INFO0402 : Méthodes de programmation orientée objet

Encapsulation

Pascal Mignot

2015-2016

UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

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

Introduction

Nous allons dans cette leçon traiter des différents moyens d'encapsuler les données et les fonctions.

Il y a différents niveaux d'encapsulation :

- **encapsulation des données :** Type de Données Abstrait (structure en C)

  Exemple : vecteur = taille du vecteur + éléments du vecteur
- **encapsulation des données et fonctions :** structure étendue en C++

  TDA + fonctions de manipulation de la TDA
- **encapsulation des données et fonctions, avec contrôle d'accès :** classe en C++

  TDA + fonctions de manipulation de la TDA + limite d'accès aux éléments de la TDA.
- **encapsulation des TDAs, classes, fonctions :** module (namespace en C++).

Structure

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

Structure

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 ?

Contenu d'une structure

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.

Définition d'une structure

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 ce 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;
```

Représentation mémoire d'une structure

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.

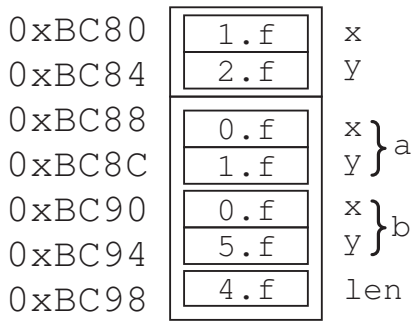
Exemple :

```
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 :



Règles d'alignement

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

Règles d'alignement :

- chaque champs de la structure est aligné.
- chaque structure est aligné sur une adresse multiple de la taille de son champ élémentaire le plus grand.
  y compris pour une structure dans une structure.
- le nombre de octets (blancs) nécessaire sont ajoutés 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.

Exemples d'alignement

Exemple 1 :

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

Exemple 2 :

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

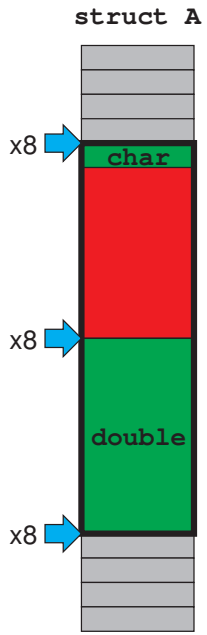
Exemples d'alignement

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; };
```

Exemples d'alignement

Exemple 1 :

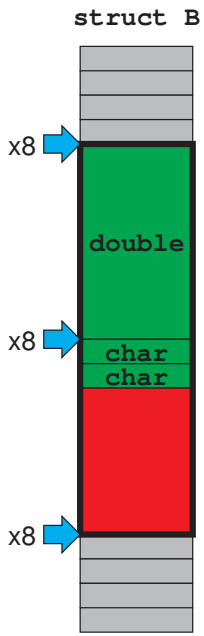
```
struct A { char a; double b; };
```

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



Exemples d'alignement

Exemple 3 :

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

Exemples d'alignement

Exemple 3 :

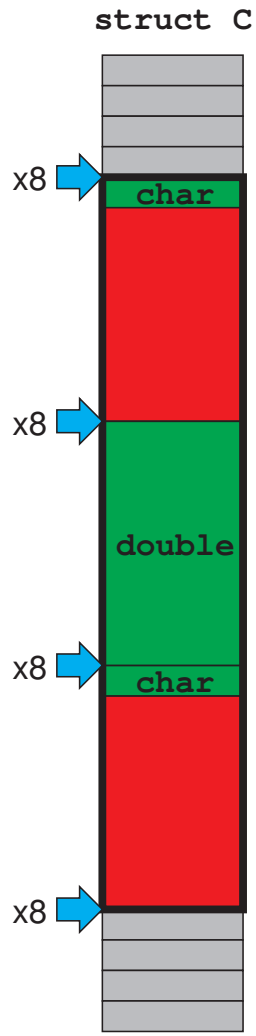
```
struct C { char a; double b; char c; };
```

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.



Règles d'alignement

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.

Opérateurs sur une structure : accès

Soit une structure `S` : `struct S { ... TYPEi NOMi; ... };`

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;
```

