Plus de C++

ISIMA - ZZ2 - 2009

Christophe Duhamel
Andréa Duhamel

Forme Normale de Coplien

- 4 méthodes
  - constructeur par défaut `A::A ();`
  - constructeur par recopie `A::A (const A &);`
  - opérateur d'affectation `A & A::operator= (const A &);`
  - destructeur `A::~A ();`
- Comportement par défaut : gestion bit à bit
- Adapté pour des classes simples
- Inapproprié lorsque
  - gestion de mémoire dynamique, pointeurs
  - gestion de fichier, socket, buffer I/O

Création d'objets (Rappel)

- Trois classes d'allocation (comme en C)
  - automatique : variable locale sur la pile
  - statique : variable globale, variable locale statique
  - dynamique : variable allouée sur le tas
    - allocation avec `new` + constructeur
    - destruction avec `delete`
- Gestion mémoire
  - statique et automatique : système
  - dynamique : construction / destruction à la main

Éléments nécessaires (1/2)

- Constructeur par défaut
  - construit un objet par défaut
  - syntaxe : `A::A();`
  - ex : initialisation dans un tableau

Point * nuage;
nuage = new Point[taille];

- Constructeur par recopie
  - construit un objet par clonage
  - syntaxe : `A::A (const A &);`
  - ex : paramètre passé par valeur

référence obligatoire !
sinon crée objet A tempo
→ boucle infinie

4

Éléments nécessaires (2/2)

- Opérateur d'affectation
  - recopie un objet dans un autre
  - syntaxe : `A &A::operator= (const A &);`
  - copie hors initialisation, gestion plus complexe

retour par référence
pour un chaînage éventuel

- Destructeur : libère les ressources de l'objet
  - libère les ressources de l'objet
  - syntaxe : `A::~A ();`

5

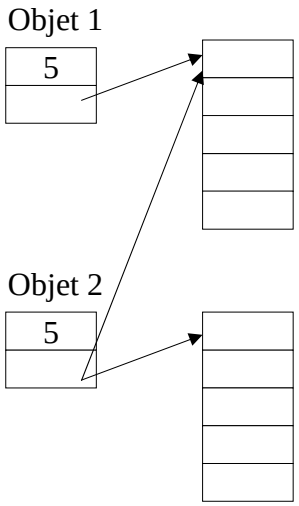
Constructeur par recopie

- Objet transmis par valeur en argument à une fonction/méthode
- Objet renvoyé par valeur comme résultat d'une fonction/méthode
- Objet initialisé lors de sa déclaration avec un autre objet de même type

6

Pourquoi c'est necessaire?

```
class A
{
  protected :
    int taille;
    int * tableau;
  public :
    A(int n) : taille(n)
    {
      tableau = new int[taille];
    }
    exemple(A a){…}
};
```

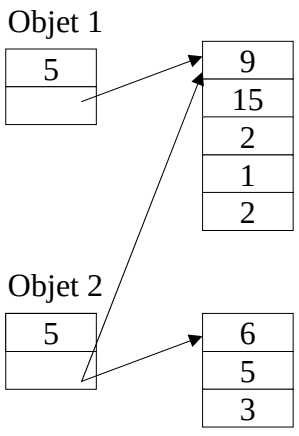



```
A objet1(5);
exemple(objet1);
```

