Programmation objet et C++

Table des sections

1. Programmation objet 4
   (a) Objet = données + fonctions 5
   (b) Un exemple 6
   (c) La métaphore des messages 8
   (d) Héritage 9
   (e) Bibliographie 12
2. C et C++ 13
   (a) “Un meilleur C” 14
   (b) Incompatibilités entre C et C++ 15
   (c) Entrées-sorties 16
   (d) Commentaires 17
   (e) Emplacement des déclarations 18
   (f) Arguments par référence 19
   (g) Arguments par défaut 20
   (h) Surcharge 21
   (i) Allocation dynamique 23
   (j) Fonctions en ligne 24
   (k) Classes et objets 25
3. Instructions 27
   (a) Identificateurs 28
   (b) Instructions 29
   (c) Itérations 30
   (d) Exceptions 31
   (e) Constantes 32
   (f) Références 34
   (g) Passage par référence 36
   (h) Retour par référence 39
   (i) Surcharge de fonctions 42
   (j) Résolution de la surcharge 45
   (k) Exemple : une calculette 48
4. Les classes 50
   (a) Objectif 51
   (b) Déclaration 53
   (c) Définition 54
   (d) Encapsulation 55

   (e) Constructeurs 57
   (f) Une classe de complexes 61
   (g) Une classe de rationnels 67
   (h) Surcharge d'opérateurs 69
   (i) Liste d'initialisation 74
   (j) Membres statiques 76
   (k) Méthodes constantes 78
   (l) La classe `string` 80

5. Héritage 84
   (a) Objectif 85
   (b) Classe composée ou classe dérivée 86
   (c) Syntaxe 87
   (d) Accès aux données et méthodes 89
   (e) Classes dérivées et constructeurs 92
   (f) Héritage multiple 94
   (g) Polymorphisme 97
   (h) Méthodes virtuelles 98
   (i) Un exemple: les expressions 106
   (j) Evaluation de polonaise postfixée, sans arbre 113
   (k) Table de fonctions virtuelles 117

6. Structures de données 119
   (a) Piles 120
   (b) Exceptions 125
   (c) Listes comme objets 130
   (d) Arbres (ou composites) 135
   (e) Visiteurs 141

7. Copie et affectation 148
   (a) Constructeurs 149
   (b) Constructeur de copie 150
   (c) Affectation 153
   (d) Conversions 155

8. Opérateurs 156
   (a) Principe 157
   (b) Opérateurs amis 159
   (c) Syntaxe 160
   (d) Opérateur d'indexation 163
   (e) Opérateurs logiques 167
   (f) Opérateur d'insertion et d'extraction 168
   (g) Opérateurs `++` et `--` 170

9. Droits d'accès 171
   (a) Objectif 172

(b) Les trois catégories 173
(c) Fonctions amies 175
(d) Accès et héritage 176

10. Flots 177
   (a) Hiérarchie de classes 178
   (b) Manipulateurs 181
   (c) Fichiers 182
   (d) Formatage des sorties 183
   (e) Entrées 189
   (f) Manipulateurs, le retour 194

11. Patrons 197
   (a) Fonctions génériques 198
   (b) Classes génériques 200

12. Conception objet 206
   (a) Objet versus interprétation 207
   (b) Réification 208
   (c) Encapsulation 209
   (d) Héritage 210
   (e) Abstraction – partition 211
   (f) Interface – implémentation 212
   (g) Universalité – généricité 213

13. Design Patterns 215
   (a) Objectifs 216
   (b) Composite 219
   (c) Visiteur 220
   (d) Adaptateur 221
   (e) Observateur 222
   (f) Itérateur 223

14. Rtti 232
   (a) Objectif 233
   (b) Réalisation 234
   (c) Exemple 235
   (d) Conversion dynamique 237

15. STL 238
   (a) Objectifs 239
   (b) Exemples 240
   (c) Conteneurs 244
   (d) Méthodes communes 246
   (e) Algorithmes 250
   (f) Objets fonctions 252
   (g) Autres algorithmes 257
   (h) Itérateurs 261

1

Programmation objet

1. Objet = données + fonctions
2. Un exemple
3. La métaphore des messages
4. Héritage
5. Bibliographie

Objet = données + fonctions

Un *objet* est une entité structurée qui

- possède des attributs (données)
- possède des opérations (méthodes)

Les opérations définies sur les données sont accessibles par l'entité elle-même.

Avantages

- rapprochement des opérations des données
- modélisation dirigée par *service* (quoi) plutôt que par *réalisation* (comment)
- programmation par contrat
- réutilisation facilitée
- modularité facilitée par l'encapsulation
- réalisation incrémentale, facilite la maintenance
- classes abstraites modélisent des services
- la délégation (mécanisme par lequel un objet ne traite pas entièrement un message, mais le transmet à un autre) renforce la responsabilité
- l'héritage encourage la généralité

Une *classe* est un modèle d'objet, qui décrit les attributs et définit les méthodes.

Un exemple

Un *point* est un couple d'entiers.

Définition de la classe

```
class Point {
  int x, y;
public:
  void setX(int xx) { x = xx; }
  void setY(int yy) { y = yy; }
  void deplace(int dx, int dy) {
    x += dx; y += dy;
  }
  void show() {
    cout << "(" << x << "," << y << ")";
  }
};
```

Utilisation

```
void main(char* argv, int argc) {
  Point a;
  a.setX(2); a.setY(3);
  a.show();
  a.deplace(4,5);
  a.show();
}
```

Résultat

```
(2,3)(6,8)
```

Programmation traditionnelle: Elle illustre la *séparation* des données et des fonctions

```
#include <iostream.h>

class Point {
public:
  int x, y;
};

void setX(Point&a, int xx) { a.x = xx; }
void setY(Point&a, int yy) { a.y = yy; }
void deplace(Point& a, int dx, int dy) {
  a.x += dx; a.y += dy;
}
void show(const Point& a) {
  cout << "(" << a.x << "," << a.y << ")";
}

void main(char* argv, int argc) {
  Point a;
  setX(a, 2);
  setY(a, 3);
  show(a);
  deplace(a, 4, 5);
  show(a);
}
```

La métaphore des messages

Métaphore = image, comparaison

Les objets communiquent par *messages*.

```
a <- deplace_toi(2,5)
```

équivaut à l'appel de méthode

```
a.deplace(2,5)
```

Avantages

- les objets sont assimilés à des entités, d'où
  - responsabilité renforcée
  - encapsulation renforcée
- améliore la conception des classes
- éclaire le mécanisme de délégation
- libère le même nom, pour même usage mais implémentation diversifiée.

Héritage

On peut déclarer une classe comme sous-classe ou *classe dérivée* d'un autre classe qui, elle, est alors appelée *classe de base*.

Une classe dérivée possède *ipso facto* les données et les méthodes de sa classe de base. Elle peut les enrichir ou les particulariser, et adapter les méthodes par réécriture (masquage).

Avantages

- *Regroupement*: données ou méthodes communes sont regroupées en une classe de base
- *Généralisation*: plusieures classes sont des cas particuliers d'un concept général
- *Spécialisation*: une classe dérivée représente un cas particulier
- *Extensibilité*: plutôt que modifier une classe existante, la dériver pour inclure de nouvelles fonctionnalités

Exemple

Une classe de cercles:

```
class Cercle {
  Point centre;
  int rayon;
public:
  void setCercle(int xx, int yy, int r) {
    centre.setX(xx);
    centre.setY(yy);
    rayon = r;
  }
  void deplace(int dx, int dy) {
    centre.deplace(dx,dy);
  }
  void show() {
    centre.show(); cout <<":"<<rayon;
  }
};
```

Une classe d'anneaux:

```
class Anneau : public Cercle {
  int prayon;
public:
  setAnneau(int xx, int yy, int r, int pr) {
    setCercle(xx, yy, r);
    prayon = pr;
  }
  void show() {
    Cercle::show();
    cout <<","<<prayon;
  }
};
```

Utilisation:

```
void main(char* argv, int argc) {
  Cercle c;
  c.setCercle(2,3,5);
  c.show();
  Anneau a;
  a.setAnneau(2,3,5,1);
  a.show();
  a.deplace(2,2);
  a.show();
}
```

Résultat:

```
(2,3):5(2,3):5,1(4,5):5,1
```

Bibliographie

G. Hansel, *Passeport pour C++, concepts et mise en pratique*, Vuibert, 1999.

E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, (traduction française) Vuibert, 1999.

2

C et C++

1. "Un meilleur C"
2. Incompatibilités entre C et C++
3. Entrées-sorties
4. Commentaires
5. Emplacement des déclarations
6. Arguments par référence
7. Arguments par défaut
8. Surcharge
9. Allocation dynamique
10. Fonctions en ligne
11. Classes et objets

“Un meilleur C”

Le langage C++ se veut un langage C amélioré.

Il possède des fonctionnalités supplémentaires, et notamment

- la surcharge de fonctions
- le passage par référence
- l’emplacement des déclarations
- l’allocation dynamique

Les apports spécifiques de C++ sont

- l’aide à l’*abstraction de données*: définition de types de données, et de leur implémentation concrète.
- l’aide à la *programmation objet*: hiérarchie de classes et héritage.
- l’aide à la *programmation générique*: classes patron et algorithmes génériques.

Incompatibilités entre C et C++

- Déclarations de fonctions

  Toute fonction doit

  – être définie avant utilisation

  – ou être déclarée par un prototype

  ```
  float fct (int, double, char*) ;
  ```

  (En C, une fonction non déclarée est supposée de type de retour `int`.)

- Une fonction qui ne retourne pas de valeur a le type de retour `void`.

- Le qualificatif `const` peut être utilisé pour une expression constante, notamment pour définir la taille d'un tableau (en C, il faut un `#define`)

  ```
  const int N = 10;
  int valeurs[N];
  ```

- Le `typedef` est implicite dans les structures (qui sont des classes). En C classique:

  ```
  typedef struct _Point {
    int x, y;
  } Point;
  ```

  En C++

  ```
  struct Point {
    int x, y;
  };
  ```

Entrées-sorties

Les entrées et sorties sont gérées dans C++ à travers des objets particuliers appelées *streams* ou *flots*. Inclure `<iostream.h>`

Deux *opérateurs* sont surchargés de manière appropriée pour les flots:

- l'opérateur d'*insertion* `<<` (écriture)
- l'opérateur d'*extraction* `>>` (lecture)

Les flots prédéfinis sont

- `cout` attaché à la sortie standard;
- `cerr` attaché à la sortie erreur standard;
- `cin` attaché à l'entrée standard.

Exemple

```
#include <iostream.h>
main() {
  cout << "Bonjour, monde !\n";
}
```

Plusieurs expressions:

```
cout << ex_1 << ex_2 << ... << ex_n ;
```

Plusieurs "lvalues":

```
cin >> lv_1 >> lv_2 >> ... >> lv_n ;
```

Les types écrits ou lus sont

char, short, int, long, float, double, char*

Exemple

```
#include <iostream.h>
int i;
main() {
  cout << "Un entier : ";
  cin >> i;
  cout << "Le carre de " << i
    <<" est " << i*i << endl;
}
```

Commentaires

// commentaire de “fin de ligne”

```
#include <iostream.h>         // pour les streams
int i;
main() {
  cout << "Un entier : ";     // affichage et
  cin >> i;                   // lecture synchronises
  cout << "Le carre de " << i
    <<" est " << i*i << endl; // "endl" insere "\n"
}
```

Les commentaires de C, de la forme `/*...*/` sont toujours valables.

Emplacement des déclarations

Une déclaration peut apparaître partout, mais doit précéder son utilisation.

```
int n;
n = 3;
int q;
q = 2*n-1;


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

Arguments par référence

Un paramètre dont le nom est suivi de `&` est transmis par référence, donc

- pas de copie de l'argument à l'appel;
- possibilité de modifier l'argument.

Déclaration

```
void echange(float&, float&) ;
```

Définition

```
void echange(float& a, float& b) {
  float t = a;
  a = b;
  b = t;
}
```

Exemple

```
float x = 2, y = 3;
echange (x, y);
cout << x <<","<< y; //Affiche 3,2
```

Passage par référence constante pour

- ne pas copier l'argument à l'appel;
- ne pas modifier l'argument :

```
void afficher(const objet&);
```

Arguments par défaut

Dans une fonction, les derniers arguments peuvent prendre des "valeurs par défaut".

Déclaration

```
float f(char, int = 10, char* = "Tout");
```

Appels

```
f(c, n, "rien")
f(c, n)    // <-> f(c, n, "Tout")
f(c)       // <-> f(c,10, "Tout")
f()        // erreur
```

Seuls les derniers arguments peuvent avoir des valeurs par défaut.

```
float f(char = 'a', int, char* = "Tout"); // erreur
```

Surcharge

Un même identificateur peut désigner plusieurs fonctions, si elles diffèrent par la liste des types de leurs arguments.

```
float max (float, float);
float max (float, float, float);
float max (int, float []);
```

Exemple

```
float max(float a, float b) {
  return (a > b) ? a : b;
}
float max(float a, float b, float c) {
  return max(a, max(b, c));
}
float max(int n, float t[]) {
  if (!n) return 0;
  float m = t[1];
  for (int i = 2 ; i < n; i++)
    m = max(m, t[i]);
  return m;
}

void main() {
  float x, y, z;
  float T[] ={11.1, 22.2, 33.3, 44.4, 7.7, 8.8 };

  x = max (1.86, 3.14);
  y = max (1.86, 3.14, 37.2);
  z = max (6, T);
  cout << x <<" "<< y <<" "<< z;
}
```

L'analyse de la signature détermine la fonction à utiliser. Les promotions et conversions usuelles s'appliquent. Le type de retour n'intervient pas

```
float min(int, float);
double min(int, float); // erreur
```

- Attention à l'ambiguïté, surtout en conjonction avec les valeurs par défaut.
- On peut aussi surcharger les opérateurs : définir par exemple l'addition de nombres complexes et représenter cette addition par le signe `+`.

Allocation dynamique

Les opérateurs **`new`** et **`delete`** gèrent la mémoire dynamiquement.

```
new chose[n]
```

alloue la place pour n éléments de type **`chose`** et retourne l'adresse du premier élément;

```
delete addresse
```

libère la place allouée par **`new`**.

```
int* a = new int; // malloc(sizeof(int))
double* d = new double[100];
                  // malloc(100*sizeof(double))
```

Exemple

```
char t[] = "Hello, world !";

char* copy(const char* t) {
  char* a;
  a = new char[1 + strlen(t)];
  strcpy(a,t);
  return a;
}
```

Fonctions en ligne

Une fonction *en ligne* (`inline`) est une fonction dont les instructions sont incorporées par le compilateur dans le module objet à chaque appel. Donc

- il n'y pas d'appel : gestion de contexte, gestion de pile;
- les instructions sont engendrées plusieurs fois;
- rappellent les macros.

Déclaration par qualificatif `inline`.

```
inline int sqr(int x) { return x*x; }
```

Classes et objets

Une *classe* est une structure, dont les *attributs* sont des *données* ou des *méthodes*. Un *objet*, ou une *instance*, est un exemplaire de cette structure.

Classe:

```
class Complexe {
  public:
    float re, im;//données
    void show(); //méthode
};
```

On déclare des objets de la classe par:

```
Complexe a, b;
```

On les manipule de manière usuelle:

```
a.re = b.im;
b.re = 7;
a.show();
```

La *définition* d'une méthode se fait soit en ligne, soit séparément; dans le deuxième cas, elle utilise l'*opérateur de portée* noté `::` pour désigner la classe.

```
void Complexe::show() {
  cout << re << ' ' << im;
}
```

A l'appel `a.show()`, la méthode est celle de la classe de `a`, donc `Complexe::show()`, et les champs `re` et `im` sont ceux de l'objet appelant, c'est-à-dire `a.re` et `a.im`.

La *surcharge d'opérateurs* permet de définir une forme agréable pour des opérations sur des objets.

```
Complexe operator+(Complexe s, Complexe t) {
  Complexe w;
  w.re = s.re + t.re;
  w.im = s.im + t.im;
  return w;
}
```

On peut alors écrire

```
Complexe a, b, c;
  ...
a = b + c;
```

3

Instructions

1. Identificateurs
2. Instructions
3. Itérations
4. Exceptions
5. Constantes
6. Références
7. Passage par référence
8. Retour par référence
9. Surcharge de fonctions
10. Résolution de la surcharge
11. Exemple : une calculette

Identificateurs

Les *identificateurs* sont comme en C formés d'une lettre ou '_', suivie de lettres, chiffres, ou '_' :

$$\texttt{[A-Za-z\_][A-Za-z0-9\_]}^*$$

- les minuscules et majuscules sont différenciées
- la longueur est arbitraire
- le nombre de caractères significatifs dépend de la machine

```
hello   fo0    _classe   ___      // ok
if     .nom    $truc$  foo^bar  // erreur
```

Identificateurs réservés :

and	and_eq	asm	auto	bitand
bitor	bool	break	case	catch
char	class	compl	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	not	not_eq
operator	or	or_eq	private	protected
public	register	reinterpret_cast	return	short
signed	sizeof	static	static_cast	struct
switch	template	this	throw	true
try	typedef	typeid	typename	union
unsigned	using	virtual	void	volatile
wchar_t	while	xor	xor_eq	

Instructions

Instructions conditionnelles

`if`
`if else`
`switch case`

Itérations

`for`
`while`
`do while`

Bloc

```
{
    déclarations et
    instructions mélangées
}
```

Ruptures de séquence

`continue;`
`break;`
`return` *expr*;
`try ...throw... catch`

Itérations

Définition d'une variable dans l'initialisateur de la boucle

```
for (int i = 0; i < 5; i++)
  cout << i;
cout << endl;
```

Dans les versions ANSI, la portée des noms déclarés dans l'initialisateur est restreinte au bloc à la clause du `for` et au corps de la boucle.

```
for (int i = 0; i < 5; i++) ;  // ok
for (int i = 0; i < 5; i++) ;  // OK !
```

Exceptions

Les *exceptions* sont un mécanisme de *déroutement conditionnel*.

- `throw` lève une exception;
- `catch` capte l'exception.

Exemple

```
void f(int i) {
  try {
    if (i)
      throw "Help!";
    cout <<"Ok!\n";
  }
  catch(char* e) {
    cout << e << "\n";
  }
}

void main() {
  f(0);
  f(1);
}
```

Résultat

```
Ok!
Help!
```

Constantes

Un identificateur dont la valeur ne change pas peut être déclaré constant en utilisant le mot-clé `const`.

Le mot-clé `const` peut intervenir dans quatre contextes:

- dans la spécification du type d'un objet;
- dans la spécification du type d'un paramètre de fonction; il indique que cette fonction n'en modifie pas la valeur;
- dans la spécification du type d'un membre de classe;
- comme qualificatif d'une méthode pour indiquer qu'elle ne modifie pas les données membres de cette classe.

Une constante doit être initialisée, et ne peut pas être affectée.

Les constantes se substituent avantageusement aux `#define`.

```
const N =10;
float a[1+N];
```

La syntaxe peut prêter à confusion:

```
const int *p;
```

se lit `(const int) *p`. Donc `p` est un pointeur vers une constante de type `int`. Avec

```
int i;
const int *p = &i;
```

la valeur de `i` ne peut pas être modifiée à travers le pointeur: on peut écrire `i = 10` mais pas `*p = 10;` !

```
int i;
int * const q = &i;
```

Ici, `q` est un pointeur constant vers un entier; l'initialisation est nécessaire. La valeur de `q` (c'est-à-dire la case pointée) ne peut être modifiée, mais on peut modifier la valeur contenue dans cette case, et écrire `*q = 10;` (Noter l'analogie avec les tableaux en C).

```
int i;
const int * const r = &i;
```

Enfin, `r` est un pointeur constant vers un objet constant.

Références

Une *entité* au sens courant du terme est une place en mémoire destinée à contenir une donnée. Une variable est un entité, mais une entité peut être *constante* si la donnée est non modifiable.

Une *référence* est un nouveau nom (synonyme, alias) attribué à une entité déjà défini.
La syntaxe est

type& *ident*=*expression*;

- Une référence doit être initialisée à sa définition.
- L'expression d'initialisation doit désigner un objet (l-value).

```
int i = 1;
int& r = i; // r et i designent le meme entier
int x = r;  // x = 1;
r = 2;      // x = 1; i = 2
int & y;    // erreur : pas d'initialisation
int* p = &r // == &i  : meme objet
```

Usage des références

- spécification des paramètres et valeurs de retour des fonctions
- surcharge d'opérateurs

Ne pas confondre

```
int&  val = 1;      // val est reference a int
&val                // adresse de val
val & 1             // ``et'' bit a bit
```

Une référence constante ne ne permet pas de changer la valeur.

```
int n;
const int& k = n;
...
++n;   // ok
++k;   // erreur
```

Passage par référence

Passage par référence d'arguments de fonctions. Rappelons:

```
int f(int k) {...}

main() {
  ...
  j = f(i) ;
  ...
}
```

Lors de l'appel `f(i)`

1. il y a création d'une nouvelle variable **`ibis`**,
2. initialisation de **`ibis`** avec la valeur de **`i`**
3. les modifications subies par **`k`** dans la définition sont subies par **`ibis`**,
4. au retour, **`ibis`** est détruite et **`i`** a conservé sa valeur

Si maintenant, `k` est déclarée en référence

```
int f(int& k) {...}
```

lors de l'appel `f(i)`

1. il y a création d'une référence `ibis` initialisée à `i`, (donc `ibis` et `i` désignent la même place);
2. les modifications subies par `k` dans la définition sont subies par `ibis`, donc par `i`.
3. au retour, `ibis` est détruite et `i` a conservé ses modifications.

```
void incr(int& a) { a++;}

main() {
  int x = 1;
  incr(x);  // x = 2
}
```

En effet, l'appel `incr(x)` réalise

– l'initialisation `int& a = x;`
– l'incrémentation `a++;`

```
#include <iostream.h>

struct Personne {
  int age;
  char* nom;
};

// Passage par reference pour changer le contenu
void change(Personne& p) {
  p.age = 45;
  p.nom = "Bjorne Stroustrup";
}

// Passage par reference pour eviter recopie
void affiche(const Personne& p) {
  cout << p.nom <<endl;
}

void main()
{
  Personne auteur;
  change(auteur);
  affiche(auteur);
}
```

Le résultat est bien entendu

```
Bjorne Stroustrup
```

Retour par référence

Passage par référence en résultat de fonctions. Rappelons:

```
int f() {... return x;}
```

- L'instruction `return x` retourne la *valeur* de `x`;
- il y a création d'une variable sur la pile contenant cette valeur;
- `f()` n'est pas une l-valeur

```
int& f() {... return x;}
```

- Ici `return x` retourne une *référence*, donc un synonyme d'un objet;
- il y a création d'une variable sur la pile contenant cette valeur;
- `f()` est une référence pour la variable `x`, et c'est une l-value.

Exemple : retourner l'objet complexe de plus grand module.