Opérateurs sur une structure : copie

Copie d'une structure :

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

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: écrire B=A copie le pointeur
```

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

  Exemple :
  ```
  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 =)

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

- initialisation d'une structure allouée dynamiquement

  Exemple :
  ```
  // en C
  Point2D *R=(Point2D*)malloc(sizeof(Point2D));
  *R = P;
  // en C++: allocation+initialisation avec P
  Point2D *R = new Point2D(P);
  ```

  Note : en C++, l'allocation et l'initialisation peut être faite en une seule instruction.

Appel de fonction

Lorsqu'une structure est :

- soit passée par valeur à une fonction

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

- soit retournée par une fonction

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

Alors, la structure est transmise par copie (=), i.e. la structure est copiée champs par champs.

Conséquence : potentiellement inefficace (autant de variable à copier que de champs dans la structure) 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.

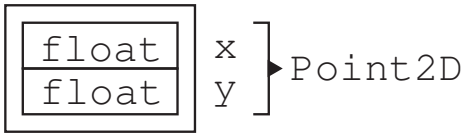
Conception des structures

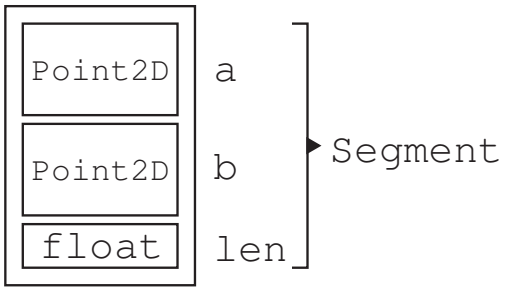
Les structures se construisent typiquement de trois manières (non exclusives) :

Façon 1 : Agrégation des données

grouper des types (élémentaires ou pas) qui forment ensemble un nouvel objet complexe.

Exemple : `Point2D`, `Segment`, ...






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

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

Conception des structures

Façon 2 : utilisation du contenu d'un tableau dynamique

La taille d'une structure étant fixe, l'ajout d'un tableau dynamique dans une structure s'effectue en ajoutant un pointeur dans la structure dont le rôle est de pointer vers le tableau.

Exemple : un vecteur, une matrice, ...

Cas du vecteur :


```
struct Vector {
    int   nb;
    float *vec;
};
```

On remarquera que les données du tableau sont stockée à l'extérieur de la structure, et doivent être allouées en plus de la structure.

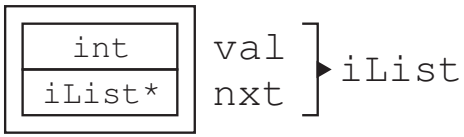
Il ne suffit donc pas de copier la structure pour dupliquer le vecteur (ce point sera revu plus tard).

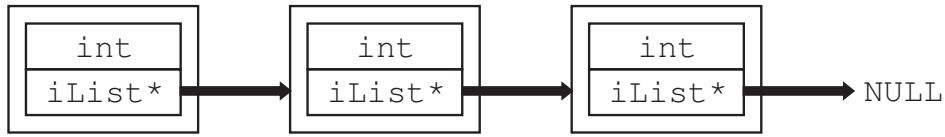
Conception des structures

Façon 3 : construire des liens entre agrégats (pointeur de la structure vers une autre structure)

Chaque structure représente un objet simple d'une structure plus complexe. Ce sont les liens entre les structures qui construisent la structure.

Exemple : une liste chainée (structure = un nœud, pointeur = lien entre éléments)






```
struct iList {
    int   val;
    iList *nxt;
};
```

Si `iList` représente un nœud de cette liste, la structure de données complète est représentée par l'ensemble des nœuds reliées entre eux par les pointeurs.

