Initialisation des attributs (1)

Initialiser les attributs d'une instance
en leur affectant individuellement une valeur,
après l'instanciation, n'est pas une technique satisfaisante:

a) elle est **fastidieuse**, en particulier lorsque le nombre d'attributs ou/et d'instances est élevé; par ailleurs, les oublis probables sont des sources supplémentaires d'erreurs.

b) elle implique que tous les attributs doivent faire partie de l'interface (i.e. soient **publics**), ce qui est clairement à éviter.[10]

Pour respecter le principe d'encapsulation, qui stipule
qu'un objet doit contenir en lui-même tout le matériel nécessaire à
son fonctionnement, il est important de le munir d'un
mécanisme d'initialisation **efficace**.

10. Ce défaut peut être atténué par l'utilisation de méthodes spécifiques («accesseurs» (*get*) et «modificateurs» (*set*)).

Initialisation des attributs (2)

Un moyen simple pour réaliser l'initialisation des attributs
est de définir une méthode dédiée à cette tâche:

```
class Rectangle {
public:
   unsigned int hauteur;    unsigned int largeur;
   void init(const int h, const int l) {
      hauteur = h; largeur = l;
   }
   ... };
```

Le mécanisme d'initialisation des instances fourni par les langages comme C++
correspond en fait à une systématisation de cette solution,
par le biais de deux types de méthodes particulières:

- **les *constructeurs***:
  méthodes responsables de l'initialisation des attributs d'une instance (i.e. réalisant toutes les opérations requises en «début de vie» de l'instance)

- **le *destructeur***:
  méthode responsable des opérations à effectuer «en fin de vie» de l'instance (i.e. lorsque l'exécution du programme quitte le bloc dans lequel l'instanciation a eu lieu).

Les constructeurs (1)

Les ***constructeurs*** sont des méthodes, invoquées **automatiquement** lors de l'instanciation, qui assurent l'initialisation des instances.

La syntaxe de déclaration d'un constructeur, pour une classe `NomClasse`, est:

```
NomClasse(<arguments>)
:   <attribut_1>(<valeur_1>),
    ...
    <attribut_n>(<valeur_n>)
{
    // autres opérations
}
```

Exemple:

```
Rectangle(int h, int l)
:  hauteur(h),
   largeur(l)
{ }
```

Un constructeur est donc une méthode **sans indication de type de retour** (pas même `void`), et **de même nom que la classe** pour laquelle la méthode est définie.[11]

11. On peut aussi considérer que le constructeur est une méthode sans nom, dont le type de la valeur de retour est la classe.

Les constructeurs (2)

Le corps du constructeur est précédé d'une section optionnelle introduite par «:», spécifiquement réservée à l'initialisation des attributs.

Les attributs non initialisés dans cette section prendront une valeur par défaut dans le cas où une telle valeur existe[12] ou resterons non initialisés dans le cas contraire.

Il est bien sûr possible de changer la valeur d'un attribut initialisé ou d'affecter une valeur à un attribut non initialisé dans le corps du constructeur[13].

Exemple:

```
Rectangle(const int h, const int l)
:  hauteur(h)          <-- initialisation
{
   int t(2*l - h % 2);
   largeur = t;        <-- affectation
}
```

largeur a une valeur indéfinie → `largeur`

12. Une valeur par défaut est une valeur donnée par le *constructeur par défaut* (c.f. suite du cours). Il n'y a pas formellement de valeurs par défaut pour les types autres que les classes, même si dans la pratique, 0 où ses différentes représentations (i.e 0.0, '\0', etc), est utilisé.

13. Remarquons cependant que les attributs déclarés constants ne peuvent être valués que dans la section d'initialisation précédant le bloc de la méthode.

Les constructeurs (3)

Comme dans le cas des autres méthodes, il est tout à fait possible de ***surcharger les constructeurs***; une classe peut donc admettre **plusieurs constructeurs**, pour autant que leurs listes d'arguments soient différentes.

