Cours de programmation orientée objet et d'algorithmique avancée de V. Jay et S. Balev

FMdKdD
fmdkdd [à] free.fr

Université du Havre
Année 2009–2010

Table des matières

Chapitre 1

Le langage C++

1.1 Particularités et spécificités de C++

Le langage C++ est un langage objet conçu par Bjarne Stroustrup, comme un sur-ensemble de C (norme ANSI).

1.1.1 Commentaires

Deux types de commentaires en C++ :
- /* ... */ commentaires sur plusieurs lignes,
- // ... commentaires sur une ligne.

1.1.2 Déclaration de fonctions

Lorsqu'on utilise, dans un programme ou dans une fonction, une fonction non encore définie, il faut la déclarer, en début de programme ou de fonction, sous la forme de son prototype :

```
type_resultat nom_fonction (type_arg1, type_arg2, ...);
```

Il est possible de donner des noms de paramètres (pour la documentation de la fonction).

1.1.3 Déclaration des variables

En C++, il n'est plus nécessaire (mais recommandé) de déclarer les variables avant la première instruction d'une fonction. La durée de vie d'une variable déclarée au milieu d'une fonction commence à la déclaration et se termine à la fin du bloc dans lequel est faite la déclaration.

Exemple.

```
int main() {
    int i;
    ...
    i = 3;
    int y = 3 * i;
    ...
}
```

ou encore

```
for (int i=0; i < N; i++) { ... }
```

Dans les dernières versions de C++, la variable i est ici considérée comme locale au **for**.

1.1.4 Allocation et désallocation mémoire

L'allocation dynamique de la mémoire en C++ est faite par l'opérateur **new** :

```
int *ad;
ad = new int;
```

ou encore

```
int *ad = new int;
```

Pour allouer un tableau dynamique de *n* entiers, on écrira par exemple :

```
int *tabdyn;
int n;
...
n = 100;
tabyn = new int[n];
```

Pour désallouer une zone mémoire allouée dynamiquement par **new**, le programmeur doit utiliser l'opérateur **delete** :

```
delete ad; // désalloue la zone pointée par ad
delete tabdyn; // désalloue la zone allouée pour le tableau
```

Remarques.

1. En C++, comme en C, la gestion dynamique de la mémoire est déléguée au programmeur (allocation et désallocation).
2. Pour désallouer un tableau d'objets, on utilisera :

   ```
   delete [] tabObj;
   ```

3. L'allocation et la désallocation d'objets font appel aux notions de constructeur et de destructeur.

1.1.5 Les constantes

En C++, le mot clé **const** est utilisé pour déclarer des constantes :

```
const int i = 10;
const float t[3] = { 1.0, 6.2, 7.3 };
```

Dans le second exemple, les valeurs du tableau sont constantes. Par exemple, l'affectation $t[2] = 5.3$ est interdite. Une variable déclarée **const** doit être initialisée à la définition.

1.1.6 Paramètres par défaut

Il est possible en C++, lors de la définition d'une fonction, d'attribuer une valeur par défaut à un ou plusieurs paramètres.

Exemple.

```
float prixTTC(float prixHT, float TVA=19.6)
  { ... }

int main () {
  float res1, res2;
  ...
  res1 = prixTTC(100, 5.5); // la valeur 5.5 remplace la valeur
                            // par défaut
  res2 = prixTTC(100); // la valeur pour la TVA sera 19.6
  ...
  return 0;
}
```

Les paramètres ayant une valeur par défaut doivent obligatoirement être déclarés en fin de liste pour éviter les ambiguïtés.

1.1.7 Les opérateurs d'entrée et de sortie

Les entrées et sorties en C++ sont gérées par les deux opérateurs >> et << respectivement.

Exemples.

```
// permet la lecture à partir du clavier (le flux d'entrée cin)
// de la valeur qui sera stockée dans la variable x
cin >> x;

// affiche à l'écran (le flux de sortie cout) les chaînes
// de caractères et les valeurs citées
cout << "Voici le nombre : " << x
     << " et son carre est : " << x * x << endl;
```

1.1.8 Notion de référence

Passage de paramètres par référence

En C, il n'existe qu'un mode de passage de paramètre, c'est le passage par valeur : une copie de la valeur du paramètre est transmise à la fonction. Il est possible de transmettre des valeurs d'adresse pour permettre la modification des variables de l'appelant.

En C++, il existe aussi le passage par référence, dans lequel une référence vers la variable de l'appelant est transmise à la fonction qui a donc accès directement à cette variable. L'opérateur de référence en C++ est noté & et est utilisé dans les définitions de fonction (ou de variable).

Exemple. Une fonction échange en C :

```
void swap(int *a, int *b) {
  int c;
  c = *a;
  *a = *b;
  *b = c;
}
```

En C++ :

```
void swap(int &a, int &b) {
  int c;
  c = a;
  a = b;
  b = c;
}
```

Autre intérêt du passage par référence : éviter la recopie d'une structure prenant beaucoup de place en mémoire. Pour empêcher la modification d'une telle structure dans la fonction, on utilise le mot clé **const** devant le paramètre dans l'en-tête de la fonction.

Exemple. `int f(const typeStruct & maStruct) { ... }`

Variable de type référence

En C++, on peut déclarer des variables « synonymes » en déclarant l'une comme référence de l'autre.

Exemple.

```
int main() {
  int nb = 1;
  int &alias = nb;
  int *ptr;
  alias = 10; // la variable désignée par alias et nb vaut maintenant 10
  nb = 20; // équivaut à : alias = nb = 20;
  ptr = &alias; // ptr pointe vers alias, donc vers nb
  *ptr = 30; // *ptr = alias = nb = 30;
}
```

Fonction retournant une référence

En C++, une fonction peut retourner une référence sur une variable (à condition qu'elle ne soit pas locale à la fonction).

Exemple.

```
const int L=3;
const int C=2;
void ecrire(int tab[L][C]);
int & place(int tab[L][C], int);

int main(void) {
  int t[L][C];
  for (int i=0; i < L; ++i)
    for (int j=0; j < C; ++j)
      t[i][j] = i + 2 * j;
  ecrire(t);
  place(t,3) = 0;
  ecrire(t);
  return 0;
```

```
}

int & place(int tab[L][C], int val) {
  for (int i=0; i < L; ++i)
    for (int j=0; j < C; ++j)
      if (tab[i][j] == val)
        return tab[i][j];
  return tab[0][0];
}

void ecrire(int tab[L][C]) {
  for (int i=0; i < L; ++i) {
    for (int j=0; j < C; ++j) {
      cout << tab[i][j] << " ";
      cout << endl;
    }
  }
}
```

Résultats affichés :

```
0 2
1 3
2 4
0 2
1 0
2 4
```

Le premier 3 trouvé, et renvoyé par référence, a été changé en 0.

1.2 Construction d'une classe en C++

1.2.1 La programmation orientée objet

Un objet associe données et traitement. Un objet laisse visible une interface représentant les opérations que l'on peut effectuer dessus.

Une classe est un modèle à partir duquel on peut définir des instances correspondant à des objets particuliers.

Avantages :

- découpage d'un problème en briques logicielles, facilement réutilisables,
- sécurité due à l'encapsulation (définir une partie privée non accessible à l'utilisateur),
- possibilité d'évolution des implémentations sans changer l'interface publique,
- meilleure gestion du travail en équipe.

1.2.2 Définition d'une classe en C++

```
class Exemple {
  private: // facultatif
    // membres cachés hors d'un objet de cette classe
    type1 donnee_membre1;
    ...
    type_2 fonction_membre2(...);
```

```
  public:
    // membres accessibles hors d'un objet de la classe
    type3 donnee_membre3;
    type4 fonction_membre4(...);
};
```

Pour un objet x de la classe Exemple, l'accès à une donnée ou une fonction membre se fait par l'opérateur . :

```
x.donnee_membre3
```

désigne la donnee_membre3 de l'objet x, instance de la classe Exemple.

