Programmation objet

Table des sections

6. Copies, affectations
   (a) Constructeurs
   (b) Constructeur de copie
   (c) Affectation
7. Surcharge d'opérateurs
   (a) Principe
   (b) Opérateurs amis
   (c) Syntaxe
   (d) Opérateur d'indexation
   (e) Opérateurs logiques
   (f) Opérateur d'insertion et d'extraction
   (g) Opérateurs `++` et `--`
   (h) Itérateurs
8. Structures de données classiques
   (a) Piles
   (b) Exceptions
   (c) Files
9. Héritage
   (a) Objectif
   (b) Classe composée ou classe dérivée
   (c) Syntaxe
   (d) Accès aux données et méthodes
   (e) Classes dérivées et constructeurs
   (f) Héritage multiple
   (g) Méthodes virtuelles
   (h) Un exemple: les expressions
10. Héritage (suite)
   (a) Table de fonctions virtuelles
   (b) Listes, deuxième version
   (c) Composites
   (d) Visiteurs
11. Droits d'accès
   (a) Objectif
   (b) Les trois catégories
   (c) Fonctions amies
   (d) Accès et héritage
12. Flots
   (a) Hiérarchie de classes
   (b) Manipulateurs
   (c) Fichiers

13. Patrons
    (a) Fonctions génériques
    (b) Classes génériques
14. STL
    (a) Objectifs
    (b) Exemples
    (c) Méthodes communes

1

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 défaut
7. Arguments par référence
8. Surcharge de fonctions
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 utilisée pour une expression constante, et utilisée pour définir la taille d'un tableau (en C, il faut un `#define`)

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

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.

```
#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*
```

```
#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 = ...;
...
int q = 2*n-1;


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

Arguments par référence

En faisant suivre le type d'un paramètre de `&`, il 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;
}
```

Appel

```
float x, y;
...
echange (x, y);
```

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
Déclaration

```
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 []);

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);
}
```

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 type[n]
```

alloue la place pour n éléments de type `type` 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.

```
class Compl {
 public:
  float re, im;
  void show();
};
```

On déclare des objets de la classe par:

```
  Compl s, t;
```

On les manipule de manière usuelle:

```
  s.re = t.im;
  t.re = 7;
  s.show();
```

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

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

A l'appel `s.show()`, les champs `re` et `im` sont ceux de l'objet appelant, c'est-à-dire `s.re` et `s.im`.

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

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

On peut alors écrire

```
  Compl s, t, w;
  ...
  w = s + t;
}
```

2

INSTRUCTIONS

1. Identificateurs
2. Instructions
3. Itérations
4. Exceptions
5. Constantes

Identificateurs

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

`[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   foO    _classe   ___       // ok
$sys    if     .nom      foo^bar   // erreur
```

Identificateurs réservés :

asm	do	int	signed	unsigned
auto	double	int	sizeof	virtual
break	else	long	static	void
case	enum	new	struct	volatile
catch	extern	operator	switch	while
char	float	private	template	
class	for	protected	this	
const	friend	public	throw	
continue	goto	register	try	
default	if	return	typedef	
delete	inline	short	union	

Nouveaux par rapport à C:

asm	catch	class	delete	friend
inline	new	operator	private	protected
public	template	this	throw	try
virtual				

Instructions

Expressions

`;`
expression`;`

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`

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

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

Dans des versions plus anciennes, la portée des noms déclarés dans l'initialisateur s'étend jusqu'à la fin du bloc *englobant* la boucle.

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

Dans les versions ANSI, la portée des noms déclarés dans l'initialisateur est restreinte au bloc *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.

```
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.

EXERCICES

1. *Calculette* : écrire un programme de calculette, capable d'évaluer les quatre opérations `+`, `-`, `*`, `/`. Prévoir le cas d'une division par 0.

Calculette

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

3

REFERENCES

1. Références
2. Passage par référence
3. Retour par référence

Références

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

Une *référence* est un nouveau nom (synonyme, alias) attribué à un objet 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.

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

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

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

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

Le résultat est

```
0x01887682 0x01887682 0x0188768A
```

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.

4

SURCHARGE DE FONCTIONS

1. Surcharge
2. Résolution

Surcharge

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 opérateurs.