```
#include <iostream.h>
#include <math.h>

struct Complexe {
  float re, im;
  Complexe(float r = 0, float i = 0) { re = r; im = i; }
  float module() const { return sqrt(re*re+im*im); }
};

Complexe& maxmod(const Complexe& x, const Complexe& y) {
  return (x.module() > y.module()) ? x : y;
}

void main()
{
  Complexe u(3,5), v(4,4);
  cout << "u.module() = " << u.module() <<" ";
  cout << "v.module() = " << v.module() << endl;
  Complexe& z = maxmod(u, v);
  cout << &z <<' '<< &u <<' '<< &v << endl;
}
```

Le résultat est

```
u.module() = 5.83095 v.module() = 5.65685
0x7ffffc80 0x7ffffc80 0x7ffffc78
```

Exemple

```
int& incr(int &x) {
  x += 10;
  return x;
}

main() {
  int i = 5;
  incr(i) +=15;
  cout << i << endl;
}
```

`incr(i)` est synonyme de `i`, et `main` l'incrémente de `15`. Le résultat est `30`.

Il ne faut pas faire retourner une référence à une variable locale.

Surcharge de fonctions

Il y a *surcharge* lorsqu'un même identificateur désigne plusieurs fonctions. Pour cela, ces fonctions doivent différer par la liste des types de leurs arguments.
La surcharge existe déjà en C: `3/5` et `3.5/5`, mais est systématisée en C++.

Usage de la surcharge:

- dans les constructeurs;
- dans les méthodes;
- dans les opérateurs.

Exemple fréquent:

donner la possibilité d'effectuer une opération avec des degrés différent d'ajustement de paramètres. L'absence d'un paramètre vaut acceptation de la valeur par défaut.

```
#include <iostream.h>

void affiche(int i)
  { cout << "Entier " << i << endl; }

void affiche(char c)
  { cout << "Caractere " << c << endl; }

void affiche(double d)
  { cout << "Reel " << d << endl; }

void affiche(char* txt, int n) {
  for (int j = 0; j < n; j++) cout << txt[j];
  cout << endl;
}

void main()
{
 affiche(3);
 affiche('x');
 affiche(1.2);
 affiche("Bonjour, monde !", 7);
}
```

Résultat

```
Entier 3
Caractere x
Reel 1.2
Bonjour
```

Le compilateur engendre une fonction par type, postfixée par un codage des types.

```
affiche__Fi
affiche__Fc
affiche__Fd
affiche__FPci
```

C'est pourquoi une fonction C externe doit être déclarée

```
extern "C" f();
```

L'analyse de la signature détermine la fonction à utiliser. Les promotions et conversions usuelles s'appliquent. Le type de retour n'intervient pas. Les conversions dites "triviales" impliquent :

- Pour tout type `T`, un `T` et un `T&` sont équivalents:

  ```
  void f(int i) {...}
  void f(int& i) {...} // erreur
  ```

- De même, un `T` et un `const T` sont équivalents, sauf pour `const T*` et `T*`
- Un type défini par `typedef` ne donne pas un type séparé

  ```
  typedef int Int;
  void f(int i) {...}
  void f(Int i) {...} // erreur
  ```

- Pointeur et tableau sont équivalents

  ```
  f(char* p)
  f(char p[])
  f(char p[12])
  ```

Résolution

Règles de choix

1. Recherche d'une correspondance exacte et conversions "triviales" :
   - `T` en `T &` et vice-versa,
   - ajout de `const`
   - transformation `T[]` en `T*`
2. Promotion (`char` en `int`, `float` en `double`...) ou dégradation.
3. Règles de conversion, surtout entre classes.
4. Règles de conversion définies par le programmeur.
5. Correspondance avec points de suspension.

Toute ambiguïté est une erreur.

Exemple

Prototypes

```
void f(int);            // fonction 1
void f(float);          // fonction 2
void f(int, float);     // fonction 3
void f(float, int);     // fonction 4
void f(int, int, int); // fonction 5
void f(...);            // defaut
```

Variables

```
int i,j;
float x, y;
char c;
double z;
```

Appels

```
f(c);          // fonction 1
f(i, j);       // erreur : en 3 ou en 4 ?
f(i, c);       // erreur : en 3 ou en 4 ?
f(c, i);       // erreur : en 3 ou en 4 ?
f(i, z);       // fonction 3
f(z, i);       // fonction 4
f(z, i, i);    // fonction 5
f("Hello");    // defaut
```

Exemple : la factorielle

```
int factorielle(int n) {
return (!n) ? 1 : n * factorielle(n - 1);
}
```

Elle se programme aussi avec une récursivité *terminale* à l'aide de

```
int Fact(int n, int p) {
  return (!n) ? p : Fact(n - 1, p * n);
}
```

On a alors `factorielle(n) = Fact(n, 1)`. On définit donc

```
int Fact(int n) {
  return Fact(n, 1);
}
```

ou mieux encore

```
int Fact(int n, int p = 1) {
  return (!n) ? p : Fact(n - 1, p * n);
}
```

Exemple : une calculette

Un programme simulant une calculette élémentaire, capable d'évaluer les quatre opérations `+, -, *, /`. Il faut prévoir le cas d'une division par 0.

```
#include <iostream.h>

void eval (float x, float y, char op, float& r) {
  switch (op) {
    case '+': r = x+y; return;
    case '-': r = x-y; return;
    case '*': r = x*y; return;
    case '/':
      if (y == 0) throw "DIV par 0";
      r = x/y; return;
  }
  throw "Operateur inconnu";
}

void main() {
  float x, y, resultat;
  char operateur, c;

  cout << "Calculette\n";
  for(;;) {
    cout << "Expression ? ";
    cin >> x >> operateur >> y;
    try {
      eval (x, y, operateur, resultat);
      cout << x << operateur << y << " = " << resultat << endl;
    }
    catch (char* message) {
      cerr << message << endl;
    }
    cout << "Encore (o/n)? ";
    cin >> c;
    if (c != 'o')
      break;
  }
  cout << "Au revoir !\n";
}
```

Exemple

```
Calculette
Expression ? 5+6
5+6 = 11
Encore (o/n)? o
Expression ? 6/0
DIV par 0
Encore (o/n)? o
Expression ? 5@4
Operateur inconnu
Encore (o/n)? n
Au revoir !
```

4

Les classes

1. Objectif
2. Déclaration
3. Définition
4. Encapsulation
5. Constructeurs
6. Une classe de complexes
7. Une classe de rationnels
8. Surcharge d'opérateurs
9. Liste d'initialisation
10. Membres statiques
11. Méthodes constantes
12. La classe `string`

Objectif

Une *classe* est la description d'une famille d'objets ayant même structure et même comportement.

Une classe regroupe un ensemble d'*attributs* ou *membres*, répartis en

- un ensemble de *données*
- un ensemble de fonctions, appelées *méthodes*.

Avantages

- simplifie l'utilisation des objets,
- rapproche les données et leur traitement: c'est l'objet qui sait le mieux comment gérer une demande,
- renforce la robustesse du programme
- simplifie la maintenance
- permet l'encapsulation : possibilité de ne montrer de l'objet que ce qui est nécessaire à son utilisation.
- (permet de) masquer l'implémentation

L'utilisateur dispose

- de mécanismes de construction (et de destruction) d'objets;
- de méthodes d'accès et de modification des données encapsulées.

Un *objet* est un élément de la classe. C'est une *instance* de la classe. Il est obtenu par *instanciation*.

La classe permet de produire autant d'exemplaires d'objets que nécessaire.

- Les valeurs des données membres peuvent différer d'une instance à l'autre (sauf pour des données *statiques*, de classe).
- Les méthodes sont les mêmes pour toutes les instances d'une classe. On distingue entre méthodes d'objet (d'instance) et méthodes de classe.

La *déclaration* d'une classe donne la nature des membres (type, signature), et les droits d'accès: `public`, `protected`, `private` (défaut).

La *définition* d'une classe fournit la définition des méthodes.

L'*encapsulation* peut se pratiquer en donnant à l'utilisateur

- un fichier en-tête contenant la déclaration de la classe;
- un module objet contenant la version compilée du fichier contenant la définition de la classe.

Déclaration

La syntaxe est celle des structures. Une `struct` est une classe dont tous les attributs sont publics.

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

- La classe `Point` a deux membres données privés `x` et `y`.
- Elle a trois méthodes publiques `setPoint`, `deplace` et `affiche`.

Définition

L'*opérateur de portée* `::` indique la classe à laquelle appartient la méthode.

```
void Point::setPoint(int a, int b) {
  x = a; y = b ;
}

void Point::deplace (int dx , int dy) {
  x += dx; y += dy;
}

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

Les champs `x`, `y` invoqués dans les méthodes sont ceux de l'objet qui appelle la méthode. Cet objet est explicitement accessible par le pointeur **`this`**. On peut écrire

```
void Point::setPoint(int a, int b) {
  this -> x = a; this -> y = b ;
}
```

Utilisation

```
void main() {
  Point a, b;
  a.setPoint(1, 2);
  b.setPoint(3,-5);
  a.affiche(); // x = 1, y = 2
  b.affiche(); // x = 3, y = -5
  a.deplace(1,1);
  a.affiche(); // x = 2, y = 3
}
```

Encapsulation

Principe

Il s'agit de ne montrer à l'utilisateur que ce qui lui est destiné. Dans une version plus élaborée,

- seulse des fonction d'accès et de modification sont rendues publiques;
- toute donnée est privée ou protégée;
- même la documentation ne fait état que des données et méthodes publiques.

Dans une version simplifiée, l'implémentation n'est disponible que par un module objet.

Exemple. Le projet est composé de trois fichiers

- Un fichier `point.h` de déclaration de la classe `Point`
- Un fichier `point.c` d'implémentation des méthodes de la classe
- Un fichier `main.c` d'utilisation.

Le fichier d'implémentation est à terme remplacé par un module objet `point.o` ou une bibliothèque.

Pour éviter l'inclusion multiple, un fichier d'en-tête contient une directive appropriée pour le préprocesseur.

```
// Fichier point.h
#ifndef POINT_H
#define POINT_H

class Point {
  int x, y;
 public:
  Point (int, int); // constructeur
  void deplace (int, int);
  void affiche ();
};

#endif

// Fichier point.c

#include <iostream.h>
#include "point.h"

Point::Point(int a, int b) {
  x = a; y = b ;
}

void Point::deplace (int dx , int dy) {
  x += dx; y += dy;
}

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

// Fichier main.c
#include "point.h"

void main()
{
  Point a(1,2), b(3,-5);
  a.affiche();
  b.affiche();
  a.deplace(1,1);
  a.affiche();
}
```

Constructeurs

Un *constructeur* est une méthode d'initialisation des attributs d'un objet à la création.

- En C++, un constructeur a le nom de la classe, et pas de type de retour.
- Une classe peut avoir plusieurs constructeurs.
- Un constructeur sans arguments est appelé *constructeur par défaut*.
  - ce constructeur existe implicitement, s'il est le seul constructeur.
  - la définition d'un deuxième constructeur exige que l'on définisse explicitement le constructeur par défaut si l'on veut s'en servir.

Le constructeur par défaut est utilisé

- lors de la définition d'un objet, par
  ```
  X x;
  ```
- lors d'une allocation, par
  ```
  px = new X;
  ```

Sauf écriture explicite, le constructeur par défaut n'effectue pas d'action autre que la réservation de place.

Le *destructeur* est noté `~X()` pour une classe `X`. Il est utilisé

- lorsque le programme quitte le bloc où l'objet est déclaré
- pour la destruction explicite par
  ```
  delete px;
  ```

Un destructeur libère explicitement une place allouée. Ceci est important quand un constructeur fait de l'allocation explicite.

En plus, il existe un constructeur de *copie*, un opérateur d'affectation, et des constructeurs de conversion qui seront vus plus tard.

Exemple. Remplacement de la méthode **setPoint** par un constructeur, comme dans le programme ci-dessus:

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

Point::Point(int a, int b) {
  x = a; y = b ;
}
...
void main()
{
  Point a(1.5,2.5), b(3,-5);
  ...
}
```

Le constructeur **Point** rend inopérant le constructeur par défaut s'il n'est pas redéfini.

Le constructeur est employé en passant les arguments en paramètre. C'est une abbréviation de

```
Point a = Point(1.5,2.5), b = Point(3,-5);
```

Variations

- Un constructeur simple peut être défini *en ligne*, par

```
class Point {
  int x, y;
 public:
  Point (int a, int b) {x = a; y = b;}
  ...
};
```

- Au lieu de construire l'objet puis d'*affecter* des valeurs aux champs, on peut *initialiser* les champs avec les valeurs, par

```
Point (int a, int b) : x(a), y(b) {}
```

Voici des exemples d'emploi de cette classe

1.- Les constructeurs:

```
main () {
  Point p;                // par defaut
  Point q(4,5);           // deuxieme
  Point* a = new Point;      // par defaut
  Point* b = new Point(4,5); // deuxieme
  delete a, b;         // liberation
}
```

2.- Tableaux d'objets:

```
main () {
  const int taille = 7;
  Point* a = new Point[taille];  // par defaut
  for (int i=0; i < taille ; i++)
    cout << i << a[i].x << a[i].y << endl;
  delete [] a;
}
```

La *libération* d'un tableau d'objets par `delete []`

- parcourt le tableau et appelle le destructeur sur chaque élément du tableau;
- puis libère le tableau.

Exemple de constructeur et destructeur

La mention explicite du constructeur par défaut et du destructeur signifie leur redéfinition.

```
class Id {
  string nom;
public:
  Id();
  ~Id();
};
```

Le constructeur et le destructeur sont définis comme suit:

```
Id::Id() {
  cout << "Entrez votre nom : ";
  cin >> nom;
}

Id::~Id() {
  cout <<"Mon nom est "<< nom <<endl;
}
```

Le programme se réduit à:

```
void main()
{
  Id pierre, paul;
}
```

Voici une trace d'exécution:

```
Entrez votre nom : Pierre
Entrez votre nom : Paul
Mon nom est Paul
Mon nom est Pierre
```

Une classe de complexes

```
class Complexe {
  double re, im;
public:
  Complexe() { re = im = 0;}
}
```

Le constructeur par défaut `Complexe()` est explicité pour fixer des valeurs de départ. Ainsi

```
Complexe s, t;
```

définit deux complexes `s`, `t` initialisés à zéro.
Un constructeur (de conversion) est utile pour convertir un couple de `double` en complexes:

```
Complexe::Complexe(double x, double y) {re = x; im = y;}
```

La conversion d'un réel en complexe est faite en mettant `im` à zéro:

```
Complexe::Complexe(double x) {re = x; im = 0;}
```

Si les fonctions sont courtes, on les définit en ligne. Ensemble:

```
class Complexe {
  double re, im;
  Complexe() { re = im = 0;}
  Complexe(double x, double y) {re = x; im = y;}
  Complexe(double x) {re = x; im = 0;}
}
```

En utilisant les arguments par défaut, les trois constructeurs se réduisent en un seul.

Ici, on remplace l'affectation par l'initialisation, et on ajoute une méthode:

```
class Complexe {
  double re, im;
  Complexe(double r = 0, double i = 0) : re(r), im(i) {}
  double getModule() { return sqrt(re*re + im*im); }
}
```

On s'en sert

- à l'initialisation, par exemple

  ```
  Complexe h, a(3,5), b = Complexe(2,4);
  Complexe c(5), d = Complexe(6), e = 7;
  ```

  Par conversion de type, `7` est transformé en `Complexe(7)` puis copié.

- pour la réalisation d'une conversion dans une expression:

  ```
  void f(Complexe z) {...}
    ...
  f(3.14)
  ```

- par appel explicite dans une instruction

  ```
  r = Complexe(3,4).module();
  ```

Une implantation plus fournie. La déclaration est:

```
#include <iostream.h>
#include <math.h>

const double pi = 3.14159265358979323846;
// Premiere implementation :
// re, im sont la partie reelle et la partie imaginaire;

class Complexe {
public :
  Complexe(double re = 0, double im = 0);
  Complexe(double rho, double theta, int polaire);
  double getRe();
  double getIm();
  double getModule();
  double getArg();
  void setRe(double);
  void setIm(double);
  void setModule(double);
  void setArg(double);
  void showXY() { cout << getRe() << " " << getIm() <<" "; }
  void showRT() { cout << getModule() << " " << getArg() <<" "; }
private:
  double re, im; // Premiere implementation
  static double getArgFromCart(double x, double y);// helper
};
```

Même si **re** et **im** sont privés ils sont “vus” par un utilisateur. Voici l’implémentation.

```
Complexe::Complexe(double re, double im): re(re), im(im){}
Complexe::Complexe(double rho, double theta, int polaire) {
  re = rho*cos(theta);
  im = rho*sin(theta);
}
double Complexe::getRe() { return re; }
double Complexe::getIm() { return im; }
double Complexe::getModule() { return sqrt(re*re+im*im); }
```

```
double Complexe::getArg() { return getArgFromCart(re,im); }
void Complexe::setRe(double x) { re = x; }
void Complexe::setIm(double y) { im = y; }
void Complexe::setModule(double r) {
  double m = getModule();
  if(!m)
    re = r;
  else {
    re = re*r/m; im = im*r/m;
  }
}
void Complexe::setArg(double arg) {
  double r = getModule();
  re = r* cos(arg);
  im = r* sin(arg);
}

double Complexe::getArgFromCart(double x, double y) {
  if (!x) return atan(y/x);
  if (y >0) return pi/2;
  if (y < 0) return -pi/2;
  return 0;
}
```

Quelques appels:

```
void main()
{
  Complexe a = Complexe(3,4);
  cout << a.getModule() << endl;
  cout << a.getArg() << endl;
  a.setModule(10);
  a.showXY();a.showRT();
  a.setArg(pi/2);
  a.showXY();a.showRT();
}
```

Voici une deuxième implémentation, en coordonnées polaires.

Aucun intitulé de méthode n'est modifié

```
#include <iostream.h>
#include <math.h>

const double pi = 3.14159265358979323846;

class Complexe {
public :
  Complexe(double re, double im = 0);
  Complexe(double rho, double theta, int polaire);
  double getRe();
  double getIm();
  double getModule();
  double getArg();
  void setRe(double);
  void setIm(double);
  void setModule(double);
  void setArg(double);
  void showXY() { cout << getRe() << " " << getIm() <<" "; }
  void showRT() { cout << getModule() << " " << getArg() <<" "; }
private:
  double rho, theta; // Deuxieme implementation
  static double getArgFromCart(double x, double y);
};
```

L'implantation change évidemment.

```
Complexe::Complexe(double re, double im) {
  rho = sqrt(re*re+im*im);
  theta = getArgFromCart(re,im);
}
Complexe::Complexe(double rho, double theta, int polaire):
  rho(rho), theta(theta) {}

double Complexe::getRe() { return rho*cos(theta); }
double Complexe::getIm() { return rho*sin(theta); }
double Complexe::getModule() { return rho; }
double Complexe::getArg() {return theta; }
void Complexe::setRe(double re) {
  double im = getIm(); rho = sqrt(re*re+im*im);
  theta = getArgFromCart(re,im);
}
void Complexe::setIm(double im) {
  double re = getRe(); rho = sqrt(re*re+im*im);
  theta = getArgFromCart(re,im);
}
```

```
void Complexe::setModule(double r) { rho = r; }
void Complexe::setArg(double arg) { theta = arg; }

double Complexe::getArgFromCart(double x, double y) {
  if (!x) return atan(y/x);
  if (y >0) return pi/2;
  if (y < 0) return -pi/2;
  return 0;
}
```

En revanche, rien ne se voit dans le programme:

```
void main()
{
  Complexe a = Complexe(3,pi/4,0);
  cout << a.getModule() << endl;
  cout << a.getArg() << endl;
  a.setModule(10);
  a.showXY();a.showRT();
  a.setArg(pi/2);
  a.showXY();a.showRT();
  Complexe b(5);
  b.setArg(pi/4);
  b.showXY();b.showRT();
}
```

Un *attribut* n'a pas d'existence en dehors d'un objet. L'implantation n'assure pas l'existence physique d'un attribut.

Par exemple, le *module* n'est accessible que par `getModule()` et `setModule()`, mais il n'est pas assuré qu'il existe une données membre qui maintient la valeur courante.

On appelle *attribut calculé* un attribut qui n'est pas physiquement présent dans une donnée membre. Si l'implantation cache les données (*data hiding*), un utilisateur ne peut pas voir la différence entre un attribut présent et un attibut calculé.

Une classe de rationnels

L'*objectif* est de montrer l'utilisation de surcharges, cette fois-ci sur les opérateurs arithmétiques et d'entrée-sortie.

Un *nombre rationnel* est toujours réduit, et le dénominateur est toujours positif. D'où la nécessité d'une méthode de réduction.

```
class Rat {
  static int pgcd(int,int);
public:
  int num, den;
  void red();  // reduit la fraction
  Rat(int n = 0) : num(n), den(1) {}
  Rat(int n, int d) : num(n), den(d) { red();}
};
```

Il y a deux constructeurs; seul le deuxième fait usage du réducteur.

```
Rat a;      // a = 0
Rat b(3);   // b = 3
Rat c(3,2); // c = 3/2
Rat d = 7;  // d = 7
```

Par conversion de type, `7` est transformé en `Rat(7)` puis `d` est initialisé à cette valeur.

La réduction fait appel à une fonction de calcul de pgcd:

```
void Rat::red() {
  int p = pgcd( (num > 0) ? num : -num, den);
  num /= p; den /= p;
}

int Rat::pgcd(int a, int b) {
  return (!b) ? a : pgcd(b, a%b);
}
```

Une *méthode* déclarée `static`

- peut être appelée sans référence à une instance
- peut être appelée par référence à sa classe
- n'a pas de pointeur `this` associé.

Une *donnée* déclarée `static` est commune à toutes les instances.

Surcharge d'opérateurs

Comme toute fonction, un opérateur peut également être surchargé. Pour surcharger un opérateur *op* on définit une nouvelle fonction de nom

`operator`*op*

Par exemple `operator=` ou `operator+`. La syntaxe est

type `operator`*op*`(`*types des opérandes*`);`

Par exemple

```
Rat operator+(Rat,Rat);
```

L'expression

```
a+b
```

équivaut à

```
operator+(a, b)
```

Un opérateur surchargé peut se définir

- au niveau global,
- comme membre d'une classe.

Apport de la surcharge: le calcul s'écrit comme pour un type élémentaire !

Nous définissons les quatre opérations arithmétiques par:

```
Rat operator+(Rat a, Rat b) {
  return Rat(a.num*b.den + a.den*b.num, a.den*b.den);
}

Rat operator-(Rat a, Rat b) {
  return Rat(a.num*b.den - a.den*b.num, a.den*b.den);
}

Rat operator/(Rat a, Rat b) {
  return Rat(a.num*b.den, a.den*b.num);
}

Rat operator*(Rat a, Rat b) {
  return Rat(a.num*b.num, a.den*b.den);
}
```

Le moins unaire se définit par:

```
Rat operator-(Rat a) {
  return Rat(-a.num, a.den);
}
```

Les rationnels passé en argument ne sont pas modifiés, on peut donc les spécifier `const Rat&`.

