Programmation des Interfaces Graphiques en langage C++

Stéphane Derrode

École Centrale de Lyon
Bât. E6, 2eétage

stephane.derrode@ec-lyon.fr
emmanuel.dellandrea@ec-lyon.fr

30 janvier 2021

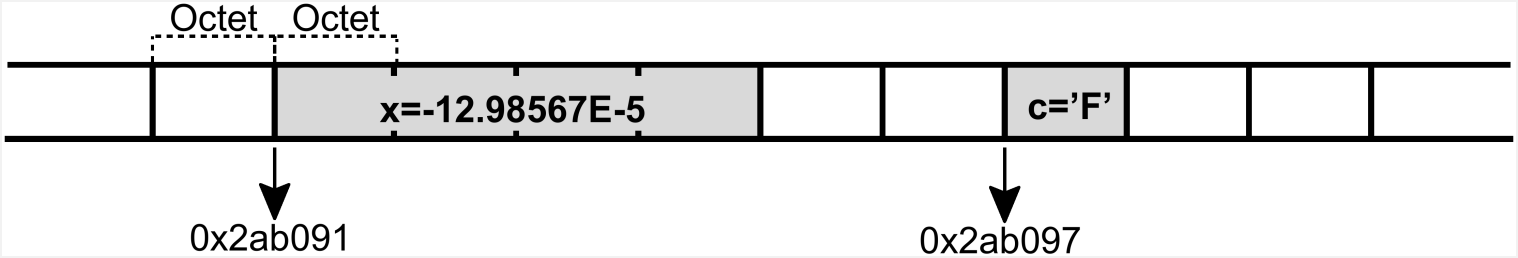
Première partie I

En guise de préliminaires

1. Pointeurs et références
   - Pointeurs
   - Références
   - Variables, pointeurs et références constants
2. Tableaux et allocation de mémoire
   - Tableaux et allocation statique de memoire
   - Allocation dynamique de mémoire
3. Fonctions : arguments, sur-définition et patrons
   - Rôle d'une fonction
   - Arguments d'une fonction

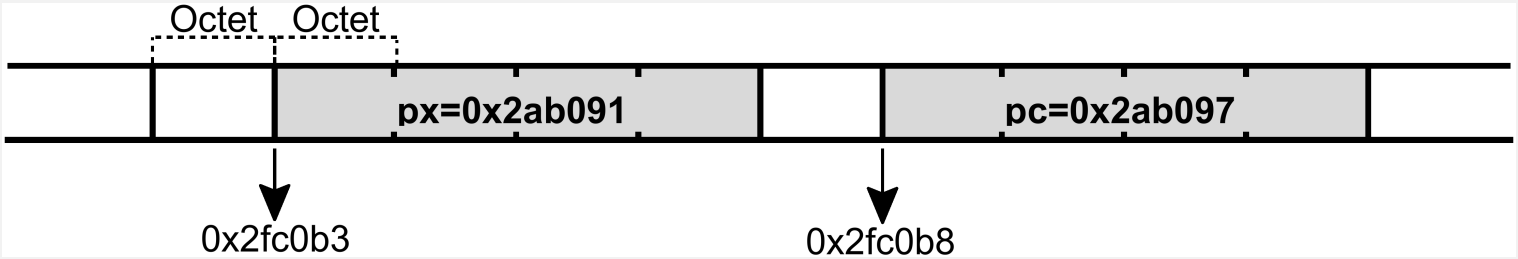
Adresses mémoire et pointeurs

- Variables : `float x=-12.98587E-5; char c='F';`



- Pointeurs :

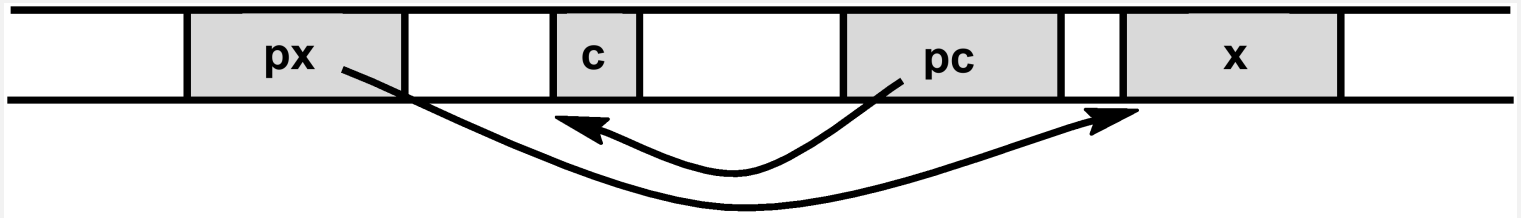
Déclaration : `float *px;` et `char *pc;`
Initialisation : `px = &x;` et `pc = &c;`



- Remarque : La taille d'un pointeur, c'est à dire la quantité de mémoire réservée au stockage d'une adresse, est de 4 octets (quelque soit le type sur lequel il pointe) :

`sizeof(char*)==sizeof(double*)==4` `sizeof(char)==1; sizeof(double)==8`

Schéma global :



Exemples de pointeurs

```
int x=1, y=2;
int *pi;        // pi pointe sur un int
pi  = &x;       // pi pointe sur x
y   = *pi;      // y reçoit la valeur pointée par pi (x)
*pi = 0;        // x vaut 0
```

Remarques :

- `&*p` est identique à `p` (pour `p` un pointeur)
- Opérateurs sur les pointeurs : `+`, `-`, `++`, `-`, `+=`, `-=`, `==`, `!=` ...
- `p=NULL` (identique à l'initialisation à `0` d'une variable entière ou flottante). `if (p==NULL) ...` est identique à `if (p) ...`

Les références (C++)

Les références sont des synonymes (ou alias) d'identificateurs. Elles permettent de manipuler une variable sous un autre nom que celui sous laquelle cette dernière a été déclarée.

Exemples de références

```
int n;
int &r = n;     // r est une référence sur n
int *p = &n;    // p est une pointeur sur n

n = 3;
cout << *p;     // Imprime 3
cout << r;      // Imprime 3
```

Impossible !

```
int &r;      // Initialisation obligatoire
int &n = 3; // Init. d'une référence à une constante
```

Intérêt principal : passage d'arguments par référence dans les fonctions - Tr. 15.

Les variables constantes : `const`

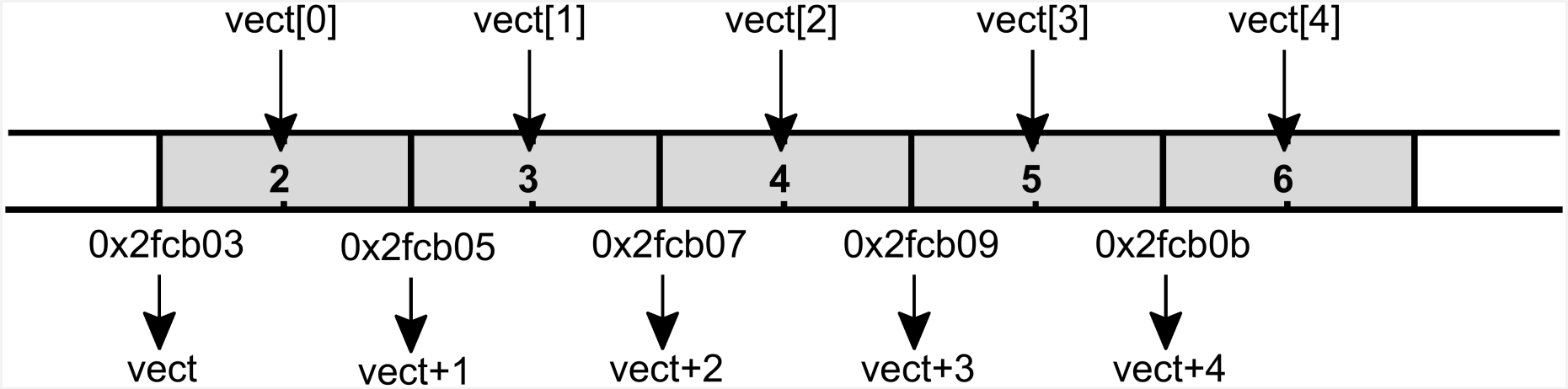
➽ Le mot clef `const` indique d'une variable qu'elle ne peut pas être modifiée une fois initialisée : `const double i = 5.5;`.