```
#include <iostream.h>

void affiche(int i)
  { cout << "Entier valeur " << 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(1.0);
 affiche("Bonjour, monde !", 7);
}
```

Résultat

```
Entier 3
Caractere x
Reel 1
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&` ne doivent pas être en même position:

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

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

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

- Pointeur et tableau sont identiques

```
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 entière (`char` en `int`, `float` en `double`...).
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
```

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);   // fonction 3
f(i,z);   // conversion degradante de z en float
f(z,z);   // ambiguite des degradations
```

Exemple

```
void f(int)
  { cout << "Entier\n"; }
void f(...)
  { cout << "Hello !\n"; }
```

Variables

```
int i;
float y;
char c;
double z;
```

Appels

```
f(i);   // Entier
f(c);   // Hello !
f(z);   // Hello !
f(y);   // Hello !
```

Bien entendu, la forme elliptique joue le rôle du `default` dans un aiguillage. Elle sert parfois dans le traitement des exceptions.

Exemple : la factorielle

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

Elle se programme 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 `fact(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);
}
```

5

LES CLASSES COMME STRUCTURES DE DONNEES

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

Objectif

Le premier des paradigmes de la programmation objet est l'*encapsulation*. C'est la possibilité de ne montrer de l'objet que ce qui est nécessaire à son utilisation. D'où

- simplification de l'utilisation des objets
- meilleure robustesse du programme
- simplification de la maintenance.

Elle a pour conséquence de

- rapprocher les données et leur traitement: c'est l'objet que sait le mieux comment gérer une demande,
- masquer l'implémentation.

Sont fournis à l'utilisateur:

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

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* si le choix de la fonction est déterminé par le *type* des paramètres d'appel;
- Il y a *polymorphisme* si le choix de la fonction est déterminé par la *valeur* des paramètres d'appel.

La surcharge peut être réalisée sur des fonctions globales, des méthodes, des opérateurs; le polymorphisme se met en place par l'héritage de méthodes virtuelles.

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*.

Un *objet* est 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* se réalise 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 Pt {
  float x, y;
 public:
  void init (float, float);
  void deplace (float, float);
  void affiche ();
};
```

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

Elle utilise l'*opérateur de portée* `::` pour indiquer la classe à laquelle appartient la méthode.

```
void Pt::init(float a, float b) {
  x = a; y = b ;
}

void Pt::deplace (float dx , float dy) {
  x += dx; y += dy;
}

void Pt::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 Pt::init(float a, float b) {
  this -> x = a; this -> y = b ;
}
...
```

```
void main() {
  Pt a, b;
  a.init(1.5,2.5);
  b.init(3,-5);
  a.affiche();
  b.affiche();
  a.deplace(0.1,0.1);
  a.affiche();
}
```

Résultat

```
x = 1.5 , y = 2.5
x = 3 , y = -5
x = 1.6 , y = 2.6
```

Encapsulation

Le projet est composé de trois fichiers

- Un fichier `pt.h` de déclaration de la classe `Pt`
- Un fichier `pt.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 `pt.o` ou une bibliothèque.

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

```
--------------------------
// Fichier pt.h

#ifndef PT_H
#define PT_H

class Pt {
  float x, y;
 public:
  Pt (float, float);
  void deplace (float, float);
  void affiche ();
};

#endif
--------------------------
// Fichier pt.c

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

Pt::Pt(float a, float b) {
  x = a; y = b ;
}

void Pt::deplace (float dx , float dy) {
  x += dx; y += dy;
}

void Pt::affiche() {
  cout << "x = " << x <<" , y = " << y << endl;
}
--------------------------
// Fichier main.c

#include "pt.h"

void main()
{
  Pt a(1.5,2.5), b(3,-5);
  a.affiche();
  b.affiche();
  a.deplace(0.1,0.1);
  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 `init` par un constructeur:

```
class Pt {
  float x, y;
 public:
  Pt (float, float);
  void deplace (float, float);
  void affiche ();
};

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

Le constructeur `Pt` rend inopérant le constructeur par défaut s'il n'est pas redéfini.
Le constructeur est employé en passant les arguments en paramètre.
Variations

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

```
class Pt {
  float x, y;
 public:
  Pt (float a, float 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

```
class Pt {
 public:
  float x, y;
  Pt () {};
  Pt (float a, float b) : x(a), y(b) {}
  ...
};
```

Voici des exemples d'emploi de cette classe

1.- Les constructeurs:

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

2.- Tableaux d'objets:

```
main () {
  const int taille = 7;
  Pt* a = new Pt[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 se fait par `delete []` qui

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

Voici une petite classe. La mention explicite du constructeur par défaut et du destructeur signifie leur redéfinition.

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

Le constructeur et le destructeur sont définis comme suit:

```
Id::Id() {
  char t[20];
  cout << "Entrez votre nom : ";
  cin >> t;
  nom = new char[1+strlen(t)];
  strcpy(nom,t);
}

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

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
```

```
class Compl {
  float re, im;
  Compl() { re = im = 0;}
}
```

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

```
Compl s, t;
```

définit deux complexes `s`, `t` initialisés comme indiqué.
Un constructeur (de conversion) est utile pour convertir un couple de `float` en complexes:

```
Compl::Compl(float x, float y) {re = x; im = y;}
```

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

```
Compl::Compl(float x) {re = x; im = 0;}
```

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

```
class Compl {
  float re, im;
  Compl() { re = im = 0;}
  Compl(float x, float y) {re = x; im = y;}
  Compl(float 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 Compl {
  float re, im;
  Compl(float r = 0, float i = 0) : re(r), im(i) {}
  float module() { return sqrt(re*re + im*im); }
}
```

On s'en sert

- à l'initialisation, par exemple

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

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

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

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

- par appel explicite dans une instruction

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

Une classe de rationnels

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

Dans notre implémentation, 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) {
  return a = Rat(a.num*b.den + a.den*b.num, a.den*b.den);
}
```

L'un des appels

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

- construit un rationnel temporaire,
- le copie dans `a` passé par référence;
- en transmet la référence en sortie.

Mais 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 (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 ! De même pour l'affichage:

```
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;
}
```

et le résultat est:

```
B(0) = 1/1
B(1) = -1/2
B(2) = 1/6
B(3) = 0/1
B(4) = -1/30
B(5) = 0/1
B(6) = 1/42
...
```

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 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 peut modifier (quand c'est permis) la valeur d'une donnée membre par une fonction de modification (*setter function*);

Cette discipline s'étend au nom des fonctions. Une proposition:

- une donnée membre (privée) finit par le caractère de soulignement: `xyz_`.
- la "getter function" a le nom `getxyz()`;
- la "setter function" a le nom `setxyz()`.

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;}
};
```

6

COPIE ET AFFECTATION

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

Constructeurs

Les constructeurs se classent syntactiquement en 4 catégories

1. le constructeur *par défaut*, sans argument,
2. les constructeurs *de conversion*, à un argument
3. le constructeur *de copie*,
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 bit à bit une zone de mémoire dans une autre (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;  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 possédant:

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

```
#include <iostream.h>

class Pt {
  static char u;
  char no;
  int x, y;
public:
  Pt(int abs = 1, int ord = 0) {
    x = abs; y = ord;
    no = u++;
    cout <<"Construction de "<< no <<" : "<< x <<" "<< y << endl;
  }

  Pt(const Pt& p) {
    x = p.x; y = p.y;  no = u++;
    cout << "Recopie      de " << p.no << " en " << no << endl;
  }
  ~Pt() {
    cout << "Destruction  de " << no << endl;
  }
  Pt& operator=(const Pt& p) {
    x = p.x; y = p.y;
    cout << "Affectation  de " <<p.no << " a " << no << endl;
    return *this;
  }
};

char Pt::u= 'a';
```

Voici un programme utilisant cette classe

```
void Fn(Pt, Pt&, Pt*);

void main()
{
  cout << "entree\n";
  Pt a;
  Pt b = 3;
  Pt c = Pt(0,2);
  {
    Pt d = Pt(a);
    Pt e = a;
  }
  Pt f;
  Pt* p = new Pt(6,7);
  Fn(a, b, p);
  c = Pt(5,5);
  c = f;
  Pt j = 0;
  cout << "sortie\n";
}

void Fn(Pt p, Pt& q, Pt *r) {
  cout << "entree f\n";
  delete r;
  cout << "sortie f\n";
}
```

```
entree
Construction de a : 1 0        //  Pt a;
Construction de b : 3 0        //  Pt b = 3;
Construction de c : 0 2        //  Pt c = Pt(0,2);
Recopie      de a en d         //  Pt d = Pt(a);
Recopie      de a en e         //  Pt e = a;
Destruction  de e              //  sortie du bloc
Destruction  de d
Construction de f : 1 0        //  Pt f;
Construction de g : 6 7        //  Pt* p = new Pt(6,7);
Recopie      de a en h         //  Fn(a, b, p);
entree f
Destruction  de g              //  delete r;
sortie f
Destruction  de h              //  de la copie de a
Construction de i : 5 5        //  Pt(5,5); temporaire
Affectation  de i a c
Destruction  de i              //  du temporaire
Affectation  de f a c
Construction de j : 0 0
sortie
Destruction  de j              //  dans l'ordre inverse
Destruction  de f              //  de leur creation
Destruction  de c
Destruction  de b
Destruction  de a
```

Affectation

Il ne faut pas confondre copie et affectation.

- le constructeur de copie sert à l'initialisation et pour le passage de paramètres. Il retourne l'objet.
- l'opérateur d'affectation `=` pour l'affectation dans une expression. Il retourne 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, ce qui pose les mêmes problèmes que pour la copie. La signature de l'opérateur est

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

et l'expression `y = x` équivaut à

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

Le résultat de l'affectation est donc dans l'objet appelant.

Voici une petite classe `Id`:

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

Pour illustrer le rôle de la copie, voici quatre fonctions

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

Les résultats des quatre appels sont