```
Rat operator+(const Rat& a, const Rat& b) {...}

Rat operator-(const Rat& a, const Rat& b) {...}
...
```

L'opérateur d'incrémentation `+=` se surcharge également:

```
Rat& operator+=(Rat& a, const Rat& b) {
  a.num = a.num*b.den + a.den*b.num;
  a.den = a.den*b.den; a.red();
  return a;
}
```

L'un des appels

```
a += b; //ou
a = operator+=(a,b);
```

- construit dans `a` passé par référence, la valeur incrémentée,
- et en transmet la référence en sortie.

Ici, une définition en méthode est plus logique (voir plus loin).

Les opérateurs d'entrée et de sortie se surchargent également:

```
ostream& operator<<(ostream& out, Rat r) {
  return out << r.num << "/" << r.den;
}
istream& operator>>(istream& in, Rat& r) {
  in >> r.num >> r.den; r.red();
  return in;
}
```

Les opérateurs d'entrée et de sortie prennent et retournent la *référence* au flot en argument, pour éviter une copie.
Noter la référence dans la lecture !

Et voici un test:

```
void main()
{
   Rat a,b,c;
   cout << "1 - 1/2 = "<<1 - Rat(1)/2<<endl;
   cout << "Donner deux rationnels : "; cin >> a >> b;
   cout << "J'ai lu: " << a <<" "<< b;
   cout << endl;
   cout <<  -a<< endl;
   cout <<  -a +1<< endl;
   cout <<  -a +1 - (Rat) 1/2<< endl;
   cout <<  -a  +1 - (Rat) 1/2 +a << endl;
   c = -a +1 - (Rat) 1/2 +a +b*2;
   cout << "\nResultat " << c << endl;
}
```

Ceci donne:

```
1 - 1/2 = 1/2
Donner deux rationnels : 2 5 8 4
J'ai lu: 2/5 2/1
-2/5
3/5
1/10
1/2

Resultat 9/2
```

Voici un emploi, proche d'un calcul formel. Les *nombres de Bernoulli* B_n sont des nombres rationnels qui vérifient

$$B_n = \sum_{k=0}^{n} \binom{n}{k} B_k$$

avec $B_0 = 1$, $B_1 = -1/2$. On montre que $B_{2n+1} = 0$ pour $n > 1$, et que

$$B_{n-1} = -\frac{1}{n} \sum_{k=0}^{n-2} \binom{n}{k} B_k$$

C'est cette dernière formule que nous calculons:

```
void Bernoulli(int N, Rat B[]) {
  B[0] = 1;
  B[1] = Rat(-1,2);
  for (int n = 2; n<=N ; n++)
    B[n] = 0;
  for (int n = 3; n<= N; n += 2) {
    Rat s;
    for (int k = 0; k<= n-2; k++)
      s += binomial(n,k)*B[k];
    B[n-1] = -s/n;
  }
}
```

Il faut noter l'apport de la surcharge: le calcul s'écrit comme pour un type élémentaire !

```
void main() {
  const N = 22;
  Rat B[N+1];

  Bernoulli(N, B);
  for (int i = 0; i<=N; i++)
    cout <<"B("<<i<<") = "<< B[i] << endl;
}
```

Liste d'initialisation

Un constructeur peut recevoir une *liste d'initialisations* qui sont effectuées à la construction de l'objet.

Ceci est important quand un ou des champs de l'objet sont eux-mêmes des objets.

La construction d'un objet par un constructeur *sans* liste d'initialisation se fait en trois étapes:

- réservation de la place pour l'objet;
- construction des champs par appel du constructeur par défaut de chacun des objets;
- exécution du corps de la fonction.

En présence d'une liste d'initialisation, les constructeurs par défaut sont remplacés par les appels spécifiés dans la liste.

Exemple : une classe de points

```
class Point {
    int x, y;
  public:
    Point(int abs = 0, int ord = 0);
};
```

et une classe de segments, formées de deux points.

```
class Segment {
  Point d,f;
public:
  Segment(int dx,int dy, int fx, int fy);
};
```

Première implémentation du constructeur

```
Segment::Segment(int dx,int dy, int fx, int fy) {
  d = Point(dx,dy); f = Point(fx,fy);
}
```

La construction

```
Segment k = Segment(1,2,3,4);
```

provoque:

- deux appels au constructeur par défaut `Point()` pour l'initialisation de `d` et de `f`, et ceci avant d'entrer dans le corps du constructeur de segments;
- les appels de constructeurs `Point(1,2)` et `Point(3,4)` pour construire des objets temporaires;
- deux emplois de l'opérateur d'affectation;
- deux appels au destructeur pour détruire les objets temporaires.

Deuxième implémentation

```
Segment::Segment(int dx,int dy, int fx, int fy)
  : d(dx,dy), f(fx,fy) {}
```

La construction ne nécessite que deux appels de constructeurs, pour construire `d` à partir de `(dx,dy)` et de `f` à partir de `(fx,fy)`.

Utiliser le plus souvent possible les listes d'initialisation.

Membres statiques

Parmi les attributs, on distingue

- les membres de classe, qualifiés de *statiques*;
- les membres d'objet.

Une donnée membre *statique* est spécifiée par **`static`**.

- Elle n'est instanciée qu'une seule fois;
- Elle est commune à toutes les instances de la classe;
- Si un objet la modifie, elle est modifiée pour tous les objets.
- Elle est initialisée avant utilisation.

Une donnée statique et constante est une donnée immuable.
Un méthode *statique* est spécifiée par **`static`**.

- Elle ne fait pas référence à un objet (pas de **`this`**);
- Elle est appelée comme une fonction globale;
- S'il faut préciser sa classe, son nom est préfixé à l'aide de l'opérateur de portée : `Math::pgcd()`.

```
#include <iostream.h>

class XY {
 public:
   static int app;
   XY() { cout << "+ : " << ++app << endl; }
   ~XY() { cout << "- : "<< app-- << endl; }
};

int XY::app = 0; // doit etre initialise
```

```
void main() {
  XY a, b, c;
  {
    XY a, b;
  }
  XY d;
}
```

Résultat:

```
+ : 1
+ : 2
+ : 3
+ : 4
+ : 5
- : 5
- : 4
+ : 4
- : 4
- : 3
- : 2
- : 1
```

Méthodes constantes

Une méthode qui ne modifie pas l'objet appelant peut recevoir le qualificatif **`const`** en suffixe. Les "getter functions" sont de ce type.

Une des disciplines de programmation objet suggère:

- les données membres sont *privées* ou *protégées*;
- on peut obtenir la valeur d'un membre donnée par une fonction d'accès (*getter function*);
- on autorise la modification d'une donnée membre en fournissant une fonction de modification (*setter function*);

Cette discipline s'étend au nom des fonctions. Plusieures variantes de cette discipline existent. *Toujours se conformer à la culture de l'entreprise.*

Une proposition:

- une donnée membre (privée) a un nom qui finit par un soulignement, comme **`data_`**. Ce nom est parfois précédé d'un ensemble de lettres qui décrivent son type (**`lp_data_`** pour "long pointer") appelée notation hongroise;
- la "getter function" a le nom **`getdata()`**, ;
- la "setter function" a le nom **`setdata()`**.

Une autre proposition (Java):

- le nom d'une donnée membre commence par une minuscule; chaque mot qui suit commence par une majuscule : **`rayonGlobal`**.

- la “getter function” a le nom **`getRayonGlobal()`**;
- la “setter function” a le nom **`setRayonGlobal()`**.

La “getter function” ne modifie pas la donnée. Elle reçoit le suffixe **`const`**.

```
class Rat {
protected :
  int num_, den_;
public:
  ...
  int getnum() const { return num_;}
  int getden() const { return den_;}
  ...
  void setnum(int num){ num_ = num;}
  void setden(int den){ den_ = den;}
};
```

La classe `string`

La classe **`string`** de C++ facilite la manipulation de chaînes de caractères.

Pour employer la classe **`string`**, inclure le fichier en-tête **`string`**.

```
#include <string>
```

L'implémentation de la classe **`string`** est complexe, et basée sur plusieurs autres classes. Nous décrivons ses possibilités comme si elle était implémentée directement.

Constructeurs

`string()`
: contructeur par défaut, initialise une chaîne vide.

`string(const char* c)`
: construit une chaîne en l'initialisant avec `c`.

`string(const string& a)`
: contructeur par copie, initialise la chaîne avec une copie de `a`.

```
string c;           // vide
string a = "Lotus";// string("Lotus")
string b("bleu");   // string("bleu")
string d(b);        // copie
for (int i = 0; i < a.size() ; i++)
  cout << a[i] << ' ';
```

affiche `L o t u s`.

Noter que

- `size()` retourne la taille.
- `a[i]` rend le caractère correspondant à l'indice, par surcharge de l'opérateur d'indexation.
- `at()` joue le même rôle: `a.at(i)` équivaut à `a[i]`, mais en plus il y a contrôle de la validité de l'indice.

Opérateurs

- `"+"` pour la concaténation.

  ```
  string a = "Lotus", b("bleu");
  string c = a + " " + b;
  cout << c << endl;
  ```

  donne `Lotus bleu`.

- `"+="` de concaténation incrémentale:

  ```
  string a = "Lotus"; a += " bleu";
  ```

  donne à nouveau `Lotus bleu`.

- les 6 opérateurs de comparaison comme opérateurs globaux, et notamment `==` pour la comparaison de chaînes.

  ```
  string a = "Lotus bleu";
  cout ("bleu" > a) << endl;
  ```

  donne `1`. Aussi,

  ```
  string x;       //Chaine vide
  string y = "";  //Chaine contenant mot vide
  cout<< x.size() << y.size() << (x==y) << endl;
  ```

  donne (heureusement) `001`.

Autres méthodes

`replace(int d, int n, string b)`

remplace dans la chaîne appelante `n` caractères à partir de la position `d` par la chaîne `b`.

```
string a = "Lotus bleu ciel", b = "jaune";
a.replace(6,4, b);
```

donne `Lotus jaune ciel`.

`compare(const string& b)`

retourne un entier négatif, nul, positif si la chaîne appelante est inférieure, égale, supérieure à `b`.

```
string x = "adam", y = "eve", z = "evelyne";
cout << x.compare(y) <<' '<< y.compare(z) <<' '
  << z.compare(z) <<' '<< z.compare(x)<< endl;
```

donne `-4 -4 0 4`.

Conversion en chaînes C

`const char* c_str()`

copie les caractères dans une chaîne C constante, terminée par le caractère nul.

```
string s = "coup de main";
s[4] = 0; // caractere nul
cout << s << endl;
const char* c = s.c_str();
cout << c << endl;
```

donne

```
coupde main
coup
```

`int copy(char *t, int n, int d = 0)`

copie dans la chaîne existante **`t`** au plus **`n`** caractères à partir de la position **`d`**.

```
string s = "coup de main";
char txt[20];
s.copy(txt, 4, 8);
txt[4] = 0;
cout << txt << endl;
```

donne **`main`**.

5

Héritage

1. Objectif
2. Classe composée ou classe dérivée
3. Syntaxe
4. Accès aux données et méthodes
5. Classes dérivées et constructeurs
6. Héritage multiple
7. Polymorphisme
8. Méthodes virtuelles
9. Un exemple: les expressions
10. Evaluation de polonaise postfixée, sans arbre
11. Table de fonctions virtuelles

Objectif

L'*héritage* est l'un des principes fondamentaux de la programmation objet. Il a pour objectif de hiérarchiser les classes et les objets.

L'héritage est mis en œuvre par la construction de *classes dérivées*.

Une classe dérivée

- contient les données membres de sa classe de base;
- peut en ajouter de nouvelles;
- possède *a priori* les méthodes de sa classe de base;
- peut redéfinir (*masquer*) certaines méthodes;
- peut ajouter de nouvelles méthodes.

L'héritage peut être simple ou multiple.

Avantages:

- Une classe dérivée modélise un *cas particulier* de la classe de base, et peut donc utiliser la classe de base;
- l'héritage encourage la recherche de la *généralité* et de la *généricité*;
- une hiérarchie de classes facilite la solution de problèmes complexes;
- facilite la maintenance, le développement et les extensions;
- peut ajouter de nouvelles méthodes.

Classe composée ou classe dérivée

Une classe est *composée* si certains de ses membres sont eux-mêmes des objets.

```
class DescrHabitant {
protected:
  string nom;
  string domicile;
};

class Carte {
  DescrHabitant h;
  int age;
  float age;
};
```

La description d'un habitant fait *partie* d'une carte d'identité; correct car une carte d'identité n'est pas un cas particulier d'un habitant.

```
class DescrResident : public DescrHabitant{
protected:
  string residence;
};
```

La description d'un résident est *plus complète* que celle d'un habitant: l'information supplémentaire est l'adresse de sa résidence (qui peut être différente de celle de son domicile).

Syntaxe

La syntaxe pour la définition d'une classe dérivée est

`class` *classe-derivee* `:` *protection* *classe-de-base* `{...};`

Les types de protection sont `public`, `protected`, `private`. En général, on choisit `public`.

Autres exemples

- Une voiture comporte quatre (ou cinq) roues, donc

```
class Automobile {
  Roue roues[5];
  ...
}
```

- Un segment comporte deux points;
- Un anneau est un cas particulier d'un cercle;
- Une forme contient un rectangle englobant:

```
class Shape {
  Rect r;
  ...
}
```

- Un triangle est une forme particulière:

```
class Triangle : public Shape {
  ...
}
```

Discussion : un carré est-il un cas particulier d'un rectangle ?

Accès aux données et méthodes

Les attributs de la classe de base peuvent être désignés directement ou par le nom de la classe de base, en employant l'opérateur de portée `::`

```
class DescrHabitant {
  string nom;
  string domicile;
public:
  DescrHabitant();
  void show();
};

class DescrResidence : public DescrHabitant {
  string residence;
public:
  DescrResidence();
  void show();
};
```

Implantations

```
DescrHabitant::DescrHabitant() {
  cout << "Entrez votre nom : "; cin >> nom;
  cout << "Entrez votre domicile : "; cin >> domicile;
}
void DescrHabitant::show() {
  cout << "Nom: \t\t" << nom << endl;
  cout << "Domicile \t" << domicile << endl;
}

DescrResidence::DescrResidence() {
  cout << "Entrez votre residence : "; cin >> residence;
}
```

```
void DescrResidence::show() {
  DescrHabitant::show();
  cout << "Residence \t" << residence << endl;
}
void main()
{
  DescrHabitant pierre;
  pierre.show();
  DescrResidence paul;
  paul.show();
  pierre = paul;
  pierre.show();
}
```

Dans la définition de **`DescrResidence::show()`**, l'appel est fait à la méthode de la classe de base.

Dans le constructeur de **`DescrResidence`**, il y a appel du constructeur par défaut de la classe de base.

```
Entrez votre nom : Dupont
Entrez votre domicile : Paris
Nom:            Dupont
Domicile        Paris
Entrez votre nom : Duval
Entrez votre domicile : Marne
Entrez votre residence : Champs
Nom:            Duval
Domicile        Marne
Residence       Champs
Nom:            Duval
Domicile        Marne
```

Le *principe de substitution* s'énonce comme suit: toute occurrence d'un objet de la classe de base peut toujours être remplacée par un objet d'une classe dérivée.

La redéfinition d'une méthode dans une classe dérivée est le *masquage*. Dans

```
class Base { ...
  public : void f(char* c);...
};
classe Der : public Base {
  public : void f(int i);...
};
```

il y a masquage, et non surcharge, même si les signatures des méthodes sont différentes. Les fonctions ne sont pas dans le même espace de "portée"

Pour un objet **d** de la classe **Der**,

```
int i;...
d.f(i);
```

désigne la méthode de **Der::f(int)**. L'appel

```
d.f("Ben"); // erreur
```

est rejeté, car il n'y a pas de méthode **Der::f(char*)**. On peut écrire

```
d.Base::f("Ben");
```

Classes dérivées et constructeurs

La construction d'un objet d'une classe dérivée comporte

- la réservation de la place pour l'objet;
- l'appel d'un constructeur approprié (constructeur par défaut) de la classe de base;
- appel des constructeurs pour les membres objets;
- l'exécution du corps de la fonction constructeur.

Un constructeur autre que celui par défaut est indiqué dans la *liste d'initialisation*.

```
class Animal {
  protected:
    string nom;
  public:
    Animal(string n) : nom(n) {}
    void show() {cout << nom <<endl;}
};

class Ours: public Animal {
    string couleur;
  public:
    Ours(string n, string c) : Animal(n), couleur(c) {}
    void show() { cout << nom <<' '<< couleur << endl; }
};
```

```
void main()
{
  Animal a("Elephant"); a.show();
  Ours o("Panda","blanc"); o.show();
}
```

La sortie est

```
Elephant
Panda blanc
```

- La méthode **`show()`** de la classe dérivée *masque* la méthode de la classe de base, mais
- Dans une fonction, un paramètre formel objet de la classe de base peut être substitué par un objet de la classe dérivée (principe de substitution). Il y a *conversion* de l'objet dérivé en objet de la classe de base.

Ainsi

```
o.Animal::show();
```

affiche simplement **`Panda`**. Définissons une fonction globale

```
void affiche(Animal* a) {
  cout <<"Nom : "; a ->show();
}
```

Le résultat est

```
Nom : Panda
```

car il y a conversion de type. (Si, en revanche, la méthode **`show()`** de la classe **`Animal`** est définie **`virtual`**, la conversion n'a pas le même effet.)

Héritage multiple

Une classe peut hériter de plusieurs classes. Les classes de bases sont alors énumérées dans la définition.

```
class D : public B1, public B2 {...}
```

L'héritage multiple est utile dans l'emploi d'interfaces (voir plus loin).

Exemple : Listes chaînées

Une *liste chaînée* comporte deux parties de nature différente, les données et le chaînage.

• Une *liste "pure"* est formée de cellules dont chacune ne contient qu'un pointeur vers la suivante. Le constructeur insère une cellule.

```
typedef class Cellule* Liste;
class Cellule {
  protected:
    Liste suiv;
  public:
    Cellule(Liste a) : suiv(a) {}
};
```

- Un *"Entier"* a une donnée membre unique.

```
class Entier {
  protected:
    int val;
  public:
    Entier(int i) : val(i) {}
    void show() { cout << val << ' ';}
};
```

- Une classe de *listes d'entiers* dérive des deux classes:

```
typedef class CelluleEntiers* ListeEntiers;
class CelluleEntiers : public Cellule, public Entier {
public:
  CelluleEntiers(Liste a, int i) : Cellule(a),Entier(i) {}
  void show();
};
```

La classe a deux champs

- le champ `int val` hérité de la classe `Entier`;
- le champ de `Liste suiv` hérité de la classe `Liste`.

La fonction d'affichage dépend des deux classes de bases:

- de `Entier` pour l'affichage des éléments;
- de `Cellule` pour le chaînage.

Elle s'écrit

```
void CelluleEntiers::show() {
  Entier::show();
  if (suiv)
    ((ListeEntiers) suiv) -> show();
};
```

La conversion de `suiv` du type `Liste` en type `ListeEntiers` est nécessaire pour la cohérence.

```
void main() {
  ListeEntiers a = 0; int j;
  for (int i = 0; i < 5; i++) {
    cin >> j;
    a = new ListeEntiers(a, j);
  }
  a -> show();
}
```

Polymorphisme

L'*édition de lien* est l'opération qui associe, à l'appel d'une fonction, l'adresse du code de la fonction.

On distingue

- La liason *statique* (ou précoce): effectuée à la compilation
- La liason *dynamique* (ou tardive) : effectuée à l'exécution.

Le *polymorphisme* est la capacité, pour une fonction, d'être interprétée différemment en fonction de l'objet auquel elle s'applique.
On distingue

- Le polymorphisme *statique* : c'est la surcharge. Le sens de la fonction est déterminé statiquement par la signature des arguments à l'appel.
- Le polymorphisme *dynamique* : c'est le véritable polymorphisme. Le sens de la fonction est déterminé par le type effectif de l'objet à l'appel.

Le polymorphisme ne s'applique qu'aux méthodes d'une classe.
Avantages:

- encourage l'abstraction,
- facilite l'extensibilité.

Méthodes virtuelles

Les *méthodes virtuelles* sont des fonctions membres réalisant le polymorphisme, c'est-à-dire le choix de la méthode en fonction du type des arguments *à l'appel* : un argument d'une classe dérivée utilise les méthodes de sa classe dérivée, et non de la classe du paramètre formel. En C++ (et en Java), le polymorphisme n'est réalisable que sur le "zéro-ieme" argument, c'est-à-dire sur l'objet appelant la méthode.

De plus, en C++ deux conditions doivent être réunies pour que le polymorphisme prenne effet (en Java, c'est le comportement par défaut)

- La méthode doit être déclarée virtuelle, par le préfixe `virtual`
- La méthode est appelée sur l'objet manipulé par pointeur ou par référence, pour qu'il n'y ait pas d'édition de lien statique.

Déclaration de trois classes

```
class X {
public :
  int i;
  virtual void print();
  void show();
  X() { i=5;}
};

class XX : public X {
public :
  int j;
  virtual void print();
  void show();
  XX() { j=7;}
};

class XXX : public XX {
public :
  int k;
  void print();
  void show();
  XXX() { k=9;}
};
```

Chacune (sauf `XXX`) a une méthode `print` virtuelle, et une méthode `show` non virtuelle.

Définition des diverses méthodes. Noter : on ne répète pas le mot-clé **virtual** dans la définition.

```
void X::print() {
  cout << "La classe est X " << i <<endl; }
void X::show() {
  cout << "La Classe est X " << i <<endl; }
void XX::print() {
  cout << "La classe est XX "<<i<<" "<<j<<endl; }
void XX::show() {
  cout << "La Classe est XX "<<i<<" "<<j<<endl; }
void XXX::print() {
  cout <<"La classe est XXX "<<i<<" "<<j<<" "<<k<<endl;}
void XXX::show() {
  cout << "La Classe est XXX "<<i<<" "<<j<<" "<<k<<endl;
}
```

La méthode utilisée dépend de la classe de l'objet, et la classe de l'objet est définie à la déclaration.

Forcer la conversion est possible, mais risqué car on accède à une zone mémoire non allouée.

```
void main() {
  X z;
  XX zz;
  XXX zzz;

  z.print(); z.show();
  zz.print(); zz.show();
  zzz.print(); zzz.show();
  z = zz; z.print(); z.show(); // Conversion, ok
  zz = (XX) z; zz.print(); zz.show(); // danger
  zzz = (XXX) zz; zzz.print(); zzz.show(); // danger
}
```

```
La classe est X 5
La Classe est X 5
La classe est XX 5 7
La Classe est XX 5 7
La classe est XXX 5 7 9
La Classe est XXX 5 7 9
La classe est X 5
La Classe est X 5
La classe est XX 5 25720520
La Classe est XX 5 25720520
La classe est XXX 5 25720520 25720524
La Classe est XXX 5 25720520 25720524
```

Utilisation de pointeurs vers des objets.

```
void main() {
  X* a;  XX* aa;  XXX* aaa;
  a = new X; a -> print(); a -> show();
  aa = new XXX; aa -> print(); aa -> show();
  aaa = (XXX*) new X; aaa -> print(); aaa -> show();
}
```

Les résultats sont:

```
La classe est X 5
La Classe est X 5
La classe est XXX 5 7 9
La Classe est XX 5 7
La classe est X 5
La Classe est XXX 5 969289320 14
```

- La méthode *virtuelle* appliquée est celle de la classe de l'objet à sa *création*.
- La méthode non virtuelle est celle de la classe de la variable désignant l'objet.

Plus formellement:

Si une classe `Base` contient une fonction `f` spécifiée `virtual`, et si une classe `Der` dérivée contient une fonction `f` de même type, l'appel de `f` sur un objet de classe `Der` invoque `Der::f` *même si l'accès se fait à travers un pointeur ou une référence sur* `Base`.

Le polymorphisme (dynamique) *ne s'applique pas* aux paramètres de fonctions: la sélection en cas de surcharge est faite sur le type statique du pointeur.