7

Affectation : problème voisin

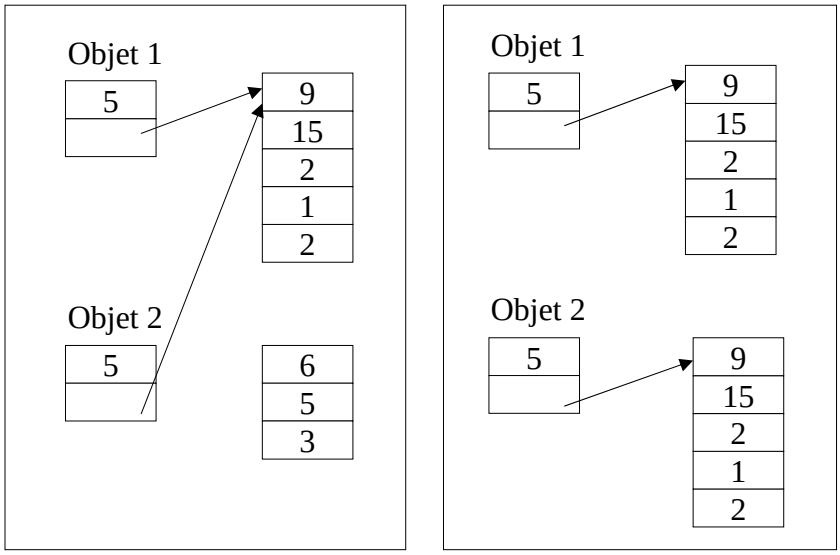
```
class A
{
  protected :
    int taille;
    int * tableau;
  public :
    A(int n) : taille(n)
    {
      tableau = new int[taille];
    }
    exemple(A a){…}
};
```




```
A objet1(5), objet2(3);
objet2 = objet1;
```

8

Affectation : problème voisin



9

Une classe **`Chaîne`**

- Problème : gestion dynamique de la longueur
- Solution
  - allocation dynamique d'un tableau de caractères
  - un attribut taille et un attribut tableau
- Gestion complexe de mémoire
  - à la mort de l'objet, effectuer la désallocation
  - lors de la copie d'une chaîne dans une autre
    - ne pas recopier le pointeur sur le tableau (copie physique/ fausse copie)
    - copier le tableau (copie logique/duplication)

10

Déclaration de **Chaîne**

```
#ifndef __CHAINE_H__
#define __CHAINE_H__

class Chaine
{
  private:
    int   taille;
    char *tableau;

  public:
    Chaine(int t=0);
    Chaine(const char *str);
    Chaine(const Chaine &);
    Chaine & operator=(const Chaine &);
    ~Chaine();
    int getTaille(void) const;
    const char * str();
};

#endif
```

11

Définition de **Chaîne** (2/2)

```
Chaine::Chaine (const Chaine & c)
{
  taille = c.taille;
  if (taille)
  {
    tableau = new char[taille];
    strcpy(tableau, c.tableau);
  }
}
Chaine & Chaine::operator= (const Chaine & c)
{
  if (this != &c)
  {
    if (tableau)
      delete[] tableau;
    taille = c.taille;
    if (taille)
    {
      tableau = new char[taille];
      strcpy(tableau, c.tableau);
    }
  }
  return *this;
}
```

constructeur par recopie

important !

factoriser

opérateur d'affectation

pour le chaînage des opérations : `c3 = c2 = c1;`

12

Initialisation ou affectation ?

- Principe

  une initialisation se fait uniquement dans une définition

- exercice

```
T t1;
T t2(params);
T t3(t1);
T t4 = t1;
T t5();
t1 = t4;
```

```
T::T();
T::T(lst_params);
T::T(const T &);
T::T(const T &);
T t5(void);
T & T::operator=(const T &);
```

13

Passage d'objets par référence

- Lors d'un passage par valeur
  - création d'un objet temporaire
  - initialisation par le constructeur par recopie
- Toujours passer les objets par référence
  - évite une création d'objet inutile
  - préserve le polymorphisme
  - passer en référence constante si objet non modifiable
- Lors du renvoi d'un objet
  - passer par référence si on renvoie l'objet courant (**`this`**)
  - sinon, toujours par copie (car destruction de l'objet local)

14

La surcharge d'opérateurs

- La langage C réalise déjà la surdéfinition de certains opérateurs, par exemple :
  - (a + b) le symbole "+" effectue une opération qui dépends du type de a et b…
  - (*) peut désigner une multiplication ou une indirection (a = *ptr).
- Surdéfinir des opérateurs existant (unaire ou binaire) pour qu'ils portent sur au moins un objet.
- Exemple : soit a et b du type nombres complexes, on va pouvoir donner signification à des expressions, telles que (a+b), (a-b), (a*b), (a,/b),…