```
y=vv(x);  // deux copies, une destruction
y=vr(x);  // une copie
y=rv(x);  // une copie, une destruction, Argh!
y=rr(x);  // pas de copie
```

Dans le troisième appel, on détruit la variable créée sur la pile pour conserver la copie, après avoir passé son nom en référence ! Un “bon” compilateur émet au moins un avertissement.

Conversions

Pour les conversions définies par le programmeur, il y a deux possibilités:

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

Un *opérateur de conversion* d'une classe `C`

- est une fonction membre de `C`;
- a le prototype

  ```
  operator T()
  ```

  où `T` est le type cible (classe ou type de base.
- 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 int() { return (r.num) ? 1 : 0;}
}
```

7

SURCHARGE D'OPERATEURS

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

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* si le choix de la fonction est déterminé par le *type* des paramètres d'appel;
- Il y a *polymorphisme* si le choix de la fonction est déterminé par la *valeur* des paramètres d'appel.

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
```

mais il ne peuvent être surchargés.

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 Num() {return num;}
          int Den() {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.Num()*b.Den() + a.Den()*b.Num();
  return Rat(n, a.Den()*b.Den());
}
```

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);
}
```

Pour cela, 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);
}
```

Pour

```
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
```

Mais

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

car vaut

```
1.operator-(b)
```

qui n'est pas défini.

Opérateur d'indexation

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

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

Ainsi

```
Vec 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;
  Vec u(N, 7);
  Vec 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& o , Vec& u) {
  o << "(";
  for (int i=0; i<u.taille();i++) o << u[i] << " ";
  return o << ")" << endl;
};
istream& operator>>(istream& I , Vec& u) {
  for (int i=0; i<u.taille();i++) I >> u[i];
  return I;
};
```

On obtient alors les sorties

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

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.

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

```
#include <iostream.h>

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

On s'en sert traditionnellement par:

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

  P.push("Hello world !");
  P.push("Bonjour, monde !");
  y = P.pop(); x = P.pop();
}
```

On définit 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;
}
```

et on peut écrire:

```
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);`

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(++num,++den);}
  Rat operator++(int z) { return Rat(num++,den++);}
};
```

permet d'écrire:

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

Itérateurs

Le *parcours* d'un conteneur se formalise par l'introduction de classes spécifiques, appelées *Enumération* en Java et *Itérateurs* en C++.
Nous allons implémenter d'abord les Enumérations.

Un *énumérateur* est attaché à une classe (dans notre cas, des vecteurs de flottants). Il a deux méthodes

- `int hasMoreElements()` qui retourne vrai s'il reste des éléments à énumérer
- `float& nextElement()` qui fournit l'élément suivant

et bien sûr un constructeur. Voici la déclaration:

```
class Enumerator {
  Vec* v;       // objet e'nume're'
  int courant;  // indice interne
public:
  int hasMoreElements();
  float& nextElement();
  Enumerator(Vec* vv) :v(vv),courant(0){}
};
```

Si l'on dispose d'un énumérateur `j`, on peut écrire

```
while (j.hasMoreElements())
  cout << j.nextElement()<<' ';
```

La classe `Vec`

```
class Vec {
protected:
  float *a;
  int n;
public:
  int taille() const {return n;}
  Vec(int t = 10);    // constructeur
  Vec(int t, float v) // constructeur
  float& operator[](int i) {return a[i];}
  Enumerator elements() {return Enumerator(this);}
};
```

La méthode `Enumerator elements()` retourne un énumérateur associé (c'est la terminologie Java ...). On s'en sert dans

```
void main()
{
  Vec x(5,5.0);
  Enumerator j = x.elements();

  while (j.hasMoreElements())
    cout << j.nextElement()++ <<' ';
  cout << endl;

  for (Enumerator i = x.elements();i.hasMoreElements();)
    cout << i.nextElement() <<' ';
  cout << endl;
}
```

et le résultat

```
5 5 5 5 5
6 6 6 6 6
```

Les deux méthodes manquantes s'écrivent

```
int Enumerator::hasMoreElements() {
  return (courant != v->taille());
}
int& Enumerator::nextElement() {
  return (*v)[courant++];
}
```

Les *itérateurs* sont d'une construction semblable, mais sont plus élaborés.

- La classe sur laquelle on itère fournit des itérateurs par deux méthodes:
  - `Iterator begin()` pour démarrer;
  - `Iterator end()` pour indiquer le dépassement.
- La classe d'itérateurs fournit
  - la méthode d'avancement `Iterator& next()`
  - la méthode d'accès à l'élément courant `float& valeur()`

On s'en sert par exemple dans:

```
Iterator i;
for (i= x.begin(); i!=x.end(); i=i.next())
  cout <<  i.valeur() <<' ';
  cout << endl;
```

Cette écriture oblige à surcharger l'opérateur `!=` pour les itérateurs. Dans notre exemple, on a

```
class Vec {
  float *a;
  int n;
  Iterator debut;
  Iterator fin;
public:
  int taille() const {return n;}
  ...
  Iterator& begin()  { return debut; }
  Iterator& end() { return fin; }
};
```

et la classe d'iterateur

```
class Iterator {
  Vec* v;       // objet e'nume're'
  int courant;  // indice interne
public:
  Iterator(){};
  Iterator(int c,Vec* vv) : courant(c), v(vv){};
  Iterator& next() { ++courant; return *this;}
  int operator==(const Iterator& j);
  int operator!=(const Iterator& j);
  float& valeur();
};
```

avec les définitions que voici

```
int Iterator::operator==(const Iterator& j) {
  return (courant == j.courant);
}
int Iterator::operator!=(const Iterator& j) {
  return !(*this == j);
}
float& Iterator::valeur() {
  return (*v)[courant];
}
```

Il est très tentant

- de remplacer `next()` par un opérateur d'incrémentation,
- de remplacer `valeur()` par un opérateur de déréférencement.

```
Iterator& Iterator::operator++() {
  ++courant; return *this;
}
float& Iterator::operator*() {
  return valeur();
}
```

On s'en sert dans

```
for (Iterator i= x.begin(); i!=x.end(); ++i)
  cout << *i <<' ';
cout << endl;
```

Les constructeurs de la classe `Vec` doivent initialiser les itérateurs membres `debut` et `fin`.

```
Vec::Vec(int t = 10) : debut(0,this), fin(t,this){
    a = new float[n = t];
  }

Vec::Vec(int t, float v): debut(0,this), fin(t,this){
    a = new float[n = t];
    for (int i=0; i<n; i++) a[i] = v;
  }
```

Une implémentation générique (indépendante des types actuels) est donnée dans la STL (Standard Template Library).

8

STRUCTURES DE DONNEES

1. Piles
2. Exceptions
3. Files

Piles

L'implémentation d'une pile est cachée. Les opérations sont

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

Ici, les piles sont *exogènes* : seul un pointeur est stocké, et il n'y a pas de copie.

```
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 naturellement en surchargeant les opérateurs de flots `<<` et `>>` .
D'après les recommandations faites pour la surcharge, 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;
}
```

avec le résultat

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

- 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 par `g++ -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";}
  }
}
```

Implémentation des piles par liste chaînée.

On utilise une classe interne `Cellule`.

```
#include <iostream.h>

class Pile {
  public :
    Pile() { p = 0; }
    void push(char* x) { p = new Cellule(x, p); }
    char* top() { return p -> cont; }
    char* pop();
  private :
    struct Cellule {
      Cellule* suiv;
      char* cont;
      Cellule(char* c, Cellule* s) :
        cont(c), suiv(s){}
    };
    Cellule * p;
};

char* Pile::pop() {
  Cellule* q = p;
  char* c = q -> cont;
  p = p -> suiv;
  delete q;
  return c;
}
```

EXERCICES

1. *Polonaise postfixée* : écrire un programme qui lit une expression postfixée sur une ligne, et en affiche la valeur.

Polonaise postfixée

```
#include <ctype.h>
#include <iostream.h>
#include <iomanip.h>

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

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

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

int Pile::top() {
  return p[sp];
}

int Pile::pop() {
  return p[sp--];
}

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

int getlex(char& x, int& i) {
  while (cin.get(x)) {
    if (x == '\n')
      return -1;
```

```
    if (x != ' ')
      break;
  }
  if (('0' <= x) && (x <= '9')) {
    cin.putback(x);
    cin >> i;
    cout << "Je lis un entier " << i << endl;
    return 0;
  }
  cout << "Je lis un char " << x << endl;
  return 1;
}

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;
}

void main()
{
  Pile p;
  char x;
  int i;
  for (;;) {
    switch (getlex(x,i)) {
      case -1:
        return;
      case 0:
        p.push(i);
        p.imprimer();
        cout << endl;
        break;
      case 1:
        int a = p.pop();
        int b = p.pop();
        int c = apply(op, b, a);
        p.push(c);
        p.imprimer();
        cout << endl;
    }
  }
}
```

9

HERITAGE

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. Méthodes virtuelles
8. Un exemple: les expressions

Objectif

L'*héritage* est le deuxième des paradigmes de la programmation objet. Il a pour but de "placer les problèmes à leur juste niveau".