Pour définir la fonction membre fonction_membre4() de la classe Exemple, on utilisera la notation :

```
type4 Exemple::fonction_membre4(...) { ... }
```

Exemple d'une classe en C++

```
#include <iostream>

using namespace std;

class Vecteur {
  int taille;
  float *valeur;

  public:
    void initialise(int, float);
    void ajoute(int, float);
    void affiche();
};

void Vecteur::initialise(int a, float b) {
  taille = a;
  valeur = new float[taille];
  for (int i=0; i < taille; ++i)
    valeur[i] = b;
}

void Vecteur::affiche() {
  for (int i=0; i < taille; ++i)
    cout << valeur[i] << " ";
  cout << endl;
}

void Vecteur::ajoute(int i, float b) {
  valeur[i] += b;
}

int main() {
  Vecteur v;
  v.initialise(10, 2.7);
  v.affiche();
  return 0;
}
```

Remarques. 1. Chaque classe possède automatiquement une donnée cachée : le pointeur **this** qui pointe vers l'objet courant. Par exemple, ajoutons la fonction membre suivante à la classe Vecteur :

```
class Vecteur {
  ...
  public:
  ...
    bool estLeMeme(const Vecteur &);
  ...
};
```

que l'on définit par :

```
bool estLeMeme(const Vecteur &v) {
  // this contient l'adresse de l'objet courant
  return this == &v;
}
```

Cette fonction permet de savoir si l'objet courant, pointé par **this**, et l'objet passé en paramètre sont le même objet en mémoire.

2. On peut déclarer des membres à l'aide du mot clé **static** dans une classe. Ces membres sont alors communs à tous les objets. Si un objet change la valeur d'une donnée déclarée avec **static** alors cette donnée est modifiée pour tous les objets de la classe. Par exemple, un compteur d'objet dans une classe :

```
static int nbobjets;
```

L'initialisation de cette variable devra se faire à l'extérieur de la classe :

```
int Vecteur::nbobjets = 0;
```

Par défaut, une variable déclarée avec **static** est initialisée à 0.

Constructeurs et destructeur

Lorsqu'on définit un objet, un constructeur est appelé automatiquement par le langage après la réservation mémoire de l'objet. De même lorsqu'un objet n'existe plus (hors de portée de son bloc de définition), un destructeur est appelé automatiquement. L'écriture d'un destructeur est nécessaire chaque fois qu'un objet a une donnée membre de type pointeur avec allocation dynamique. Le constructeur est déclaré **public** et porte le nom de la classe.

Exemple. Dans la classe Vecteur, la fonction membre initialise () sera remplacée par un constructeur :

```
class Vecteur {
  int taille;
  float *valeur;

  public:
    Vecteur(int, float); // constructeur
    void ajoute(int, float);
    void affiche();
};
```

```
...

Vecteur::Vecteur(int a, float b) {
  taille = a;
  valeur = new float[taille];
  for (int i=0; i< taille; ++i)
    valeur[i] = b;
}

...

int main() {
  Vecteur v(10, 2.7);
  v.affiche();
  return 0;
}
```

Dans la définition d'un constructeur, on peut faire suivre l'en-tête par des affectations de donnée membres de la manière suivante :

```
Vecteur::Vecteur(int a, float b) : taille(a) {
  valeur = new float[taille];
  ...
}
```

Plusieurs affectations peuvent éventuellement être effectuées, séparées par des virgules.

Il peut y avoir plusieurs constructeurs pour une classe. Ils portent tous le même nom et se différencient par le nombre ou le type de leurs arguments. On parle alors de surcharge.

Exemple. Dans la classe Vecteur :

```
Vecteur::Vecteur() {
  taille = 1;
  valeur = new float;
}

int main() {
  Vecteur v1; // appelle le constructeur sans arguments
  Vecteur v2(5, 1.0); // appelle le constructeur à deux arguments
}
```

Il est parfois nécessaire de définir un destructeur, qui se nomme obligatoirement :

```
~nom_de_la_classe
```

Exemple. Dans la classe Vecteur, il faut définir un destructeur qui libère l'espace pointé par valeur, alloué dynamiquement dans les constructeurs.

```
class Vecteur {
  int taille;
  float *valeur;
```

```
  public:
    Vecteur(int, float);
    Vecteur();
    ~Vecteur();
};

Vecteur::~Vecteur() {
  delete valeur;
}
```

Lorsqu'on veut initialiser un objet avec un autre objet de même classe, un constructeur particulier est appelé automatiquement par le langage. De la même manière, lorsqu'une fonction reçoit un paramètre, ou retourne en résultat un objet, le constructeur de copie par défaut est invoqué automatiquement. Ce constructeur de copie réalise une copie membre à membre, dite de surface, de l'objet. Cela pose problème pour les objets avec des données membres de type pointeur nécessitant une allocation dynamique.

Exemple.

```
Vecteur v1(3,1);
Vecteur v2 = v1; // ou Vecteur v2(v1);
```

v1 et v2 ne sont pas disjoints : si l'on modifie v1, on modifie v2 ; si l'on détruit v2, v1 n'a plus ses composantes, la mémoire étant récupérée par le destructeur de v2.

Exemple. Dans la classe Vecteur :

```
// prototype du constructeur de copie :
Vecteur(const Vecteur &);
// passage par référence nécessaire pour éviter la copie de v
// et sa destruction en fin de fonction

// définition :
Vecteur::Vecteur(const Vecteur &v) {
  taille = v.taille;
  valeur = new float[taille];
  for (int i=0; i < taille; ++i)
    valeur[i] = v.valeur[i];
}
```

Les constructeurs et le destructeur peuvent être définis directement dans la déclaration de la classe. On parle alors de fonction **inline**. Chaque appel d'une fonction **inline** est remplacé par le code du corps de la fonction, ce qui gagne du temps à l'éxécution mais augmente la taille du code généré.

```
class Vecteur {
  int taille;
  float * valeur;

  public:
    Vecteur(int n=1, float b=0.) {
      taille = n;
      valeur = new float[taille];
```

```
    for (int i=0; i < taille; ++i)
      valeur[i] = b;
  }

  Vecteur(const Vecteur &v) {
    taille = v.taille;
    valeur = new float[taille];
    for (int i=0; i < taille; ++i)
      valeur[i] = v.valeur[i];
  }

  ~Vecteur() {
    delete valeur;
  }
}
```

Surcharge des opérateurs

La surcharge de fonction consiste à définir plusieurs fonctions de même nom mais se différenciant par leurs paramètres (en nombre ou en type).

Surcharge d'opérateurs arithmétiques Les opérateurs arithmétiques classiques $(+, -, *, /)$ peuvent être surchargés pour s'appliquer à des objets d'une classe donnée.

Exemple. Dans la classe Vecteur, on souhaite pouvoir écrire l'expression, de type Vecteur, v1 + v2 où v1 et v2 sont deux vecteurs. Il y a deux possibilités pour cela :

1. On définit une fonction **operator**+ dans la classe Vecteur :

   ```
   Vecteur
   operator+(const Vecteur &v1, const Vecteur &v2) {
     // addition de deux vecteurs supposés de même taille
     Vecteur v(v1.taille);
     for (int i=0; i < taille; ++i)
       v.valeur[i] = v1.valeur[i] + v2.valeur[i];
     return v;
   }
   ```

   Cette fonction sera déclarée amie de la classe Vecteur.

2. On définit une fonction membre de la classe Vecteur : v1 + v2 sera interprété comme v1.**operator**+(v2). La déclaration du prototype dans la définition de la classe s'écrira :

   ```
   class Vecteur {
     ...
     public:
     ...
     Vecteur operator+(const Vecteur &v) const;
   }
   ```

   et la fonction membre s'écrit :

   ```
   Vecteur Vecteur::operator+(const Vecteur &v) const {
     // addition de vecteur v au vecteur courant,
   ```

```
    // supposés de même taille
    Vecteur resultat(taille);
    for (int i=0; i < taille; ++i)
      resultat.valeur[i] = valeur[i] + v.valeur[i];
    return resultat;
  }
```

Remarques.
1. Le paramètre v est transmis par référence pour éviter une copie et avec le mot clé **const** pour empêcher toute modification.
2. Le mot clé **const** en fin de prototype et d'en-tête de la fonction indique que le vecteur courant peut être un vecteur constant. Cela permet d'additionner des vecteurs constants.