```
void afficher(Base* b) {...}
void affficher(Der* d) {...}
main() {
Base* x = new Der;
afficher(x); } // utilise la premiere fonction !
```

Retours sur les animaux

```
class Animal {
  protected:
    string nom;
  public:
    Animal(string n) : nom(n) {}
    virtual void show() { cout << nom << endl; }
};

class Ours: public Animal {
    string couleur;
  public:
    Ours(string n, string c) : Animal(n), couleur(c) {}
    void show() { cout << nom <<' '<< couleur << endl; }
};
```

Quel est le résultat de:

```
Animal* o = new Ours("Panda","blanc");
o->show();
```

Même résultat pour

```
Ours b = Ours("Ours","brun");
Animal& O = b;
O.show();
```

Données groupées

On veut grouper, dans un tableau, des textes et des entiers (comme par exemple dans un tableur). Les données de base dérivent d'une classe commune:

```
class Item {
 public:
  virtual void show() = 0;
};

class Texte: public Item {
 protected:
  string t;
 public:
  Texte(string t) : t(t) {}
  void show() {cout <<"Texte : " << t <<endl ; }
};

class Entier: public Item {
  int val;
 public:
  Entier(int val) : val(val) {}
  void show() { cout << "Entier : " << val << endl; }
};
```

La classe `Item` est *abstraite*, sa fonction `show()` est *virtuelle pure*, la classe `Item` ne peut être instanciée, elle correspond à un *concept*.

```
void main() {
  Item* donnee[10];
  donnee[0] = new Texte("Bonjour");
  donnee[1] = new Entier(2000);
  donnee[2] = new Texte("monde");

  for (int i = 0; i < 3; i++)
    donnee[i] -> show();
}
```

virtuelle pure ou virtuelle par défaut ?

On aurait aussi pu définir

```
class Item {
 public:
  virtual void show(){};
};
```

Alors, la fonction **show** n'est plus virtuelle pure, la classe **Item** peut être instanciée, et la définition de **show** donne un *comportement par défaut*.

```
class Figure {
 public:
  virtual void draw() =0;  //pure
  virtual void paint(){};  //rien par defaut
};

class Rectangle : public Figure{
 public:
  void draw() {...};  //masque
  void paint(){...};  //masque aussi
};

class Segment : public Figure {
 public:
  void draw() {...};  //masque
  // ne masque pas paint
};
```

Un exemple: les expressions

Evaluation des expressions arithmétiques, sous le double aspect:

- Construction de l'arbre d'expression,
- Evaluation de cet arbre.

L'expression est lue sous forme *postfixée*. L'analyseur construit progressivement la pile des arbres des sous-expressions reconnues, et à la fin imprime la valeur de l'expression.

Voici la fonction principale

```
void main()
{
  Expression e;
  cout << "Donnez une expression postfixee\n";
  e = construction();
  cout << "Expression : ";
  e -> print();
  cout << "\nValeur : "<<e -> eval()<<endl;
}
```

Une *expression* est un concept *abstrait* : seules existent des sommes, produits etc, et des nombres. On définit donc les expressions comme classes abstraites.

Voici un exemple d'exécution:

```
Donnez une expression postfixee
3 4 + 2 - 2 / !
Je lis un entier 3
Pile : 3   . = 3
Je lis un entier 4
Pile : 4   3   . = 4
Je lis un char +
Pile : + 3 4   . = 7
Je lis un entier 2
Pile : 2    + 3 4   . = 2
Je lis un char -
Pile : - + 3 4 2   . = 5
Je lis un entier 2
Pile : 2    - + 3 4 2   . = 2
Je lis un char /
Pile : / - + 3 4 2 2   . = 2
Je lis un char !
Pile : ! / - + 3 4 2 2   . = 2
Expression : ! / - + 3 4 2 2
Valeur : 2
```

Voici la déclaration de la classe des expressions.

```
typedef class ExpressionR* Expression;
class ExpressionR {
public:
  virtual void print() =0;
  virtual int eval() =0;
};
```

Chaque opération arithmétique est représentée par une sous-classe. Il y en a 6:

```
Addition
Soustraction
Multiplication
Division
Factorielle
Simple
```

Les quatre premières sont “binaires”, la `Factorielle` est “unaire”, la dernière est zéroaire.

Voici la déclaration des sous-classes (Une seule des classes d'opérateurs binaires est montrée).

```
class Soustraction: public ExpressionR {
    Expression gauche, droite;
  public:
    Soustraction(Expression, Expression);
    int eval();
    void print();
};

class Factorielle: public ExpressionR {
    Expression unique;
  public:
    Factorielle(Expression);
    int eval();
    void print ();
};

class Simple: public ExpressionR { //Zeroaire
    int valeur;
  public:
    Simple(int);
    void print();
    int eval();
};
```

Définition des méthodes des classes.

```
Soustraction::Soustraction(Expression a,Expression b) :
  gauche(a), droite(b) {}
void Soustraction::print() {
  cout << "- " ; gauche -> print(); droite -> print();
}
int Soustraction::eval() {
  return gauche -> eval() - droite -> eval();
}


Factorielle::Factorielle(Expression a): unique(a){}
void Factorielle::print() {
  cout << " ! " ; unique -> print();
}
int Factorielle::eval() {
  int n, v = unique -> eval();
  n = v;
  while (--v) n *= v;
  return n;
}


Simple::Simple(int n) : valeur(n) {}
void Simple::print() { cout << valeur << " " ;}
int Simple::eval() { return valeur;}
```

L'analyse syntaxique d'une expression utilise une fonction globale barbare `getlex(x,i)` qui retourne

- `-1` à la fin de la lecture;
- `0` si le lexème lu est un entier;
- `1` si le lexème est un des caractères désignant un opérateur binaire ou la factorielle.

La construction de l'arbre est fonction du lexème retournée.

```
Expression construction() {
  Pile p; char x;
  Expression g, d; int i;
  for (;;) {
    switch (getlex(x,i)) {
    case -1:
      return p.pop();
    case 0:
      p.push(new Simple(i)); break;
    case 1:
      if (x == '!')
        p.push(apply(x,p.pop()));
      else {
        d = p.pop();
        g = p.pop();
        p.push(apply(x, g, d));
      }
    }
  }
  return NULL;
}
```

La pile est une *pile d'expressions*.

La fonction **apply** retourne un nœud approprié en fonction de l'opérateur.

```
Expression apply (char x, Expression a, Expression b=0)
{
  switch (x) {
    case '+':
      return new Addition(a,b);
    case '-':
      return new Soustraction(a,b);
    case '*':
      return new Multiplication(a,b);
    case '/':
      return new Division(a,b);
    case '!':
      return new Factorielle(a);
  }
  return NULL;
}
```

Note : D'autres problèmes se traitent de façon semblable. Il y a toute la famille de construction et d'évaluation d'arbres. Voir le design pattern "Composite"

Evaluation de polonaise postfixée, sans arbre

Variation du programme précédent: un programme qui lit une expression postfixée sur une ligne, et en affiche la valeur. Il ne construit pas l'arbre.

On définit une classe abstraite de lexèmes, ayant deux sous-classes : entiers et opérateurs, pour une meilleure lecture.

```
/////////////////////////////
//
// Polonaise, avec lecture plus sophistiquee
// sans construction d'arbre.
//
#include <ctype.h>
#include <iostream.h>
#include <iomanip.h>
/////////////////////////////
// Classes
//
class Pile;
class Lexeme;
class Entier;
class Operateur;
/////////////////////////////
// Methode globale
//
Lexeme* makeLexeme();  //lecture
/////////////////////////////
// Classe Pile d'entiers
//
class Pile {
 public :
  Pile(int n = 10);
  void push(int x);
  int isEmpty();
  int top();
  int pop();
  void imprimer();
 private :
  int sp;
  int * p;
};
```

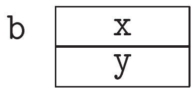
```
////////////////////////////
// Classe Lexeme
//
class Lexeme {
 public:
  virtual void eat(Pile&) =0;
};
/////////////////////////////
// Classe Entier : public Lexeme
//
class Entier : public Lexeme {
  int val;
 public :
  Entier(int i) { val = i;}
  void eat(Pile& p) {
    p.push(val);
    p.imprimer(); cout << endl;
  }
};
/////////////////////////////
// Classe Operateur : public Lexeme
//
class Operateur : public Lexeme {
  char op;
  static int apply (char x, int a, int b) {
    switch (x) {
    case '+': return a + b;
    case '-': return a - b;
    case '*': return a * b;
    case '/': return a / b;
    }
  return 0;
  }
 public:
  Operateur(char op) { this -> op = op;}
  void eat(Pile& p) {
    int a = p.pop();
    int b = p.pop();
    int c = apply(op, b, a);
    p.push(c);
    p.imprimer(); cout << endl;
  }
};
/////////////////////////////
Lexeme* makeLexeme() {
  char x;
  int i;
  while (cin.get(x)) {
    if (x == '\n')
```

```
      return 0;
    if (x != ' ')
      break;
  }
  if (('0' <= x) && (x <= '9')) {
    cin.putback(x);
    cin >> i;
    cout << "Je lis un entier " << i << endl;
    return new Entier(i);
  }
  cout << "Je lis un char " << x << endl;
  return new Operateur(x);
}

/////////////////////////////
Pile::Pile(int n) {
  sp = 0; p = new int[n];
}

void Pile::push(int x) { p[++sp] = x; }

int Pile::isEmpty() { return (sp == 0); }

int Pile::top() {
  if (sp == 0)
    throw "Pile Vide";
  return p[sp];
}

int Pile::pop() {
  if (sp == 0)
    throw "Pile Vide";
  return p[sp--];
}

void Pile::imprimer() {
  cout << "\t";
  for (int i= sp; i>0; i--)
    cout << p[i]<< " ";
  cout << "@";
}
/////////////////////////////

void main() {
  Pile p; int valeur;
  Lexeme* lex = makeLexeme();
  try {
    for (; lex != 0; lex = makeLexeme())
      lex -> eat(p);
```

```
    valeur = p.pop();
    if (!p.isEmpty())
      throw "Expression surdefinie";
  }
  catch(char * e) {
    cout << e << endl; exit(1);
  }
  cout << "Valeur = " << valeur << endl;
  exit(0);
}
```

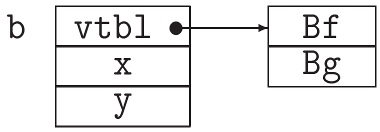
Table de fonctions virtuelles

```
class Base {
  int x, y;
  int H(void);
};

Base b;
```



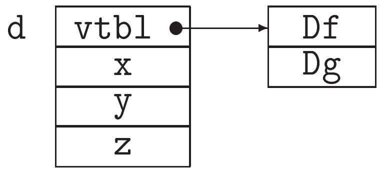
L'appel `b.H()` est remplacé par l'adresse de `H` à la compilation.

```
class Base {
  int x, y;
  int H(void);
  virtual int f(void);
  virtual int g(void);
};
```



L'appel `b.g()` est remplacé par `b.vtbl[1]` à la compilation.

```
class Der : Base {
  int x, y;
  int f(void);
  int g(void);
  int z;
};
```



L'appel `d.g()` est remplacé par `d.vtbl[1]` à la compilation.

Soit maintenant

```
Base * p = new Der;
```

L'affectation au pointeur `p`

- restreint l'accès aux champs de `Base`, i.e. `p -> z` est une erreur;
- ne modifie pas le contenu des champs de l'objet pointé: `p -> g()` vaut `p -> vtbl[1] = Dg()`.

Une conversion `b = d` remplace la table de `Der` par la table de `Base`.

On appelle *ligature dynamique* ou *résolution* ou *typage dynamique* le calcul de l'adresse de la fonction appelée en fonction du type de l'objet appelant, calcul fait à l'exécution.

6

Structures de données

1. Piles
2. Exceptions
3. Listes comme objets
4. Arbres (ou composites)
5. Visiteurs

Piles

Les opérations d'une pile sont

- `void push(element)` pour empiler;
- `element top()` pour récupérer l'élément en haut de pile;
- `element pop()` supprime et retourne l'élément en haut de pile;

Dans une structure *exogène*, seul un pointeur vers l'élément est stocké, dans une structure *endogène*, il y a copie.

La version la plus simple:

```
class Pile {
  int sp;
  char* K[10];
public :
  Pile() : sp(0) {}
  void push(char* x) { K[sp++] = x; }
  char* top() { return  K[sp - 1]; }
  char* pop() { return  K[--sp]; }
};
```

La valeur de `sp` est l'indice de la première place disponible dans `K`.

Voici une petite fonction utilisant une pile.

```
void main() {
  Pile P;
  char* x; char* y;

  P.push("Hello world !");
  P.push("Bonjour, monde !");
  x = P.top();
  cout << x << endl;
  y = P.pop();
  x = P.pop();
  cout << x << ' ' << y << endl;
}
```

Avec le résultat

```
Bonjour, monde !
Hello world ! Bonjour, monde !
```

L'ajout et le retrait d'un élément s'écrivent parfois en surchargeant les opérateurs de flots `<<` et `>>`. On les surcharge en *méthodes*:

```
Pile& Pile::operator<<(char* x) {
  push(x);
  return *this;
}
Pile& Pile::operator>>(char*& x) {
  x = pop();
  return *this;
}
```

On écrit alors:

```
void main() {
  Pile P;
  char* x; char* y;

  P << "Hello world !" << "Bonjour, monde !";
  P >> x >> y;
  cout << x << ' ' << y << endl;
  //Affiche: Bonjour, monde ! Hello world !
}
```

Pile par liste de cellules

On réalise une pile par une liste chaînée de cellules

```
class Pile {
 public :
  Pile(): sp(NULL) {};
  void push(string x) { sp = new Maillon(x, sp); }
  bool isEmpty() { return (!sp); }
  string top() { return sp -> cont; }
  string pop() {
    string x = sp -> cont; Maillon* t = sp;
    sp = sp -> next;
    delete t; return x;
  }
  void imprimer() {
    for (Maillon* t = sp; t; t = t -> next)
      cout << t -> cont <<" ";
    cout << endl;
  }
 private :
  class Maillon {
  public:
    Maillon* next;
    string cont;
    Maillon(string c, Maillon* n = NULL) : cont(c), next(n) {}
  };
  Maillon* sp;
};

void main() {
  Pile p;
  p.push("Bonjour, monde");
  p.imprimer();
  p.push("Hello, world");
  p.imprimer(); //etc
}
```

Question : Quelle implantation choisir ?

Réponse : aucune des deux, mais une classe abstraite, représentant les services que rend une pile:

```
class PileAbstraite {
 public :
  virtual void push(string x) =0;
  virtual bool isEmpty() =0;
  virtual string top() =0;
  virtual string pop() =0;
  virtual void imprimer() =0;
};
```

utilisée soit avec

```
void main() {
  PileAbstraite* p = new Pile();
  // On choisit l'implementation concrete
  p -> push("Bonjour, monde");
  p -> imprimer();
  p -> push("Hello, world");
  p -> imprimer();
  cout << p -> top();
  string x = p -> pop();
  p -> imprimer();
}
```

soit avec

```
void main() {
  PileAbstraite& p = Pile();
  // On choisit l'implementation concrete
  p.push("Bonjour, monde");
  p.imprimer();
  p.push("Hello, world");
  p.imprimer();
  cout << p.top();
  string x = p.pop();
  p.imprimer();}
```

Exceptions

- Quand la pile est vide, on ne peut faire ni `pop()` ni `top()`
- Quand la pile est pleine, on ne peut pas faire de `push()`

Dans ces cas:

- Ce sont les méthodes de la classe qui doivent *détecter* ces situations;
- mais c'est le programme *appelant* qui dont en tirer les conséquences.

Les méthodes doivent *signaler* les débordements.

Les *exceptions* constituent le moyen privilégié de gestion d'erreurs, mais servent aussi comme une structure de contrôle générale.

Les *exceptions* sont un mécanisme de *déroutement conditionnel*.

- Une instruction `throw` lève une exception;
- Un bloc `try ... catch` capte l'exception.

```
void f(int i) {
  if (i)
      throw "Help!";
  cout <<"Ok!\n";
}

void main() {
  try {
    f(0);
    f(1);
    f(0);
  }
  catch(char *) {
    cout << "A l'aide !\n";
  }
}
```

Compilé (parfois avec l'option **`-fhandle-exceptions`**), le résultat est

```
Ok!
A l'aide !
```

Trois acteurs interviennent dans le mécanisme des exceptions:

- on *lance* ou *lève* une exception par une instruction formée de `throw` suivi d'une expression;
- on effectue une exécution conditionnée d'une suite d'instructions dans un bloc commençant par `try`;
- on *traite* ou *gère* une exception par une clause `catch` de capture. Les clauses de capture suivent physiquement le groupe `try`.

Les exceptions sont levées, directement ou indirectement, dans le bloc `try`.

Le bloc `try` est suivi d'une ou plusieurs clauses `catch`. Chacune est un *gestionnaire d'exception*. Un gestionnaire est paramétré par un type (suivi éventuellement d'un argument).

La séquence d'exécution est la suivante:

- si aucune exception n'est levée dans le bloc `try`, le contrôle passe à l'instruction suivant les gestionnaires d'exception;
- si une exception est levée, le bloc `try` est quitté, on n'y revient pas, et le contrôle passe au premier gestionnaire rencontré et dont le type s'"accorde" à l'argument du `throw`.
- après exécution du gestionnaire, le contrôle passe à l'instruction suivant le dernier gestionnaire.
- si aucun gestionnaire ne convient, une fonction prédéfinie `terminate()` est appelée.

Une exception est interceptée en filtrant un *type*.

A la levée, c'est un *objet* qui est passé, et non un type.

Une fonction *peut* (en Java *doit*) spécifier les exceptions qu'elle est susceptible de lever et qu'elle ne traite pas. Par exemple

```
void push(char *) throw (Overflow);
```

Deux types d'exceptions sont *définies*, comme classes:

```
class Underflow{};
class Overflow{};
```

La classe pile est redéclarée:

```
class Pile {
private:
  int sp;
  char* K[4];
public:
  Pile() : sp(0) {}
  char* top() throw (Underflow);
  char* pop() throw (Underflow);
  void push(char* x) throw (Overflow);
};
```

Les méthodes sont redéfinies:

```
char* Pile::top() throw (Underflow) {
  if (sp == 0) throw Underflow(); // objet de la classe
  return K[sp-1];
}
char* Pile::pop() throw (Underflow){
  if (sp == 0) throw Underflow();
  return K[--sp];
}
void Pile::push(char* x) throw (Overflow) {
  if (sp > 10) throw Overflow();
  K[sp++] = x;
}
```

C'est l'utilisateur de la classe qui capte les messages:

```
void main()
{
  Pile P;
  char c;
  for (;;) {
    cin >> c;
    try {
      switch(c){
      case '+':
        P.push("Hello world!"); // peut lever exception
        break;
      case '-':
        cout << P.pop() << endl;// peut lever exception
      }
    }
    catch (Overflow o)  { cout << "Pile deborde!\n";}
    catch (Underflow u) { cout << "Pile vide!\n";}
  }
}
```

Listes comme objets

On peut voir les listes

- comme mécanisme de chaînage d'objets;
- comme une réalisation des suites finies d'objets.

Dans le premier cas, une liste est un pointeur vers son premier élément, et le chaînage est réalisé par une classe de "cellules".

Dans le deuxième cas, on a "réifié" la liste en objet. Le mécanisme de chaînage est secondaire (ce pourrait être un tableau). On s'intéresse ici à

- l'itération;
- le transfert des opérations des éléments sur les listes (le fameux `mapcar`);
- et on verra comment on obtient gratis des listes de listes (un avant-goût des "composites").

Listes (exogènes) de pointeurs vers des “objets” (ici une classe d’entiers).

```
class Objet {
protected:
  int t;
public:
  Objet(int t = 0) : t(t) {}
  virtual void show() { cout <<' '<< t ;}
};
```

Une *liste* contient un pointeur vers une première cellule, un pointeur vers la dernière, et un entier contenant la taille.

La classe des *maillons* est locale à la classe des listes.

```
class Liste {
protected:
  class Maillon {
  public:
    Maillon* suiv;
    Maillon* prec;
    Objet*   d;
    Maillon(Maillon* p, Maillon* s, Objet* d)
      : suiv(s), prec(), d(d) {}
   };
  Maillon* prem;
  Maillon* dern;
  int taille;
...
};
```

Constructeur par défaut:

```
Liste::Liste() : prem(NULL), dern(NULL), taille(0) {}
```

Deux méthodes d'interrogation naturelles:

```
int getTaille() const { return taille;}
int isEmpty() const { return taille==0;}
```

On manipule par des méthodes usuelles:

```
void addHead(Objet* n);
void addTail(Objet* n);
Objet* getHead();
Objet* removeHead();
```

Par exemple:

```
void Liste::addHead(Objet* n) {
  Maillon* nouv = new Maillon (NULL, prem, n);
  if (prem != NULL)
    prem -> prec = nouv;
  else // liste vide
    dern = nouv;
  prem = nouv;
  taille++;
}
```

Noter que `removeHead()` supprime le maillon et retourne l'adresse de l'objet.

L'impression se fait en appelant la méthode adéquate pour chaque objet de la liste:

```
void Liste::show() {
  cout <<"(";
  for (Maillon* p = prem; p; p = p -> suiv)
    p -> d -> show();
  cout <<")";
}
```

Exemple:

```
Liste a;

a.addHead(new Objet(2));
a.addTail(new Objet(5));
a.addTail(new Objet(6));
cout << "Taille : " << a.getTaille() << endl; // 3
a.show();cout << endl; // ( 2 5 6)
```

Listes récursives

Une liste récursive est une liste dont les éléments peuvent être des listes, comme

```
( 4( 1 3) 2 5 6)
```

Ceci est réalisé en déclarant qu'une liste *est un cas particulier* d'un objet.

```
class Liste : public Objet {
...
};
```

Rien ne doit être modifié dans les fonctions précédentes.

En fait, le constructeur de la classe `Liste` fait appel au constructeur par défaut de la classe `Objet` (qui doit être défini). Ainsi

```
Liste a, b, c;
a.addHead(new Objet(2)); // (2)
a.addTail(new Objet(5)); // (2 5)
b.addHead(new Objet(7));
b.addHead(new Objet(8)); // (8 7)
c.addHead(new Objet(3));
c.addHead(&b);           // ((8 7) 3)
c.addHead(new Objet(1)); // (1 (8 7) 3)
a.addHead(&c);
a.addHead(new Objet(4));
a.show(); cout << endl;
```

donne

```
( 4( 1( 8 7) 3) 2 5)
```

Arbres (ou composites)

- Les listes récursives précédentes sont un cas particulier du schéma conceptuel *Composite*.
- Ce schéma conceptuel décrit une représentation d'informations hiérarchisées.

Exemples d'application:

- Un *élément graphique* peut être
  - une ligne
  - un rectangle
  - un texte
  - ou une figure qui groupe des éléments graphiques
- Un arbre est
  - une feuille
  - un groupe d'arbres

L'objectif de *Composite* est de composer des objets en une structure arborescente, en vue de pouvoir traiter de manière uniforme un objet et un groupe d'objet.

Structuration: La hiérarchie des classes comporte

- une classe (abstraite) `Composant` (élément graphique, sommet)
- des classes dérivées `Feuilles` (ligne, rectangle, texte, ou feuille d'un arbre)
- une classe dérivée `Composite` (figure, ou nœud)

Avantage : un client manipule les objets en tant que composants.

Une arborescence de cercles et de carrés

Description

- Les sommets forment une `class Forme`.
- Les feuilles sont soit de la `class Carre`, ou de la `class Cercle`.
- Les nœuds sont de la `class Groupe`.

La hiérarchie est formée de

```
class Forme {...}
class Carre : public Forme {...}
class Cercle : public Forme {...}
class Groupe : public Forme {...}
```

Les opérations sont en général *virtuelles* sur la classe `Forme`. Ici l'affichage et le calcul de l'aire:

```
class Forme {
  public:
    virtual void show() = 0;
    virtual float getAire() = 0;
};
```

La classe `Groupe` a un mécanisme de parcours. Ici un simple tableau de taille arbitrairement fixée à `12`.