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.

Une classe dérivée modélise un *cas particulier* de la classe de base, et est donc enrichie d'informations supplémentaires.

L'emploi de l'héritage conduit à un style de programmation par raffinements successifs et permet une programmation incrémentale effective.

Dans la conception, la tâche principale est la *réification*: opération qui transforme en "chose" (Petit Robert), c'est-à-dire la définition de la hiérarchie de classes.

La *délégation* fait usage du *polymorphisme* réalisé en C++ par les méthodes virtuelles.

Classe composée ou classe dérivée

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

```
class Habitant {
protected:
  char* nom;
  char* adresse; //domicile
};

classe Carte {
  Habitant h;
  int age;
  float age;
};
```

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

```
class Resident : public Habitant{
protected:
  char* adresse; //residence
};
```

Un résident est un *cas particulier* 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`.

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

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

- Un cercle coloré est un cas particulier d'un cercle, doté en plus d'une couleur, donc

```
class CercleCol : public Cercle {
  Color couleur; // de la classe Color
  ...
}
```

- Une forme contient un rectangle englobant:

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

- Un triangle est une forme particulière:

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

Accès aux données et méthodes

Les champs 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 Habitant {
protected:
  char* nom;
  char* adresse; //domicile
};

class Resident : public Habitant{
protected:
  char* adresse; //residence
};
```

Dans

```
main() {
  Resident r;
  ...
  cout << r.adresse << r.Habitant::adresse;
  ...
}
```

le premier désigne l'adresse de résidence, le deuxième l'adresse du domicile. Les mêmes règles s'appliquent aux méthodes.

Dans

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

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

Pour un objet `d` de la classe `D`,

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

désigne la méthode de `D::f(int)`. L'appel

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

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

```
d.B::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:
    char nom[20];
  public:
    Animal(char* n) {strcpy(nom,n);}
    void show() {cout << nom <<endl;}
};

class Ours: public Animal {
    char couleur[20];
  public:
    Ours(char*n, char*c) : Animal(n) {strcpy(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. 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 {...}
```

Ce qui a été dit précédemment s'applique, avec quelques nuances.

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:

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

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

La ligne

```
  ((ListeEntiers *) suiv) -> show();
```

convertit le type "adresse de liste" en "adresse de liste d'entiers" pour pouvoir appeler la méthode.

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

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 la méthode appelante.

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`
- L'objet est appelé par pointeur ou par référence, pour qu'il n'y ait pas de conversion implicite.

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.

```
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
  zz = (XX) z; zz.print(); zz.show();
  zzz = (XXX) zz; zzz.print(); zzz.show();
}
```

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

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`.

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 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;
}
```

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.

```
class ExpressionR {
public:
  virtual void Print() =0;
  virtual int eval() =0;
};

typedef ExpressionR* Expression;
```

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

```
Addition
Soustraction
Multiplication
Division
Fact
Simple
```

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 Fact: public ExpressionR {
    Expression unique;
  public:
    Fact(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();
}


Fact::Fact(Expression a): unique(a){}
void Fact::print() {
  cout << " ! " ; unique -> print();
}
int Fact::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;}
```

La fonction `apply` demande la création d'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 Fact(a);
        }
        return NULL;
}
```

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 pile est une *pile d'expressions*.

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

Note : D'autres problèmes se traitent de façon semblable. Il y a toute la famille de construction et d'évaluation d'arbres. Deux exemples :

- Calcul d'un automate fini à partir d'une expression rationnelle : dans ce cas, c'est l'arbre de l'expression rationnelle qui est construit, l'évaluation consistant à construire l'automate.
- De manière plus générale, la construction d'un arbre de dérivation par analyse syntaxique.
  - Toute variable correspond à une classe;
  - Toute règle est un constructeur.

EXERCICES

1. *Polonaise postfixée* : écrire un programme qui lit une expression postfixée sur une ligne, et en affiche la valeur. On définira une classe de lexèmes, ayant deux sous-classes : entiers et opérateurs, pour une meilleure lecture.