Exemple:

```
class Rectangle {
   public:
   int hauteur, largeur;

   Rectangle(const int c)
   : hauteur(c), largeur(c) { }

   Rectangle(const int h, const int l)
   : hauteur(h), largeur(l) { }

   // autres méthodes ........
   unsigned int surface() {...}
   void changeHauteur(...) {...}
};
```

Initialisation par constructeur

La syntaxe de la déclaration avec initialisation d'une instance est identique à celle d'une déclaration avec initialisation d'une variable ordinaire.

Ainsi, pour une classe nommée `NomClasse`, définissant le constructeur `NomClasse(`arg_1`,` arg_2`)`, la déclaration-initialisation de l'instance `instance` est:

`NomClasse instance(`$valarg_1$`,` $valarg_2$`);`

où $valarg_1$ et $valarg_2$ sont les valeurs d'initialisation des arguments.

Exemple:

Avec la définition de la classe `Rectangle`, donnée auparavant, la déclaration avec initialisation de deux instances `r1` et `r2`, où `r1` est un rectangle de 18 par 5 et `r2` un carré de 5 de côté, sera obtenue par:

```
Rectangle r1(18,5);
Rectangle r2(5);
```

Masquage d'attributs et constructeurs

Comme dans le cas des autres méthodes, lorsqu'un constructeur possède un argument dont l'identificateur **masque** l'un de ses attributs, il est possible d'utiliser le mot-clef `this` pour référencer de façon non ambigüe l'attribut de l'instance.

Exemple:

```
Rectangle(const int h, const int largeur)
   :  hauteur(h)
   { this->largeur = (2*largeur - h % 2); }
```

Remarquons que la désambiguïsation à l'aide de `this` n'est pas nécessaire dans la section d'initialisation, puisque les éléments initialisés sont obligatoirement les attributs:

Exemple:

```
Rectangle(const int hauteur, const int largeur)
   :  hauteur(hauteur),
      largeur(largeur)
   { }
```

désignent obligatoirement les attributs

Arg à valeur par défaut et constructeurs (1)

Comme les autres méthodes et fonctions en général,
les constructeurs peuvent admettre des arguments pour lesquels
une **valeur par défaut** est spécifiée.

Exemple:

```
Rectangle(const int h, const int l = 10)
   :  hauteur(h),
      largeur(l)
   { }
```

Dans ce cas, la déclaration d'une instance `rect` de la forme: `Rectangle rect(5);`
est alors équivalente à la déclaration: `Rectangle rect(5,10);`

Les précautions à prendre lorsque l'on donne une valeur par défaut à un argument de constructeur sont les mêmes que pour les autres méthodes ou fonctions. Il faut particulièrement veiller à ne pas créer d'ambiguïtés lors de l'utilisation simultanée du mécanisme de surcharge, puisque chaque argument avec valeur par défaut induit deux signatures différentes pour la méthode.

Constructeur par défaut (1)

Le ***constructeur par défaut*** est un constructeur autorisant une instanciation de la forme:

Forme générale:

<identificateur de classe> **<***identificateur d'instance***>;**

Exemple:

```
Rectangle r;
```

Un *constructeur par défaut* est donc un constructeur sans arguments, ou un constructeur dont tous les arguments ont des valeurs par défaut:

```
Rectangle()
:  hauteur(0),
   largeur(0)
{ }
```

```
Rectangle(const int dim = 10)
:  hauteur(dim),
   largeur(dim)
{ }
```

Deux constructeurs par défaut.

La mise en garde précédente est illustrée ici par le fait que ces deux constructeurs ne peuvent être définit simultanéments pour la même classe (l'instanciation «`Rectangle r;`» serait en effet ambigüe).

Constructeur par défaut (2)

Lorsqu'on ne définit aucun constructeur pour une classe[14], le compilateur se charge de **générer automatiquement** un constructeur par défaut.

Naturellement, ce constructeur est minimaliste, et il se contente d'initialise à leur valeur par défaut les attributs pour lesquels une telle valeur existe.