```
class Groupe : public Forme {
    Forme* element[12];
    int nb;
  public :
    Groupe() : nb(0) {}
    void add(Forme* f) { element[nb++]= f; }
    float getAire();
    void show();

};
```

Il est plus approprié d'utiliser un conteneur de la STL:

```
class Groupe : public Forme {
    Vector<Forme*> element;
  public :
    Groupe() {}
    void add(Forme* f) { element.push_back(f); }
    float getAire();
    void show();

};
```

Les opérations sur un groupe font appel aux opérations sur les éléments:

```
float Groupe::getAire() {
  float s = 0;
  for (int i = 0; i < nb ; i++)
    s += element[i] -> getAire();
  return s;
}
void Groupe::show() {
  cout << "Forme (\n";
  for (int i = 0; i < nb ; i++)
   element[i] -> show();
  cout << ")\n";
}
```

Avec un vecteur, on utilise la méthode **`element.size()`** à la place de l'entier **`nb`**.

Quelques exemples d'appels

```
main() {
  Cercle * c; Carre * r; Groupe g;

  c = new Cercle(5); r = new Carre(3);
  c -> show(); r -> show();
  g.add(c); g.add(r); g.show();
  Groupe * h = new Groupe;
  h -> add(new Carre(6));
  g.add(h); g.show();
  cout << "Aire totale = " << g.getAire() << endl;
}
```

Résultat:

```
Cercle 5 78.5397
Carre 3 9
Forme (
   Cercle 5 78.5397
   Carre 3 9
)
Forme (
   Cercle 5 78.5397
   Carre 3 9
   Forme (
      Carre 6 36
   )
)
Aire totale = 123.54
```

Un annuaire

L'annuaire associe des noms et des numéros de téléphone. Ces couples sont associés aux *feuilles* d'un arbre. Les nœuds contiennent juste le nom de la plus grande feuille du sous-arbre gauche.

Il y a donc une classe **Sommet** dont dérivent une classe **Noeud** et une classe **Feuille**. L'arbre est représenté par la classe **Arbre**. Voici les définitions

```
class Sommet {
 protected:
  string nom;
 public:
  Sommet(string nom);
  virtual void show(int d = 0) =0;
  virtual long findTel(string nom) =0;
  virtual Sommet* add(string nom, long tel) =0;
};

class Noeud : public Sommet {
  Sommet *gauche, *droit;
 public:
  Noeud(string nom, Sommet *gauche, Sommet*droit);
  Sommet* add(string n, long tel);
  void show(int d = 0);
  long findTel(string n);
};

class Feuille : public Sommet {
  long tel;
 public :
  Feuille(string nom, long tel);
  void show(int d = 0);
  Sommet* add(string n, long tel);
  long findTel(string n);
};
```

```
class Arbre {
  Sommet* racine;
 public:
  Arbre();
  void show();
  void add(string nom, long tel);
  long findTel(string nom);
};
```

On s'en sert par exemple comme suit

```
main() {
  Arbre a;
  a.add("Pierre", 1234);
  a.add("Paul", 1301);
  a.add("Alex", 1111);
  a.add("Vero", 1700);  a.show();
  cout << a.findTel("Pierre") << endl;
  cout << a.findTel("Marie") << endl;
}
```

Résultat:

```
Paul
   Alex
      Alex = 1111
      Paul = 1301
   Pierre
      Pierre = 1234
      Vero = 1700
1234
-1
```

D'autres modèles d'arbre existent.

Visiteurs

Le polymorphisme en C++ (et en Java) est limité: il ne s'applique qu'à l'objet appelant (le 0-ième argument). Sur les autres arguments s'applique la surcharge. En Smalltalk, le polymorphisme est total.

Reprenons l'évaluation des expressions polonaises postfixées.

- La classe `Lexeme` a les deux classes dérivées `Entier` et `Operateur`.
- L'évaluation se fait dans une boucle :

  ```
  void main() {
    Pile p; int valeur;
    Lexeme* lex = makeLexeme();
    for (; lex != 0; lex = makeLexeme())
      lex -> eat(p);
    ...
  }
  ```

- Logiquement, c'est la pile qui mange le lexème. On aurait dû écrire `p.eat(lex)`, avec deux méthodes "dérivées"

  ```
  eat(Operateur*)
  eat(Entier*)
  ```

  Or, le compilateur demande une méthode de signature

  ```
  eat(Lex*)
  ```

  que l'on est bien incapable de fournir. Tout ceci parce que le polymorphisme ne s'applique pas à l'*argument* de `eat()`.

Le schéma conceptuel des *Visiteurs* sert à rémédier à cela par l'inversion des arguments.

Dans ce schéma

- la classe (abstraite) `Lexeme` a une méthode (virtuelle) traditionnellement appelée `accept(Pile)`.
- Chaque classe dérivée (concrète) implémente cette méthode, par inversion des arguments:

```
void Operateur::accept(Pile& p) {
  p.eat(this);
}
void Entier::accept(Pile& p) {
  p.eat(this);
}
```

- La classe `Pile` implémente deux méthodes

```
void Pile::eat(Entier* e) {
  push(e->val);
}
void Pile::eat(Operateur* o) {
  int a = pop();
  int b = pop();
  int c = o->apply(b, a);
  push(c);
```

L'évaluateur envoie le "message"

```
lex -> accept(p);
```

Si `lex` pointe sur un Operateur `o`, ceci devient

```
o -> accept(p);
```

et donc

```
p.eat(o);
```

et la méthode `Pile::eat(Operateur*)` s'applique.

Le tour est joué !

Annexe 1 : Liste du programme des formes

```
////////////////////////
//
// Composite de formes
//
#include <iostream.h>
#include <iomanip.h>
#define decal(d) cout << setw(d) <<""
////////////////////////
//
// Forme --- Cercle
//       |-- Carre
//       |-- Groupe
//
////////////////////////
// class Forme
//
class Forme {
  public:
    virtual void show(int = 0) = 0;
    virtual float getAire() = 0;
};
////////////////////////
// class Carre
//
class Carre : public Forme {
    int cote;
    float aire;
  public:
    Carre(int cote) {
      this -> cote = cote;
      this -> aire = cote*cote;
    }
    float getAire() { return aire; }
    void show(int d) {
      decal(d);
      cout << "Carre " << cote <<" " << aire << endl;
    }
};
////////////////////////
// class Cercle
//
class Cercle : public Forme {
    int rayon;
    float aire;
  public:
    Cercle(int rayon) {
      this -> rayon = rayon;
```

```
      this -> aire = 3.14159* rayon*rayon;
    }
    float getAire() { return aire; }
    void show(int d) {
      decal(d);
      cout << "Cercle " << rayon <<" " << aire << endl;
    }
};
////////////////////////
// class Groupe
//
class Groupe : public Forme {
    Forme * element[12];
    int nb;
  public :
    Groupe() : nb(0) {}
    void add(Forme* f) { element[nb++]= f; }
    float getAire() {
      float s = 0;
      for (int i = 0; i < nb ; i++)
s += element[i] -> getAire();
      return s;
    }
    void show(int d) {
      decal(d); cout << "Forme (\n";
      for (int i = 0; i < nb ; i++)
element[i] -> show(d + 3);
      decal(d); cout << ")\n";
    }
};
////////////////////////
// main()
//
main() {
  Forme* c, *r;
  Groupe* g;

  c = new Cercle(5);
  r = new Carre(3);
  g = new Groupe();
  c -> show(); r -> show();
  g -> add(c); g -> add(r); g -> show();
  Groupe* h = new Groupe;
  h -> add(new Carre(6));
  g -> add(h); g -> show();
  cout << "Aire totale = " << g -> getAire() << endl;
}
```

Annexe 2 : Liste du programme annuaire

```
////////////////////////
//
// Arbre, informations aux feuilles
//
#include <iostream.h>
#include <iomanip.h>
#include <string>
#define decal(d) cout << setw(d) <<""
////////////////////////
//
// Arbre (racine)
// Sommet
//   Noeud
//   Feuille
//
////////////////////////
// class Arbre
//
class Sommet;
class Arbre {
  Sommet* racine;
 public:
  Arbre();
  void show();
  void add(string nom, long tel);
  long findTel(string nom);
};
////////////////////////
// class Sommet
//
class Sommet {
 protected:
  string nom;
 public:
  Sommet(string nom);
  virtual void show(int d = 0) =0;
  virtual long findTel(string nom) =0;
  virtual Sommet* add(string nom, long tel) =0;
};
////////////////////////
// class Noeud
//
class Noeud : public Sommet {
  Sommet *gauche, *droit;
 public:
  Noeud(string nom, Sommet *gauche, Sommet*droit);
  Sommet* add(string n, long tel);
```

```
  void show(int d = 0);
  long findTel(string n);
};
////////////////////////
// class Feuille
//
class Feuille : public Sommet {
  long tel;
 public :
  Feuille(string nom, long tel);
  void show(int d = 0);
  Sommet* add(string n, long tel);
  long findTel(string n);
};
////////////////////////
// class Arbre: implantation
//
Arbre::Arbre(): racine(NULL){}

void Arbre::show() { if (racine) racine -> show(); }

void Arbre::add(string nom, long tel) {
  if (racine)
    racine = racine -> add(nom, tel);
  else
    racine = new Feuille(nom, tel);
}

long Arbre::findTel(string nom) {
  return (racine) ? racine -> findTel(nom) : -1;
}
////////////////////////
// class Sommet: implantation
//
Sommet::Sommet(string nom) : nom(nom) {}
////////////////////////
// class Noeud: implantation
//
Noeud::Noeud(string nom, Sommet *gauche, Sommet*droit)
  : Sommet(nom), gauche(gauche), droit(droit) {}
Sommet* Noeud::add(string n, long tel) {
  if (nom > n) gauche = gauche -> add (n, tel);
  if (nom < n) droit = droit -> add (n, tel);
  return this;
}
void Noeud::show(int d = 0) {
  decal(d); cout << nom << endl;
  gauche -> show(d+3);
  droit -> show(d+3);
```

```
}
long Noeud::findTel(string n) {
  if (n <= nom)
    return gauche -> findTel(n);
    return droit -> findTel(n);
}
////////////////////////
// class Feuille: implantation
//
Feuille::Feuille(string nom, long tel) : Sommet(nom), tel(tel) {}
Sommet* Feuille::add(string n, long tel) {
  if (n > nom)
    return new Noeud(nom, this, new Feuille(n, tel));
  if (n < nom )
    return new Noeud( n, new Feuille(n, tel), this);
  return this;
}
void Feuille::show(int d = 0) {
  decal(d); cout << nom <<" = " << tel << endl;
  }
long Feuille::findTel(string n) { return (n == nom) ? tel : -1; }


////////////////////////
// main()
//
main() {
  Arbre a;
  a.add("Pierre", 1234);  a.show();
  a.add("Paul", 1301);  a.show();
  a.add("Alex", 1111);  a.show();
  a.add("Vero", 1700);  a.show();
  cout << a.findTel("Pierre") << endl;
  cout << a.findTel("Marie") << endl;

}
```

7

Copie et affectation

1. Constructeurs
2. Constructeur de copie
3. Affectation
4. Conversions

Constructeurs

Les constructeurs se classent syntaxiquement en 4 catégories

1. le constructeur *par défaut*, sans argument,
2. les constructeurs *de conversion*, à un argument
3. le constructeur *de copie*, à un argument
4. les autres constructeurs.

Un constructeur de conversion sert souvent à la *promotion*.
Exemple:

```
class Rat {
  int num, den;
public:
  int num, den;
  ...
  Rat(int n)  { num = n; den = 1;}
  ...
};
```

On s'en sert pour la conversion

```
Rat r, s;
...
r = s + 7;
```

L'entier `7` est promu en `Rat(7)`.

Constructeur de copie

Chaque classe possède un constructeur de copie (ne pas confondre avec l'opérateur d'affectation).

Le constructeur de copie par défaut copie membre à membre en utilisant le constructeur de copie de chaque membre, et une copie bit à bit pour les types de base (copie *superficielle*). Ceci est insuffisant en présence d'adresses. Il faut alors allouer de nouvelles zones mémoire (copie *profonde*).

Le constructeur de copie est utilisé

- lors d'une initialisation:

  ```
  X  x = y;
  X  x(y);
  X* x = new X(y);
  ```

  où `y` est un objet *déja existant*.
- lors du retour d'une fonction retournant un objet par valeur;
- lors du passage d'un objet par valeur à une fonction.

Le constructeur de copie a deux prototypes possibles :

```
X (X&);
X (const X&);
```

L'argument est donc un objet de la classe X passé par référence (évidemment !) et éventuellement spécifié comme non modifiable dans le constructeur.

Exemples:

```
X ident;
X ibis = ident;
X ibis(ident);
X* pid = new X(ident);
```

Voici une classe `Pt` possédant:

- Un constructeur de copie `Pt(const Pt& p)`
- Un opérateur d'affectation `Pt& operator=(const Pt& p)`

```
#include <iostream.h>

class Pt {
  int x, y;
public:
  Pt(int abs = 1, int ord = 0) x(abs), y(ord) {}

  //implantation par défaut:
  Pt(const Pt& p) : x(p.x), y(p.y){}

  //implantation par défaut:
  Pt& operator=(const Pt& p) {
    x = p.x; y = p.y;
    return *this;
  }
};
```

Voici un programme utilisant cette classe, avec des fonction auxiliaires

```
void Fn(Pt p, Pt& q, Pt* r) {
  delete r;
}

Pt vv(Pt x) { return x; }
Pt vr(Pt& x) { return x; }
Pt& rv(Pt x) { return x; }
Pt& rr(Pt& x) { return x; }
```

```
void main()
{
  Pt a;               //Construction de a : 1 0
  Pt b = 3;           //Construction de b : 3 0
  Pt c = Pt(0,2);     //Construction de c : 0 2
  {
    Pt d = Pt(a);     //Recopie      de a en d
    Pt e = a;         //Recopie      de a en e
                      //Destruction  de e
  }                   //Destruction  de d
  Pt f;               //Construction de f : 1 0
  Pt* p = new Pt(6,7);//Construction de *p : 6,7

  c = f;              //Affectation  de f a c
  c = Pt(5,5);        //Construction de temp : 5 5
                      //Affectation  de temp a c
                      //Destruction  de temp

  Fn(a, b, p);        //Recopie      de a en abis
                      //Destruction  de *p
                      //Destruction  de abis

  f = vv(b);          //Recopie      de b en bis
                      //Recopie      de bis en bpile
                      //Destruction  de bis
                      //Affectation  de bpile a f
                      //Destruction  de bpile
  f = vr(b);          //Recopie      de b en bpile
                      //Affectation  de bpile a f
                      //Destruction  de bpile
  f = rv(b);          //Recopie      de b en bis
                      //Destruction  de bis
                      //Affectation  de bis a f // danger !
  f = rr(b);          //Affectation  de b a f(sans copie)
}
```

Affectation

Ne pas confondre copie et affectation!

- le constructeur de copie sert à l'initialisation et pour le passage de paramètres. Il construit l'objet.
- l'opérateur d'affectation `=` pour l'affectation dans une expression. Il retourne un objet ou une référence.

L'opérateur d'affectation est un opérateur, et peut donc être redéfini explicitement par une fonction `operator=()`.

Chaque classe possède un opérateur d'affectation par défaut.

L'affectation est superficielle comme la copie, et consiste en une affectation membre à membre.

La signature de l'opérateur est

```
X& operator=(X&)
X& operator=(const X&)
X operator=(X&)
X operator=(const X&)
```

et l'expression `y = x` équivaut à

```
y.operator=(x)
```

Le résultat de l'affectation est donc dans l'objet appelant. A quoi équivaut `x = y = z = t` ?

Voici une petite classe **Id**:

```
class Id {
  char* nom;
public:
  Id(const Id&);
  Id& operator=(const Id&);
};
```

Le constructeur de copie est:

```
Id::Id(const Id& x) {
  nom = new char[1+strlen(x.nom)];
  strcpy(nom,x.nom);
}
```

L'opérateur d'affectation est:

```
Id& Id::operator=(Id& x){
  if (this == &x) // meme objet
    return *this;
  if (x.nom == 0)
    nom = 0;
  else {
    delete nom;
    nom = new char[1+strlen(x.nom)];
    strcpy(nom,x.nom);
  }
  return *this;
}
```

La *référence* retournée est celle de l'objet appelant.

Chaque méthode contient un pointeur sur l'objet appelant la méthode; ce pointeur a pour nom **this**. L'objet lui-même est donc ***this**.

Conversions

Les conversions définies par le programmeur sont

- les constructeurs de conversion;
- les opérateurs de conversion.

Un *opérateur de conversion* d'une classe `C` *en un classe ou un type* `T`

- est une fonction membre de `C`;
- qui a le prototype `operator T()`. Le type de retour est donc `T`
- est appelé implicitement pas l'objet,
- n'est utilisé que si nécessaire.

Exemple. On veut pouvoir écrire

```
Rat r;
...
if (r) ...
```

C'est possible en convertissant `r` en boolean qui est vrai si le numérateur de `r` est non nul.

```
class Rat{
...
  operator bool() { return (num) ? 1 : 0;}
}
```

8

Opérateurs

1. Principe
2. Opérateurs amis
3. Syntaxe
4. Opérateur d'indexation
5. Opérateurs logiques
6. Opérateur d'insertion et d'extraction
7. Opérateurs `++` et `--`

Principe

Les *opérateurs* forment une famille particulière de fonctions, spécifique à C++.

L'utilisation des opérateurs par *surcharge* illustre un aspect intéressant de la programmation évoluée, indépendant de la programmation objet.

Rappel de quelques principes:

- Chaque fonction a une signature, qui est la liste des types des paramètres formels (et le type du résultat);
- Un même nom de fonction peut désigner plusieurs fonctions de signatures différentes, *sans compter le type du résultat*;
- Il y a *surcharge* (ou polymorphisme dynamique) si le choix de la fonction est déterminé par le *type* des paramètres formels;
- Il y a *polymorphisme* (dynamique) si le choix de la fonction est déterminé par le *type* de l'objet appelant.

Les *opérateurs* sont au nombre de 40:

```
+    -    *    /    %    ^    &      |    ~    !
+=   -=   *=   /=   %=   ^=   &=     |=
++   --   =    ==   !=   <    >      <=   >=
<<   >>   <<=  >>=  &&   ||
->*  ,    ->   []   ()   new  delete
```

Les cinq derniers sont: l'indirection, l'indexation, l'appel de fonction, l'allocation et la désallocation. Il n'est pas possible de changer la précédence des opérateurs, ni la syntaxe des expressions, ni de définir de nouveaux opérateurs. D'autres opérateurs existent:

```
.    ::    ?:    sizeof    .*    &
```

qui ne peuvent être surchargés. Le `&` de prise de référence n'est pas surchargeable, le "et" bit à bit l'est.

Opérateurs amis

Une fonction *amie* d'une classe a les mêmes droits qu'une fonction membre, et en particulier peut accéder aux membres privés de la classe.

```
class Rat {
 int num, den; // donc prives
public:
 int getNum() {return num;}
 int getDen() {return den;}
 Rat(int n = 0, int d = 1) : num(n),  den(d) {}
};
```

L'addition

```
Rat operator+(const Rat& a, const Rat& b) {
  int n = a.getNum()*b.getDen() + a.getDen()*b.getNum();
  return Rat(n, a.getDen()*b.getDen());
}
```

s'écrit plus efficacement et plus lisiblement:

```
Rat operator+(const Rat& a, const Rat& b) {
  return Rat(a.num*b.den + a.den* b.num, a.den*b.den);
}
```

Or **`num`** et **`den`** sont privés. Pour les rendre accessibles, l'opérateur est déclaré *ami*, dans la déclaration de la classe **`Rat`**, par

```
friend Rat operator+(Rat, Rat);
```

Une fonction amie a les mêmes droits qu'une fonction membre. Bien noter la différence entre fonction membre et fonction amie.

Syntaxe

Un opérateur est unaire, binaire.
Un opérateur peut être défini

- soit comme fonction globale à un ou deux arguments
- soit comme fonction membre avec un argument de moins.

La syntaxe est

- opérateur binaire au niveau global:

  type `operator`*op*`(`*type*`,`*type*`);`

  ```
  Rat operator+(Rat,Rat);
  ```

  L'expression `u+v` équivaut à `operator+(u,v)`.

- opérateur binaire membre de classe:

  type `operator`*op*`(`*type*`);`

  ```
  Rat& operator+=(Rat);
  ```

  L'expression `u += v` équivaut à `u.operator+=(v)`.

- opérateur unaire au niveau global:

  type `operator`*op*`(`*type*`);`

  ```
  Rat operator-(Rat);
  ```

  L'expression `-u` équivaut à `operator-(u)`.

- opérateur unaire membre de classe:

  type `operator`*op*`();`

  ```
  Rat operator-();
  ```

  L'expression `-u` équivaut à `u.operator-()`.

Conseil

- Choisir un opérateur global et ami lorsque l'opération est symétrique: `+ - * / ==`
- Choisir un opérateur membre lorsque l'opération est asymétrique: `+=`

```
class Rat {
  int num, den;
public:
  ...
  Rat operator-(Rat);               //membre : mauvais choix
  Rat& operator+=(const Rat&);      //membre :bon choix
  friend Rat operator/(Rat , Rat ); //amie : bon choix
};
```

Avec

```
Rat Rat::operator-(Rat b) {
  return Rat(num*b.den - den* b.num, den*b.den);
}
Rat& Rat::operator+=(const Rat& d) {
  num = num*d.den + d.num*den;
  den = den*d.den;
  return *this;
}
Rat operator/(Rat a, Rat b) {
  return Rat(a.num*b.den, a.den*b.num);
}
```

Exemple

```
cout << a << b << a-b << a.operator-(b)
     << a-1 << a+=2 << a;
```

Résultat

```
1/2  2/3  -1/6  -1/6  -1/2 3/2 3/2
```

Attention

```
cout << 1-a ; // erreur syntaxique
```

L'expression

```
1.operator-(a)
```

et ceci n'est pas défini : il manque la fonction

```
friend Rat operator-(int, Rat);
```

Opérateur d'indexation

L'opérateur d'indexation `[]` est surchargé dans les vecteurs et matrices.

