Travailler avec les Objets

- (signature et surcharge)
- (déclarations et définitions)
- constructeurs
- déstructeurs
- résolution de portée
- surcharge

Apr 11, 2001 12:42

Signature et Surcharge

Signature d'une fonction: le couple constitué

- du nom la fonction, et
- de la liste de types de ses arguments formels

Pour le compilateur,
c'est la signature qui compte, et ne pas le nom.

Donc,
on peut définier plusieurs fonction avec le meme nom
si les signatures sont toujours différentes.

Ce mécanisme s' appelle ***surcharge***.

Valeur par Défaut

```
void oracle (int x=42){
  cout << x << endl;
};

oracle(5);

oracle();
```

- Les arguments par défaut doivent apparaître en dernier dans la liste des arguments
- L'application crée une signature multiple !

Déclarations et Définitions (I)

Le principe de séparer la déclaration et la définition des variables est en fait aussi recommandé pour les fonctions. Cela est avantageux pour:

- la *lisibilité*
- la *modularité*

```
#include <iostream.h>
#include "local.h"
```

Déclarations et Définitions (II)

```
// toutes les déclarations
...
int f (int x);


// toutes les définitions
...
int f (int x){
  // corps
};
```

Initialisation des Attributs

Encapsulation:

Il vaut mieux éviter que tous les attributs soient publics.

Par conséquent,

on ne peut pas accéder aux attributs privés directement depuis l' exterieur . . .

Comment initialiser?

Prévoir une méthode **`set_attribut`** pour chaque attribut privé?

- trop de travail ...
  il vaut mieux le faire plus systématiquement
- trop d'accessibilité après l'initialisation

Initialisation par Méthode

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

Constructeurs

Les ***constructeurs*** représentent une
systématisation de l'initialisation (des attributs) par méthode
en »début de vie« d'un objet.

Un constructeur lui–même est une méthode

- qui ***porte le même nom que la classe***,
- qui n'a ***pas de type de retour***,
- qui n'est ***pas forcement définie*** par le programmeur,
- qui est ***invoquée automatiquement***
  lors de l'instanciation.

. . . il y a aussi des destructeurs pour la »fin de vie« . . .

Syntaxe

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

- pas nécessaire d'initialiser tous les attributs
- aussi possible dans le corps
- pas de confusion des noms des attributs avec les noms des arguments

```
NomClasse variable ( <arguments> );
```

Exemple (C–I)

```
class Rectangle {
public:
  unsigned int hauteur;
  unsigned int largeur;
  void init(const int h, const int l) {
    hauteur = h;
    largeur = l;
  }
  Rectangle (const int h, const int l)
  : hauteur(h),
    largeur(l)
  { }
  ...
};
Rectangle rect(3,5);
```

Exemple (C–II)

```
class Rectangle {
public:
   const unsigned int hauteur;
         unsigned int largeur;
   Rectangle (const int h, const int l)
   : hauteur(h)      // initialisation
   {
      int dl = 2*l;
      largeur = dl;   // affectation
   }
   ...
};
Rectangle rect(3,5);
```

Attributs déclarés constants ***seulement via initialisation***.

Surcharge

```
class Rectangle {
public:
   int hauteur, largeur;
   Rectangle (int h, int l)
   : hauteur(h), largeur(h) { }
   Rectangle (int c)
   : hauteur(c), largeur(c) { }
   ...
};
Rectangle rect(3,5);
Rectangle rect(4);
```

Une classe peut admettre ***plusieurs constructeurs***.
Mais, leur signatures doivent être differents.

Masquage

```
class Rectangle {
public:
   int hauteur, largeur;
   Rectangle (int hauteur, int largeur)
   : hauteur(hauteur)
   {
     this->largeur = largeur;
   }
   ...
};
Rectangle rect(3,5);
```

`this` est nécessaire dans l'affectation, mais pas dans l'initialisation.

Valeur–par–défaut

```
class Rectangle {
public:
   int hauteur, largeur;
   Rectangle (int h, int l=42)
   : hauteur(h), largeur(h) { }
   ...
};
Rectangle rect1(3);
Rectangle rect2(3,42);
```

`rect1` et `rect2` sont des objets tout à fait pareils,
mais ils sont quand même différents . . .

Ambiguité?

```
class Rectangle {
public:
   int hauteur, largeur;
   Rectangle (int h, int l=42)
   : hauteur(h), largeur(h) { }
   Rectangle (int c)
   : hauteur(c), largeur(c) { }
   ...
};
Rectangle rect(3);
```

`rect` ne peut pas être initialisé,
parce que le constructeur à utiliser est ambigu.