A noter que la définition de la structure est récursive (voir comment traiter ce point technique plus loin).

Conception des structures

L'objet symbolique représenté par la structure se décompose donc en trois composantes :

- une partie "statique" incluse dans la structure, et spécifique à la structure.
- une partie "dynamique" externe à la structure mais liée à elle (et uniquement à elle) par des pointeurs.
- une partie "méta-structure" à travers les liens de la structure avec d'autres structures dont le tout représente un objet symbolique plus vaste.

La cohérence de chaque instance de l'objet représenté par une structure doit être assurée par chaque fonction qui manipule l'instance.

D'autres exemples de structures seront vus en TD/TP.

Erreurs courantes

Attention aux structures qui contiennent des pointeurs

- les pointeurs d'une structure doivent toujours être initialisés (au moins à `NULL`).
- lors de la copie d'une telle structure, la copie avec l'opérateur = ne permet d'obtenir le résultat recherché.
- lors de la désallocation d'une structure, penser à libérer la partie dynamique de la structure (à savoir, la mémoire dynamique allouée utilisée seulement par la structure).

Exemple :

Pour la structure suivante :

```
struct Vector { int nb; float *vec; };
```

Soit la fonction d'initialisation :

```
Vector InitVector(int n) {
   Vector V = { n, new float[n] };
   memset(V.vec,0,n*sizeof(float));
   return V;
}
```

On considère maintenant le code suivant :

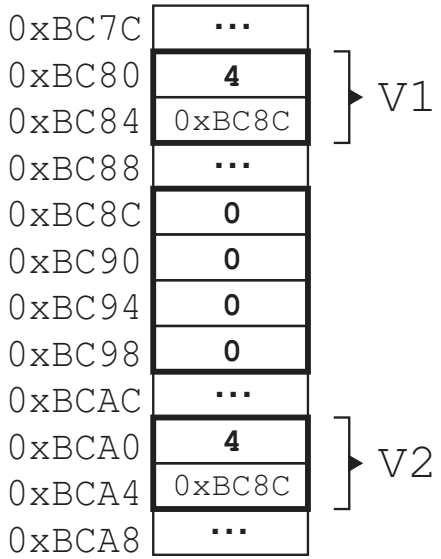
```
Vector V1 = InitVector(3);
Vector V2 = V1;
```

Pourquoi `V1` et `V2` ne sont pas des vecteurs indépendants ?

Erreurs courantes

Exemple : (suite)

Les vecteurs ne sont pas indépendants au sens suivant : les données du vecteur `V2` utilisent exactement la même table que le vecteur `V1`.



En conséquence, si le contenu du vecteur `V1` est modifié, alors le contenu du vecteur `V2` l'est aussi.

Raison : l'opérateur = ne copie que les champs, et non les données.

Il faudrait écrire une fonction de copie explicite :

```
Vector CopyVector(const Vector &Vi) {
   Vector Vo = {Vi.n, new float[Vi.n]};
   memcpy(Vo.vec,Vi.vec,Vi.n * sizeof(float));
   return Vo;
}
```

Erreurs courantes

Exemple : (suite)

Considérons le code suivant :

```
void fun1() {
    Vector V=InitVector(3);
    ...
    // fin de portée de V
}
```

Si rien n'est fait, la mémoire dynamique de la structure n'est pas libérée (et donc perdue).

Conséquence : elle doit être libérée explicitement. On propose à cet effet la fonction suivante.

```
void DeleteVector(Vector &V) {
    // libération partie dynamique du vector
    delete [] V.v;
}
```

Le code précédent doit donc être corrigé sous la forme :

```
void fun1() {
    Vector V=InitVector(3);
    ...
    DeleteVector(V);
}
```

Introduction

Une structure étendue (C++ seulement) est une structure dans laquelle la définition des fonctions s'appliquant spécifiquement à la structure est incluse dans le corps de la structure.

Des telles fonctions sont appelées des **méthodes**.