Polonaise postfixée

```
// Polonaise avec lecture
// plus sophistiquee
//
#include <ctype.h>
#include <iostream.h>
#include <iomanip.h>
class Pile;
class Lexeme;
class Entier;
class Operateur;
////////////////////////////
// Methodes globales
//
int apply (char, int, int);
Lexeme* getLex();
void eat(Lexeme*, Pile&);
////////////////////////////
// Classe Pile
//
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;
 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;
  }
};
////////////////////////////

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() {
  for (int i= sp; i>0; i--)
    cout << p[i]<< "\t";
  cout << "@";
}
////////////////////////////
Lexeme* getLex() {
  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);
}

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;
}

void eat(Lexeme* lex, Pile& p) {
  if (lex)
    lex -> eat(p);
}
////////////////////////////

void main()
{
  Pile p;
  Lexeme* lex;
  for (;;) {
    lex = getLex();
    try {
      if (lex == 0) {
```

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

10

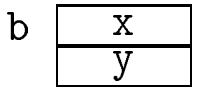
HERITAGE (suite)

1. Table de fonctions virtuelles
2. Listes, deuxième version
3. Composites
4. Visiteurs

Table de fonctions virtuelles

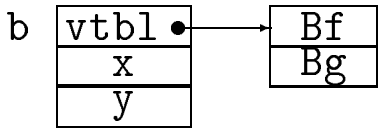
```
class Base {
  int x, y;
  int H(void);
};

Base b;
```



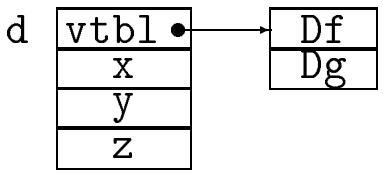
L'appel `b.H()` est remplacé par `H(b)` à la compilation.

```
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] = Bg()` 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] = Dg()` 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.

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. On s'intéresse à

- l'itération;
- le transfert des opérations des éléments sur les listes (le fameux `mapcar`);
- et comme on le verra, on obtient gratis des listes de listes.

Les constituants

Listes (exogènes) de pointeurs vers des "objets".

```
class Objet {
protected:
  int t;
public:
  Objet(int t = 0) { this -> 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.

```
class Liste {
protected:
  Maillon* prem;
  Maillon* dern;
  int 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) {
      prec = p;
      suiv = s;
      this -> d = d;
    }
   };
  Maillon* prem;
  Maillon* dern;
  int taille;
...
};
```

Le constructeur par défaut s'écrit:

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

Deux méthodes d'interrogation naturelles:

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

La manipulation

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

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;
a.show();cout << endl;
```

donne

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

donne

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

Composites

- Les listes récursives précédentes sont un cas particulier du schéma conceptuel *Composite*.
- Les *schémas conceptuels*, en anglais "design patterns" sont *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context*.

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.

Exemple: Un *élément graphique* peut être

- une ligne
- un rectangle
- un texte
- ou une figure qui groupe des éléments graphiques

Structuration: La hiérarchie des classes comporte

- une classe (abstraite) `composant` (élément graphique)
- des classes dérivées `feuilles` (ligne, rectangle, texte)
- une classe dérivée `noeud` (ou "composite", ici figure)

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

Une arborescence de cercles et de carrés

Exemple

- 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.

```
class Groupe : public Forme {
    Forme * element[12];
    int nb;
  public :
    Groupe() : nb(0) {}
    void add(Forme* f) { element[nb++]= 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 show() {
  cout << "Forme (\n";
  for (int i = 0; i < nb ; i++)
           element[i] -> show();
  cout << ")\n";
}
```

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
```

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 préfixées.

- La classe `Lexeme` a les deux classes dérivées `Entier` et `Operateur`.
- L'évaluateur est la pile, dans la boucle:

```
Pile p;
Lexeme* lex;
for (;;) {
  lex = getLex();
  if (lex == 0) { ... exit(0);}
  eat(lex,p);
}
```

- La fonction globale `eat(lex,p)` est réécrite en `lex -> eat(p)`.
- Logiquement, on aurait dû écrire `p.eat(lex)` pour deux méthodes

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

mais ceci ne se compile pas 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.

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

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

- L'évaluateur envoie le "message"

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

Si `lex` est un pointeur sur un Operateur `op`, ceci devient

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

d'où

```
p.eat(op);
```

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

Le tour est joué !

Annexe : Liste du programme composite

```
//////////////////////
//
// Composite de formes
//
#include <iostream.h>
#define decal(d) for (int i = 0; i < d ; i++) cout << ' '
//////////////////////
//
// 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() {

  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;
}
```

11

DROITS D'ACCES

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

Objectif

Les *droits d'accès* protègent les données et les méthodes, et réalisent ainsi l'encapsulation.

En général, les fonctions d'accès aux données sont publiques, et l'implantation des structures de données est cachée.

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

Autre exemple. Soit une classe

```
class Chaine {
  int lg;
  char* ch;
  ...
}
```

Problème:

- Si `lg` est privé, on ne pourra connaître la longueur,
- mais si `lg` est public, tout le monde peut changer sa valeur.