Exemples de variables `const`

```
int main() {
    double pi           = 3.14159;
    const double deuxpi = 2.0*pi;
    double un           = 1.0;

    const double* x1    = &pi;      // Ok
    const double* x2    = &deuxpi;  // Ok
    *x1   = 2.0;  // Erreur: valeur de pi non modif.
    x1    = &un;  // Ok
    *x2   = 2.0;  // Erreur: valeur de deuxpi non modif.
    x2    = &un;  // Ok

    double* const dcp1  = &pi;      // Ok
    double* const dcp2  = &deuxpi;  // Erreur
    *dcp1 = 2.0;  // Ok
    dcp1  = &un;  // Erreur: adresse de dcp1 non modif.

    return 0;
}
```

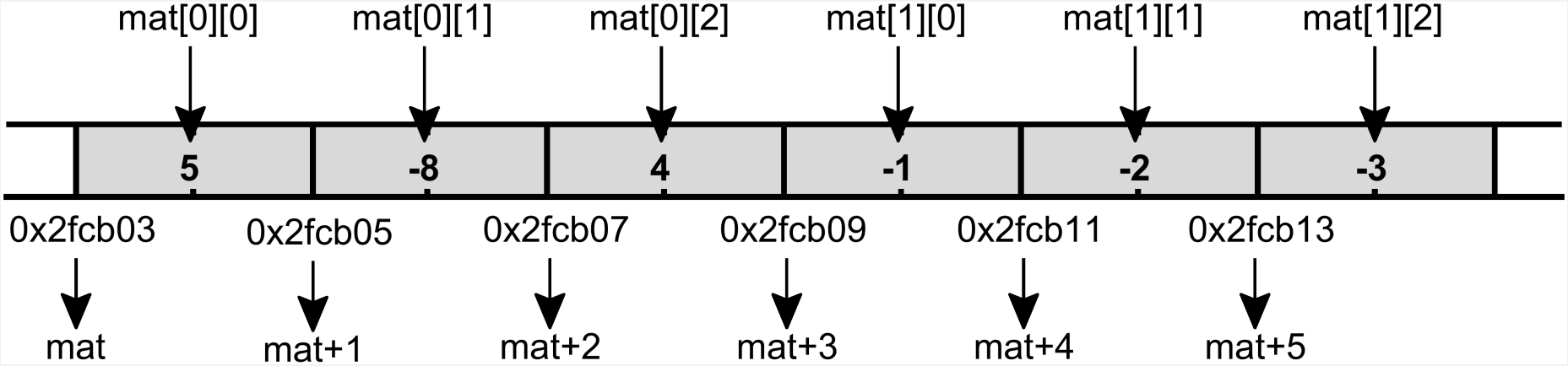
Tableaux statiques de nombres

➪ **Tableau 1D (Vecteur) :** `int vect[5] = {2, 3, 4, 5, 6};`



- `&vect[0] == vect` : Adresse du 1er élt. de `vect`.
- `&vect[i] == vect+i` : Adresse du ie élt. de `vect`.
- `vect[i] == *(vect+i)` : Valeur du ie élt. de `vect`.

⇨ Tableau 2D (Matrice) : `int mat[2][3]={{5,-8,4},{-1,-2,-3}};`



- `&mat[0][0]==mat` : Adresse du l'élt. (0,0) de `mat`.
- `&mat[i][j]== mat+i*3+j` : Adresse de l'élt. (i,j) de `mat`.
- `mat[i][j]==*(mat+i*3+j)` : Valeur de l'élt. (i,j) de `mat`.

Allocation dynamique de mémoire

- Allocation statique :
  - Pour créer et supprimer des « objets » lors de la compilation.
  - Utilise la pile (limitée en taille).
  - La quantité de mémoire à allouer doit être connue à la compilation.
- Allocation dynamique :
  - Pour créer et supprimer des « objets » lors de l'exécution.
  - Utilise le tas (limitée par la mémoire de votre PC).
  - La quantité de mémoire à allouer peut être connue à l'exécution.
- Mise en oeuvre (tableau)
  - en `C` : les fonctions `malloc()` et `free()`

```
int *t = (int*) malloc(5*sizeof(int)); ...
free(t);
```

  - en `C++` : les opérateurs `new` et `delete`

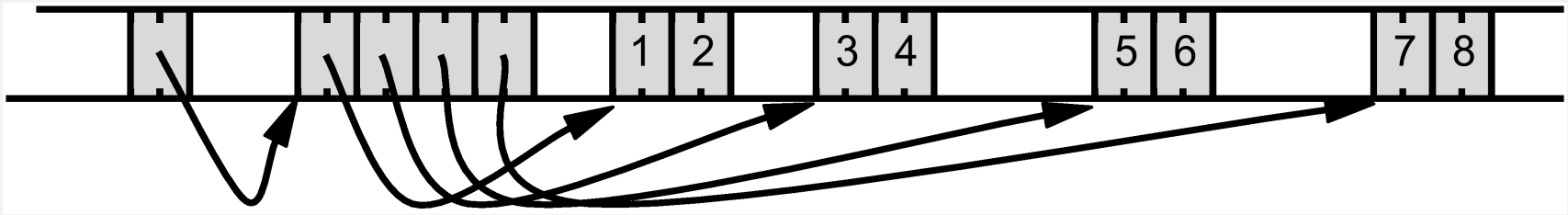
```
int *t = new int[5]; ... delete [] t;
```

Utilisation de `new` et `delete`

```
#include <iostream>
using namespace std;

int main() {
  int taille;
  cout << "Entrez la taille du vecteur : ";
  cin >> taille;
  double *tab = new double[taille];
  tab[1] = 8.5;
  ...
  delete [] tab;
  return 0;
}
```

Allocation dynamique de mémoire : Exo

⇨ **Tableau 2D (Matrice) :** Écrire un programme qui alloue dynamiquement la mémoire d'un tableau à deux dimensions, selon le schéma suivant (exemple : 4 lignes, 2 colonnes) :



$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{pmatrix}$$

Fonctions : rôles et intérêts

- **Objectifs** : Découper un programme en petites entités :
  - Indépendantes ;
  - Réutilisables ;
  - Plus lisibles.
- **Où** : Les fonctions peuvent être définies :
  - Dans le fichier principal (au dessus du programme principal) ;
  - Dans d'autres fichiers source (compilation séparée) ;
  - Dans des librairies.
- **Syntaxe d'une fonction**

```
type_de_retour nom_fonction( arguments ) {

        declarations;
        instructions;
}
```

Passage d'arguments par valeur

Exemple de passage par valeur

```
#include <iostream>
using namespace std;
// fonction Add
int Add(int val1, int val2) {
  int res;          // dec var locale
  res = val1+val2; // calcul
  return(res);      // retour
}

int main() {
  int result;
  int a = 5, b = 8;
  result = Add(a, b);
  cout << "\nResultat = " << result;
  result = Add(-8, 15);
  result = Add(-8*2-5, 6*a);

  return (0);
}
```

➪ Les arguments transmis sont recopiés dans les variables locales de la fonction.

Passage d'arguments par adresse

Exemple de passage par adresse

```
#include <iostream>
using namespace std;

void Swap(int *pt1, int *pt2) {
  int aux = *pt1;
  *pt1 = *pt2;
  *pt2 = aux;
}

int main() {
  int a = 5, b = 8;

  cout << "a = " << a << ", b = " << b;
  Swap(&a, &b);
  cout << "a = " << a << ", b = " << b;
  return (0);
}
```

➪ Les arguments transmis sont les adresses des variables. Les valeurs des variables de la fonction appellante sont modifiées.

Passage d'arguments par référence - C++

Exemple de passage par référence

```
#include <iostream>
using namespace std;

void Swap(int &q, int &p) {
  int aux = q;
  q = p; p = aux;
}

int main() {
  int a = 5, b = 8;
  cout << "a = " << a << ", b = " << b;
  Swap(a, b);
  cout << "a = " << a << ", b = " << b;
  return (0);
}
```

- La notation `int &p` signifie que `p` est un entier transmis par référence.
- Le passage par référence conserve la simplicité d'écriture du passage par valeur mais se comporte comme un passage par adresse (variable synonyme ou d'alias).

Retour de paramètres

Retour de paramètres

```
#include <iostream>
using namespace std;

int   RetValeur()     { int x = 1; return  x; }
int* RetAdresse()    { int x = 2; return &x; }
int& RetReference() { int x = 3; return  x; }

int main() {
  int a  = RetValeur();     cout << endl << "a=" << a;
  int *b = RetAdresse();    cout << endl << "*b=" << *b;
  int &c = RetReference(); cout << endl << "c=" << c;
  return 0;
}
```

Cas 2 et 3 : « returning address of local variable »

Arguments par défaut d'une fonction - C++

- Dans une fonction, les derniers arguments peuvent prendre des « valeurs par défaut ».