```
class Vecteur {
protected:
  int n;
  int *a;
public:
  int taille() const { return n;}
  Vecteur(int t = 10) : n(t), a(new int[n]) {}
  Vecteur(int t, int val) : n(t), a(new int[n]) {
    for (int i=0; i<n;i++) a[i] = val;
  }
  int& operator[](int i) { return a[i];} // surcharge
};
```

Il manque le constructeur de copie et l'opérateur d'affectation. Ainsi

```
Vecteur x;
```

crée un vecteur de taille `10`.

- *Sans surcharge*, on accède aux éléments par `x.a[i]`.
- *Avec surcharge*, l'expression `x[i]` est équivalente à `x.operator[](i)`, donc à `x.a[i]`.
- La valeur de retour est une *référence* pour pouvoir servir en "lvalue". On peut alors écrire:

```
main() {
  const int N = 4;
  Vecteur u(N, 7);
```

```
    Vecteur v(N, 5);

    for (int i = 0; i < N; i++)
      v[i] += u[i];
  }
```

On en profite pour surcharger les entrées-sorties:

```
ostream& operator<<(ostream& out , Vecteur& u) {
  out << "(";
  for (int i=0; i<u.taille();i++) out << u[i] << " ";
  return out << ")" << endl;
};
istream& operator>>(istream& in , Vecteur& u) {
  for (int i=0; i<u.taille();i++) in >> u[i];
  return in;
};
```

Remarquer qu'ici, les opérateurs d'insertion et d'extraction n'ont pas besoin d'être déclarées amis. On obtient alors les sorties

```
(7 7 7 7 )
(5 5 5 5 )
(12 12 12 12 )
```

L'opérateur d'indexation `[]` est surchargé dans les vecteurs et matrices.

Pour l'*indexation des matrices*, voici deux méthodes. Elles reposent sur le fait qu'une matrice est un vecteur de lignes.

Pour des matrices de *taille fixe*, on définit

```
class Mat {
  float a[4][4];
 public:
  ...
  float* operator[](int i) { return a[i]; }
};
```

Ainsi `x[i]` est l'adresse de la `i`-ième ligne de `a`. Le sens de `x[i][j]` résulte de l'interprétation usuelle de l'indexation.

Le résultat `x[i]` est une r-valeur, et ne peut être une référence car c'est un tableau au sens de C. En revanche, `x[i][j]` est une l-valeur comme en C.

Les matrices *dynamiques* se définissent comme suit:

```
class Mat {
  float **a;
 public:
  int nlignes, ncol;
  Mat(int l, int c) {
    nlignes = l;
    ncol = c;
    a = new float*[nlignes];
    for (int i=0; i< nlignes; i++)
      a[i] = new float[ncol];
  }
  float*& operator[](int i) { return a[i]; }
};
```

Ici aussi, `x[i]` est l'adresse de la `i`-ième ligne de `a`. Elle peut cette fois être une référence.

On peut donc écrire:

```
Mat  x(4,4);
float b[4] = {55,56,57,58};
delete x[3];
x[3] = b; // car [] retourne une reference
```

Opérateurs logiques

Tester l'égalité de deux objets par surcharge de l'opérateur `==`.

```
int operator==(const Rat& a, const Rat& b) {
  return (a.num == b.num) && (a.den == b.den) ;
}
```

Cet opérateur s'écrit bien entendu aussi pour les chaînes de caractères, vecteurs, etc.

On peut nier par

```
int operator!=(const Rat& a, const Rat& b) {
  return !(a==b) ;
}
```

Opérateurs d'insertion et d'extraction

Les opérateurs `<<` et `>>` sont souvent surchargés pour les entrées et sorties.

Une autre surcharge courante est pour l'insertion ou la suppression dans un conteneur. Mais est-ce bien raisonnable ? La surcharge doit conserver le sens original des opérateurs.

Exemple: une pile implémentée par un tableau (sans test d'erreur)

```
class Pile {
  public :
    Pile() : sp(0) {}
    void push(string x) { K[sp++] = x; }
    string top() { return  K[sp - 1]; }
    string pop() { return  K[--sp]; }
  private :
    int sp;
    string K[10];
};
```

On surcharge l'insertion et l'extraction par:

```
Pile& operator<<(Pile& p, char* x) {
  p.push(x);
  return p;
}
Pile& operator>>(Pile& p, char*& x) {
  x = p.pop();
  return p;
}
```

On peut écrire:

```
Pile p;
string x = "Hello world !", y = "Bonjour, monde !";
cout << x << ' ' << y << endl;
p << x << y;
p >> x >> y;
cout << x << ' ' << y << endl;
```

avec le résultat:

```
Hello world ! Bonjour, monde !
Bonjour, monde ! Hello world !
```

Opérateurs `++` et`--`

Il y a deux formes, préfixe et suffixe. Pour l'opérateur *préfixe*, le prototype est au niveau global:

type `operator++ (`*type*`);`

et comme membre de classe:

type `operator++ ();`

Pour l'opérateur *suffixe*, le prototype est au niveau global:

type `operator++ (`*type*`,int);`

et comme membre de classe:

type `operator++ (int);`

Il n'y a pas d'ambiguïté car `++` ne peut être redéfini sur `int`.

La valeur de l'argument passée est 0; ainsi, l'expression `x++` équivaut à `operator++(x,0)` respectivement à `x.operator(0)`.

Par exemple:

```
class Rat {
  int num, den;
public:
  ...
  Rat operator-(Rat);
  Rat operator++() { return Rat(den+num,den);}
  Rat operator++(int z) { Rat r = *this; num -= den; return r;}
};
```

permet d'écrire:

```
  cout <<"a et ++a"<< a << ++a << endl;
  cout <<"a et a++"<< a << a++ << endl;
```

9

Droits d'accès

1. Objectif
2. Les trois catégories
3. Fonctions amies
4. Accès et héritage

Objectif

Les *droits d'accès* protègent les données et les méthodes. Ils participent ainsi à l'encapsulation.

Unebonne politique consiste à déclarer que les fonctions d'accès aux données sont publiques, et que l'implantation des structures de données est cachée.

Exemple

```
class Pile {
  public :
    Pile(int n = 10);
    void push(string x); //public
    string top();        //public
    string pop();        //public
  private :
    int sp;              //prive
    string * p;          //prive
};
```

Les trois catégories

Les droits d'accès sont accordés aux fonctions membres, ou aux fonctions globales.

L'unité de protection est la classe: tous les objets d'une classe bénéficient de la même protection.

Les attributs de protection suivants sont être utilisés pour les membres d'une classe:

- un membre **`public`** est accessible à tout fonction;
- un membre **`private`** n'est accessible qu'aux fonctions membre de la classe ou aux fonctions amies;
- un membre **`protected`** n'est accessible qu'aux fonctions membre de la classe *ou des classes dérivées* ou aux fonctions amies.

Par défaut, les membres d'une classe sont privés, les membres d'une **`struct`** sont tous public.

```
class X {
    int priv; //prive par defaut
  protected:
    int prot;
  public:
    int publ;
    void m();
}
```

La fonction *membre* `X::m()` a un accès non restreint:

```
void X::m() {
  priv = 1; // ok
  prot = 1; // ok
  publ = 1; // ok
}
```

Une fonction *globale* n'accède qu'aux membres publics:

```
printf("%d", x.priv); // erreur
printf("%d", x.prot); // erreur
printf("%d", x.publ); // ok
```

Même comportement pour une fonction membre d'une classe qui n'est ni dérivée ni amie.

Fonctions amies

Une “déclaration d’amitié” incorpore, dans une classe `X`, les fonctions autorisées à accéder aux membres privés. Une fonction amie a le même statut qu’une fonction membre de `X`.

Fonction globale

```
class X { ...
  friend f(X);
}
```

Ici, `f` est autorisée à accéder aux membres privés de la classe `X`.

Fonction membre de `Y`

```
class Y { ...
  void f(X& x);
}

class X { ...
  friend Y::f(X& x);
}

void Y::f(X& x) {
  x.priv = 1; // ok
  x.prot = 1; // ok
  x.publ = 1; // ok
}
```

Classe amie

Une classe entière est amie par

```
class X { ...
  friend class Y;
}
```

Accès et héritage

Une classe dérivée est elle-même déclarée avec un mode de dérivation **`private`** (défaut), **`protected`** ou **`public`**, par

```
class Yu : public X {};
class Yo : protected X {};
class Yi : private X {};
```

La protection des membres de **`X`** se transforme pour les classes dérivées des classes dérivées.

L'accès aux membres de **`X`**, par une méthode de la classe dérivée, est la restriction commune:

- Une donnée **`private`** de **`X`** est inaccessible aux classes dérivées.
- Une donnée **`protected`** de **`X`** a la protection **`protected`** dans **`Yu`** et **`Yo`**, et **`private`** dans **`Yi`**.
- Une donnée **`public`** de **`X`** a la protection **`protected`** dans **`Yu`** et **`private`** dans **`Yi`**.

Attention : l'accessibilité concerne uniquement les membres de l'objet lui-même (**`this`**), et non pas des paramètres:

- une donnée protégée est accessible en tant que membre de l'objet;
- une donnée d'un objet de la classe de base n'est pas accessible à une fonction membre de la classe dérivée.

```
void Yu::h(X& x) {
  x.publ = 1;
  this -> prot = 2;  //ok
  x.prot = 2; // erreur: membre x.prot inaccessible
}
```

10

Flots

1. Hiérarchie de classes
2. Manipulateurs
3. Fichiers
4. Formatage des sorties
5. Entrées
6. Manipulateurs, le retour

Hiérarchie de classes

Les entrées et sorties sont gérées dans C++ à travers des objets particuliers appelés *streams* ou *flots*.

Ils ne font pas partie de la spécification de base de C++, et leur implémentation peut varier d'un compilateur à un autre.

La classe de base de la hiérarchie de classes est:

```
class ios
```

Les classes

```
class istream : public virtual ios
class ostream : public virtual ios
```

gèrent les entrées et les sorties. La classe

```
class iostream : public istream, public ostream
```

permet l'insertion et l'extraction.

Des classes particulières existent pour la gestion des fichiers:

```
class ifstream : public istream
class ofstream : public ostream
class fstream : public iostream
```

et d'autres classes pour la gestion de tableaux de caractères en mémoire.

```
class istrstream : public istream
class ostrstream : public ostream
class strstream : public iostream
```

Deux *opérateurs* sont surchargées de manière appropriée pour les flots:

- l'opérateur d'*insertion* `<<` (écriture)
- l'opérateur d'*extraction* `>>` (lecture)

Les flots prédéfinis sont

- `cout` (de la classe `ostream_withassign`) associé à la sortie standard;
- `cerr` associé à la sortie erreur standard;
- `clog` associé à la sortie erreur standard, mais les données sont bufférisées;
- `cin` (de la classe `istream_withassign`) associé à l'entrée s-tandard.

Les opérateurs d'insertion et d'extraction sont prédéfinis dans les classes `istream` et `ostream` pour les types de base, par exemple: L'opérateur retourne une référence d'un stream, et on peut donc enchaîner les ordres de lecture resp. d'écriture.

Toute nouvelle classe `X` peut surdéfinir les opérateurs d'insertion et d'extraction en fonctions amies selon le canevas:

```
istream& operator>>(istream& in, X& objet) {
  /* Lecture de l'objet */
  return in;
}

ostream& operator<<(ostream& out, const X& objet) {
  /* Ecriture de l'objet */
  return out;
}
```

Etat d'un flot

Chaque flot possède un *état* dont la valeur reflète la situation du stream. L'état du flot peut être examiné par:

```
class ios {
...
public :
  int eof() const;  // fin de fichier : 1=EOF 0 sinon
  int fail() const; // opération a échoué : 2=fail, 0=ok
  int bad() const;  // flot est inutilisable : 4=bad, 0=ok
  int good() const; // flot est en bon etat :1=ok, 0 sinon
```

Dans l'état `fail()` on sait que le flot n'est pas altéré, alors que dans `bad()` on ne peut faire cette hypothèse.
Dans une écriture comme

```
while (cin >> i) {..};
```

l'état du flot est testé, et le test est réussi ssi l'état est `good()`. Dans les autres cas, on retourne `0`.
Une *conversion* est définie pour la classe `ios`:

```
ios::operator void * ();
```

qui retourne une adresse non nulle si l'état du fichier est `good()`.

Manipulateurs

Les *manipulateurs* sont une façon agréable de formater. Inclure

```
#include <iomanip.h>
```

- Un manipulateur s'insère dans le flot comme une donnée ou une référence
- Il s'applique à partir de son insertion.

Exemple

```
cout << x << flush << y << flush;
```

Les manipulateurs prédéfinis sont:

`oct`	(io)	notation octale
`hex`	(io)	notation hexadécimale
`dec`	(io)	notation decimale
`endl`	(o)	ajoute `\n`
`ends`	(o)	ajoute `\0`
`ws`	(i)	ignore blancs en tete
`flush`	(o)	vide tampon
`setprecision(int)`	(o)	chiffres décimaux
`setw(int)`	(o)	gabarit
`setfill(char)`	(o)	caractère de remplissage

`setprecision(int n)`
fixe à `n` le nombre de chiffres d'affichage.

`setw(int n)`
fixe à `n` le nombre minimum de caractères utilisés pour le prochain affichage.

Fichiers

Les fichiers sont associés à des flots des classes **ifstream**, **ofstream** et **fstream**.

```
int main(int argc, char* argv[]) {
  if (argc != 3) error("arguments");

  ifstream depuis(argv[1]);
  ofstream vers(argv[2]);

  char c;
  while (depuis.get(c)) vers.put(c);

  if (!depuis.eof() || vers.bad())
    error("hm");
}
```

Les constructeurs peuvent prendre un deuxième argument optionnel pour des modes de lecture ou écriture que sont

```
class ios { ...
public :
  enum open_mode {
    in = 1;
    out = 2;
    ate = 4; //ouvre a la fin
    app = 010; //append
    trunc = 020; // tronque a longueur 0
    nocreate = 040; // echoue si fichier n'existe pas
    noreplace = 0100; // echoue si fichier existe
  } ...
};
```

On peut alors écrire:

```
ofstream fichier(nom, ios::out | ios::nocreate);
```

Formatage des sorties

- Formatage d'un flot par modification de l'*état de formatage* (un entier long de la classe **ios**).
- Chaque bit de l'état code un paramètre.
- Chaque paramètre a un nom symbolique
- **setf()** active le paramètre, **unsetf()** le désactive. Deux formes de **setf()** existent.
- Un paramètre activé le reste jusqu'à désactivation explicite.
- On peut obtenir la valeur de l'état par la méthode **flags()**

```
cout << hex << cout.flags() << endl;
```

Quelques paramètres

ios::skipws	0x0001	saute blancs en tête (entrée)
ios::left	0x0002	justification à gauche
ios::dec	0x0010	sortie en format décimal
ios::oct	0x0020	sortie en format octal
ios::hex	0x0040	sortie en format hexadécimal
ios::showbase	0x0080	montre 0 resp 0x
ios::showpoint	0x0100	complète les décimales
ios::fixed	0x1000	nombre fixe de décimales

Emploi

Activation par **setf**, désactivation par **unsetf**. L'activation d'un attribut l'ajoute à l'état de formatage.

Première forme. de **setf**

```
cout.setf(ios::showpos); // effet: etat |= ios::showpos;
cout.unsetf(ios::showpos); // effet: etat &= !ios::showpos;
```

Plusieurs paramètres peuvent être activés ou désactivés simultanément. Par exemple,

```
cout.setf(ios::showpos | ios::showpoint);
cout << 65.32;
```

```
-> +65.3200
```

et

```
cout.setf(ios::hex | ios::uppercase |ios::showbase);
cout << 65;
```

```
-> 0X41
```

Rien (sauf le bon sens) n'empèche d'écrire

```
cout.setf(ios::dec|ios::oct|ios::hex);
```

avec des résultats imprévisibles.

La *deuxième forme* de la méthode **setf()** prend deux arguments

```
setf(parametres, masque);
```

avec pour effet

```
unsetf(masque);
setf(parametres & masque); // attention
```

Exemple

```
cout.setf(ios::oct, ios::dec|ios::oct|ios::hex);
```

Effet

etat	=	a	b	c	d
masque	=	0	1	1	1
`unsetf`	=	a	0	0	0
paramètre	=	1	0	1	0
`setf`	=	a	0	1	0

La deuxième forme de `setf()` est utilisée avec les raccourcis suivants:

- `ios::basefield = ios::dec|ios::oct|ios::hex` pour la base de numération;

```
   cout.setf(ios::showbase);
   cout.setf(ios::hex, ios::basefield);
   cout << 65;
-> 0x41
```

- `ios::floatfield = ios::fixed|ios::scientific` pour les nombres flottants;

```
   cout.setf(ios::fixed, ios::floatfield);
   cout << 65.2 << endl;
   cout.setf(ios::scientific, ios::floatfield);
   cout << 65.2 << endl;
-> 65.200000
   6.5200000e+01
```

- `ios::adjustfield = ios::left|ios::right|ios::internal` pour la justification.

```
   cout.width(5); cout.fill('*');
   cout.setf(ios::showpos);
   cout.setf(ios::internal,ios::adjustfield);
   cout << 65;
-> +**65
```

Autres fonctions de sortie

- **put(char)** affiche son argument comme caractère:

```
   cout.put(65).put(67) << endl;
-> AC
```

- **width(int)** fixe la longueur minimale de l'affichage (par défaut 0). *Ne s'applique qu'à la donnée suivante*

```
   cout.width(8); cout.fill('*');
   cout << 65 << 66 << endl;
-> ******6566
```

- **fill(char)** fixe le caractère de remplissage. S'applique à toutes les sorties subséquentes.
- **precision(int)** fixe le nombre de chiffres (resp. chiffres après le point décimal.

Considérons les instructions suivantes:

```
cout << 1.23456789 << endl;
cout << 4.00 << endl;
cout << 5.6789E2 << endl;
cout << 0.0 << endl;
```

L'affichage par défaut est

```
1.23457
4
567.89
0
```

Précédé de `cout.precision(3);`, devient

```
1.23
4
568
0
```

Précédé de `cout.precision(3);cout.setf(ios::showpoint);`

```
1.23
4.00
568.
0.00
```

Par `cout.setf(ios::scientific, ios::floatfield);`

```
1.2345678900e+00
4.0000000000e+00
5.6789000000e+02
0.0000000000e+00
```

et avec `cout.setf(ios::fixed, ios::floatfield);`

```
1.2345678900
4.0000000000
567.8900000000
0.0000000000
```

Les fonctions **tellp** et **seekp** de la classe **ostream** positionnent l'écriture.
Servent pour l'écriture dans les fichiers (classe **ofstream** qui dérive de **ostream**).

- **tellp()** fixe la position courante dans le flot.
- **seekp** a deux formes
  - **ostream& seekp(streampos pos)**. Fixe la position absolue à **pos**.
  - **ostream& seekp(streampos pos, ios::seekdir base)**. Fixe la position, en comptant **pos** relativement à **base**. L'argument **base** peut prendre les valeurs **ios::beg**, **ios::cur** ou **ios::end**.

Exemple:

```
ofstream in("toto"); // ouvre "toto" en ecriture
in << "12345";       // ecrit dans toto
cout << in.tellp();  // affiche 5
in.seekp(3);         // positionne apres 3
in << " soleil";
```

Le fichier contient **123 soleil**.

Mêmes comportements pour les fonctions **seekg()** et **tellg()** en lecture.

Entrées

Lecture simple

```
cin >> entier >> car >> flottant;
```

donne

```
␣23A1.2    23  A  1.2
23␣A␣1.2   23  A  1.2
23␣A␣1␣.2  23  A  1
```

(blancs en tête sont sautés)

La méthode `cin.get()` vient en plusieurs variantes.

(1) `cin.get(char&)` honore les blancs: `cin.get(a).get(b);` pour l'entrée `␣A␣B` retourne `a=␣` et `b=A`.

(2) `int iostream::get()` sans argument retourne

- le dernier caractère lu, s'il y en a un;
- ou `EOF` si fin de fichier.

D'où une boucle comme

```
while((c = cin.get()) !=EOF) {...}
```

Lecture de textes

```
char txt[80];
cin >> txt;
```

donne, pour

```
␣␣␣Bonjour␣Monde
```

le résultat

```
Bonjour
```

car par défaut, les blancs en tête sont sautés et la lecture s'arrête au premier blanc. En revanche

```
cin.get(txt,11);
```

donne

```
␣␣␣Bonjour␣
```

car le nombre demandé de caractères est lu.

Le manipulateur `ws` saute les blancs en tête. Ainsi

```
cin >> ws; cin.get(txt,11);
```

donne

```
Bonjour␣Mon
```

(5) La forme générale est

```
get(char* txt, int len, delim = '\n');
```

et lit au maximum `len` caractères, et arrête la lecture au premier `delim` rencontré. Dans Gnu, `getline()` a le même comportement, à ceci près que le délimiteur est enlevé du flot.

Autres fonctions de lecture

`gcount` retourne le nombre de caractères lus par la dernière opération de lecture.

`putback(char)` et `unget` replacent le dernier caractère lu dans le flot d'entrée.

`int peek()` a le même comportement que `get()` sauf que le caractère lu n'est pas retiré du flot d'entrée.

Etat de lecture

L'état de lecture décrit la situation du flot d'entrée. Quatre fonctions en retournent les détails. Par exemple

	eof()	good()	fail()	bad()	(cin)
65	F	T	F	F	T
rien	F	F	T	F	F
Ctrl- D	T	F	T	F	F

En cas d'erreur, on passe en `fail`, et la lecture est bloquée. La méthode `clear()` remet l'état à `good`.

L'état d'un flot peut être testé par la méthode `rdstate()`.

Lecture et écriture en mémoire

Il s'agit d'analogues des `sscanf` et `sprintf` de C.

On inclut `strstream.h`.

La classe `istrstream` a deux constructeurs

- `istrstream(const char*)` qui associe un flot à la chaîne passée en paramètre.
- `istrstream(const char*, int)` qui associe le flot à la chaîne avec taille.

Exemple:

```
char txt = "ABC 1.23 45";
char t[100]; float f; int i;

istrstream in(txt); // construction

in >> t >> f >> i;
cout << t <<","<< f  <<"," << i;
```

Le résultat est `ABC,1.23,45`.

La syntaxe est la même pour la sortie: `ostrstream` remplace `istrstream`.

Lecture et écriture simultanée

Un `strstream` peut servir comme file (FIFO) pour ranger des données.

Exemple:

```
strstream es;     // cree un strstream
es << "ABC" << " " << 1.234 << " " << 5 << endl;

cout << es.rdbuf() << endl; // tout

char t[100]; float f; int i;
es.seekg(0);     // rewind
es >> t >> f >> i;
cout << t << "," << f << "," << i << endl;
```

donne

```
ABC 1.234 5
ABC,1.234,5
```

Manipulateurs, le retour

Pourquoi peut-on écrire

`cout << x << flush << y << flush;` ?

La fonction (globale) **`flush()`** est déclarée

```
ostream& flush(ostream&);
```

Posons (pour simplifier ?)

```
typedef ostream& (*Manip)(ostream&);
```

Alors **`flush`** est du type **`Manip`**.
L'opérateur d'insertion de la classe **`ostream`** est déjà surchargé:

```
ostream& operator<<(Manip f) {
  return f(*this);
}
```

Noter l'apparition du schéma conceptuel *visiteur*.
Ainsi, **`cout << endl`** est l'équivalent de **`endl(cout)`**! Par exemple, avec

```
ostream& h(ostream& out) {
  return out << " --- ";
}
```

La ligne

```
cout << "Bonjour" << h << "Monde";
```

affiche

```
Bonjour --- Monde
```

On transforme ainsi une fonction en *manipulateur*.

Une fonction avec argument dans un flot de sortie, comme

```
cout << setprecision(4) << x;
```

s'écrit en deux temps:

- la fonction `setprecision()` retourne un objet d'une nouvelle classe, contenant la fonction et l'argument;
- l'opérateur d'insertion de la nouvelle classe est surchargé pour appliquer la fonction à son argument.

Les fonctions considérées sont de type

```
typedef ostream& (*FManip)(ostream&, int);
```

La nouvelle classe est

```
class Omanip {
private:
  FManip f;
  int i;
public:
  Omanip(FManip f, int _i) : f(f), i(i) {}
  friend ostream& operator<<(ostream& os, Omanip& m)
    { return m.f(os, m.i);}
}
```

Exemple : la fonction `setprecision()` retourne un objet de la classe `Omanip` à partir d'une fonction de type `FManip`.