15

La surcharge d'opérateurs

- Forme de polymorphisme faible
  - appliquée aux opérateurs du langage
  - adapter les opérateurs à de nouveaux types
  - maintenir une syntaxe « simple » pour l'utilisateur
- Deux types d'opérateurs
  - méthode : `b + c → b.operator+(c)`
  - fonction : `b + c → +(b,c)`
- Principe

définir en méthode si `this` a un rôle prépondérant

16

La surcharge d'opérateurs

- Opérateur α externe : fonction
  - syntaxe : `T operator`α `(const T &, const T &);`
  - utilisation : `t1` α `t1` ↔ `operator`α `(t1,t2)`
  - surtout opérateurs dyadiques (2 paramètres)
  - aucun n'est prépondérant
- Opérateur α interne : méthode
  - syntaxe : `T T::operator`α `(const T &);`
  - utilisation : `t1` α `t2` ↔ `t1.operator`α`(t2)`
  - opérateurs monadiques et dyadiques
  - **`this`** est prépondérant

17

La surcharge d'opérateurs

- Opérateurs surchargeables
  - en méthode : **`=`**, **`+=`**, etc, **`()`**, **`*`**, **`[]`**, **`++`** et **`--`**, **`->`**, **`<<`** et **`>>`** (décalage), **`new`** et **`delete`**
  - en fonction : **`<<`** et **`>>`** (flux), **`+-/*`**
- Opérateurs à éviter
  - **`,`**, **`!`**, **`||`** et **`&&`**
- Opérateurs interdits
  - **`.`**, **`::`** et **`?:`** (ternaire)

18

La surcharge d'opérateurs : exemple 1

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  …
  public :

  …
    Point & operator+ (const Point & tmp)
    {
       absc = absc + tmp.absc;
       ordo = ordo + tmp.ordo;
       return *this;
    }
};
```

19

La surcharge d'opérateurs : exemple 2

```
class Point
{
  private:
    int absc;
    int ordo;
    static int nb_points;

  …
  public :

  …
};
int operator* (const Point & p1,
               const Point & p2)
{
  return (p1.getAbs() * p2.getAbs() +
          p1.getOrd() * p2.getOrd());
}
```

20

La surcharge d'opérateurs

- Conclusion
  - opérateurs monadiques et affectations en interne (méthode)
  - opérateurs dyadiques en externe (fonctions)
- Attention
  - en interne, transtypage sur le premier paramètre interdit
  - en externe, transtypage possible sur tous les paramètres
  - en externe, pas d'accès direct aux attributs des objets
  - passer par les accesseurs ou déclarer **`friend`**

21

Utilisation de `friend` (1/2)

- Autoriser l'accès pour une entité externe
  - dans la déclaration de la classe
  - classe ou fonction
  - syntaxe :

```
class A
{
    …
    friend class B;
    friend void f(int, double);
    …
};
```

  - ne se substitue pas à la déclaration du prototype
  - accès complet à la classe
  - pas transitif, ni hérité

22

Utilisation de **`friend`** (2/2)

- Avantages
  - efficacité du code, accès direct
  - notion (très limitée) de package
- Inconvénients
  - violation du principe d'encapsulation
  - pas de restriction ni de contrôle sur les accès
- Garde-fous
  - déclaration dans la classe cible
  - pas de transitivité ni d'héritage

23

Concaténation pour **`Chaîne`**

- Utiliser l'opérateur **`+`**
  - syntaxe : `ch3 = ch1 + ch2;`
  - **`this`** n'est pas prépondérant → opérateur externe

version **`friend`**

```
Chaine operator+ (const Chaine & ch1, const Chaine & ch2)
{
  Chaine tmp(ch1.taille + ch2.taille - 1);

  strcpy(tmp.tab, ch1.tab);
  strcat(tmp.tab, ch2.tab);

  return tmp; // retour par copie nécessaire
}
```

→ efficace mais doit connaître (et respecter) l'implémentation

24