- Déclaration

```
void fct(char c, int y=10, char* ch="tout") {...}
```

- Appels possibles (depuis le prg. principal)

```
fct('p', 5, "rien"); // Appel classique
fct('p', 5);         // <-> fct('p', 5, "tout")
fct('p');            // <-> fct('p', 10, "tout")
fct();               // Erreur
```

Surdéfinition de fonctions

- On appelle **surdéfinition** (ou surcharge) de fonctions, le fait que plusieurs fonctions portent le même nom.
- La distinction se fait alors par le type des arguments (analyse de la signature des arguments).

Exemple de surcharge de fonctions

```
#include <iostream>
using namespace std;
// Deux versions
void sosie(char a)   { cout << "Sosie 1 : a = " << a; }
void sosie(double a) { cout << "Sosie 2 : a = " << a; }

int main() {
  sosie('e');
  sosie(2.7);
  return (0);
}
```

➪ Attention

- Mécanismes de conversion de types. Ex. : `float` ➪ `double`.
- Possibilités d'ambiguïtés avec les arguments par défaut.
- Règles de recherche très précise utilisées par le compilateur.

Deuxième partie II

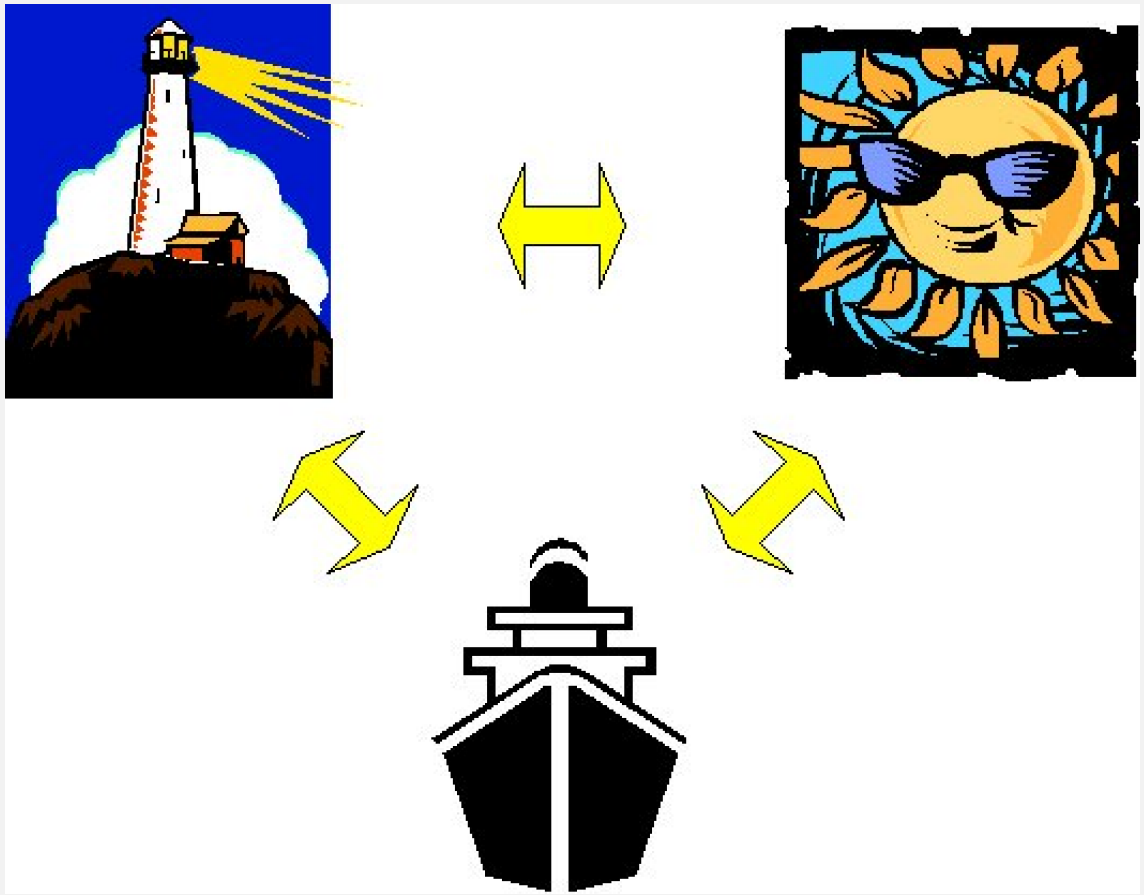
Programmation objet en C++