```
Omanip setprecision(int i) {
  return Omanip(&precision, i);
}
```

La fonction `ostream& precision(ostream&, int i)` est définie par

```
ostream& precision(ostream& os, int i) {
  os.precision(i);
  return os;
}
```

Ainsi, l'exécution de

```
cout << setprecision(4)
```

débute par l'évaluation de `setprecision(4)` qui retourne un objet temporaire de la classe `Omanip`, soit

```
temp =  Omanip(&precision, 4)
```

L'évaluation de

```
cout << temp
```

se traduit par l'appel de

```
precision(cout, 4)
```

Une définition générique est contenue dans le fichier `iomanip.h`

11

Patrons

1. Fonctions génériques
2. Classes génériques

Fonctions génériques

Une *fonction générique* ou modèle de fonction, est une fonction dans laquelle le type de certains arguments est paramétré.

La syntaxe la plus simple est

template <class *paramètre***>** *définition de fonction*

Exemple:

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

On peut alors écrire

```
int i = max(34,45);
double d = max(4.5,3.4);
```

Le compilateur construit deux instances de la fonction **max**, de prototypes

```
int max(int, int)
double max(double, double)
```

Notes:

1. Le compilateur n'engendre le code d'une fonction avec des arguments de type donné que lorsqu'il y a un appel de cette fonction. Potentiellement, le nombre de fonctions définies par un patron est illimité.
2. Dans la définition d'un patron, le mot-clé **`class`** n'a pas sa signification habituelle: il indique qu'un paramètre de type va suivre.
3. Le compilateur ne fait pas les conversions standard. Ainsi, on ne peut pas écrire avec le patron ci-dessus:

   ```
   char c;
   int i,j;
   ...
   j = max(i,c);
   ```

Il y a des patrons à plusieurs paramètres:

```
template <class T, class U>
T max(T a, U b) { return (a>b) ? a : b; }
```

ce qui permet d'interpréter correctement le **`max(i,c)`** ci-dessus.

Classes génériques

Les *classes génériques* ou *patrons* ou *modèles* de classes permettent de définir des classes paramétrées par un type ou par une autre classe. Ceci permet d'obtenir une certaine généricité et est donc très utile pour les structures de données.

Par exemple, une pile d'entiers ne diffère d'une pile de caractères que par le type de ses éléments. Si l'on dispose d'une pile paramétrable, on peut écrire:

```
main()
{
  ...
  Pile <int> S;
  Pile <char> Op;
  ...
  char op = Op.pop();
  S.push(x*y);
  ...
}
```

Exemple: un programme rudimentaire d'évaluation d'expressions arithmétiques complètement parenthésées. Il utilise deux piles: une pile d'opérandes et une pile d'opérateurs.

```
void main() {
  char c;
  Pile <int> S;
  Pile <char> Op;

  while (cin.get(c)) {
    if (c == '\n') break;
    switch(c) {
      case '(': break;
      case '+':
      case '*':
        Op.push(c);
        break;
      case ')':
        int x = S.pop();
        int y = S.pop();
        switch (char op = Op.pop()) {
          case '+' : S.push(x+y); break;
          case '*' : S.push(x*y);
        };
        break;
      default :
        if (('0'<=c) && (c<='9'))
          S.push(c-'0');
    }
  }
  cout << S.pop();
}
```

Définition

Une classe *générique* ou *modèle* ou *patron*, en anglais *template* est une classe paramétrée par un type.

Exemple

```
template <class T> class Pile {
  private:
    T p[10];
    int sp;
  public
    Pile() {sp = 0;}
    void push(T x) { p[sp++] = x; }
    T pop(void) {return p[--sp]; }
};
```

On remarque

- Un nouveau mot-clé `template`;
- Le paramètre est entre chevrons;
- Définition de classe concrète par spécification d'un type.

  ```
  Pile<int> S;
  ```

  ou fréquemment

  ```
  typedef Pile<int> PileEntiers;
  PileEntiers S;
  ```

Plusieurs paramètres peuvent intervenir:

```
template <class T, int Max>
class Pile {
private:
  T p[1+Max];
  int sp;
  int maximum;
public
  Pile() : maximum(Max), sp(0) { }
  void push(T x) { p[sp++] = x; }
  T pop(void) { return p[--sp]; }
  int IsFull(void) { return sp == maximum;}
};
```

Si l'on sépare les déclarations des définitions, il convient de répéter le mot-clé `template` à chaque définition.

Concrétisation et spécialisation

```
template <class T, class U, int N>
class Patron {
  // ...
}
```

La définition d'un constructeur s'écrit, en dehors de la déclaration:

```
template <class T, class U, int N>
Patron<T, U, N>::Patron() {...}
```

On *concrétise* le patron en donnant des types "concrets":

```
class Patron<int, float, 5> C1;
class Patron<Point, int, 100> C2;
```

si `Point` est une classe. On peut substituer à un paramètre un autre patron.
On *spécialise* est redéfinissant une partie de la classe. Par exemple, pour un patron

```
template <class T>
class Point {
  // ...
}
```

on précise la définition dans le cas des caractères

```
class Point<char> {
  // redefinition ou extension
}
```

On peut ne définir qu'une seule méthode par

```
void Point<char>::doit(char)...
```

Une pile générique

Voici une pile générique, implémentée à l'aide d'une liste chaînée.

```
template <class Arg>
class Pile {
public :
  Pile() { p = NULL; }
  void push(Arg x) { p = new Cellule(x, p); }
  Arg top() { return p -> cont; }
  Arg pop();
private :
  struct Cellule {
    Cellule* suiv;
    Arg cont;
    Cellule(Arg c, Cellule* s) : cont(c), suiv(s) {}
  };
  Cellule * p;
};
```

Pour une méthode définie hors de la déclaration (comme la méthode `pop()`), on doit répéter l'en-tête complet:

```
template <class Arg>
Arg Pile<Arg>::pop() {
  Cellule* q = p;
  Arg c = q -> cont;
  p = p -> suiv;
  delete q;
  return c;
}
```

Cette méthode réalise la suppression explicite de la cellule, mais bien entendu pas de l'argument.

12

Conception objet

1. Objet versus interprétation
2. Réification
3. Encapsulation
4. Héritage
5. Abstraction – partition
6. Interface – implémentation
7. Universalité – généricité

Objet versus interprétation

Une *interprétation* est un service réalisé par une fonction ou méthode.

Exemple :

- 4 catégories d'objet : triangle, segment, rectangle, cercle.
- 4 interprétations : tracer, colorer, remplir, épaissir.

Style objet

Chaque interprétation est réalisé par une méthode, pour chaque classe, d'où

- ajout d'une nouvelle catégorie : implémenter toutes les interprétations.
- ajout d'une nouvelle interprétation : ajouter une nouvelle méthode dans chaque classe.

Style interprétation

Chaque interprétation est réalisé par une fonction qui est particularisée pour chaque catégorie (aiguillage), d'où

- ajout d'une nouvelle catégorie : revoir toutes les interprétations.
- ajout d'une nouvelle interprétation : ajouter une nouvelle fonction unique.

Réification

Dans la conception, la première tâche est la *réification*: opération qui transforme en “chose” (Petit Robert), c’est-à-dire la définition de la hiérarchie de classes.

Analyse:

- Quelles sont les classes requises ?
- Existe-t-il des classes reliées ? Peut-on les dériver ?
- Quelles sont les fonctions membres utiles ? Lesquelles virtuelles, virtuelles pures ?
- Quelles sont les données membres ?
- Protection : public, protégé privé ?

Quand utiliser les classes ?

- Une classe représente un *concept*.
- Une classe peut encapsuler des décisions de conception incertaines (et donc susceptibles de changer) ou dépendant de l’environnement.
- Une classe peut masquer la complexité de programmation
- Une classe représente un type de données de portée générale.
- Un classe peut être une interface à une bibliothèque.
- Une classe peut permettre l’usage d’une notation plus appropriée.
- Une classe ne contenant que des attributs statiques sert de réceptacle aux données et fonctions globales.

Encapsulation

L'*encapsulation* est le procédé qui

- ne rend visible à l'utilisateur qu'une partie des données (*data hiding*);
- permet l'accès et la modification des données uniquement par des méthodes spécifiques (*getter* et *setter*).

Exemple : un compte en euro et francs

Une classe **`Compte`** fournit des comptes qui peuvent être crédités en francs ou en euro, par les méthodes

```
float getSoldeE() const;
float getSoldeF() const;
void crediteE(float);
void crediteF(float);
void debiteE(float);
void debiteF(float);
```

Deux implémentations sont possibles: mémoriser le montant

- en euros, et le convertir en francs si nécessaire;
- en francs, et le convertir en euros si nécessaire.

Par l'encapsulation, l'implémentation est cachée. Attributs *calculés* (comme dans l'implémentation des complexes).

Héritage

Une classe dérivée représente

- un cas particulier
- un enrichissement
- une variante

de la classe de base. Elle contient une information supplémentaire qui diminue la liberté des objets.

Exemples de cas particuliers

1) Une classe `Texte` contenant …un texte, et une classe `TexteEnMaj` identique à `Texte`, sauf qu'elle affiche son texte en majuscules.
La classe dérivée n'a pas de donnée propre, et laisse inchangée toutes les méthodes sauf celle d'affichage qui est masquée.

2) Un carré est un cas particulier d'un rectangle, un cercle un cas particulier d'une ellipse.
Les méthodes de la classe `Carre` sont celles de `Rectangle`, mais en veillant maintenir la particularité des carrés. Il faudra donc en réécrire. (enrichissement par contrainte).

Exemple d'enrichissement

Un animal *coloré* est un enrichissement d'un animal.

Abstraction – partition

Une classe base, dont la fonction est de grouper les caractéristiques communes d'un certain nombre de classes, en est une *abstraction*. Une abstraction contient fréquemment des méthodes virtuelles.

Lorsque l'ensemble des posibilités concrètes est couvert par les classes concrètes, il s'agit d'une *partition*. La classe abstraite n'est alors jamais instanciée, et ses méthodes sont souvent virtuelles pures.

Exemples d'abstractions

1) Une classe de base `Forme` et des classes dérivées `Carre`, `Cercle`.

2) Une classe de base `Cellule` (d'un tableur), et des classes dérivées `Nombre`, `Texte`, `Formule`.

Abstraction et définition récursive (Composite)

L'abstraction d'une définition récursive fait d'un conteneur d'objets un objet lui-même. Un objet est alors un arbre, dont les nœuds sont des conteneurs et les feuilles des objets “terminaux”.

1) Une classe de base `Item` et des classes dérivées `Conteneur` et `ItemA`, `ItemB`,... (Une classe conteneur et plusieurs classes de feuilles)

2) Une classe `Expression` et des classes dérivées `Somme`, `Produit`, `Entier`. Ici, `Somme` et `Produit` sont des classes de conteneurs, et `Entier` la classe de feuilles.

3) Une classe de base `Terme` (logique). Des classes dérivées `Constante`, `Variable`, `FonctUnaire` (comme `not`), `FonctBinaire` etc.

4) Arbres binaires : une classe de base `Sommet`, et des classes dérivées `Noeud` (conteneur de deux sommets) et `Feuille`.

Interface – implémentation

Un *type abstrait de données* est un ensemble (dynamique) muni d'un ensemble d'opérations.

Un type abstrait de données est représenté par une *interface*, c'est-à-dire par une classe dont les méthodes sont virtelles pures.

Type abstrait : File de priorité

Trois méthodes

```
void add(Item) = 0;
void remove() = 0;
Item getMinimum() = 0;
```

Type abstrait : Dictionnaire

Trois méthodes

```
void add(Item) = 0;
void remove(Item) = 0;
bool find(Item) = 0;
```

Une *structure de données* est un ensemble de données et de fonctions qui réalisent un type abstrait de données.

Une structure de données est représenté par une classe dérivée de l'interface qui implémente ses méthodes.

Universalité – généricité

Une classe *universelle* peut être utilisée avec n'importe quel type de données. Le type des données est en général `void *` en C++, et `Object` en Java. Les types doivent être convertis au type actuel.

Pile universelle

```
class Pile {
  public :
    Pile() : sp(0) {}
    void push(void* x) { p[sp++] = x; }
    void* top() { return  p[sp -1]; }
    void* pop() { return  p[--sp]; }
  private :
    int sp;
    void* p[10];
};
```

Une classe *générique* est paramétrée par un type. La concrétisation s'obtient en substituant un type concret au paramètre du type.

Pile générique

```
template <class T> class Pile {
  public
    Pile() : sp(0) {}
    void push(T x) { p[sp++] = x; }
    T top() { return  p[sp -1]; }
    T pop() {return p[--sp]; }
  private:
    T p[10];
    int sp;
};
```

La pile universelle s'obtient en concrétisant la pile générique par

```
typedef Pile<void*> PileUniverselle;
```

13

Design pattern

1. Objectifs
2. Composite
3. Visiteur
4. Adaptateur
5. Observateur
6. Itérateur

Objectifs

Un *design pattern* ou *schéma de conception* est une solution éprouvée pour traiter une situation typique et qui se rencontre fréquemment.

Avantages

- la palette des design patterns existants facilite l'identification d'un cas typiques,
- l'identification d'un design pattern facilite la communication, et la programmation
- le traitement par un design pattern conserve la généralité et la flexibilité de la conception.

Principe général : programmer avec des interfaces (classes abstraites) de préférence à des implémentations: renforce le service (quoi) plutôt que la méthode (comment).

Une classe abstraite se compose de signatures d'opérations et de constantes. Une classe concrète implémente la classe abstraite. Plusieures implémentations possibles.

Ne pas confondre les niveaux de conception:

- *architecture* : structure globale d'un projet, souvent en couche;
- *design pattern* : solution fine d'une situation, indépendamment du langage;
- *idiome* : solution spécifique à un langage donné.

Plusieurs design patterns sont des idiomes du langage Smalltalk.

Adaptateur (Adapter, Wrapper)

Une classe qui pourvoit une classe existante avec une nouvelle interface. Sert à adapter ou encapsuler une classe existante.

Décorateur (Decorator)

Permet de modifier les propriétés d'un objet indépendamment de sa classe.

Façade (Facade)

Une classe qui interface un sous-système, formé d'un ensemble de classes.

Mémento (Memento)

Permet de restaurer des états antérieurs d'objets, et d'implémenter le undo.

Observateur (Observer)

Mécanisme pour informer un ensemble d'objets que l'état de l'objet observé a changé (un modèle — plusieures vues).

Proxy

Fournit un cadre à un autre objet et permet de le contrôler.

Itérateur (Iterator)

Fournit un moyen d'accéder séquentiellement aux objets d'une collection sans exposer l'implémentation concrète.

Visiteur (Visitor)

représente une façon d'effectuer une opération sur un objet sans que l'objet connaisse la nature de l'opération.

Composite (Composite)

structure arborescente, de sorte que groupe d'objet puisse être utilisé de la même manière qu'un objet singleton.

Composite

Le principe de ce design pattern a été exposé lors de la programmation d'arbres.

Visiteur

Le principe de ce design pattern a été exposé lors de la programmation de l'évaluation d'expressions postfixées

Adaptateur

Observateur

Itérateur

Le *parcours* d'un conteneur se formalise par l'introduction de classes spécifiques, appelées *Enumération* en Java et *Itérateurs* en C++.
Une implémentation générique (indépendante des types actuels) est donnée dans la STL (Standard Template Library).

Un *itérateur* est attaché à une classe. Les valeurs d'un itérateur sont des *positions*. Concrètement, un itérateur est un pointeur ou une variable d'indice, et ses valeurs sont des adresses ou des indices.
Un itérateur a les méthodes suivantes:

- **`first()`** qui lui affecte la première position
- **`first()`** qui le fait avancer à la position suivante
- **`bool isDone()`** qui indique que le parcours est achevé
- **`Item& currentItem()`** qui retourne une référence à l'élément à la position courante.

Exemple: Si on dispose d'un itérateur sur une classe de vecteurs.

```
void print(VIterator& i) {
  for (i.first(); !i.isDone(); i.next())
    cout << i.currentItem() << " ";
  cout << endl;
}

Vecteur x; ...
VIterator i = x.createIterator(); // itérateur sur x
print(i);
```

Pour ce faire, voici une classe de vecteurs de **string**.

```
class Vecteur {
protected:
  string *a;
  int n;
public:
  int taille() const {return n;}
  Vecteur::Vecteur(int t) { a = new string[t]; n = 0; }
  string& operator[](int i) const {return a[i];}
  VIterator createIterator() const;
  void add(string x) { a[n++] = x;}
};
```

La classe **VIterator** d'itérateurs de vecteurs s'écrit

```
class VIterator {
  const Vecteur* v;   // objet énuméré
  int courant;        // indice interne
public:
  void first() { courant = 0; }
  void next() { courant++; }
  bool isDone() const { return courant >= (v->taille()); }
  string& currentItem() const { return (*v)[courant]; }
  VIterator(const Vecteur* v) : v(v), courant(0){}
};
```

La méthode **createIterator()** est facultative. Elle s'écrit:

```
VIterator Vecteur::createIterator() const {
  return VIterator(this);
}
```

Le programme

```
void main() {
  Vecteur x;
  x.add("Alpha"); x.add("Beta");
  VIterator i = x.createIterator();
  // ou aussi VIterator i = VIterator(&x);
  for (i.first(); !i.isDone(); i.next())
    i.currentItem() +=" !";
  print(i);
}
```

fournit le résultat

```
Alpha ! Beta !
```

De même, on peut itérer sur les listes chaînées. Voici une classe de listes de **`string`**

```
class Liste {
protected:
  class Maillon {
  public:
    Maillon* suiv;
    string cont;
    Maillon(string c, Maillon* n = NULL) : cont(c), suiv(n) {}
  };
  Maillon* tete;
public:
  Liste() : tete(NULL) {}
  void add(string c) { tete = new Maillon(c, tete);}
  LIterator createIterator() const;
  friend class LIterator;
};
```

La classe **LIterator** d'itérateurs de listes s'écrit

```
class LIterator {
  const Liste* v;           // objet énuméré
  Liste::Maillon* courant;  // indice interne
public:
  void first() { courant = v -> tete; }
  void next() { courant = courant -> suiv; }
  bool isDone() { return courant == NULL; }
  string& currentItem() { return courant -> cont; }
  LIterator(const Liste* v) : v(v), courant(v -> tete){}
};
```

On définit comme ci-dessus:

```
void print(LIterator& i) {
  for (i.first(); !i.isDone(); i.next())
    cout << i.currentItem() << " ";
  cout << endl;
}
```

Seule l'initiale du nom de la classe d'itérateur change ! Le même programme principal s'applique:

```
void main() {
  Liste x;
  x.add("Alpha"); x.add("Beta");
  LIterator i = LIterator(&x);
  //ou LIterator i = x.createIterator();
  for (i.first(); !i.isDone(); i.next())
    i.currentItem() += " !";
  print(i);
}
```

Abstraction

L'exemple conduit naturellement à une double abstraction

- Une classe abstraite d'itérateurs
- Une classe abstraite de conteneurs d'une suite d'éléments

Le type de base retenu dans cet exemple est **string**. Mieux : des classes patron, paramétrées par le type de base.

Profil de la classe d'itérateurs:

```
class Iterator {
public:
  virtual void first() =0;
  virtual void next() =0;
  virtual bool isDone() const =0;
  virtual string& currentItem() const =0;
};
```

Classe de conteneurs, très simplifiée:

```
class Sequence {
public:
  virtual Iterator* createIterator() const =0;
  virtual void add(string x) =0;
  void printIt();
};
```

Il est plus commode de gérer des pointeurs.

La méthode **printIt()** est une *template method* ou méthode générique : elle est définie en utilisant des fonctions virtuelles pures.

```
void Sequence::printIt();{ // template method
  Iterator* i = createIterator();
  for (i -> first(); !i -> isDone(); i -> next())
    cout << i -> currentItem() << " ";
  cout << endl;
  delete i;
}
```

Il suffit maintenant

- de dériver `Vecteur` et `Liste` de `Sequence`
- de dériver `VIterator` et `LIterator` de `Iterator`

La méthode globale `print` a pour argument une référence d'itérateur

```
void print(Iterator& i) {
  for (i.first(); !i.isDone(); i.next())
    cout << i.currentItem() << " ";
  cout << endl;
}
```

La même suite d'instructions s'applique aux vecteurs et aux listes

```
void main() {
  Sequence* x = new Vecteur();
  //resp. Sequence* x = new Liste();
  x -> add("Alpha"); x -> add("Beta");
  Iterator* i = x ->createIterator();
  for (i -> first(); !i->isDone(); i -> next())
    i->currentItem() +=" !";
  Iterator* k = x -> createIterator();
  print(*k);
  x -> printIt();
```

Annexe 1 : Sequences et itérateurs

```
#include <iostream.h>
#include <string>
////////////////////////
// Iterator
class Iterator {
public:
  virtual void first() =0;
  virtual void next() =0;
  virtual bool isDone() const =0;
  virtual string& currentItem() const =0;
};
////////////////////////
// Sequence
class Sequence {
public:
  virtual Iterator* createIterator() const =0;
  virtual void add(string x) =0;
  void printIt() { // template method
    Iterator* i = createIterator();
    for (i -> first(); !i -> isDone(); i -> next())
      cout << i -> currentItem() << " ";
    cout << endl;
    delete i;
  }
};
////////////////////////
// Vecteur : Sequence
class Vecteur : public Sequence {
protected:
  string *a;
  int n;
public:
  int taille() const {return n;}
  Vecteur(int t = 10){ a = new string[t]; n = 0; }
  string& operator[](int i) const {return a[i];}
  Iterator*  createIterator() const;
  void add(string x) { a[n++] = x;}
};
////////////////////////
// VIterator : Iterator
class VIterator : public Iterator {
  const Vecteur* v;   // objet énuméré
  int courant;        // indice interne
public:
  void first() { courant = 0; }
  void next() { courant++; }
  bool isDone() const { return courant >= (v->taille()); }
```

```
  string& currentItem() const { return (*v)[courant]; }
  VIterator(const Vecteur* v) : v(v), courant(0){}
};
Iterator* Vecteur::createIterator() const { return new VIterator(this); }
/////////////////////////
// Liste : Sequence
class Liste : public Sequence {
  class Maillon {
    Maillon* suiv;
    string cont;
  public:
    Maillon(string c, Maillon* n = NULL) : cont(c), suiv(n) {}
    friend class LIterator;
  };
protected:
  Maillon* tete;
public:
  Liste() : tete(NULL) {}
  void add(string c) { tete = new Maillon(c, tete);}
  Iterator* createIterator() const;
  friend class LIterator;
};
/////////////////////////
// LIterator : Iterator
class LIterator : public Iterator {
  const Liste* v;           // objet énuméré
  Liste::Maillon* courant;  // indice interne
public:
  void first() { courant = v -> tete; }
  void next() { courant = courant -> suiv; }
  bool isDone() const { return courant == NULL; }
  string& currentItem() const { return courant -> cont; }
  LIterator(const Liste* v) : v(v), courant(v -> tete){}
};
Iterator* Liste::createIterator() const { return new LIterator(this); }
/////////////////////////
// print
void print(Iterator& i) {
  for (i.first(); !i.isDone(); i.next())
    cout << i.currentItem() << " ";
  cout << endl;
}
/////////////////////////
// main
void main() {
  Sequence* x = new Vecteur();
  x -> add("Alpha"); x -> add("Beta");
  Iterator* i = x ->createIterator();
  for (i -> first(); !i->isDone(); i -> next())
```

```
    i->currentItem() +=" !";
  Iterator* k = x -> createIterator();
  print(*k);
  x -> printIt();