Parmi ces méthodes, certaines auront des appels automatiques :

- les constructeurs qui permettront d'indiquer comme une instance de la structure doit être initialisée.
- le constructeur par copie qui indique comment copier correctement une structure.
- le destructeur qui indique comment détruire une instance de la structure.
- les opérateurs qui permettent d'indiquer comment appliquer les opérations usuelles (affectation, opération arithmétique, comparaison, ...) sur la structure.

Remarque : A noter que l'encapsulation d'une méthode dans un structure ne modifie la taille de celle-ci (*i.e.* son `sizeof`). Attention, ce n'est pas le cas pour une classe donc au moins l'une des méthodes est virtuelle.

Méthodes

Les méthodes peuvent être définies de deux façons différentes :

- **Définition dans la structure**
  La méthode est incluse directement dans la définition de la structure : une méthode est une fonction dont le code est défini dans une structure). Ce type de déclaration est en général réservé aux fonctions `inline` de la structure.

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

- **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`)
  - le définition de la méthode est dans le code (dans le `.cpp`)

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

```
#include "Vector.h"

// définit Norm dans Vector
float Vector::Norm(void) {
    ...
}
```

Méthodes

Remarques sur la définition d'une méthode :

- 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 structure.
- Dans une méthode, les champs de la structure 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);
}
```

  Attention de 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 Warning sur certains compilateurs).
- Si `a` est une instance d'une structure `A`, dans laquelle est définie une méthode `fun(...)`, alors l'appel de la méthode `fun` sur `a` s'effectue avec : `a.fun(...)`.

```
Complex Z = {1.f,2.f};
float NormZ = Z.Norm();
```

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

Constructeurs

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.

- Deux constructeurs définis par défaut : le constructeur par défaut et le constructeur par copie. **Note :** trois en 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.

Constructeurs par défaut

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).
- il s'exécute en appelant le constructeur par défaut sur chacun de ses membres (non statiques).
- le code du constructeur par défaut d'une classe `A` est `A() {}`.

Le constructeur par défaut n'est automatiquement défini que si aucun autre constructeur n'est défini.

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).

Constructeurs

Exemple :

```
// Complex.h
struct Complex {
    // champs
    float r,i;
    // constructeur
    inline Complex() { r=i=0.f; };
    inline Complex(float v) { r=i=v; };
    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;       // Pz0 pointe sur 0+0i;
    Complex *Pz1 = new Complex(2.f); // Pz1 pointe sur 2+2i;
    Complex *PTz = new Complex[10];  // PTz pointe sur 10 Complex
                                     // initialisés avec 0+0i;
    ...
}
```

Copie

un objet peut être copié de deux façons :

- **par initialisation :** à savoir un objet est initialisé au moment où il est créé (i.e. associé au **constructeur par copie**).
- **par assignation :** on veut copier un objet dans un autre objet qui existe déjà (i.e. associé à l'**opérateur =**, voir la partie sur les opérateurs).

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 : 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.

Copie

Règles :

- toute structure qui contient un pointeur (y compris dans l'un de ses agrégats) devrait définir le constructeur et l'assignation par copie.
- 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 est donc potentiellement différente de l'assignation par copie.

Remarques :

- S'il n'est pas défini, le constructeur par copie (par défaut) recopie juste tous les champs de la structure. Idem pour la copie par assignation.
- Le prototype du constructeur par copie **doit être** `A(const A&)` (exception : cas de l'idiome copy-and-swap).

Constructeur par déplacement

En C^{++}_{11}, en raison de la possibilité de faire référence à une rvalue (*i.e.* à droite d'une assignation) 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 assignation operator) :** qui permet d'assigner le résultat de la rvalue directement dans un objet déjà existant.

  prototype : `T &operator=(T &&)`

Plus de détails seront donnés ultérieurement.

En conséquence, il est désormais d'implémenter les 5 opérations suivantes pour une classe `T` :

- le constructeur par copie `T(const T&)` et par déplacement `T(T &&)`,
- l'assignation par copie `T& operator=(const T&)` et par déplacement `T&operator=(T&&)`,
- le destructeur `~T()`.

Limites des constructeurs

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 structure. 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.

Limites des constructeurs

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);
    ...
}
```