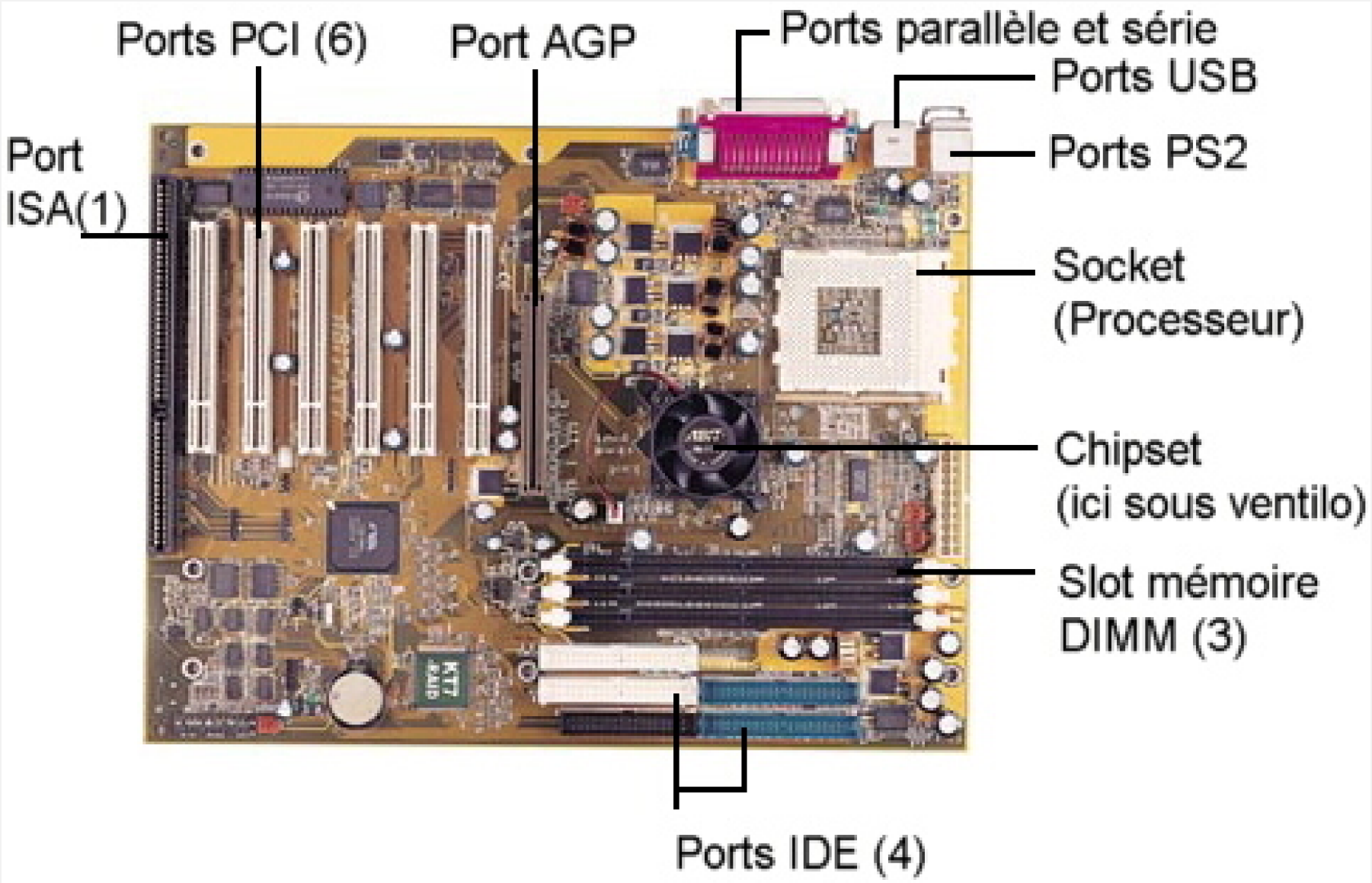
4. Le paradigme objet

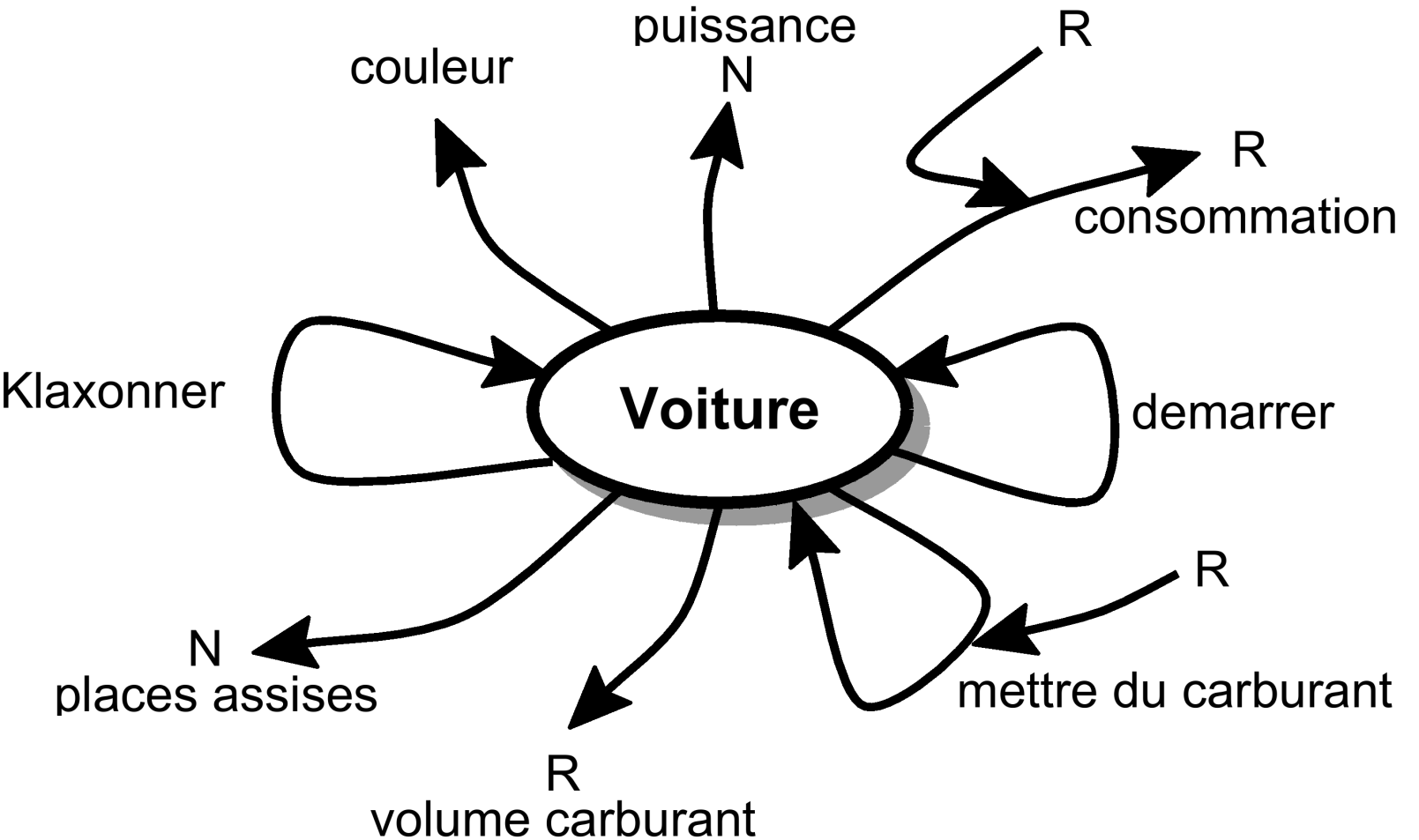
5. Programmation par objets

Le paradigme objet

➽ Le monde réel est composé d'objets autonomes qui sont en relation (communication) et coopèrent.




Ports PCI (6)
Port AGP
Ports parallèle et série
Ports USB
Port
ISA(1)
Ports PS2
Socket
(Processeur)
Chipset
(ici sous ventilo)
Slot mémoire
DIMM (3)
Ports IDE (4)

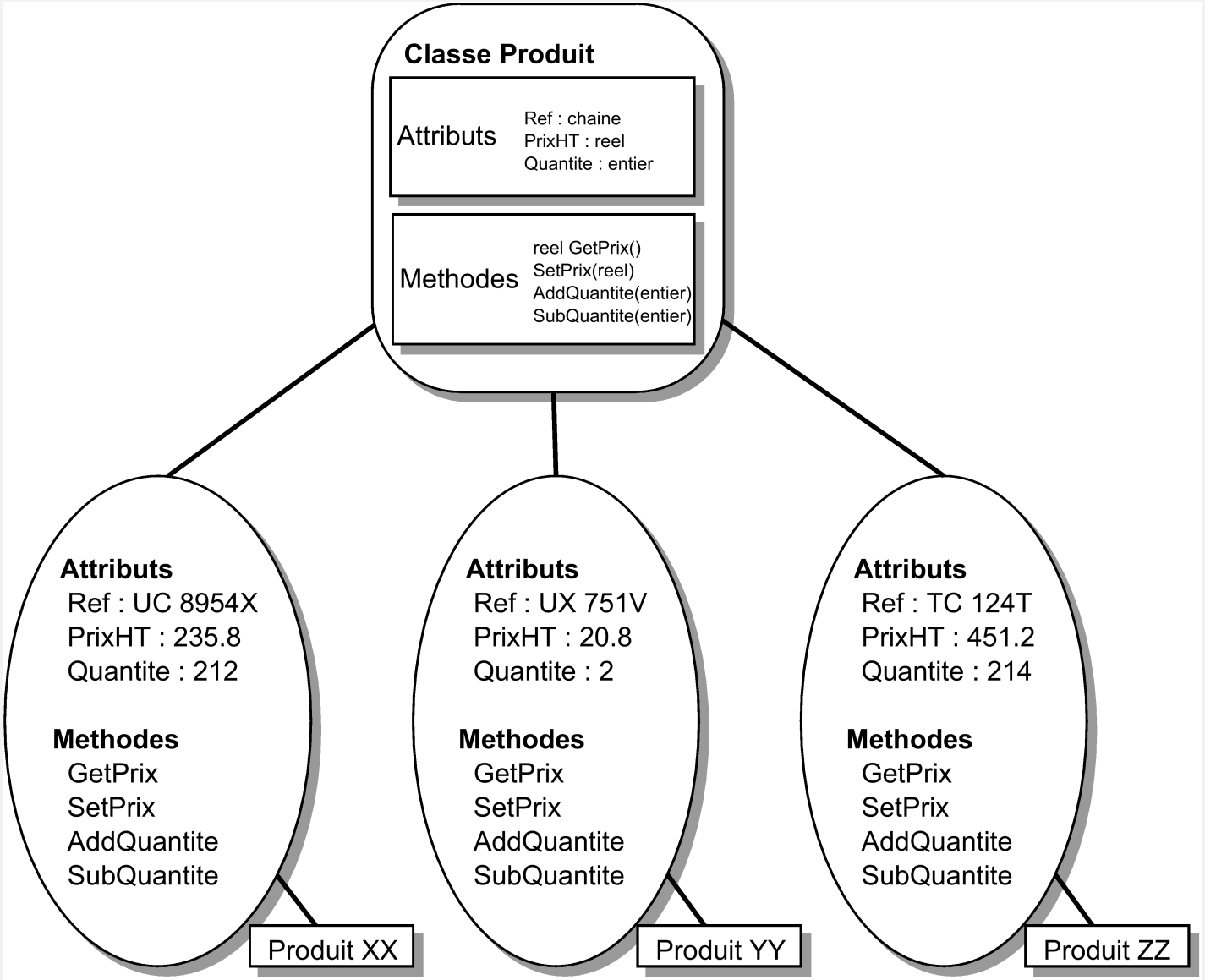

couleur
puissance
N
R
R
consommation
Klaxonner
Voiture
demarrer
R
mettre du carburant
N
places assises
R
volume carburant

Programmation par objets

➽ Classes : C'est une abstraction qui permet de modéliser et/ou décrire un ensemble d'objets de façon générale. La classe explicite le comportement et le rôle des instances.

➽ Instances/Objets : Les instances sont toutes différentes : leur état est différent mais elles ont toutes le même comportement. On décrit parfois la classe comme une usine qui construit des objets. En statistique, on pourrait considérer qu'un objet est une réalisation d'une classe (comme x est une réalisation de la V.A. X).