Surcharge de l'opérateur d'affectation Lors de l'affectation entre objets, un opérateur d'affectation par défaut est prévu par le langage, mais fait, comme le constructeur de copie par défaut, une copie du surface de l'objet. On obtiendra donc deux objets partageant leurs zones allouées dynamiquement pour des pointeurs membres. Il faut alors surcharger l'opérateur d'affectation pour les classes utilisant des structures dynamiques. L'affectation v1 = v2 sera interprétée comme v1.**operator**=(v2). Dans la classe Vecteur :

```
class Vecteur {
  ...
  Vecteur & operator=(const Vecteur &v);
}
```

Définition :

```
Vecteur & Vecteur::operator=(const Vecteur &v) {
  if (this != &v) {
    if (taille != v.taille) {
      delete valeur;
      valeur = new float[taille = v.taille];
    }
    for (int i=0; i < taille; ++i)
      valeur[i] = v.valeur[i];
  }
  return *this;
}
```

Remarques.
1. La transmission de l'argument doit se faire par référence car on a besoin de l'adresse de l'objet transmis pour le test **this** != &v qui permet de s'assurer qu'on ne fait pas l'affectation d'un objet sur lui-même. Si c'est le cas, il ne faut surtout pas détruire cet objet.
2. Le qualificatif **const** du paramètre permet d'avoir en argument, donc en partie droite d'affectation, un vecteur constant. Le mot clé **const** assure également qu'il n'y aura pas de modification de l'argument.
3. L'opérateur d'affectation renvoie une référence pour permettre les affectations en cascade du type : v1 = v2 = v3.

Forme canonique d'une classe Lorsqu'une classe gère des pointeurs et des structures dynamiques, il est nécessaire de la munir d'au moins les quatre fonctions membres suivantes :

- constructeur
- destructeur
- constructeur de copie
- opérateur d'affectation

qui permettent une gestion propre de la mémoire utilisée par les objets.

```
class T {
  ...
  public:
    T(...); // constructeur
    T(const T &); // constructeur de copie
    ~T(); // destructeur
    T & operator=(const T &); // opérateur d'affectation
};
```

Une manière élégante d'écrire ces fonctions est la suivante :

```
class T {
  public:
    T(...);
    T(const T &);
    ~T();
    T & operator=(const T &);
};

T::T(const T &o) { copy(o); }
T::~T() { free(); }
T & T::operator=(const T &o) {
  if (this != &o) {
    free();
    copy(o);
  }
  return *this;
}
```

Les deux seules fonctions à écrire spécifiquement pour la classe T, ou pour toute autre classe, sont les fonctions copy() et free() qui seront des fonctions membres privées de la classe.

Surcharge des opérateurs d'accès On souhaite avoir accès à l'une des composantes d'un vecteur. Cela peut se faire par une fonction membre, mais également par une surcharge de l'opérateur d'accès []. Cela permettra d'écrire v[i] plutôt que v.valeur[i], d'autant que valeur est une donnée privée de Vecteur.

Dans la classe Vecteur :

```
class Vecteur {
  ...
  public:
    float & operator[](int);
  ...
};
```

Définition :

```
float & Vecteur::operator[](int i) {
  return valeur[i];
}
```

Le fait de retourner une référence permet d'utiliser le résultat à gauche d'une affectation :

```
v[i] = 3.2; // v.operator[](i) = 3.2;
```

Pour pouvoir manipuler des vecteurs constants, donc interdits d'affectation, on écrira une seconde surcharge de l'opérateur [] ne retournant pas de référence :

```
// prototype
float operator[](int) const;

// définition
float Vecteur::operator[](int i) const {
  return valeur[i];
}
```

Cette surcharge de l'opérateur [] peut être utilisé dans les autres fonctions membres :

```
Vecteur Vecteur::operator+(const Vecteur &v) {
  Vecteur resultat(taille);
  for (int i=0; i < taille; ++i)
    resultat[i] = (*this)[i] + v[i];
  return resultat;
}
```

Fonctions et classes amies

On dit qu'une fonction est amie d'une classe si elle peut accéder toute la partie privée de la classe. On dit qu'une classe est amie d'une autre classe si toutes ses fonctions peuvent accéder à la partie privée de cette classe. Les déclarations d'amitié doivent se faire dans la classe qui autorise l'accès à sa partie privée.

```
class B {
  ...
  friend void fonction(...);
  friend class C;
};
```

La notion d'amitié est contraire à la notion d'encapsulation, donc à utiliser uniquement si nécessaire.

Exemple. On souhaite définir, dans la classe Vecteur, la multiplication d'un vecteur par un scalaire. La multiplication par un scalaire à droite sera une fonction membre de la classe Vecteur. L'expression v * a, avec v étant un vecteur et a un réel, sera interprétée comme v.**operator***(a). Mais pour la multiplication par un scalaire à gauche, a * v, le vecteur n'est plus le premier argument de * et donc la fonction ne pourra pas être membre de la classe Vecteur. Il faut donc définir une surcharge de l'opérateur * hors de la classe Vecteur :

```
Vecteur operator*(float alpha, const Vecteur &v) {
  Vecteur result(taille);
  for (int i=0; i < taille; ++i)
    result[i] = alpha * v[i];
  return result;
}
```

Cette fonction accède aux données privées de Vecteur, elle doit donc être amie de la classe Vecteur. Cette déclaration se fait dans la classe Vecteur :

```
class Vecteur {
  public:
    friend Vecteur operator*(float, const Vecteur &);
};
```

Pour éviter que la surcharge de l'opérateur $*$ pour la multiplication d'un Vecteur par un scalaire à gauche ne soit déclarée amie de la classe Vecteur, on ajoute une fonction membre size() retournant la taille d'un vecteur. Dans la classe Vecteur :

```
int size() const;
```

Définition hors de la classe :

```
int Vecteur::size() const {
  return taille;
}
```

v. taille dans la surcharge de $*$ sera remplacé par v.size().

Surcharge des opérateurs d'entrée et de sortie

cin (resp. cout) est un objet appelé flot prédéfini d'entrée (resp. de sortie) appartenant à la classe istream (resp. ostream) de flots d'entrée (resp. de sortie).

Écrire

```
cin >> var;
```

correspond à l'appel de l'opérateur >> :

```
operator>>(cin, var);
```

Pour surcharger >>, il faut nécessairement écrire une fonction hors classe qui aura le prototype suivant :

```
istream & operator>>(istream & entree, nom_classe & objet);
```

Le flot d'entrée entree peut être cin ou bien un fichier. On peut tester si ce flot est le flot prédéfini d'entrée cin ou non :

```
if (&cin == &entree) ...
```

La fonction de surcharge de >> devra se terminer par :

```
return entree;
```

On procède de la même manière pour l'opérateur de sortie <<.

Exemple. Dans la classe Vecteur, on souhaite surcharger l'opérateur d'entrée >> de manière à permettre la lecture d'un vecteur à partir du clavier ou à partir d'un fichier.

```
istream & operator>>(istream & entree, Vecteur &v) {
  int ntaille;

  if (&entree == &cin)
    cout << "donner la taille du vecteur : ";
  entree >> ntaille;
  Vecteur nv(ntaille);
  if (&entree == &cin)
    cout << "donner les composantes du vecteur" << endl;
  for (int i=0; i < ntaille; ++i)
    entree >> nv[i];
  v = nv;
  return entree;
}
```

Grâce à la variable ntaille et à la surcharge de l'opérateur d'accès [], la fonction surcharge de l'opérateur >> n'a pas besoin d'être amie de la classe Vecteur.

Exercice. Écrire la surcharge de l'opérateur de sortie << pour la classe Vecteur.