Plus précisément, les attributs sont simplement déclarés, et ceux pour lesquels un constructeur par défaut existe sont initialisés par l'invocation de ce constructeur:

Exemple: *dans cet exemple, la classe C est refusée par le compilateur, qui ne sait comment initialiser les instances de B.*

```
class A {
public:
   int a;
   A():a(10) { }
};
```

*constructeur par défaut **explicite***

`A obj1;` *est licite*
`A obj2(...);` *est illicite*

```
class B{
public:
   int b;
   B(int b):b(b) {}
};
```

***pas** de constructeur par défaut*

`B obj1;` *est illicite*
`B obj2(...);` *est licite*

```
class C{
public:
   A objA;
   B objB;
};
```

+ auto-généré

```
C()
: objA(),
  objB()
{ }
```

*constructeur par défaut **implicite***

`C obj1;` *est licite*
`C obj2(...);` *est illicite*

14. Et uniquement dans ce cas.

Constructeur de copie (1)

Le constructeur de copie est un autre constructeur particulier, qui permet de créer une copie d'une instance (i.e. d'initialiser une instance en utilisant les attributs d'une autre, de même type):

Forme générale:

```
<id de classe>(const <id de classe>& obj)
: …
{ … }
```

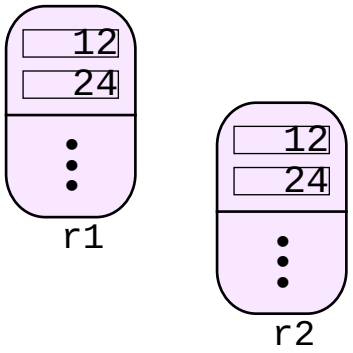
Exemple:

```
Rectangle(const Rectangle& obj)
:  hauteur(obj.hauteur),
   largeur(obj.largeur)
{ }
```

L'invocation du constructeur de copie se fait par une instruction de la forme:

```
Rectangle r1(12,24);
Rectangle r2(r1);
```

Après ces instructions, on obtient la situation suivante:

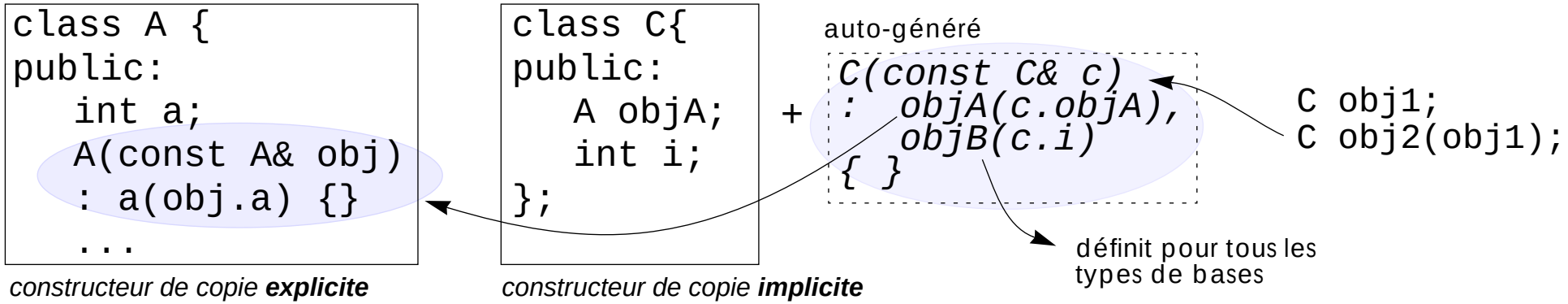


`r1` et `r2` sont deux instance **distinctes** de la classe `Rectangle`, mais équivalentes pour ce qui est de leurs attributs (*égalité profonde*)

Constructeur de copie (2)

Si un constructeur de copie n'est pas explicitement défini, le compilateur en générera un de manière automatique.

Le constructeur généré procède à une initialisation membre à membre des attributs, en invoquant le constructeur de copie de chacun d'eux (copie *de surface*):



Si le constructeur généré par le compilateur suffit dans la plupart des cas, il existe néanmoins des situations ou le programmeur est obligé de fournir lui-même la définition de ce constructeur.

Parmi ces situations, la plus fréquente est celle d'instances réalisant des allocations dynamiques de mémoire (abordé plus tard dans le cours); un autre cas (académique) pour lequel la copie *de surface* n'est pas désirée est celui du comptage des instances (c.f. exemple ci-après, en fin de chapitre «Constructeurs, Destructeurs»).