➽ État d'un objet : C'est l'ensemble des caractéristiques instantanées d'un objet, c'est à dire l'ensemble des valeurs des attributs. Toutes les instances d'une classe ont les mêmes attributs avec des valeurs variables, uniques pour chaque instance.


Classe Produit
Attributs
Ref : chaine
PrixHT : reel
Quantite : entier
Methodes
reel GetPrix()
SetPrix(reel)
AddQuantite(entier)
SubQuantite(entier)
Attributs
Ref : UC 8954X
PrixHT : 235.8
Quantite : 212
Methodes
GetPrix
SetPrix
AddQuantite
SubQuantite
Produit XX
Attributs
Ref : UX 751V
PrixHT : 20.8
Quantite : 2
Methodes
GetPrix
SetPrix
AddQuantite
SubQuantite
Produit YY
Attributs
Ref : TC 124T
PrixHT : 451.2
Quantite : 214
Methodes
GetPrix
SetPrix
AddQuantite
SubQuantite
Produit ZZ

- En première approche,
  - les objets sont à la programmation objet ce que les variables sont à la programmation structurée,
  - les classes sont à la programmation objet ce que les types sont à la programmation structurée.

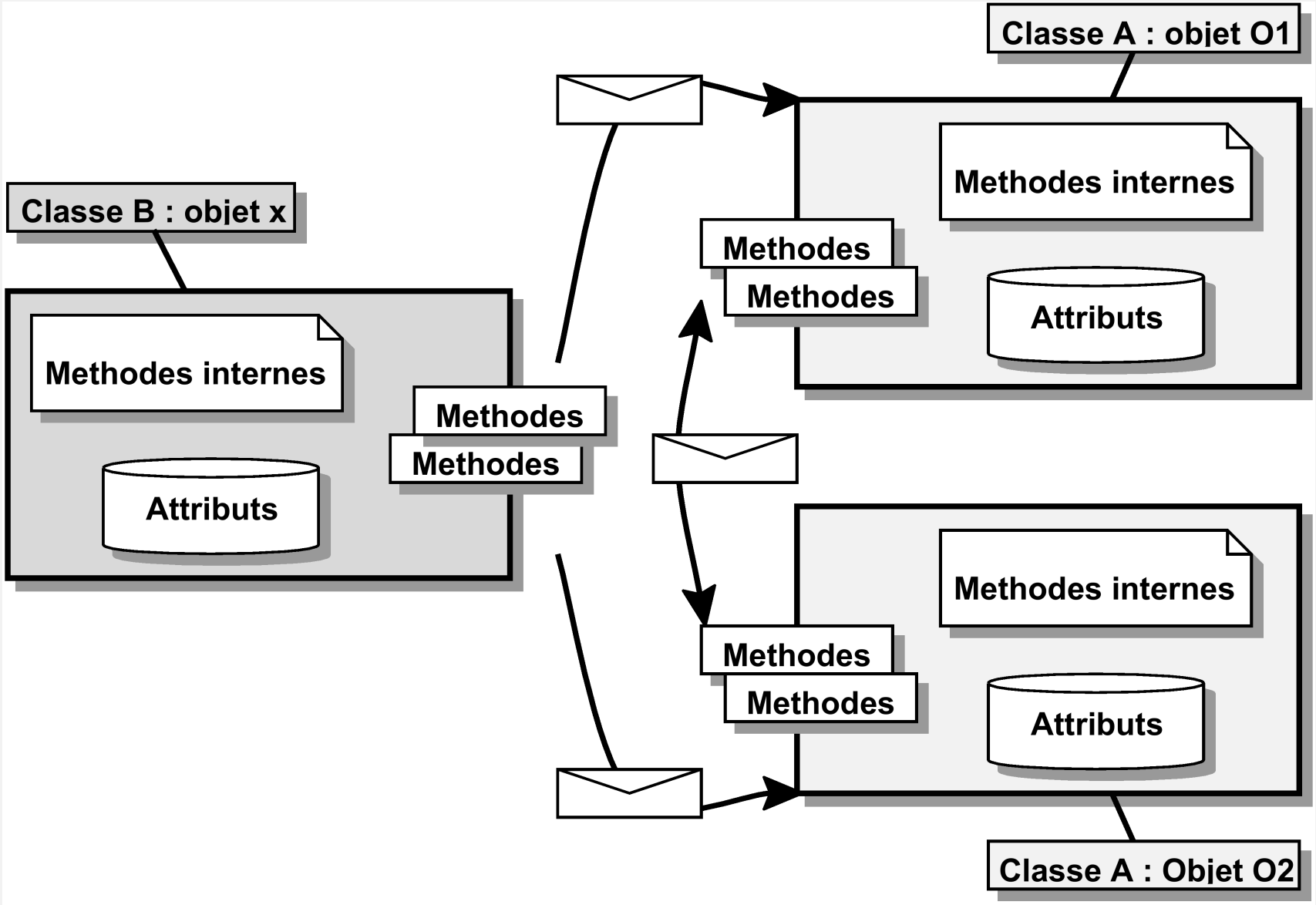
Programmation procédurale	**Programmation objet**
Variable	Objet
Type	Classe

Interface et communication

Les objets

- **L'encapsulation** permet à l'objet de cacher son fonctionnement interne. Ainsi les changements internes n'ont pas d'influence sur le code extérieur à l'objet. Les objets encapsulent les données (« **données membres** » ou « **attributs** ») et les traitements (« **méthodes membres** ») (Partie privée - « private »).

- **L'interface** permet à l'objet de communiquer avec d'autres objets par l'intermédiaire d'envoi de **messages** à l'aide de **méthodes** (partie publique - « public »). Envoyer un **message** à un objet, c'est lui demander d'exécuter une de ses **méthodes**.


Classe A : objet O1
Methodes internes
Methodes
Methodes
Attributs
Classe B : objet x
Methodes internes
Attributs
Methodes
Methodes
Methodes internes
Methodes
Methodes
Attributs
Classe A : Objet O2

Troisième partie III

La programmation en C++

Rappel sur les structures

➽ **Structures :**

- Agrégat de plusieurs objets de types quelconques (appelés *champs* de la structure).
- Exemple : Voiture={marque, puissance, année, ...}
- Une structure est un type et **non** une variable.

➽ **Syntaxe :** Point du plan (coordonnées discrètes)

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

➽ Pour déclarer une variable du type `point` (en `C++` le mot-clef `struct` n'est pas obligatoire !) :

- `struct point coord;`
- `point TabCoord[3];`

- On peut déclarer et initialiser les champs d'une structure :
  - `point coord = {2, 9};`
  - `point TabCoord[3] = {{2, -1}, {4, -9}, {-1, 1}};`

- Pour accéder à un champ de la structure :
  - `cout « coord.x;`
  - `coord.x = 5;`
  - `root = coord.x*coord.x+coord.y*coord.y;`
  - `cout « TabCoord[1].y;`

- L'affectation entre deux structures de même type est permise :

```
point coord1 = {3, 8}, coord2;
coord2 = coord1;
```

Par contre, il n'est pas possible de tester l'égalité, la supériorité, ... de deux structures. Il faut comparer individuellement chaque champs.

Structure, pointeurs et mémoire

- Déclaration d'un pointeur sur une structure : `point *p;`

- Initialisation d'un pointeur sur une structure :

```
point coord1;
p = &coord1;
```

- Comment accéder au contenu de la valeur pointée ?

```
(*p).x == coord1.x
```

Remarque : Plutôt que `(*p).x`, on préférera écrire `p->x`.
Exemple : `cout « (*p).x « ", " « p->x « ", " « coord1.x;`
affiche 3 fois la même valeur.