Chaine d'initialisation

Le but des chaines d'initialisation est d'utiliser les constructeurs de champs d'une structure 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.
- Cette initialisation est potentiellement plus rapide car elle peut se faire en place (en particulier lorsque celle-ci est utilisée en argument de `return`).
- L'ordre dans laquelle est donnée la liste d'initialisation est sans importance : elle est toujours effectuée dans l'ordre de déclaration des champs dans la structure.

Destructeur

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

Remarques :

- le nom du destructeur est : `~A()` (par d'argument, pas de retour).
- le destructeur par défaut ne fait rien (à savoir `~A() {};`).
- il ne peut y avoir qu'un seul destructeur défini.
- si un destructeur est défini, il 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` sur une instance ou un tableau d'instances de la structure.

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()
```

Gestion mémoire non typée

Dans certains cas, on voudrait allouer un bloc mémoire brut, afin de l'utiliser pour gérer une mémoire locale et y stocker des données.

- En C, utiliser `malloc`/`free` et caster le résultat de l'allocation en `void*`.
- En C++, utiliser la version générique des opérateurs `new`/`delete` :
  `void* operator new(size_t count)` (aussi en version `new[]`)
  `void operator delete(void* ptr)` (aussi en version `delete[]`).

Exemple :

```
// allocation
void *data = (void *)malloc(nb_elts * elts_size);
// desallocation
free(mem);
```

```
// allocation
void *data  = ::operator new(nb_elts * elts_size);
// desallocation
::operator delete(data);
```

Notes :

- Utiliser `T* reinterpret_cast<const T*>(void*)` pour caster les emplacements mémoire `void` en type `T` souhaité.
- Les pointeurs renvoyés par ces fonctions fonctionnent pour tout alignement $\leq$ `alignof(std::max_align_t)`.
- Il est également possible d'utiliser les appels C dans un code C++.

Alignements spécifiques

L'approche est un peu différente si des alignements spécifiques sont requis (exemple : alignement SSE, ligne de cache ou page de la mémoire virtuelle) :

- en C, utiliser `void *aligned_alloc(size_t alignment,size_t size)` (libération avec `free`), où `size` est un multiple de `alignment`.
- en C^{++}_{11}, il est possible de définir un type qui respecte les règles d'alignement souhaité avec `std::aligned_storage`.

  Utiliser alors les fonctions d'allocation standard sur un objet de ce type.

Exemple :

```
// allocation
void *data = (void *)aligned_alloc(alignment,blocksize);
// desallocation standard avec free(mem)
```

```
// définition du type T avec l'alignement souhaité
typename std::aligned_storage<sizeof(T), alignof(T)>::type *data;
// allocation/desallocation standard avec ce type.
```

Note : La fonction `size_t alignof(Type)` permet de retourner l'alignement nécessité par un type `Type`.

Constructeurs et destructeur en place

Truc 2 : Allocation d'un objet sur un emplacement mémoire déjà alloué :
Utiliser :

- pour allouer un bloc de mémoire non typé, voir ci-avant.
- la construction d'un objet de type `Type` à l'adresse mémoire `prt` dans un bloc déjà alloué peut être effectué avec : `Type* new(ptr) Type(constructor args)`
- la destruction d'un objet sans libérer la mémoire occupée celui-ci est effectué par un appel explicite au destructeur.

Exemple :