Destructeurs (1)

Le constructeur a pour tâche l'initialisation de l'instance.
En d'autre termes, il créé l'ensemble des éléments informatiques avec lesquels les méthodes pourront opérer.

Parfois, cet initialisation implique la mobilisation de ressources – fichiers, périphériques, portion de mémoire, etc – qu'il est important de libérer après usage.

Dans cette optique, se pose alors le même problème que lors de l'initialisation:

l'invocation explicite par le programmeur de méthodes de libération de ressources n'est pas une solution satisfaisante (fastidieuse, source d'erreur, et affaiblissant l'encapsulation).

La solution fournie par C++ est similaire à celle à base de constructeurs: une méthode particulière, appelée ***destructeur***, est définie.
Cette méthode est invoquée automatiquement en fin de vie de l'instance, et son rôle principal est **d'assurer la libération des ressources** éventuellement mobilisées.

Destructeurs (2)

La syntaxe de déclaration d'un destructeur,
pour une classe nommée `NomClasse`, est:

```
~NomClasse()
{
   // opérations
}
```

Le destructeur d'une classe est donc une méthode **sans argument**
dont le nom est celui de la classe, préfixé du signe «~» (tilde)

Comme le destructeur n'admet pas d'argument, il n'est pas possible de le surcharger.
Toute classe admet donc exactement **un** destructeur; s'il n'est pas défini
explicitement par le programmeur, c'est le compilateur qui se chargera
d'en générer une version minimaliste.

Destructeurs (3): exemple

Le problème du comptage des instances (en activité) d'une classe constitue un exemple (académique) d'utilisation des destructeurs pour lequel l'implémentation réalisée par le compilateur n'est pas acceptable:

Pour connaître à tout moment le nombre d'instance de la classe `Rectangle`, il suffit d'utiliser une variable globale de type entier long comme compteur:

⇨ le constructeur incrémente la variable,

⇨ tandis que le destructeur la décrémente:

Exemple:

```
long counter(0); // variable globale
// ...
class Rectangle
{
// ...
   Rectangle(): hauteur(0),largeur(0) { ++counter; } // constructeur
   ~Rectangle() { --counter; } // destructeur
   // ...
};
```

Destructeurs (4): exemple - suite

On constate immédiatement, on considérant l'exemple précédant,
que tous les cas de figure ne sont pas envisagés,
et que le comptage n'est pas toujours correcte:

```
void main()
{
   Rectangle r1;
   {
      Rectangle r2(r1);
      Rectangle r3(r2);
      // ...
   }
}
```

counter

0
1
1
1
-1
(-2)

La copie d'un rectangle en un autre échappe au compteur d'instance,
puisque nous n'avons pas définit le comportement du *constructeur de copie*.

Constructeurs et Destructeurs

Une règle quasi absolue veut que si l'on
doit spécifier le comportement de l'une ou l'autre
des méthodes traditionnellement générées par le compilateur
(constructeur par défaut, de copie et destructeur[15]), alors,
on doit spécifier le comportement de **toutes** ces méthodes.

```
long counter(0);
class Rectangle
{
public:
   int hauteur;
   int largeur;
   Rectangle(const int dim = 0)
   :  hauteur(dim),
      largeur(dim)
   {
      ++counter;
   }
   Rectangle(const Rectangle& r)
   :  hauteur(r.hauteur),
      largeur(r.largeur)
   { ++ counter; }

   // et ainsi de suite avec tous
   // les constructeurs.

   ~Rectangle() { --counter; }

   // autres méthodes ...
};
```

15. Formellement, cette énumération devrait comporter des éléments supplémentaires (tels l'opérateur d'affectation), mais qui ne seront étudiés que plus tard dans le cours.

Opérateur de résolution de portée «::»

Lorsque les définitions des classes deviennent complexes,
la lisibilité du code diminue, car on perd la **vue d'ensemble des comportements** définit dans une classe.