Utilisation des fichiers

Exemple.

```
#include <fstream>
using namespace std;
#include "Vecteur.h";

int main() {
  Vecteur v1, v2;
  ifstream in;
  otstream out;

  in.open("input.dat");
  out.open("output.dat");

  in >> v1;
  in >> v2;
  Vecteur v3 = 2 * v1 + v2;
  out << "v1 : " << endl << v1;
  out << "v2 : " << endl << v2;
  out << "calcul de 2 * v1 + v2 : " << endl << v3;

  in.close();
  out.close();

  return 0;
}
```

Fichier input.dat :

```
4
2 3 4 1
4
1 2 3 5
```

Fichier output.dat :

```
v1 :
taille du vecteur : 4
composantes :
2 3 4 1
v2 :
taille du vecteur : 4
composantes :
1 2 3 5
calcul de 2 * v1 + v2 :
taille du vecteur : 4
composantes :
5 8 11 7
```

1.3 Généricité

C++ permet la généricité, c'est à dire l'écriture de fonctions et de classes dépendant d'un ou plusieurs types non précisés au moment de la définition, et spécifiés lors de l'utilisation de ces fonctions et classes.

Exemple d'une fonction minimum :

– sur des entiers :

```
int min(int a, int b) {
  return a > b ? b : a;
}
```

– sur des réels :

```
float min(float a, float b) {
  return a > b ? b : a;
}
```

Duplication de code identique dans le corps des deux fonctions.

Écriture d'une fonction min() générique :

```
template <typename T> T min(T a, T b) {
  return a > b ? b : a;
}
```

Utilisation :

```
int main() {
  int n=4, p=12;
  float x=3.5, y=7.2;

  cout << "minimum de" << n << " et "
       << p << " : " << min(n,p) << endl;
  cout << "minimum de" << x << " et "
       << y << " : " << min(x,y) << endl;

  return 0;
}
```

À l'appel de min(), le compilateur reconnaît le type des arguments et construit l'appel de min avec ce type à la place du type générique T. Le type générique T doit apparaître pour au moins un argument de la fonction. T peut être remplacé

par n'importe quel type, y compris un type classe, pourvu que la fonction ait un sens sur les objets de cette classe. Dans l'exemple de la fonction min(), il faut avoir surchargé l'opérateur < pour les objets de la classe.

Patron de classe On peut également définir des classes dépendant d'un ou plusieurs types génériques.

Exemple. Une classe Vecteur générique :

```
template <typename T> class Vecteur {
  private:
    int taille;
    T *valeur;
  public:
    Vecteur(int=1, T=0);
    Vecteur(const Vecteur &);
    ~Vecteur();
    Vecteur & operator=(const Vecteur &);
    ...
    T & operator[](int) const;
    Vecteur operator*(T);
};
```

Pour définir les fonctions membres en dehors de la classe, on écrira :

```
template <typename T>
type_resultat Vecteur<T>::nom_fonction(...) {...}
```

Par exemple :

```
template <typename T>
T & Vecteur<T>::operator[](int i) const {
  return valeur[i];
}
```

Utilisation de la classe générique :

```
int main() {
  Vecteur<int> x(10);
  Vecteur<float> y(20);
  // les deux vecteurs sont ensuite utilisés
  // comme des vecteurs d'une classe non générique

  return 0;
}
```

Si l'on imbrique des templates, il faut mettre un espace entre les chevrons finaux pour éviter les écritures ambiguës :

```
Vecteur < Vecteur<int> > x; // x est un vecteur de vecteurs d'entiers
```

Remarque. Il est possible d'avoir plusieurs types génériques dans une définition de classe ou de fonction.

```
template <typename U, typename V, typename T> class Exemple
{
  T x;
```

```
  U y;
  V z;
};

int main() {
  Exemple<int, int, float> obj;
  return 0;
}
```

Spécialisation dans un patron de classe Il est possible de spécialiser une fonction membre d'une classe générique pour un type particulier.

Exemple.

```
template <typename T> class Point {
  T x;
  T y;

  public:
    Point(T a=0, T b=0) { // Point(T a=0, T b=0) : x(a), y(b)
      x = a; y = b;
    }
    void affiche();
};

template <typename T> void Point<T>::affiche() {
  cout << x << " " << y << endl;
}

// ajout d'une fonction affiche() spécialisée pour le type char
void Point<char>::affiche() {
  cout << (int)x << " " << (int)y << endl;
}
```

Utilisation :

```
int main() {
  Point<int> a(3,5);
  Point<char> b('c','d');

  a.affiche(); // fonction affiche() générique avec le type int
  b.affiche(); // fonction affiche() spécifique au type char

  return 0;
}
```

Chapitre 2

L'héritage en C++

C'est un concept fondamental de la programmation orientée objet. L'héritage permet une construction de classes par affinage. Une classe (dite dérivée en C++) peut hériter d'une (ou plusieurs) classe, dite de base en C++, c'est à dire avoir les propriétés de celle-ci en plus des propriétés propres à elle-même.

2.1 Exemple d'héritage simple

Une classe PointCol hérite d'une classe Point :

```
class Point {
  int x;
  int y;

  public:
    void initialise(int,int);
    void deplace(int,int);
    void affiche();
};

void Point::deplace(int deltax, deltay) {
  x += deltax;
  y += deltay;
}

class PointCol : public Point {
  char couleur;
  public:
    void colore(char cl) {
      couleur = cl;
    }
};
```

2.2 Relations entre classes de base et dérivées

```
class PointCol : public Point
```

Signifie que PointCol hérite de Point et que cet héritage est **public**, c'est à dire que les membres de PointCol ont accès aux membres publics de Point mais pas aux membres privés. Ajout, dans la classe PointCol, d'une fonction d'affichage d'un point coloré :

```
void PointCol::afficheCol() {
  affiche(); // appel de affiche() de Point
  cout << "couleur : " << couleur << endl;
}
```

Code de affiche() dans Point :

```
void Point::affiche() {
  cout << "x : " << x ", y : " << y << endl;
}
```

2.3 Redéfinition des fonctions membres

On souhaite avoir dans la classe PointCol une fonction affiche(), de même nom que dans la classe de base. On peut pour cela redéfinir la fonction affiche() pour les points colorés. Si, dans cette fonction, on veut utiliser la fonction affiche() de la classe Point, on utilisera l'opérateur de résolution de portée. Ce sera aussi possible dans l'utilisation de cette fonction.

Définition de affiche() dans la classe PointCol :

```
void PointCol::affiche() {
  Point::affiche();
  cout << "couleur : " << couleur << endl;
}
```

Utilisation de affiche() :

```
PointCol pc;
...
pc.affiche(); // c'est PointCol : :affiche() qui est appelée sur pc
pc.Point::affiche(); // appel de Point : :affiche() sur pc
```

2.4 Constructeurs et destructeurs

```
class Base {
  ...
  public:
  Base(...);
  ~Base();
  ...
};

class Derivee : public Base {
  ...
  public:
    Derivee(...);
    ~Derivee();
  ...
};
```

La création d'un objet de la classe Derivee nécessite d'abord la création d'un objet de la classe Base. Le constructeur de la classe Base sera appelé automatiquement avant celui de la classe Derivee. À l'inverse, lors de la destruction d'un objet de la classe Derivee, c'est le destructeur de la classe Derivee qui sera appelé automatiquement en premier, puis celui de la classe Base. Les destructeurs sont donc appelés du plus spécifique au moins spécifique, l'ordre inverse d'appel des constructeurs.

2.4.1 Cas des constructeurs avec arguments

```
class Point {
  int x;
  int y;

  public:
    Point(int,int);
};

class PointCol : public Point {
  char couleur;

  public:
    PointCol(int,int,char);
}
```

Pour indiquer que le constructeur de PointCol doit transmettre ses deux premiers arguments au constructeur de Point, il faut écrire l'en-tête de PointCol() de la manière suivante :

```
PointCol::PointCol(int abs, int ord, char cl)
: Point(abs, ord)
{
  couleur = cl;
}
```

2.5 Contrôle des accès

Dans une dérivation **public** :
- Toute fonction membre de la classe dérivée a accès aux membres **public** de la classe de base.
- Tout objet instance de la classe dérivée a accès aux membres **public** de sa classe de base.