- Allocation dynamique de mémoire pour une structure :

```
point *p = new point;
p->x = 5; p->y = 8;
delete p;
```

Structure et fonctions membres

Déclaration	Initialisation

`struct point {`
`  // données`
`  int x;`
`  int y;`
`  // Méthodes`
`  void init (int x1, int y1);`
`  void deplace (int dx, int dy);`
`  void affiche ();`
`};`
	`point a;`
	`a.x =  5;`
	`a.y = -2;`
	`a.init (-5, 4);`
	`a.deplace (2, -1);`
	`a.affiche ();`

- C'est l'encapsulation !
- Définition des méthodes de la structure `point`

```
void point::init (int x1, int y1) {
  x = x1; y = y1;
}
void point::deplace (int dx, int dy) {
  x += dx; y += dy;
}
void point::affiche () {
  cout << "Je suis en " << x << ", " << y << endl;
}
```

Exemple récapitulatif

```
struct point {
  int x, y;
  void init (int x1, int y1);
  void deplace (int dx, int dy);
  void affiche ();
};

void point::init (int x1, int y1) {
  x = x1; y = y1;
  }
void point::deplace (int dx, int dy) {
  x += dx; y += dy;
  }
void point::affiche () {
  cout << "Je␣suis␣en␣" << x << ",␣" << y << endl;
}

void main () {
  point a;
  a.init(5,2);       a.affiche();
  a.deplace(-1,-5); a.affiche();
}
```

Notions de classe et d'objet

- De la structure à la notion de classe
  - Le mot clé `struct` devient `class`
  - Membres publics (`public`) et membres privés (`private`)

```
class point {
  public :
    void init (int x1, int y1);      // Méthode
    void deplace (int dx, int dy);   // Méthode
    void affiche ();                 // Méthode
  private :
    int x, y;                        // Attributs
};
```

Exemple récapitulatif

```
#include<iostream>
using namespace std;
class point {
  public :
    void init (int x1, int y1);
    void deplace (int dx, int dy);
    void affiche ();
  private :
    int x, y;
};
```

```
void point::init(int x1, int y1) {
  x = x1; y = y1;
}
void point::deplace(int dx, int dy) {
  x += dx; y += dy;
}
void point::affiche() {
  cout << "Je suis en " << x;
  cout << ", " << y << endl;
}
```

```
int main () {
  point A, B;
  A.init(5,2);
  A.affiche();
  A.deplace(-1,-5);
  B.x = -2; // Impossible car privé
  B   = A;  // OK : Recopie
  return 0;
}
```

Notion de constructeur

➽ **Problème** : Un `point` n'est utilisable que si l'utilisateur

1. a initialisé son `point` grâce à la méthode `init(...)`, puis
2. l'a utilisé ou modifié grâce aux méthodes `deplace(...)` et `affiche()`.

➽ **Constructeur** : On peut obliger l'utilisateur à initialiser son `point` dès la phase de déclaration en utilisant un **constructeur** :

- C'est une méthode particulière qui porte le même nom que la classe, c'est à dire `point`.
- Elle ne comporte **jamais** de valeur de retour !

➽ **Trois catégories :**

- Constructeur par défaut - Tr. 37 ;
- Autres constructeurs - Tr. 38 ; Avec valeurs par défaut - Tr. 39 ;
- Constructeur de recopie - Tr. 45 ;

Constructeur par défaut

➽ Le constructeur **par défaut** ne possède aucun argument. Il initialise un objet et ses attributs avec des valeurs par défaut.

```
class point {
  public :
    point (); // constructeur par défaut
    void deplace (int dx, int dy);
    void affiche ();
  private :
    int x, y;
};
```

```
...

point::point () {
  x = 0;
  y = 0;
}

...
```

➽ **Utilisation :** `point B;`. L'utilisation de `B.affiche();` affichera `0,0` sur l'écran. L'utilisation de `B.deplace(5,3)` permettra à l'utilisateur de la classe d'initialiser le `point` aux coordonnées souhaitées.

Autres constructeurs

```
class point {
  public :
    point ();
    point (int a, int b);
    point (int ab);
    void deplace (int dx, int dy);
    void affiche ();
  private :
    int x, y;
};
```

```
...

point::point() {
  x = 0; y = 0;
}
point::point(int ab) {
  x = ab; y = 0;
}
point::point(int a, int b) {
  x = a; y = b;
}
...
```

➽ **Utilisation** : `point A(3,8);` ou `point B(3);`. Il y a recopie des paramètres passés en arguments dans les attributs correspondants de l'objet.

Constructeurs avec valeurs par défaut

➽ Plutôt que de définir trois constructeurs comme dans l'exemple précédent (surcharge), on peut utiliser des valeurs par défaut dans les arguments :

- **Déclaration :**

```
point(int a=0, int b=0);
```

- **Implémentation :**

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

➽ Ainsi, les trois manières de créer un objet `point` restent possible :

- `point A;` ➽ `0,0`
- `point B(4);` ➽ `4,0`
- `point C(-2,8);` ➽ `-2,8`

Exemple récapitulatif

```
// Fichier point.h
#include<iostream>
using namespace std;
class point {
  public :
    point(int a=0, int b=0);
    void deplace(int dx=-8, int dy=4);
    void affiche();
  private :
    int x, y;
};
```

```
// Fichier point.cc
#include "point.h"

point::point (int a, int b) {
    x = a; y = b;
  }
void point::deplace(int dx, int dy) {
    x += dx; y += dy;
  }
void point::affiche () {
    cout << x << "," << y << endl;
  }
```

```
// Programme principal
#include<iostream>
using namespace std;
#include "point.h"

int main () {

  // Allocation statique
  point A(5,-2), B;
  A.deplace(-5,+2); B.deplace(18);
  A.affiche(); B.affiche();

  // Allocation dynamique
  point *pC=new point(2); // Constructeur
  pC->affiche();
  delete pC; // Destructeur par défaut (cf. plus loin)

  return 0;
}
```

Destructeur d'une classe

➽ Par opposition au **constructeur**, un **destructeur** est une méthode particulière permettant de détruire un objet :

- soit à la fin du bloc d'instructions où le bloc a été créé (appel implicite au destructeur) lorsque l'objet a été créé de manière statique.
- soit à l'aide de l'opérateur `delete` (appel explicite au destructeur) lorsque l'objet a été créé de manière dynamique (opérateur `new`).

➽ Si `X` est le nom de la classe, le constructeur est `X::X(...)` et le destructeur est `X::~X()`. Il ne possède *aucun* argument et ne comporte jamais de valeur de retour.

➽ Comme pour les constructeurs, il existe un **destructeur par défaut**. Il n'est donc pas nécessaire de le programmer, sauf ...

➽ **Attention** : Lorsque la classe utilise de la mémoire allouée dynamiquement (opérateur `new`), on doit remplacer le destructeur par défaut en le surchargeant (sinon on a une destruction superficielle). Le destructeur a la charge de libérer la mémoire allouée dynamiquement.

➽ **Exemple** : classe `Vect`. Création d'une classe qui gère un vecteur de nombres réels dont la taille est fixée par un argument transmis au constructeur.

```
int main() {
    // Statiquement
    Vect T(10);
    T.Set(3, 8.7);
    // Dynamiquement
    Vect* P=new Vect(10);
    P->Set(4, -1.2);
    delete P;
}
```

```
class Vect {
  public:
    Vect(int size=10);
    ~Vect();
    void Set(int i, double v);
  private:
    double *TAB;
    int Taille;
};
```

```
Vect::Vect(int size) {
  if  (size>0) Taille = size;
  else         Taille = 10;
  TAB = new double [Taille];
}

Vect::~Vect(){
  if (TAB != NULL) {
    delete [] TAB;
    TAB = NULL;
  }
}

void Vect::Set(int i, double v){
  ...
}
```

➽ Le test permet d'éviter d'essayer de détruire de la mémoire qui n'aurait pas été allouée (dans ce cas, `TAB==NULL`).

Constructeur de recopie

➽ Un **constructeur de recopie** est un constructeur qui initialise un objet d'une certaine classe avec un autre objet de la même classe.

➽ Exemple :

```
point a(3, -8);
point b(a); // utilisation du constr. de recopie
a.affiche(); b.affiche();
```

➽ Le constructeur de recopie est nécessaire pour

- La transmission par valeur d'un objet en argument d'une fonction : `void fct (point a);`
- Le retour par valeur d'un objet comme résultat d'une méthode ou fonction : `point fct (int h);`
- L'initialisation d'un objet à partir des caractéristiques d'un autre objet : cf. exemple ci-dessus.