Il est toutefois possible de retrouver cette vue d'ensemble,
en **écrivant les définition des méthodes à l'extérieur** de la déclaration de la classe

Ceci est rendu possible grâce à l'opérateur de résolution de portée «::»,
qui permet de relier la définition d'une méthode
à la classe pour laquelle elle est définie.

Grace à cet opérateur, le programmeur peut se contenter
d'indiquer les **prototypes des méthodes** dans la déclaration de la classe,
et écrire les définitions correspondantes, à l'extérieur de la déclaration,
sous la forme de définition de fonctions de nom:

<nom de classe>`::`*<nom de fonction>*`(`*<arg$_1$>*`,`*<arg$_2$>*`, ...)`
`{ ... }`

Opérateur «::» (2)

Par exemple, la déclaration de la classe `Rectangle` pourrait être:

```
class Rectangle
{
   public:

   // déclaration des attributs
   int hauteur;
   int largeur;

   // prototypage des constructeurs & destructeur
   Rectangle(const int dimensions = 0);
   Rectangle(const int hauteur, const int largeur);
   Rectangle(const Rectangle& obj);
   ~Rectangle();

   // prototypage des méthodes
   int surface() const;
};
```

Opérateur «::» (3)

Accompagné des définitions «externes» des méthodes:

```
Rectangle::Rectangle(const int dimensions)
:  hauteur(dimensions),
   largeur(dimensions)
{ }

Rectangle::Rectangle(const int hauteur, const int largeur)
:  hauteur(hauteur),
   largeur(largeur)
{ }

// ...

Rectangle::~Rectangle()
{ }

int Rectangle::surface() const
{
   return (hauteur * largeur);
}
```

Surcharge des opérateurs (1)

C++ fournit une fonctionalité extêmement utile pour les applications: la possibilité donner un sens aux opérateurs du langage (tels `+`, `*`, `/`, ...) appliqués à des types quelconques.

Cette fonctionnalité est habituellement appelée: ***surcharge des opérateurs***

Elle correspond de fait aux surcharges de fonctions ou de méthodes, mais réalisée sur les fonctions et méthodes qui implémentent le fonctionnement de ces opérateurs.

On peut en particulier donner un sens à des opérateurs appliqués aux instances d'une classe:

Supposons par exemple que l'on veuille définir une addition pour les rectangles, permettant d'utiliser une expression intuitive de la forme: `rect1 + rect2`

Surcharge des opérateurs (2)

Pour cela, il faut surcharger l'opérateur «+» dans la classe `Rectangle`, par le biais de la définition de la méthode prédéfinie:

`<`*type*`> operator+(<`*type*`> <`*membre_droit*`>)`

Exemple:

Prototypage dans la classe `Rectangle`:

```
Rectangle operator+(const Rectangle& mbrDroit) const;
```

Définition hors de la classe:

```
Rectangle Rectangle::operator+(const Rectangle& r) const
{
   return Rectangle(hauteur + r.hauteur, largeur + r.largeur);
}
```

Si `rect1` et `rect2` sont des instances de `Rectangle` initialisée par respectivement
`Rectangle rect1(2,3); Rectangle rect2(3,4);`
alors, l'expression «`rect1 + rect2`» correspond à une nouvelle instance de la classe `Rectangle`, de hauteur 5 et de largeur 7.

Surcharge des opérateurs (3)

La surcharge des autres opérateurs (`-`, `*`, `/`, ...) se fait par le biais des diverses méthodes associées:

`operator-(`*<second argument>*`)`
`operator*(`*<second argument>*`)`
`operator/(`*<second argument>*`)`
...

Où *<second argument>* désigne le membre droit de l'opérateur (dans le cas des opérateurs diadiques)

Remarquons qu'il est possible de **surcharger multiplement** les opérateurs (règles usuelles de la surcharge), par exemple pour donner un sens aux expressions suivantes:

```
rect1 * rect2;
  rect2 * 2;
```

Mais dans ce dernier cas, la commutativité de l'opérateur «*» n'est pas respectée, l'expression «`2 * rect2`» n'ayant pas de sens.[16]

16. Il existe toutefois un moyen pour donner un sens à une telle expression, et ainsi garantir les propriétés des opérateurs arithmétiques, mais cela dépasse largement le cadre de ce cours.