2.5.1 La notion de protection

Pour permettre l'accès, pour une classe dérivée, à certains membres privés de la classe de base, il faut déclarer ces membres **protected** dans la classe de base.

Exemple.

```
class Point {
  protected:
    int x;
    int y;

  public:
```

```
    Point(...);
    void affiche();
};

class PointCol : public Point {
  char couleur;

  public:
    void affiche() {
      cout << "position : " << x << ", " << y;
      cout << "et couleur : " << couleur << endl;
    }
}
```

Remarques.
1. Les fonctions amies d'une classe dérivée ont les mêmes autorisations d'accès que les fonctions membres de cette classe (notamment pour l'accès aux membres protégés).
2. Les déclarations d'amitié ne s'héritent pas.

2.5.2 Dérivation publique, protégée ou privéé

Dérivation publique

Situation classique d'héritage.

Dérivation privée

Elle interdit aux instances d'une classe dérivée l'accès aux membres **public** de la classe de base.

Exemple. Si PointCol hérite de Point de manière privée, alors une instance de PointCol ne peut pas appeler la fonction membre deplace(), qui est pourtant un membre **public** de Point.

C'est utile quand les méthodes d'une classe sont redéfinies entièrement dans une classe dérivée, il n'y a pas lieu de donner un accès aux méthodes de la classe de base.

Dérivation protégée

Dans ce mode de dérivation, les membres **public** de la classe de base seront **protected** dans la classe dérivée.

Remarque. Il ne faut pas confondre le mode de dérivation et le statut des membres d'une classe.

2.6 Constructeur de copie

Si une classe B hérite d'une classe A, si B n'a pas de constructeur de copie, le constructeur de copie par défaut est appelé. Pour la partie héritée de A de toute instance de B, le constructeur de copie de A sera appelé s'il existe, sinon ce sera le constructeur de copie de défaut.

Si B possède un constructeur de copie, celui-ci sera appelé mais pas celui de A. Autrement dit, le constructeur de copie d'une classe dérivée devra prendre en charge la totalité de la copie de l'objet, et non plus seulement de sa partie héritée.

On peut toutefois utiliser un constructeur de A dans le constructeur de copie de B :

```
B(B &x) : A(...)
```

Dans ce cas, si l'on transmet au constructeur de A l'objet x lui-même, il y a alors conversion de x au type de A et donc appel du constructeur de copie de A.

```
B(B &x) : A(x) // conversion de x au type A, transmission
                // au constructeur de copie de A
{
  // copie de la partie spécifique à B
}
```

2.7 Opérateur d'affectation

Si B hérite de A,

1. Si B n'a pas de surcharge de l'opérateur =, l'affectation se fait alors membre à membre. La partie héritée de A étant gérée par l'affectation définie dans A (surchargée ou par défaut).
2. Si B possède une surcharge de l'opérateur =, alors seule l'affectation de B sera appelée et pas celle de A. L'opérateur = surchargé dans B doit donc prendre en charge totalement l'affectation d'objets de B, même pour leur partie héritée de A.

2.8 Héritage et patron de classe

Trois cas sont possibles :

1. Classe « ordinaire » dérivée d'une classe patron, c'est à dire d'une « instance » particulière d'un patron de classe. Soit A un patron de classe :

   ```
   template <typename T> class A { ... };
   ```

   alors la déclaration

   ```
   class B : public A<int> { ... };
   ```

   désigne une seule classe B héritant de la classe A<**int**>.
2. Patron de classe dérivé d'une classe « ordinaire ».

   ```
   template <typename T> class B : public A { ... };
   ```

   On obtient une famille de classes (dépendant du type T). L'aspect générique est introduit au moment de la dérivation.
3. Patron de classe dérivé d'un patron de classe. Soit A défini par

   ```
   template <typename T> class A { ... };
   ```

   On peut alors :

– définir une nouvelle famille de classes dérivées par :

```
template <typename T> class B : public A<T>
  { ... };
```

On obtient autant de classes dérivées que de classes de base possibles.

– définir une nouvelle famille de classes dérivées par la déclaration suivante :

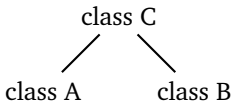
```
template <typename T, typename U>
  class B : public A<T>
```

Chaque classe de base possible peut engendrer une famille de classes dérivées (dépendant du type générique U).

2.9 Héritage multiple

C++ offre la possibilité de l'héritage multiple, c'est à dire qu'une classe peut dériver de plusieurs classes de base.

2.9.1 Mise en œuvre



Déclaration :

```
class C : public A, public B { ... };
```

Exemple.

```
class PointColore : public Point, public Couleur { ... };

class Point {
  protected:
    int x,y;
  ...
};

class Couleur {
  protected:
    char couleur;
  ...
};
```

Les attributs x, y et couleur pourraient être déclarés **private** si l'on ne souhaite pas que les instances des classes dérivées y accèdent.

Les constructeurs de classes de base sont appelés dans l'ordre précisé dans la déclaration d'héritage de la classe dérivée, et avant celui de la classe dérivée. Les destructeurs sont appelés dans l'ordre inverse.

Pour un appel de fonction, il est possible de préciser à quelle classe appartient la fonction membre appelée :

Exemple. Si la classe Point et la classe Couleur ont chacune une fonction membre affiche (), on peut alors définir une fonction membre affiche () dans la classe PointColore comme suit :

```
void PointColore::affiche() {
  Point::affiche();
  Couleur::affiche();
}
```

Remarque. Le mode de dérivation (publique, protégée ou privée) doit être précisé pour chaque classe de base dont hérite une classe dérivée.

On peut également distinguer les données membres de clases de base dans une classe dérivée :

```
class A {
  protected:
    int x;
  ...
};

class B {
  protected:
    int x;
  ...
};

class C : public A, public B {
  int x;
  ...
};
```

C possède par héritage deux données membres x. Pour les distinguer, on utilisera A::x pour la donnée membre x héritée de A, et B::x pour celle héritée de B. Ceci sera vrai également pour les utilisateurs de C si les données membres x de A et de B sont publiques.

2.9.2 Classes déclarées virtuelles dans l'héritage

Dans le schéma d'héritage multiple suivant :

```
class A {
  protected:
    int x;
  ...
};
class B : public A { ... };
class C : public A { ... };
class D : public B, public C { ... };
```

D hérite deux fois de A. Les fonctions membres de A ne sont pas dupliquées dans D (elles existent au niveau de A). En revanche, les données membres de A apparaîtront deux fois dans D. POur les distinguer, on écrira : A::B::x et A::C::x, ou directement B::x et C::x si ni B ni C ne contiennent de donnée membre x.

Le plus souvent, on ne souhaite pas cette duplication. Pour l'éviter, on déclarera A virtuelle (mot clé **virtual**) au niveau de l'héritage :

```
class B : public virtual A { ... };
class C : public virtual A { ... };
class D : public B, public C { ... };
```

Cette déclaration de A **virtual** pour B et C signifie que les données membres de A ne seront introduites qu'une seule fois dans les descendants communs éventuels de B et C. Cela ne change rien sur le comportement de A, B et C.

S'il y a transmission d'arguments à faire entre constructeurs, cela doit se faire au niveau de D pour B et A ou pour C et A :

```
D(int n, int p, int q) : B(n,p,q), A(n,p) { ... };
```

Il faut alors prévoir un constructeur sans argument dans A, car il n'est plus possible de transmettre des informations des constructeurs de B ou de C vers celui de A.

Les constructeurs des classes déclarées **virtual** seront toujours appelés en premier.

2.10 Héritage et typage dynamique : les fonctions membres virtuelles

Il y a deux situations où le typage dynamique est nécessaire.

