une référence sur une rvalue, alors il peut être déplacé (donc modifié).

Si le contexte permet de déterminer qu'il est possible de déplacer une `lvalue` lors de l'appel d'une fonction, alors il faut que :

- cette `lvalue` soit explicitement convertie en référence sur une `rvalue` lors du passage de l'argument : effectué avec la fonction `std::move`.
- la `rvalue` obtenue soit passée à une fonction dont une surcharge gère le déplacement (= accepte comme paramètre une référence à une rvalue).

Note : la fonction `std::move` ne fait rien d'autre que de convertir l'argument en une référence à une rvalue, le rendant ainsi candidat pour un déplacement, mais n'effectue pas le déplacement.

Exemple :

```
int fun(A &&x) { ... }
int fun(const A &x) { ... }
```

```
A    x;
int  y1 = fun(x);             // call fun(A&)
int  y2 = fun(std::move(x)); // call fun(A&&)
```

Application : template swap

Typiquement, l'implémentation typique de swap est la suivante :

```
template <class T> void swap(T &x, T &y) {
                   T tmp(x); x=y; y=temp; }
```

Noter que ce code nécessite :
- un constructeur par copie, et deux copies par assignation, donc particulièrement inefficace pour beaucoup de type `T`.
- le type `T` doit être copiable (i.e. CPC et APC existent).

Avec C_{11}^{++}, le code de swap est désormais :

```
template <class T> void swap(T &x, T &y) {
   T tmp(std::move(x)); // CPD si possible, CPC sinon
   x=std::move(y);      // APD si possible, APC sinon
   y=std::move(temp);   // APD si possible, APC sinon
}
```

Ce code convertit tous les arguments des constructeurs ou assignations en référence sur une `rvalue` avec `std::move` de façon à permettre aux fonctions appelées d'effectuer un déplacement s'il est implémenté, et ainsi éviter les copies.

Remarque : attention au retour par valeur
- **éviter de le retour par valeur constante :**
  Exemple : `const std::string getMsg() { return "Hello"; }`
  L'objet constant retourné ne peut pas être utilisé comme une source pour un déplacement (`const` donc non modifiable).
  ⇒ Mauvaise interaction avec la sémantique de déplacement.
- **ne jamais mettre un `std::move` lors d'un retour par valeur.**
  Exemple : `A getA() { return std::move(A()); }`
  Or, le type retourné par `move` est `A&&` et non `A` (donc différent).
  En conséquence, la RVO/NRVO ne peut pas être appliquée.

Conséquences :
- un paramètre qui doit être déplacé ne doit pas être déclaré comme constant, sinon il sera copié au lieu d'être déplacé,
- `std::move` ne déplace rien, et ne garantit pas que l'objet sera déplacé, ni qu'il sera éligible pour un déplacement.
- une fonction qui prend en paramètre une référence sur une `rvalue` ne garantit pas que l'objet passé sera déplacé.

11.8 Référence universelle

Cette section doit absolument être complétée par la lecture de la section 10.3 de la leçon 6. En effet, le recyclage des `rvalue`s sur tous les paramètres sur lesquels il peut être réalisé suppose une explosion combinatoire des codes à écrire pour gérer toutes les combinaisons de `lvalue`s et `rvalue`s sur ces paramètres.
Une solution pour générer ces codes consiste à utiliser les références universelles décrites dans la leçon sur les templates.