Solution:

- `lg` est un membre privé;
- on définit une fonction membre `getLg()` *publique* qui retourne la valeur de `lg`.

```
int Chaine::getLg() { return lg; }
```

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
```

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...);
}
```

`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 `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
}
```

12

Flots

1. Hiérarchie de classes
2. Manipulateurs
3. Fichiers

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 hiérarchie de classes comprend:

```
class ios
```

C'est la classe de base, contenant les fonctionnalités principales.

```
class istream : public virtual ios
```

classe gérant les entrées;

```
class ostream : public virtual ios
```

classe gérant les sorties;

```
class iostream : public istream, public ostream
```

une classe qui 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.

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

Les opérateurs d'insertion et d'extraction sont prédéfinis dans les classes `istream` et `ostream` pour les types de base, par exemple:

```
class istream : public virtual ios {
...
public :
  istream& operator>>(char *);
  istream& operator>>(char&);
  istream& operator>>(short&);
  istream& operator>>(int&);
  ...
}
```

L'opérateur retourne une référence d'un stream, et on peut donc enchaîner les ordres de lecture resp. d'écriture.

Pour une nouvelle classe `X`, surdéfinition en fonctions amies selon le canevas

```
istream& operator>>(istream& in, X& objet) {
  // Lecture objet
  return in;
```

```
}

ostream& operator<<(ostream& out, X& objet) {
  // Ecriture de l'objet
  return out;
}
```

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
  int fail() const; // la prochaine operation echouera
  int bad() const;  // le flot est endommage
  int good() const; // le flot est en bon etat
}
```

L'opération précédente a réussi si l'état est `good()` ou `eof()`. 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* de la classe `cin` retourne un entier. Cet entier est non nul si l'état 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`		ajoute `\0`
`ws`	(i)	ignore blancs en tete
`flush`	(o)	vide tampon
`setprecision(int)`	(o)	chiffres après ,
`setw(int)`	(o)	gabarit
`setfill(char)`	(o)	remplissage

Définir ses propres manipulateurs

On peut écrire

```
cout << x << flush << y << flush;
```

parce que la méthode `flush()` est déclarée

```
ostream& flush(ostream&);
```

Il suffisait donc de surcharger l'opérateur d'insertion (comme méthode de la classe `ostream`) avec

```
ostream& operator<<(ostream& (*manip)(ostream &))
  { return (*manip)(*this); }
```

comme c'est déjà fait dans `iostream.h`. On transforme ainsi une fonction membre en *manipulateur*. Noter l'apparition du schéma conceptuel *visiteur*.

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& (*F)(ostream&, int);
```

La nouvelle classe est

```
class Omanip {
private:
  F f;
  int i;
public:
  Omanip(F _f, int _i) : f(_f), i(_i) {}
  friend ostream& operator<<(ostream& os, Omanip& m)
    { return m.f(os, m.i);}
}
```

La fonction `setprecision()` est définie pour retourner un objet de la classe `Omanip` à partir d'une fonction de type `F`. On suppose donné

```
ostream& precision(ostream&, int i);
```

et on définit

```
Omanip setprecision(int i) {
  return Omanip(&precision, i);
}
```

Dans notre cas, la fonction `precision()` est définie par

```
ostream& precision(ostream&, int i) {
  os.precision(i);
  return os;
}
```

Une définition générique est contenue dans les fichiers usuels:

```
template <class T>
class OMANIP {
public:
  OMANIP(ostream &(*f)(ostream&, T), T v) :
    func(f), val(v) {}
  friend ostream &operator<<(ostream& s, OMANIP<T>& m)
    { return m.func(s, m.val); }
private:
  ostream &(*func)(ostream&, T);
  T val;
};
```

et de même pour `IMANIP` et `SMANIP`.

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);
```

13

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);
```

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; }
```

ou pire

```
template <class T, class U, class R>
R max(T a, U b) { return (a>b) ? a : b; }
```

qui équivaut presque au

```
#define max(a,b) (a>b) ? a : b
```

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);
  ...
}
```

Voici un programme rudimentaire d'évaluation d'expressions arithmétiques complètement parenthésées. Il utilise une pile d'opérandes et une pile d'opérateurs. (Observer que les parenthèses ouvrantes sont inutiles.)

```
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 ou une classe. La syntaxe est illustrée sur l'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;
```

D'autres 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.

Instanciation et *spécialisation.* Soit

```
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 instancie 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
}
```

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 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.

14

Standard Template Library

1. Objectifs
2. Exemples
3. Méthodes communes

Objectifs

La *bibliothèque générique standard* ou *STL* fournit un ensemble de structures de données et d'algorithmes génériques. La syntaxe la plus simple est

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
```

```
#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`.

```
#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.

```
#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
```