2.10.1 Utilisation de pointeurs

```
class Point {
  protected:
    int x,y;
  public:
    void affiche();
  ...
};

class PointCol : public Point {
  protected:
    char couleur;
  public:
    void affiche();
};
```

Si l'on définit :

```
Point p;
PointCol pc;
Point *adp = &p;
```

alors l'appel

```
adp->affiche(); // (*adp).affiche();
```

appelle naturellement la méthode affiche() de Point. En revanche, si l'on fait

```
adp = &pc;
```

ce qui est possible puisqu'un PointCol est un Point, alors l'appel

```
adp->affiche();
```

appellera encore la méthode affiche() de Point car adp est de type Point* et l'appel de la fonction a été prévu à la compilation donc en fonction du type Point* de adp (typage statique).

Pour permettre le typage dynamique, et donc l'appel de affiche() de PointCol, il faut déclarer affiche() **virtual** dans Point, donc dans la classe de base.

```
class Point {
  protected:
    int x,y;
  public:
    virtual void affiche();
};

class PointCol {
  public:
    void affiche();
};

int main() {
  Point p;
  PointCol pc;
  Point *adp = &p;
  PointCol *adpc = &pc;
  adp->affiche(); // appelle affiche() de Point
  adpc->affiche(); // appelle affiche() de PointCol
  adp = adpc; // l'affectation inverse serait rejetée
  adp->affiche(); // appelle affiche() de PointCol
}
```

Remarque. En Java, la ligature est toujours dynamique.

2.10.2 Délégation

Une fonction membre affiche() appelle une autre fonction membre identifie() qui précisera ce qu'il y a de particulier à faire pour chaque classe dérivée. Seule cette fonction membre idenfie() sera redéfinie dans les classes dérivées, mais pas affiche().

```
class Point {
  ...
  public:
    void idenfie() {
      cout << "Je suis un point" << endl;
    }

    void affiche() {
      identifie();
      cout << "x, y : " << x << ", " << y << endl;
    }
};

class PointCol : public Point {
  ...
```

```
  public:
    void identifie() {
      cout << "Je suis un point de couleur " << couleur
           << endl;
    }
};
```

Si l'on définit

```
PointCol pc(3,5,'b');
```

l'appel

```
pc.affiche();
```

fournira le résultat

```
Je suis un point
x, y : 3, 5
```

car l'appel à identifie () appelle la fonction membre identifie () de Point et pas celle de PointCol. Pour permettre la ligature dynamique, on déclare identifie () **virtual** dans la classe de base Point. Alors l'appel à identifie () dans affiche () ira chercher la fonction membre de la classe de l'objet appelant.

```
class Point {
  public:
    virtual void identifie() { ... }
    ...
};
```

Si l'on effectue :

```
int main() {
  PointCol pc(3,5,'b');
  Point p(7,2);
  pc.affiche(); // appel de affiche() de Point qui appelle
                // identifie() de PointCol
  Point *adp = &pc;
  adp->affiche();
}
```

On obtient :

```
Je suis un point de couleur : b
x, y : 3, 5
Je suis un point de couleur : b
x, y : 3, 5
```

Remarques.

1. Il n'est pas obligatoire de redéfinir une fonction déclarée **virtual** dans toutes les classes dérivées.
2. Un constructeur ne peut pas être **virtual**. En revanche, un destructeur peut être **virtual** (pour une classe de base avec des fonctions virtuelles). Ça peut être nécessaire si l'on manipule des objets de la classe dérivée, par l'intermédiaire de pointeurs sur la classe de base. Lors de la récupération mémoire d'un de ces pointeurs, le fait d'avoir un destructeur déclaré **virtual** dans la classe de base assure que le destructeur de la classe dérivée sera bien appelé pour détruire correctement l'objet de la classe dérivée pointé par ce pointeur de la classe de base.

2.11 Fonctions membres virtuelles pures et classes abstraites

Pour définir en C++ une classe abstraite, c'est à dire qu'on ne peut pas instancier mais qui peut donner naissance à d'autres classes par héritage, il faut que cette classe contienne au moins une fonction virtuelle pure, c'est à dire dont la définition est nulle (qui vaut zéro, et non pas qui est vide).

Une fonction virtuelle est rendue pure par un initialisateur `= 0` :

```
virtual void f(...) = 0; // pas de définition, elle est égale à 0
```

Ainsi :

- Toute classe contenant au moins une fonction membre virtuelle pure est considérée comme abstraite (pas d'instanciation possible).
- Une fonction membre virtuelle pure doit forcément être définie dans les classes dérivées, ou redéclarée virtuelle pure. Dans ce cas, cette classe dérivée est elle aussi abstraite.

Exemple.

```
class A { // abstraite à cause de f et g, virtuelles pures
  ...
  public:
    virtual void f() = 0;
    virtual void g() = 0;
};

class B : public A {
  public:
    void f();
};
```

Puisqu'il y a redéfinition de `f()`, elle n'est plus virtuelle pure. En revanche, `g()` reste virtuelle pure, puisque non redéfinie, sa redéclaration en virtuelle pure est implicite pour les compilateurs modernes.

```
class C : public B {
  public:
    void g();
};
```

`g()` est redéfinie dans `C`, `f()` est redéfinie dans `B`, donc `C` n'est plus abstraite, elle peut être instanciée.

Chapitre 3

Java

3.1 Caractéristiques

- Orienté objet, à l'exception des types primitifs pour des raisons de performance,
- Universel,
- Propriétaire pour garder le contrôle, mais à licence libre pour permettre l'utilisation sans contraintes,
- Repose sur une machine virtuelle,
- Syntaxe proche du C++,
- Bibliothèque standard riche (IHM, réseaux, bases de données, structures de données, ...),
- Absence de pointeurs de C/C++, pour des raisons de sécurité et au prix de la performance apportée par l'utilisation raisonnée des pointeurs.
- Pas d'héritage multiple, mais possibilité d'implémenter plusieurs interfaces,
- Généricité, depuis 2004.

3.2 Compilateur et machine virtuelle

En C++, la compilation d'un fichier source créée un fichier objet en code machine, qui sera lié avec d'autres fichiers objets et bibliothèques pour créer un éxécutable destiné à la machine cible.

En Java, le fichier source est compilé en bytecode destiné à la machine virtuelle. Il n'y a pas d'édition de liens : lors du lancement d'une classe Java sur la machine virtuelle, les appels à d'autres classes chargent leurs bytecode dynamiquement.

L'approche de la compilation en C++ est plus performante, mais nécessite de recompiler le programme pour toutes les machines cibles. En revanche, la machine virtuelle de Java est universelle, mais au coût d'une performance moindre en raison de la couche supplémentaire nécéssaire entre le machine cible et le bytecode. La différence de performance s'est nettement rétrecie avec l'arrivée du compilateur « Just in time » en Java, mais il y a toujours un coût.

3.3 Variables

Trois types de variables en Java :
– Variables d'instance (attributs, champs),
– Variables de classe (**static**),
– Variables locales au méthodes.

3.3.1 Types

Java est un langage fortement typé : chaque variable a un type.

Types de base

Les types primitifs de Java sont :
– byte,
– **short**,
– **int**,
– **long**,
– **float**,
– **double**,
– **char**,
– boolean

À l'inverse du C, les tailles des types primitifs ne sont pas laissées à l'appréciation du compilateur en Java :
– 1 octet : byte, boolean,
– 2 octets : **short**, **char**,
– 4 octets : **int**, **float**,
– 8 octets : **long**, **double**.

Les boolean ne représentent que deux valeurs, donc ne nécessitent qu'un bit. Pour des raisons matérielles, ils sont le plus souvent codés sur un octet, mais cela dépend de l'implémentation de la machine virtuelle utilisée.

Références

Le seul mode de passage d'objets en Java est le passage par référence, et non par copie, qui est le mode de passage des types primitifs.