```
// allocation: place pour 256 éléments de type A
// (aucun constructeur appelé)
A* mem = reinterpret_cast<A*>(::operator new(256*sizeof(A)));
// appel du constructeur A(x) sur le 16ème élément
A *a16 = new(&mem[15]) A(x);
...
// appel du destructeur sur ce même élément
a16->~A();
// il est possible de réutiliser cet emplacement
...
// l'emplacement mémoire reste alloué tant que l'on
// ne libère pas le bloc avec:
::operator delete(mem);
```

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 : [], *, &, ->, .

A savoir, si après la définition d'une structure `Vector`, on peut définir :

- l'opérateur == pour comparer deux vecteurs `a` et `b` (dont la définition signifie : même taille + même valeur pour chacune des composantes.
- l'opérateur + pour ajouter deux vecteurs, composante par composante.
- 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).

Voir la page wikipédia en anglais sur "Operators in C and C++ " pour voir les méthodes à définir afin de surcharger les opérateurs.

Opérateurs

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.

- 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.

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;
}
```

Opérateurs

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 (par exemple : pour une matrice `a`, `a *= a`).
- le type de chaque paramètre peut être différent.
- l'opération peut ne pas être symétrique (à savoir on peut faire en sorte que `a+b` donne un résultat différent de `b+a`).
- lorsque les types des paramètres sont différents, il faut définir deux opérateurs pour que l'opération soit symétrique.
  Exemple : `Vector operator*(Vector,float)` et `Vector operator*(float,Vector)`.
- En général, toujours préférer la fonction externe lorsqu'elle existe : le prototype est plus intuitif.
  sauf dans le cas polymorphique où la version interne peut être virtualisée.
- Lorsqu'un opérateur renvoie le constructeur d'une structure, le compilateur peut construire le résultat directement dans la structure résultat (cas du complexe `z` dans le transparent précédent).

Opérateurs

Les opérateurs sont pratiques :

- facilitent l'écriture d'expressions arithmétiques ou logiques.
- permettent de changer le sens de certains opérateurs.

mais ont un côté obscur :

- l'effet de 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`).
- on peut se mettre à écrire autre chose que ce que l'on croit écrire.

En conséquence, il est fortement déconseillé de les utiliser dans un premier temps car ils sont juste une facilité d'écriture.

Tant que vous ne serez pas au point en C++, vous risquez de passer plus de temps à déboguer une erreur liée à l'utilisation d'un opérateur qu'à écrire explicitement les choses.

En C_{11}^{++}, des optimisations utilisant le passage par référence à une rvalue peuvent être effectuée.

Membres et fonctions statiques

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.

Membres et fonctions statiques

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()) ...
}
```

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).

Constante de structure

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.

Constante de structure

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

cas 1 : Définition et initialisation dans la structure, déclaration à l'extérieur de la structure.
Approche limitée aux types simples.

```
// 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.

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

// stack.cpp
const int Stack::Size = 32;
```

Méthode constante

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];
    };
};
```

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.

Classe

Inconvénient d'une structure : la boîte est ouverte.

A savoir, n'importe quel bout de code qui a accès à la structure 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.

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).

Définition : exemple

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.

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 `asserts` afin de faire en sorte que $0 \leq$ `i` $<$ `n`.

Fonctions et classes amies

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; };
```

Fonctions et classes amies

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;
};
```

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 `Discs` ou des `Squares`.

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; };
}
```

Gestion des 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.

Gestion des membres par défaut

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;`

Gestion des membres par défaut

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;
```

Gestion des membres par défaut

- 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
```

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.

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.

Définition d'un espace de nommage

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).

Définition d'un espace de nommage

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) {
    ...
}
```

Accès à un 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);
    ...
}
```

Accès à un espace de nommage

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).

Structuration

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 à un espace de nommage

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`.

  Exemple :
  ```
  {   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.

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.

namespace anonyme

L'intérêt d'un `namespace` anonyme (sans nom) est de cloisonner les déclarations qui y sont contenu dans l'environnement dans lequel 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

Les outils vus jusqu'à présent permettent de structurer les données, et d'encapsuler les fonctions avec les données.

Mais, nous n'avons pas encore vu les fonctionnalités avancées des classes (héritage, virtualisation, polymorphisme).

Ceci est l'objet de la leçon suivante.