Constructeur–par–défaut (I)

```
class Rectangle {
public:
   int hauteur, largeur;
   Rectangle ()
   : hauteur(0), largeur(0) { }
   ...
   Rectangle (int x=42)
   : hauteur(x), largeur(x) { }
};
Rectangle rect;
```

Constructeur par défaut est un constructeur qui n’attend pas d’arguments.

Constructeur–par–défaut (II)

```
class A {
public:
   int ia;
};
```

```
class B {
public:
   A a;
};
```

Il n' est pas toujours nécessaire de définir un constructeur.
Le compilateur va essayer d'en générer un par défaut.
Celui initialise à leur valeur par défaut les attributs pour lesquels une telle valeur existe.

```
A a;
A a(42);
B b;
B b(42);
```

Constructeur–par–défaut (III)

```
class A {
public:
   int ia;
   A():ia(10){};
};
class B {
public:
   int ib;
   B(int ib):ib(ib){};
};
class C {
public:
   A a;
   B b;
};
```

```
A a;
A a(42);
B b;
B b(42);
C c;
C c(42);
```

Il n' est pas toujours nécessaire de définir un constructeur.
Le compilateur va essayer d'en générer un par défaut.

Constructeur de Copie (I)

```
class Rectangle {
public:
   int hauteur, largeur;
   Rectangle (const Rectangle &r)
   : hauteur(r.hauteur),
     largeur(r.largeur) { }
   ...
};
Rectangle rect1;
Rectangle rect2(rect1);
```

Constructeur de copie est un constructeur qui permet d'accéder à un objet existant de la meme classe.

Constructeur de Copie (II)

```
class A {
public:
   int ia;
   A(const A &a):ia(a.ia){};
};
class B {
public:
   int ib;
   A a;
};
B b1;
B b2(b1);
```

Si on ne définit pas de constructeur par copie,
le compilateur va essayer d'en générer un par défaut.

Destructeurs

Les ***destructeurs*** représentent un complément aux constructeurs: ils sont éxecuté en »fin de vie« d'un objet.

Un destructeur lui–même est une méthode

- qui ***porte le nom "~nom_de_la_classe"***,
- qui n'a ***pas de type de retour***,
- qui n'a ***pas d'arguments***,
- qui n'est ***pas forcement définie*** par le programmeur,
- qui est ***invoquée automatiquement*** lors de la destruction (fin du bloc, delete).

Les destructeurs servent à la libération des ressources (fichiers, périphériques, ...) éventuellement mobilisées.

Syntaxe

```
~NomClasse ( )
{
  // corps avec opérations
};
```

Comme il n'y a pas d'argument,
on ne peut pas surcharger les déstructeurs,
donc, toute classe n'admet qu'un seul déstructeur.

Exemple (D–I)

Comptage du nombre des instances.

```
long counter = 0;   // variable globale
class Rectangle {
public:
   int hauteur,largeur;

   Rectangle ()
   : hauteur(0), largeur(0) { ++counter; }

   ~Rectangle ()
   { --counter; }
   ...
};
```

Exemple (D–II)

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

Résolution de Portée (::)

Suivant le principe de séparation de la déclaration et de la définition d'une fonction, les méthodes peuvent etre prototypées à l'intérieur d'une classe, mais définies à l'exterieur de cette classe.

- *lisibilité*
- *modularité*

```
#include <iostream.h>
#include "local.h"
```

L'opérateur "`::`" permet de ce faire ...

Syntaxe

```
class NomClasse {
   ...
   type nomfonction(<arguments>);
   ...
};


...

type NomClasse::nomfonction(<arguments>)
{
   // corps avec
   // autres opérations
};
```

Exemple (R–I)

```
class Rectangle {
public:

   // déclaration des attributs
   int hauteur,largeur;

   // déclaration des {con,dé}structeurs
   Rectangle (const int h, const int l);
   ~Rectangle ();

   // déclaration des méthodes
   int surface ();

};
```

Exemple (R–II)

```
// définition des {con,dé}structeurs

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

Rectangle::~Rectangle () { }


// définition des méthodes

int surface () {
  return ...?...
};
```

Surcharge des Opérateurs (+,–,*,/,...)

Comment définir `rect1 + rect2` ?

En interprétant

- `rect1` comme objet
- `+` comme méthode de rect1
- `rect2` comme argument

Exemple (S)

```
class Rect {
public:

  // déclaration des méthodes
  Rect operator+ (const Rect &r2);

};

// définition des méthodes
Rect Rect::operator+ (const Rect &r2){
  return Rect( haut+r2.haut , larg+r2.larg );
};
```

Série

`http://lampwww.epfl.ch/cours/mp2/`
`serie1/donnee.html`

- partie obligatoire
- partie facultative
- corrigé: la semaine prochaine
- (continuation avec QC++ facultative)