Si l'on éxécute le programme suivant :

```
class MonInt {
  private int val;
  public MonInt() { val=0; }
  public MontInt(int x) { val=x; }
  public int get() { return val; }
  public void incr() { val++; }

  public static void main(String args[]) {
    int x=2;
    int y=x;
    y++;
    System.out.println(x);
    System.out.println(y);
```

```
        MonInt mx = new MonInt(2);
        MonInt my = mx;
        my.incr();
        System.out.println(mx.get());
        System.out.println(my.get());
    }
}
```

L'affichage sera :

```
2
3
3
3
```

Car my et mx sont des références sur le même objet en mémoire, donc l'appel my.incr() incrémente le membre val de l'unique classe MonInt référencée par my et mx. À l'inverse, l'instruction y = x copie la valeur de x dans la zone mémoire allouée pour y, zone différente de celle de x.

3.4 Visibilité

Il y a quatre types de visibilité en Java, différenciés par des mots clés. Par ordre de croissance de liberté :

- **private** : visible dans la classe,
- par défaut (aucun mot clé) : visible dans le package,
- **protected** : visible dans le package et dans les sous classes,
- **public** : visible partout.

3.5 Cycle de vie des objets

3.5.1 Création

En Java, l'opérateur de création d'objets est le mot clé **new**. Contrairement au C++, l'instruction

```
MonInt mx;
```

n'alloue aucune instance de classe, seule la référence est créée. Pour instancier la classe, il faut utiliser **new** :

```
MonInt mx = new MonInt();
```

3.5.2 Utilisation

3.5.3 Recyclage

Il n'y a pas d'opérateur **delete** en Java, car la gestion de la mémoire est déléguée à la machine virtuelle. Les objets alloués sur le tas seront nettoyés par le *garbage collector* dès qu'ils ne seront plus référencés.

Le mécanisme employé est assimilable à un compteur de références aux objets alloués, qui sont considérés comme perdus lorsque le compteur tombe à zéro. Dans

la pratique, des cas épineux peuvent se présenter : deux objets contenant chacun une référence à l'autre ont leur compteur à un, sans pour autant être accessibles par le programme. Le *garbage collector* est capable de résoudre ces problèmes, mais à un certain coût. Il est donc avisé de faire quelques gestes « écologiques », en mettant les références à null lorsque les objets crées ne sont plus désirés.

Lorsque le *garbage collector* s'apprête à libérer un objet, il commence par appeler la méthode finalize () définit dans Object. Par défaut, cette méthode ne fait rien, mais dans le cas d'une classe qui utilise des ressources externes, il peut être utile de s'assurer que ces ressources sont libérées lors de la destruction de l'objet.

Le programme suivant

```
class MonInt {
  private int val;
  private static int nbObjets = 0;

  public MonInt() {
    this(0);
  }

  public MontInt(int x) {
    val = 0;
    ++nbObjets;
  }

  protected void finalize() {
    --nbObjets;
  }

  public static getNbObjets() {
    return nbObjets;
  }

  public static void main(String args[]) {
    for (int i=0; i < 10....0; ++i) {
      MonInt a = new MonInt();
      ...
      System.out.println(MonInt.getNbObjets());
    }
  }
}
```

Aura une sortie semblable à

```
1
2
3
...
10216
5398
5399
5400
...
```

La chute du nombre d'objets est dûe à l'action du *garbage collector*, qui est un thread lancé par la machine virtuelle, donc n'agit que par moments. Le nombre d'objets

alloués ne tombe pas à zéro à ce moment car le *garbage collector* n'a qu'un temps restreint afin de ne pas bloquer l'éxécution du programme en cours.

Il est possible d'appeler explicitement le garbage collector avec l'instruction

```
System.gc();
```

3.6 Variables et méthodes d'instance et de classe

Pour déclarer des variables ou méthodes de classe, on emploie le mot clé **static**. S'il s'agit d'une variable, toutes les instances de la classe la partagent alors, et une modification faite par une instance, modifie la valeur pour toutes les instances de la même classe. Si c'est une méthode de classe, elle ne peut agir sur une instance en particulier. En particulier, une méthode de classe n'a accès qu'aux variables et méthodes de la classe déclarées avec **static**.

Pour appeler une méthode de classe, ou accèder à une variable de classe, on utilise le nom de la classe plutôt que le nom d'une variable d'instance :

```
Classe c = new Classe();
c.variableStatic; // déconseillé
c.methodeStatic(); // déconseillé
Classe.methodeStatic(); // préféré
Classe.variableStatic; // préféré
```

3.7 Héritage

Toutes les classes peuvent hériter d'une autre avec le mot clé extends. L'héritage multiple n'est pas permis comme en C++. Par défaut, toutes les classes qui n'héritent pas spécifiquement d'une autre sont descendantes de la classe Object. Il y a donc un arbre d'héritage en Java.

```
class Employe {
  private String nom;
  private int salaire;

  public Employe(String n, int s) {
    nom = n;
    salaire = s;
  }
  ...
  public double prime() {
    return salaire * 0.1;
  }
}

class Secretaire extends Employe {
  int nbLang;

  public Secretaire(String n, int s, int l) {
    super(n,s);
    nbLang = l;
  }
```

```
}

class Directeur extends Employe {
  Secretaire secretaire;

  public Directeur(String n, int s, Secretaire se) {
    super(n,s);
    secretaire = se;
  }

  public double prime() {
    return 20 * super.prime();
  }
}
```

Les directeurs et secrétaires étant tous des employés, on peut faire :

```
Employe[] personnel = new Employe[3];
personnel[0] = new Employe(...);
personnel[1] = new Secretaire(...);
personnel[2] = new Directeur(...);

double totalPrimes = 0;
for (int i=0; i < personnel.length; ++i)
  totalPrimes += personnel[i].prime();
```

Le mot clé final est utilisé pour déclarer des attributs non modifiables, des méthodes non surchargeables, et lorsqu'il qualifie une classe, elle ne peut plus avoir de descendants.

3.8 Classes abstraites

```
public abstract class Horloge {
  private int h, m;

  public int getH() { return h; }
  public int getM() { return m; }
  ...
  public void avancer(int min) { ... }
  ...
  public abstract void afficher();
}

Horloge h = new Horloge(12,45); // Erreur

public class Horloge24 extends Horloge {
  public void afficher() {
    System.out.println(getH() + ":" + getM());
  }
}

public class Horloge12 extends Horloge {
  public void afficher() {
    if (getH() > 12)
```

```
      System.out.println(getH() - 12 + ":" getM() + "pm");
    else
      System.out.println(getH() + ":" getM() + "am");
  }
}

Horloge[] tab = new Horloge[10]; // Ok, pas d'instanciation
tab[0] = new Horloge12(...);
tab[1] = new Horloge24(...);
...
for (int i=0; i<10; ++i)
  tab[i].afficher();

void toto(Horloge h) {
  ...
  h.afficher();
}

toto(new Horloge12(13,45));
toto(new Horloge24(13,45));
```

3.9 Interfaces

```
public interface Audible {
  public void ecouter();
}

public class HorlogeAudible extends Horloge implements Audible {
  public void afficher() { ... }
  public void ecouter() { ... }
  ...
}

Audible a = new Audible(); // Erreur
Audible a = new HorlogeAudible(); // Ok

Audible[] tab = new Audible[10];
tab[0] = new HorlogeAudible();
tab[1] = new Chanson();
...
for (int i=0; i < 10; ++i) {
  tab[i].ecouter(); // Ok
  tab[i].getH(); // Erreur
}
```