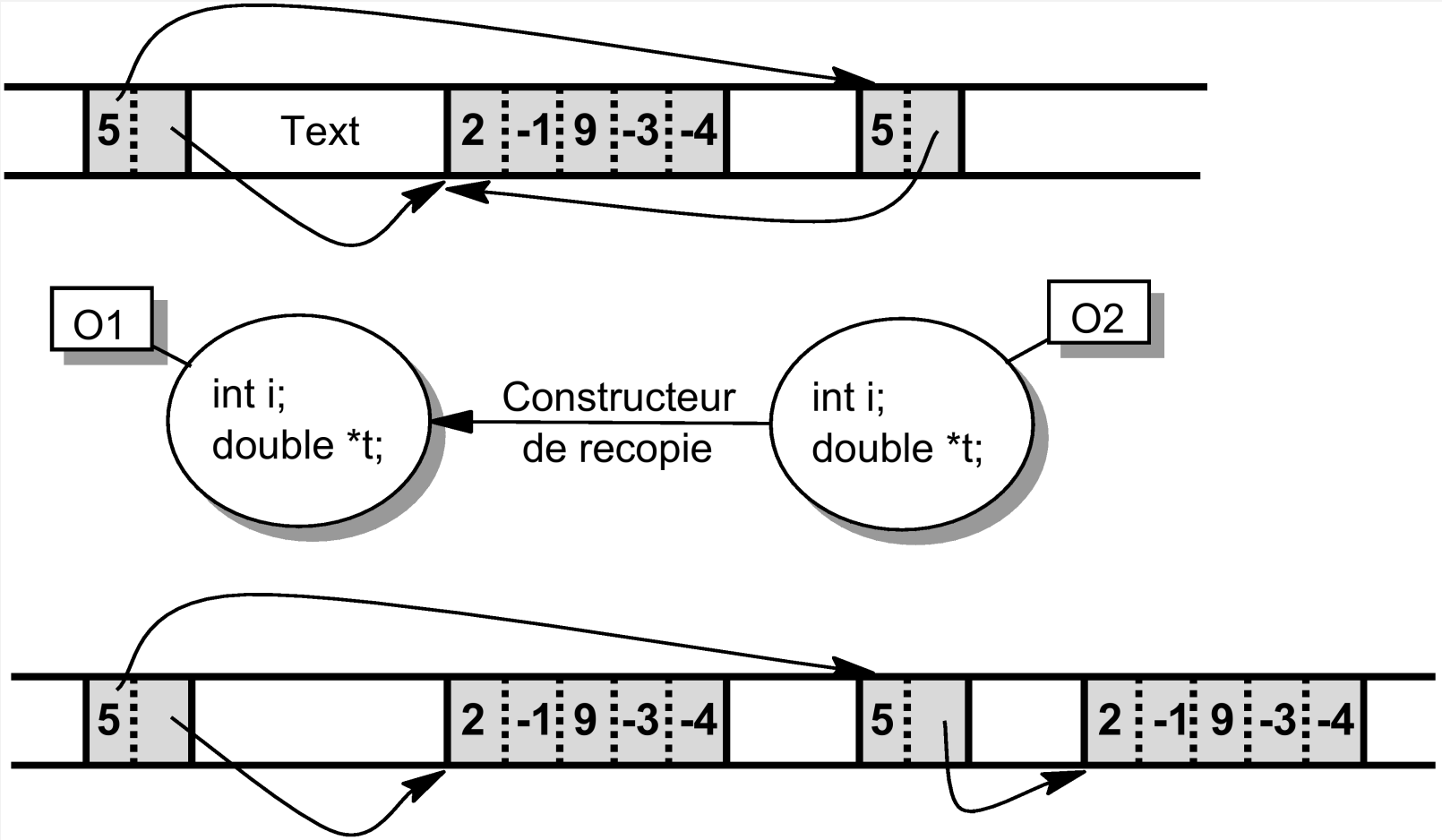
➽ Chaque classe est *automatiquement* équipée d'un **constructeur de recopie par défaut**. Celui-ci réalise la copie d'un objet à partir d'un autre, c'est à dire la recopie de leurs attributs privés et publics. Il n'est donc pas nécessaire *a priori* de la programmer, sauf ...

➽ Attention : La recopie n'est que *superficielle* (copie membre à membre) ! Le constructeur de recopie par défaut ne fonctionne pas correctement dans le cas de classes gérant de la mémoire allouée dynamiquement.

➽ Solution : Pour les classes utilisant l'allocation dynamique de mémoire, il faut (absolument) coder son propre constructeur de recopie. Comme tout constructeur, il porte le nom de sa classe. Il prend comme argument un objet du même type que la classe. Cet argument est le plus souvent passé par référence constante (pour éviter une copie alors même que l'on est en train de définir la façon de faire une copie). Exemple : `Vect::Vect(const Vect& V1);`

Exemple : Classe « vecteurs de `double` »

```
void main () {

  Vect T1(5); // constructeur
  T1.affiche();
  Vect T2(T1); // constructeur de recopie
  T2.affiche();

  // Modif. de T2
  T2.set(1, -1.0);

  // Affichage des tableaux
  T1.affiche(); T2.affiche();
}
```


5
Text
2
-1
9
-3
-4
5
O1
int i;
double *t;
Constructeur
de recopie
O2
int i;
double *t;
5
2
-1
9
-3
-4
5
2
-1
9
-3
-4

```
class Vect {
  public :
    Vect(int s);
    Vect(const Vect &P); // constructeur de recopie
    ~Vect();             // destructeur
    void set(int i, double v);
  private :
    int    Taille;       // taille du vecteur
    double *TAB;         // adresse du début du vecteur
};
```

```
#include "Vect.h"
Vect::Vect(int size) {
  Taille = size;
  TAB    = new double[Taille];
}
Vect::Vect(const Vect &P) {
  Taille = P.Taille;
  TAB    = new double[Taille];
  for(int i=0; i<Taille; i++) TAB[i] = P.TAB[i];
}
Vect::~Vect() {
  if (TAB != NULL) { delete [] TAB; TAB = NULL; }}
void Vect::set(int i, double v) {
  if (i>=0 && i<Taille) TAB[i]=v;}
```

Opérateur d'affectation

➽ L'**opérateur d'affectation =** joue un rôle particulier, semblable à celui du constructeur de recopie, puisqu'il permet d'initialiser un nouvel objet à partir d'un objet déjà initialisé :

```
point c(3,5), z;
z = c;
```

➽ **Attention :** Ne pas confondre *recopie* et *affectation* :

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

➽ Il existe un **opérateur = par défaut** comme il existe un **constructeur de recopie par défaut**. La recopie d'un objet par l'opérateur = n'est que superficielle (copie membre à membre). Lorsque la classe possède des attributs alloués dynamiquement, il faut surcharger l'opérateur = en implémentant une méthode appelée `operator=()` qui peut prendre l'une des signatures suivantes :

```
void operator=(X&); ou  X& operator=(X&);
```

et l'expression `y=x` équivaut à `y.operator=(x)`. L'objet appelant est `y` et l'objet passé en paramètre est `x`. Le résultat de l'affectation se fait dans l'objet appelant.

Exemple

```
class Vect {
  public :
    Vect(int s);
    Vect(const Vect &P); // constructeur de recopie
    Vect& operator=(const Vect &P); // opérateur d'affectation
    ~Vect();             // destructeur
    void affiche();
    void set(int i, double v);
  private :
    int    Taille;       // taille du vecteur
    double *TAB;         // adresse du début du vecteur
};
```

```
#include "Vect.h"

Vect& Vect::operator=(const Vect &P) {
  if (this != &P) { \\auto-référence
    delete [] TAB;
    Taille = P.Taille;
    TAB = new double[Taille];
    for (int i=0; i<Taille; i++) TAB[i] = P.TAB[i];
  }
  return *this;
}
```

Fonction amie d'une classe

- Une **fonction amie** d'une classe (mot clef `friend`)
  - est une *fonction*, non pas une *méthode*,
  - est déclarée dans le fichier d'en-tête de la classe,
  - a les mêmes droits qu'une fonction membre, et peut donc accéder aux attributs privés.

- **Exemple :** Fonction `coincide` qui teste si deux objets de la classe `point` sont identiques :
  - Fonction membre

```
int temp = B.coincide1(A);
```

  - Fonction indépendante amie de la classe `point`

```
int temp = coincide2(B,A);
```

```
class point {
  public :
    point (int x1=0, int y1=0);
    int coincide1(point A);
    friend int coincide2(point B, point A);
  private :
    int x, y;
};
```

```
int point::coincide1(point A){
  if( (x==A.x) && (y==A.y) )
    return 1;
  else
    return 0;
}
int coincide2(point B, point A) {
  if ( (A.x==B.x) && (A.y==B.y) )
    return 1;
  else
    return 0;
}
```

- Nous verrons d'autres exemples avec les opérateurs amis - Tr. 57

- Il existe d'autres situations d'amitiés telles que :
  - Une fonction membre d'une classe, amie d'une autre classe.
  - Une fonction amie de plusieurs classes.
  - Une classe amie : toutes les fonctions membres d'une classe sont amies d'une autre classe.

Les opérateurs

- Notion d'opérateur - Tr. 56
- Opérateurs amis - Tr. 57
- Opérateurs membres - Tr. 59
- Choix de la syntaxe des opérateurs - Tr. 61
- Autres opérateurs - Tr. 62

Notions de surcharge d'opérateur

➽ Problème : On aimerait pouvoir faire la somme de deux objets en utilisant le signe +. Cette surcharge de l'opérateur + est déjà réalisée pour les nombres entiers, les nombres réels, ...