  Sequence* y = new Liste();
  y -> add("Alpha"); y -> add("Beta");
  Iterator* j = y ->createIterator();
  for (j -> first(); !j->isDone(); j -> next())
    j->currentItem() +=" !";
  Iterator* l = y -> createIterator();
  print(*l);
  y -> printIt();
}
```

14

Rtti

1. Objectif
2. Réalisation
3. Exemple
4. Conversion dynamique

Objectif

Obtenir le type effectif d'un objet, à l'exécution.

Par exemple, si un pointeur est déclaré d'un certain type, il peut désigner un objet d'un autre type, notamment d'un type dérivé.

Les dernières versions permettent

- de connaître le type effectif d'un objet,
- de tester si une conversion de type est licite.

Le mécanisme utilisé s'appelle *run-time type information* (*RT-TI*).

Il faut inclure `<typeinfo>` et (parfois) compiler avec l'option `-frtti`

Réalisation

C++ dispose d'une structure de bibliothèque

```
type_info
```

Ce n'est pas exactement une classe. La fonction

```
typeid(type_name)
```

retournant un objet de la structure **type_info**. L'argument de **typeid()** est, un peu comme pour **sizeof** de C, soit un objet, soit un nom de classe.

Exemple. Considérons deux classes

```
class Cellule {
 public:  virtual void f(){}
};
class Entier : public Cellule {};
```

et, dans un programme,

```
Cellule* n = new Entier();
```

Les résultats de

```
cout << (typeid(*n) == typeid(Entier)) << endl;
cout << (typeid(*n) == typeid(Cellule)) << endl;
```

sont `1` et `0`. Les opérations autorisées sur la structure **type_info** sont

- la comparaison par `==` et `!=`;
- la prise de nome, par la méthode **name()**: le résultat est une chaîne de caractères, dont la forme exacte dépend de l'implémentation.

Exemple

```
#include <typeinfo>
#include <iostream.h>

class Cellule {
 public:
  virtual void  show() =0;
};

class Entier : public Cellule {
 public:
  void  show() { cout << "Entier" <<endl; }
};

class Texte : public Cellule {
 public:
  void  show() { cout << "Texte" <<endl; }
};

int  main() {
  Cellule* n = new Entier();
  Cellule* t = new Texte();
  n -> show();
  t -> show();
  cout <<typeid(*n).name()<<endl;
  cout <<typeid(Entier).name()<<endl;
  cout <<typeid(*t).name()<<endl;
  cout <<typeid(Texte).name()<<endl;

  cout << (typeid(*n) == typeid(Entier)) << endl;
  cout << (typeid(*t) == typeid(Cellule)) << endl;
}
```

Le résultat est

```
Entier
Texte
TD.6Entier
TD.6Entier
TD.5Texte
TD.5Texte
1
0
```

En revanche, le programme suivant

```
#include <typeinfo>
#include <iostream.h>

class Cellule {};
class Entier : public Cellule {};
class Texte : public Cellule {};

int  main() {
  Cellule* n = new Entier();
  Cellule* t = new Texte();
  cout <<typeid(*n).name()<<endl;
  cout <<typeid(Entier).name()<<endl;
  cout <<typeid(*t).name()<<endl;
  cout <<typeid(Texte).name()<<endl;
  cout << (typeid(*n) == typeid(Entier)) << endl;
  cout << (typeid(*t) == typeid(Cellule)) << endl;
}
```

a le résultat

```
TD.7Cellule
TD.6Entier
TD.7Cellule
TD.5Texte
0
1
```

En effet, si une classe ne contient aucune fonction virtuelle, la classe indiquée par `typeid` est la classe de l'indentificateur, et non celle à la création.

Conversion dynamique

Lorsqu'on a une classe `Der` dérivée d'une classe de base `Base`, une instruction du genre

```
Base* b = new Der;
```

retourne dans `b` une pointeur vers un objet de la classe `Der`. Seuls les membres de cet objet qui figurent dans `Base` sont accessibles par `b`. Il est donc naturel de forcer la conversion de `b` en un pointeur vers la classe `Der` par

```
(Der *) b
```

Par défaut, aucun test de la validité de cette conversion n'est faite dans C++. La technique du *dynamic cast* sert à cela.

La syntaxe est la suivante. Soit `Base` une classe contenant au moins une fonction virtuelle, et `Der` une classe qui en dérive. Alors on écrit

```
Base* b = new Der;
Der* d = dynamic_cast<Der *>(b);
```

Le résultat du forçage dynamique est un pointeur non nul en cas de réussite, et `0` en cas d'échec.

En absence de fonction virtuelle, il y a erreur à la compilation.

15

Standard Template Library

1. Objectifs
2. Exemples
3. Conteneurs
4. Méthodes communes
5. Algorithmes
6. Objets fonctions
7. Autres algorithmes
8. Itérateurs

Objectifs

La *bibliothèque générique standard* ou *STL* fournit un ensemble de structures de données et d'algorithmes génériques.

Aspects principaux

- STL est une norme ANSI (juillet 1994).
- STL est efficace, en évitant l'héritage et les fonctions virtuelles.
- STL offre 10 familles de "conteneurs": **`vector`**, **`deque`**, **`list`**, **`multiset`**, **`set`**, **`multimap`**, **`map`**, **`stack`**, **`queue`**, **`priority_queue`**.
- STL utilise des *itérateurs* qui sont une extension des pointeurs. Un pointeur peut accéder aux données, et parcourir un conteneur.
- STL fournit 70 algorithmes. Un algorithme n'est pas une fonction membre. Il opère sur les conteneurs par itérateurs.
- STL connaît les objets fonctions. Ces objets encapsulent des fonctions.
- STL introduit un mécanisme spécifique de gestion de mémoire par *allocateurs* qui se superposent à **`new`** et **`delete`**.

Exemples

Utilisation d'un vecteur

```
#include<iostream.h>
#include<vector.h>

main() {
  vector<int> v;        // vecteur d'entiers vide

  for (int i=0; i < 10 ; i++)
    v.push_back(i*i);    // a la fin
  int k;
  cout << "Un entier : ";
  cin >> k;
  cout << "v[" << k << "] = " << v[k] << endl;
  cout << "Taille de v " << v.size() << endl;
}
```

Résultat:

```
Un entier : 7
v[7] = 49
Taille de v 10
```

Utilisation d'un itérateur

```
#include<iostream.h>
#include<vector.h>

main() {
  vector<int> v;       // vecteur d'entiers vide

  for (int i = 0; i < 10 ; i++)
    v.push_back(i*i);
  vector<int>::iterator i;
  for (i = v.begin(); i != v.end(); i++)
    cout << *i <<' ';
  cout << endl;
}
```

Résultat :

```
0 1 4 9 16 25 36 49 64 81
```

La ligne

```
vector<int>::iterator i;
```

définit un itérateur dans la classe des vecteurs d'entiers.

Les fonctions `v.begin()` et `v.end()` retournent respectivement la première "position" et la position après la dernière dans `v`. L'écriture `*i` accède à l'élément en position `i`.

Utilisation d'un ensemble ordonné

```
#include<iostream.h>
#include<set.h>

main() {
  typedef set<int, less<int> > EnsOrd;
  EnsOrd s;

  for (int j = 6; j >=0 ; j--)
    s.insert(j*j);
  for (int j = 6; j >=0 ; j--)
    s.insert(j*j*j);
  EnsOrd::iterator i;
  for (i = s.begin(); i != s.end(); i++)
    cout << *i <<' ';
  cout << endl;
}
```

Résultat:

```
0 1 4 8 9 16 25 27 36 64 125 216
```

Il est utile d'introduire des **`typedef`**.

Noter que le parcours d'un ensemble ordonné fournit les éléments dans l'ordre et sans répétition.

`less<int>` est une classe qui fournit un comparateur d'entiers.

Utilisation d'un algorithme

```
#include<iostream.h>
#include<vector.h>
#include<algorithm>

main() {
  vector<int> v;

  for (int j = 6; j >=0 ; j--)
    v.push_back(j*j);
  for (int j = 6; j >=0 ; j--)
    v.push_back(j*j*j);
  for (int j = 0 ; j < 14 ; j++)
    cout << v[j] << ' ';
  cout << endl;
  sort(v.begin(), v.end());  // tri
  vector<int>::iterator i;
  for (i = v.begin(); i != v.end(); i++)
    cout << *i <<' ';
  cout << endl;
}
```

Résultat:

```
36 25 16 9 4 1 0 216 125 64 27 8 1 0
0 0 1 1 4 8 9 16 25 27 36 64 125 216
```

Conteneurs

Trois catégories de conteneurs

- séquentiels : accès linéaire
- associatifs : accès par clés
- adaptateurs : accès réglementé

L'ensemble des conteneurs est partagé en

- First class collection
  - séquentiels
    - `vector` tableaux
    - `deque` file bilatère
    - `list` liste doublement chaînée
  - associatifs
    - `set` ensemble sans répétition
    - `multiset` ensemble avec répétition
    - `map` ensemble sans répétition de couples (clé, valeur)
    - `multimap` ensemble avec répétition de couples (clé, valeur)
- Adaptateurs
  - `stack` pile
  - `queue` file
  - `priority_queue` file de priorité

Quelques exigences

Une classe concrète dont les instances sont stockées dans un conteneur STL doit posséder

- un constructeur par défaut,
- un constructeur de copie,
- un opérateur d'affectation (`operator=`).

Si l'on se sert de comparaison, elle doit posséder

- un opérateur de comparaison (`operator<`)
- un opérateur de test d'égalité, et de différence (`operator==`)

Méthodes communes

Toute collection STL possède 10 méthodes, de même nom dans chaque collection, et du même effet. Ce sont

Constructeur par défaut	comme `vector()`
Constructeur de copie	comme `vector()`
Destructeur	comme `~vector()`
`empty()`	vrai si vide
`size()`	taille actuelle
`max_size`	taille maximale
`=`	`operator=()` d'affectation
`==`	test d'égalité
`<`	comparateur
`swap`	échange de contenu

Exemple:

```
#include<iostream.h>
#include<vector.h>

main() {
  vector<int> v;        // vecteur d'entiers vide
  cout << "Vide =  " << v.empty() << endl;
  cout << "Taille de v = " << v.size() << endl;
  cout << "Taille maximale =  " << v.max_size() << endl;
  v.push_back(42);
  cout << "Taille de v = " << v.size() << endl;
  cout << "v[" << 0 << "] = " << v[0] << endl;
}
```

Résultat:

```
Vide =  1
Taille de v = 0
Taille maximale =  1073741823
Taille de v = 1
v[0] = 42
```

Exemple:

```
#include<iostream.h>
#include<vector.h>

void print(char* nom, vector<double>& w) {
  cout << nom << " = ";
  for (int i = 0; i < w.size(); i++)
    cout << w[i] << ' ';
  cout << endl;
}

main() {
  vector<double> v, z;
  v.push_back(32.1); v.push_back(40.5);
  print ("v",v);
  z.push_back(98.7); print("z",z);
  z.swap(v);        // echange de contenus
  print ("v",v); print("z",z);
  v = z;            // transfert
  print ("v",v);
}
```

Résultat:

```
v = 32.1 40.5
z = 98.7
v = 98.7
z = 32.1 40.5
v = 32.1 40.5
```

Exemple:

```
#include<iostream.h>
#include<vector.h>

template <class T>
void print(char* nom, vector<T>& w) {
  cout << nom << " = ";
  for (int i = 0; i < w.size(); i++)
    cout << w[i] << ' ';
  cout << endl;
}

main() {
  vector<char> v;
  v.push_back('h'); v.push_back('i');
  print ("v",v);
  vector<char> z(v); // copie
  z[1] = 'o';
  print("z",z);
  cout << "(v == z) = " << (v == z) << endl;
  cout << "(v < z) = " << (v < z) << endl;
}
```

Résultat:

```
v = h i
z = h o
(v == z) = 0
(v < z) = 1
```

Algorithmes

La bibliothèque STL contient, en plus des conteneurs, 70 algorithmes. Certains sont plutôt élémentaires, d'autres plus élaborés.

for_each

L'algorithme

```
for_each(begin, end, fonction)
```

applique la **`fonction`** à tous les éléments de l'intervalle commençant en **`begin`** et finissant juste avant **`end`**.

```
#include <iostream.h>
#include <vector>
#include <algorithm>
int nombres[] = {1,3,5,7,9};
void cube(int i) { cout << i*i*i<<" "; }

main() {
  vector<int> v(nombres, nombres+5);
  for_each(v.begin(), v.end(), cube);
  cout << endl;
}
```

Le programme affiche

```
1 27 125 343 729
```

Noter:

- On peut initialiser un vecteur avec un intervalle de données d'un tableau.
- La fonction `cube` donnée en paramètre est appliquée à chaque élément du vecteur.

Les fonctions qui peuvent être appliquées sont générales. Plus précisément, la signature de `for_each` est

```
template<class It, class F>
F for_each ( It d, It f, F doit )
```

où

`d` et `f` sont des itérateurs d'entrée,

`doit` est une fonction, soit parce que c'est une fonction ordinaire, soit un objet fonction.

Objets fonctions

Un *objet-fonction* est un objet qui est en même temps une fonction. Cette propriété est obtenue en surchargeant l'opérateur `()` d'application de fonctions.

L'opérateur d'application de fonction peut être surchargé dans toute classe, comme fonction membre d'une classe seulement .

Pour un objet `f`,

```
f.operateur()(x,y,z,...)
```

peut s'écrire

```
f(x,y,z,..)
```

Le *nom* de l'objet tient donc lieu du du nom de fonction.

Le prototype est

```
type operator()(parametres) {...}
```

Un premier exemple

Une classe `Cube`:

```
class Cube {
 public:
  void operator()(int i) { cout << i*i*i<<' '; }
};
```

Un objet

```
Cube doit;
```

est aussi une fonction. L'effet de

```
doit(4);
```

est le même que le `cube(4)` précédent, car `doit(4)` se développe en `doit.operator(4)`.

On peut donc remplacer

```
for_each(v.begin(), v.end(), cube);
```

par

```
Cube doit;
for_each(v.begin(), v.end(), doit);
```

ou encore par

```
for_each(v.begin(), v.end(), Cube());
```

en utilisant le constructeur par défaut `Cube()` pour fournir un objet.

Un deuxième exemple

Voici une classe qui fournit des objets multiplieurs:

```
class Multiplieur {
  int facteur;
 public:
  Multiplieur(int f = 1) : facteur(f) {}
  void operator()(int& x) { x *= facteur; }
};
```

En déclarant

```
Multiplieur parcinq = Multiplieur(5);
```

L'objet fonction **`parcinq`** multiple par `5` l'entier qui lui est passé en référence. Si `x` est un entier qui vaut `12`, après

```
parcinq(x)
```

`x` contient la valeur `60`.

Si `v` est un vecteur d'entiers comme avant, la séquence

```
Multiplieur parcinq = Multiplieur(5);
Multiplieur pardeux(2);

for_each(v.begin(), v.end(), parcinq);
for_each(v.begin(), v.end(), pardeux);
for_each(v.begin(), v.end(), Multiplieur(10));
```

a pour effet de multiplier par `100` chaque élément du tableau.

Un troisième exemple

Une classe générique de comparateurs fournit des objets fonctions capables de comparer, paramétrés par le type des objets à comparer.

```
template<class T>
class Comparator {
public:
  bool operator()(const T& i, const T& j) const {
    return (i<j)? 1:0;
  }
};
```

Elle repose sur la surcharge de l'opérateur de comparaison `<` dans la classe concrète prise pour `T`.

Voici une classe de points avec trois opérateurs.

```
class Point {
  int x, y;
public:
  bool operator==(Point q) {
    return (x==q.x) && (y == q.y);
  }
  bool operator!=(Point q) { return !operator==(q);}
  bool operator<(Point q) {
    return (x < q.x) || ((x == q.x) &&(y < q.y));
  }
  Point(int x, int y): x(x), y(y){}
};
```

On peut désormais construire des objets fonction de la classe concrète **`Comparator<Point>`**. En voici l'usage:

```
Comparator<Point> pointless; // de points
Point p(3,4), q(4,5);
cout << (p < q) ;
cout << pointless(p,q) ;
cout << Comparator<Point>()(p,q) ;
```

Les trois évaluations donnent le même résultat.

La classe patron STL prédéfinie **`less`** est bâtie sur ce schéma. Ainsi, **`less<int>`** est la classe concrète obtenue en instanciant la classe paramètre en **`int`**, et **`less<int>()`** fournit un objet fonction de cette classe.

Autres algorithmes

count

L'algorithme

```
count(begin, end, valeur, resultat);
```

compte le nombre d'éléments de l'intervalle commençant en **begin** et finissant juste avant **end** qui sont égaux à **valeur**. Il incrémente la référence **resultat** de cette valeur.

```
#include <iostream.h>
#include <algorithm>
int nombres[] = {1,5,3,5,5,7,5,9};
main() {
  int n = 8;
  count(nombres, nombres+8, 5, n);
  cout <<"n = " << n<< endl;
}
```

affiche `n = 12` (8+4).

count_if

L'algorithme

```
count_if(begin, end, predicat, resultat);
```

compte le nombre d'éléments de l'intervalle commençant en **begin** et finissant juste avant **end** pour lesquels **predicat** est vrai. Il incrémente la référence **resultat** de cette valeur.

```
#include <iostream.h>
#include <math.h>
#include <algorithm>
#define N 100000

bool estPremier(int n) {
  int r = (int) sqrt(n);
  for (int i = 2; i <= r; i ++)
    if ( (n % i) == 0) return false;
  return true;
}
int a[N];
main() {
  for (int i = 2; i < N; i++) a[i] = i;
  int n = 0;
  count_if(a+2, a+N, estPremier, n);
  cout << "Il y a " << n
    << " nombres premiers entre 2 et "<< N << endl;
}
```

affiche par exemple

```
Il y a 9592 nombres premiers entre 2 et 100000
```

generate

L'algorithme

```
generate(begin, end, fonction);
```

affecte aux éléments de l'intervalle commençant en **begin** et finissant juste avant **end** les valeurs successives retournées par **fonction**.

```
#include <iostream.h>
#include <stdlib.h>
#include <algorithm>
#define N 20

int mrand() { return rand()% N; }
int entiers[N];
main() {
  generate(entiers, entiers+N, mrand);
  for (int i = 0; i < N; i++)
    cout << entiers[i] << " ";
  cout << endl;
}
```

affiche

```
9 4 1 18 17 5 2 13 3 8 1 5 7 6 17 5 3 12 14 16
```

Un autre exemple: génération de nombres de Fibonacci

La classe suivante

```
class Fibon {
  int a, b;
public:
  Fibon() : a(0),b(1){}
  int operator()() {
    b= a+b; a = b-a;
    return a;
  }
};
```

définit des objets fonction qui retournent, à chaque appel, le nombre de Fibonacci suivant. On s'en sert dans

```
#include <iostream.h>
#include <algorithm>
#include <vector>
#define N 20

main() {
  vector<int> a(N);
  Fibon gen;
  generate(a.begin(), a.end(), gen);
  for (int i = 0; i < N; i++)
    cout << a[i] << " ";
  cout << endl;
}
```

avec, pour résultat

```
1 1 2 3 5 8 13 21 34 55 89 144 233
377 610 987 1597 2584 4181 6765
```

Itérateurs

Un *itérateur* est un moyen de parcours, comme dans

```
for (i = 0; i < N; i++) cout << t[i];
```

ou, de manière équivalente sur une liste

```
for (p = tete; p != NULL; p = p->suiv)
  cout << p -> nom;
```

Les itérateurs forment une généralisation commune de ces mécanismes.

- Chaque classe de conteneur définit sa classe **`iterator`**.
- La valeur d'un itérateur est une *position*.
- Les opérateurs `++` et `--` sont surchargés.
- Les opérateurs `==` et `!=` sont surchargés.
- Il existe des fonctions **`begin()`** et **`end()`** qui retournent la position du premier élément, et la position juste après le dernier élément.
- Si `i` est un itérateur, la valeur déréférencée `*i` est une référenc à l'élément du conteneur en position `i`.

Catégories d'itérateurs

La bibliothèque C++ distingue 5 catégories d'itérateurs, selon leurs fonctionnalités.

- `input_iterator` ou itérateur *en entrée* permettant de lire dans un conteneur.
- `ouput_iterator` ou itérateur *de sortie* permettant d'écrire dans un conteneur.
- `forward_iterator` et `backward_iterator` sont des itérateurs *unidirectionnels*, permettent de traverser un conteneur dans un sens seulement.
- `bidirectional_iterator` ou itérateur *bidirectionnel* permettent l'accès à un conteneur dans les deux sens.
- `random_access_iterator` ou itérateur *à accès direct* permet l'accès en temps constant à tout élément du conteneur.

Itérateurs particuliers

Ils sont aussi appelés des *adaptateurs d'itérateurs*. Ce sont

- les itérateurs *rétrogrades* qui parcourent en sens inverse;
- les itérateurs d'*insertion*, qui préalablement à l'insertion alloue la place où se refa l'écriture.
- les itérateurs d'*entrée-sortie*, qui permettent de considérer les flots comme des conteneurs.

Itérateurs rétrogrades

Un itérateur rétrograde parcourt un conteneur de la fin vers le début, au moyen de deux fonctions particulières que sont **rbegin()** et **rend()**.

```
int nombres[] = {1,5,3,5,5,7,5,9};

main() {
  vector<int> v(nombres, nombres+8);
  vector<int>::reverse_iterator i:
  for (i=v.rbegin(); i!=v.rend(); i++)
    cout << *i << " ";
}
```

affiche

```
9 5 7 5 5 3 5 1
```

Itérateurs d'insertion

Tout itérateur de sortie permet d'écrire dans un conteneur.

```
vector<int> v(10), w;
copy(v.begin(), v.end(), w.begin())
```

recopie les `10` éléments de `v` dans `w`. Mais comme la taille de `w` est nulle, il y a

```
Segmentation fault (core dumped)
```

Un itérateur d'insertion réserve la place avant insertion.

Trois classes d'itérateurs d'insertion sont définies:

- `insert_iterator` insère un élément juste avant la position qu'il désigne.

  ```
  typedef vector<int> Vec;
  Vec w;
  Vec::iterator i= w.begin();
  insert_iterator<Vec> j(w,i + 2);
  ```

  l'initialisation se fait avec le conteneur et la valeur. Ainsi

  ```
  copy(v.begin(), v.end(), j);
  ```

  insère les éléments de `v` dans `w` entre son 2ième et son 3ième élément.

- `back_insert_iterator` insère à la fin du conteneur récepteur;

- `front_insert_iterator` insère au début du conteneur récepteur.

Itérateurs d'entrée-sortie

Deux classes patrons sont définies

- **`istream_iterator`** avec les constructeurs

  ```
  istream_iterator()
  istream_iterator(istream& flot)
  ```

- **`ostream_iterator`** avec les constructeurs

  ```
  ostream_iterator(ostream& flot)
  ostream_iterator(ostream& flot, char* separation)
  ```

Pour les *itérateurs d'entrée*, le premier constructeur donne un itérateur "fin de fichier", le deuxième permet de lire dans un flot.

```
istream_iterator<int> fin;
istream_iterator<int> i(cin);
while( i != fin) {...}
```

Pour les *itérateurs de sortie*, la chaîne de séparation est insérée après la sortie de chaque élément:

```
ostream_iterator<int> show(cout, " : ");
ostream_iterator<int> doit(cout, "-");
copy(a.begin(), a.end(), show);
copy(a.begin(), a.end(), doit);
```

donne

```
1 : 1 : 2 : 3 : 5 : 8 : 13 :
1-1-2-3-5-8-13-
```