3.10 Généricité

```
// Java 1.4
public class Paire {
  private Object a;
  private Object b;

  public Paire(Object a, Object b) {
```

```
      this.a = a;
      this.b = b;
    }

    public Object getA() { return a; }
    public Object getB() { return b; }
    public void setA(Object a) { this.a = a; }
    public void setB(Object b) { this.b = b; }
  }

  Paire p1 = new Paire(3.2,1.5); // Types de base non admis
  Paire p1 = new Paire(new Double(3.2), new Double(1.5)); // Ok
  Paire p2 = new Paire("toto", new Integer(2)); // Ok

  int age = new (Integer)(p2.getB()).int_Value(); // Ok
  String nom = (String)(p2.getA()); // Ok
  String nom = (String)(p1.getA()); // Erreur à l'éxécution

  // Java 1.5
  public class Paire<A,B> {
    private A a;
    private B b;

    public Paire(A a, B b) {
      this.a = a;
      this.b = b;
    }

    public A getA() { return a; }
    public B getB() { return b; }
    public void setA(A a) { this.a = a; }
    public void setB(B b) { this.b = b; }
  }

  Paire<double, double> p1; // Erreur, types de base
  Paire<Double, Double> p1 = new Paire<Double, Double>(3.2, 1.1); //
Ok
  Paire<String, Integer> p2 = new Paire<String, Integer>("toto", 2); //
Ok

  double x = p1.getA(); // Ok
  int age = p2.getB(); // Ok
  String nom = p2.getA(); // Ok
  String nom = p1.getA(); // Erreur à la compilation

  class Pomme extends Fruit { ... }
  class Cheval extends Animal { ... }
  ...
  void toto1(Paire<Animal, Fruit> p) { ... }

  Paire<Animal, Fruit> p1 = ...
  Paire<Cheval, Pomme> p2 = ...
  toto1(p1); // Ok
  toto1(p2); // Erreur
```

```
p1 = p2; // Erreur

void toto2(Paire<? extends Animal, ? extends Fruit> p) { ... }
toto2(p1); // Ok
toto2(p2); // Ok

Paire<?, ?>
```

équivaut à

```
Paire<? extends Object, ? extends Object>

<? super Toto>
```

indique toutes les classes qui précèdent Toto dans l'arbre d'héritage.

3.11 Boucles

Nouvelle boucle depuis la version 5.
Avant :

```
int[] tab = new int[...];
...
int somme = 0;
for(int i=0; i < tab.length; ++i)
  somme += tab[i];
```

Après :

```
int [] tab = new int[...];
...
int somme = 0;
for (int x : tab)
  somme += x;
```

Pour les collections :

```
Collection<Employe> c = ...
Iterator<Employe> it = c.iterator();
double salaireTotal = 0;
while (it.hasNext()) {
  Employe e = it.next();
  salaireTotal += e.getSalaire();
}
```

Devient :

```
for (Employe e : c)
  salaireTotal += e.getSalaire();
```

3.12 Java Collection Framework

Collection : regroupement d'objets manipulables (recherche, ajout, suppression, union, intersection).
Les collections sont : des interfaces, des implémentations, des algorithmes.
Avantages :

- moins d'effort,
- plus d'efficacité,
- liens plus faciles entre les API,
- code réutilisable.

3.12.1 Interface Collection

```
interface Collection<E> {
  // opérateurs de base
  int size();
  boolean isEmpty();
  boolean contains(Object o);
  boolean add(E e); // facultatif
  boolean remove(Object o); // facultatif
  Iterator<E> iterator();

  // opérateurs de masse
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c); // facultatif
  boolean removeAll(Collection<?> c); // facultatif
  boolean retainAll(Collection<?> c); // facultatif
  void clear();

  // opérateurs tableau
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

3.12.2 Interface Iterator

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove(); // facultatif
}

Collection<Employe> c = ...
...
Iterator<Employe> it = c.iterator();
while(it.hasNext())
  Employe e = it.next();
  // traitement de e
  ...
}
```

remove() supprime le dernier élément renvoyé par next(), si pas d'appel de next() entre deux appels de remove() : IllegalStateException.

3.12.3 Exemple

```
public class MyCollection<E> implements Collection<E> {
  private static final int INITIAL_CAPACITY = 10;
  private static final int CAPACITY_INCREMENT = 10;
```

```
private E[] elements;
private int size;

private void addNoCheck(E e) { elements[size++] = e; }
private void fastRemove(int i) {
  System.arrayCopy(elements, i+1, elements, i, size-i-1);
  size--;
  elements[size] = NULL;
}

private int indexOf(Object o) {
  if (o == NULL) {
    for (int i=0; i < size; ++i)
      if (elements[i] == NULL)
        return i;
  }
  else
    for (int i=0; i < size; i++)
      if (o.equals(elements[i]))
        return i;
  return -1;
}

private void allocElements(int n) {
  elements = (E[])(new Object[n]);
}

private void incrementCapacity(int c) {
  E[] addElements = elements;
  allocElements(elements.length + c);
  System.arrayCopy(oldElements, 0, elements, 0, size);
}

// Constructeurs
public MyCollection(int n) {
  allocElements(n);
  size = 0;
}

public MyCollection() {
  this(INITIAL_CAPACITY);
}

public MyCollection(Collection<? extends E> c) {
  this(11 * c.size() / 10);
  for (E e : c)
    addNoCheck(e);
}

// Opérations de base
public int size() { return size; }
public boolean isEmpty() { return size == 0; }
public boolean contains(Object o) { return indexOf(o) != -1; }
```

```
public boolean add(E e) {
  if (size == elements.length)
    incrementCapacity(CAPACITY_INCREMENT);
  addNoCheck(e);
  return true;
}

public boolean remove(Object o) {
  int i = indexOf(o);
  if (i == -1)
    return false;
  fastRemove(i);
  return true;
}

// Opérations de masse
public boolean containsAll(Collection<?> c) {
  for (Object o : c)
    if (!contains(o))
      return false;
  return true;
}

public boolean addAll(Collection<? extends E> c) {
  int missing = size + c.size() - elements.length;
  if (missing > 0)
    incrementCapacity(11 * missing / 10);
  for (E e : c)
    addNoCheck(e);
    return c.size() > 0;
}

public boolean removeAll(Collection<?> c) {
  boolean res = false;
  for (Object o : c )
    while(remove(o))
      res = true;
  return res;
}

public boolean retainsAll(collection<?> c) {
  boolean res = false;
  int i = 0;

  while (i < size) {
    if (!c.contains(elements[i])) {
      fastRemove(i);
      res = true;
    }
    else
      i++;
  }
  return res;
}
```

```
public void clear() {
  for (int i=0; i < size; ++i)
    elements[i] = NULL;
  size = 0;
}

// Opérations de tableau
public Object[] toArray() {
  Object[] res = new Object[size];
  System.arrayCopy(elements, 0, res, 0, size);
  return res;
}

public <T> T[] toArray(T[] a) {
  if (a.length < size)
    a = (T[])(new Object[size]);
  System.arrayCopy(elements, 0, a, 0, size);

  if (a.length > size)
    a[size] = NULL;

  return a;
}

// Itérateur
public Iterator<E> iterator() {
  return new MyIterator();
}

private class MyIterator implements Iterator<E> {
  private int iprevious;
  private int inext;

  public MyIterator() {
    iprevious = 1;
    inext = 0;
  }

  public boolean hasNext() {
    return inext < size;
  }

  public boolean E next() {
    if (inext == size)
      throw new NoSuchElementException();

    iprevious = inext;
    inext++;
    return elements[iprevious];
  }

  public void remove() {
    if (iprevious == -1)
```

```
        throw new IllegalStateException();
      fastRemove(iprevious);
      iprevious = -1; // pour éviter les appels successifs
      inext--;
    }
  }
}

public class MyCollection<E> extends AbstractCollection<E>
```

Deux méthodes abstraites : size() et iterator().
Dans AbstractCollection :

```
public boolean isEmpty() {
  return size() == 0;
}

public boolean contains(Object o) {
  Iterator<E> it = new Iterator();
  while(it.hasNext())
    if (o.equals(it.next()))
      return true;
  return false;
}
```

3.12.4 Set

Pas de répétition.

```
s1.containsAll(s2); // s2 ⊂ s1
s1.addAll(s2); // s1 ← s1 ∪ s2
s1.retainsAll(s2); // s1 ← s1 ∩ s2
s1.removeAll(s2); // s1 ← s1 \ s2
```

3.12.5 List

```
E get(int i);
E set(int i, E e);
void add(int i, E e);
E remove(int i);
int indexOf(Object o);
int lastIndexOf(Object o);
ListIterator<E> listIterator();
ListIterator<E> listIterator(int i);
List<E> subList(int from, int to);
```