➽ Exemple :

```
point b(5,4), c(-2,1), a;
a = b + c;
```

➽ Remarque : Nous utilisons ici l'opérateur d'affectation = et l'opérateur d'addition +. Cela équivaut à `a.operator=(b.operator+(c))`

➽ Un opérateur peut être surdéfini comme un opérateur ami ou comme un opérateur membre.

Opérateur ami d'une classe

- Un opérateur ami est une fonction *amie* pour ce qui concerne les opérateurs.

- Exemple : Addition de deux objets de type `point`.

```
#include <iostream>
using namespace std;
class point {
public:
   point(int x1=0, int y1=0);
   friend point operator+ (const point& A, const point& B);
   void affiche();
 private :
   int x,y;
 };
```

```
#include "point.h"
point::point(int x1, int y1){ x=x1; y=y1;}
point operator+ (const point& A, const point& B){
  point C(A.x+B.x,A.y+B.y);
  return C;
}
void point::affiche(){cout << endl << x << "," << y;}
```

```
#include "point.h"
void main() {
  point A(3,5);
  point B(A); // Constr. de recopie par défaut
  point C;
  C = A+B;    // Opér. d'addition (amie) +
              // Opér. d'affectation par défaut

  A.affiche(); B.affiche(); C.affiche();
}
```

Opérateur membre d'une classe

- Un opérateur membre est une fonction *membre* pour ce qui concerne les opérateurs.

- Exemple : Opération `+=` entre deux objets de type `point`.

```
#include <iostream>
using namespace std;
class point {
 public:
   point(int x1=0, int y1=0);
   point& operator+=(const point& A);
   void affiche();
 private :
   int x,y;
};
```

```cpp
#include "point.h"
 point::point(int x1, int y1){
   x=x1; y=y1;
 }
 point& point::operator+= (const point &A){
   x+=A.x; y+=A.y;
   return *this;
 }
 void point::affiche(){
   cout << endl << x << "," << y;
 }
```

```cpp
#include "point.h"
void main() {
   point A(3,5);
   point B(8,4);
   B += A;
   A.affiche();
   B.affiche();
}
```

Synthèse sur la syntaxe des opérateurs

- Différents cas de figure :
  - Un opérateur est soit unaire (`-A`), soit binaire (`A+B`).
  - Un opérateur peut être défini
    - soit comme une *fonction globale* à un ou deux arguments,
    - soit comme une *fonction membre* avec un argument de moins.

	Binaire (`A+B`)	**Unaire** (-A)
Fonction	`point operator+(point A, point B)`	`point operator-(point A)`
Membre	`point point::operator+(point A)`	`point point::operator-()`

Autres opérateurs

- Que faire dans les cas suivants (classe `point`) :
  - `B=A-1` : ami `friend point operator-(const point& A, int i);` ou membre `point point::operator-(int i);`
  - `B=1-A` : ami `friend point operator-(int i, const point& A);` uniquement.
  - `B = A++` ou `B = ++A` ...

- On peut surcharger les opérateurs suivants (cf. TD)
  - les opérateurs d'insertion « et d'extraction » pour afficher ou récupérer un `point` par exemple.
  - les opérateurs `+`, `-`, `*`, `/`, `%` et `+=`, `-=`, `*=`, `/=`, `%=` entre :
    - [un objet] et [un `double` ou un `float` ou un `int`],
    - [un `double` ou un `float` ou un `int`] et [un objet].
  - les opérateurs logique `&&`, `||`, `==`, `!=`, `!`, `>`, `>=`, `<`, `<=`
  - l'opérateur d'indexation `[]` que l'on peut utiliser dans la classe `Vect` pour obtenir le ième élément d'un tableau.

Flots d'entrée - sortie

➽ Flot d'entrée = « canal recevant » - Flot de sortie = « canal fournissant ».
Ce canal peut être un périphérique (écran, clavier, imprimante) ou un fichier.

➽ Flot de sortie : Classe `ostream` (ex : `cout`) :

- `ostream& operator« (expression);`
- `ostream& put(char c);` Transmet au flot le caractère `c`.
- `ostream& write(void* adr, int taille);` envoie `taille` caractères prélevés à partir de l'adresse `adr`.

➽ Flot d'entrée : Classe `istream` (ex : `cin`) :

- `istream& operator» (type_de_base &);` Le type de base doit être un pointeur
- `istream& get(char& c);` Extrait le caractère `c` du flot d'entrée.
- `istream& getline(char* ch, int taille, char delim);` Lit les caractères sur le flot et les place à l'adresse ch. La lecture s'arrête si le caractère `delim` est rencontré ou si `taille-1` caractères ont été lus.
- `istream& read(void* adr, int taille);` lit `taille` caractères sur le flot et les range à partir de l'adresse `adr`.

- Surdéfinition de « pour une classe `T` quelconque.

```
ostream& operator<< (ostream& sortie, T& objet) {
  //Envoie sur le flot des membres de l'objet en utilisant << sur
      des type de base. Ex. :
  sortie << objet.taille << ", " << objet.name;
  return sortie;
}
```

On peut envoyer un objet la classe `T` sur un flot : `T a; cout « a;`

- Surdéfinition de » pour une classe `T` quelconque.

```
istream& operator>> (istream& entree, T& objet) {
  //Lecture sur le flot de type de base qui sont affectés à des
      attributs de l'objet. Ex. :
  entree >> objet.taille >> objet.name;
  return entree;
}
```

Les fichiers

- La classe `ofstream`, dérivant de `ostream`, permet de créer un flot de sortie associé à un fichier : `ofstream flot(char *nomfich, mode_ouverture)`. La fonction membre `seekp(deplacement, origine)` permet d'agir sur le pointeur de fichier.
- La classe `ifstream`, dérivant de `istream`, permet de créer un flot d'entrée associé à un fichier : `ifstream flot(char *nomfich, mode_ouverture)`. La fonction membre `seekg(deplacement, origine)` permet d'agir sur le pointeur de fichier.
- Modes d'ouverture

Mode	**Action**
`ios::in`	Ouverture en lecture (obligatoire pour `ifstream`)
`ios::out`	Ouverture en écriture (obligatoire pour `ofstream`)
`ios::app`	Ouverture en ajout de données (écriture à la fin)
`ios::trunc`	Si le fichier existe, son contenu est perdu
`ios::binary`	À n'utiliser qu'avec windows qui fait la distinction entre modes binaire et texte

- la méthode `close()` permet de fermer le fichier.

Introduction aux classes génériques

➽ Les **classes génériques**, ou *patrons*, ou *modèles* de classes (ou « template ») permettent de définir des classes paramétrées par un `type` ou par une autre `class` :

- un `point` peut être constitué de coordonnées entières ou réelles,
- un `vecteur` peut être constitué d'éléments de type `char` ou `float`.

➽ Sans la notion de classes générique, nous serions amenés à écrire les mêmes méthodes pour plusieurs classes, alors que seul le type diffère. Par contre, si on dispose d'une classe `point` paramétrable, on pourra par exemple écrire :

```
void main() {
  point<double> V1(3.5, -5.6); V1.affiche();
  point<char>   V2('r', 'T');  V2.affiche();
}
```

```
# Fichier Point.h

#include<iostream>
using namespace std;

template <class T>
class point
{
public:
   point(T x1, T y1){x = x1; y = y1;}
   void deplace(T d){x += d; y += d;}
   void affiche(){cout <<x<<", "<<y<<endl;}
   T getX(){return x;}

private:
   T x, y;
};
```

La classe est entièrement développée dans le fichier ".h" !

Classe composée ou classe dérivée ?

➽ Une classe est composée si des attributs sont des objets :

```
class Date {
  public:
    Date(int j, int m, int
         a);
    ...
  protected :
    int jour, mois, annee;
};
```

```
class Individu {
  public:
    Individu (char* Name, Date& d);
    ...
  private :
    char Nom[100];
    Date DN;
};
```

Un `Individu` a une `Date` de naissance : composition.

➽ Une classe est dérivée si elle apporte une information supplémentaire à la classe de base :

```
class point {
 public:
   ...
 protected :
   int x, y;
};
```

```
class pointcol : public point {
  public:
    ...
  private :
    short couleur;
};
```

Un `pointcol` est constitué d'un `point` et d'un attribut de couleur.

Classe composée et liste d'initialisation

```
class Date {
  public:
    Date(int j, int m, int a);
    int comDate(const Date& d);
    void afficher();
  protected :
    int jour, mois, annee;
};
```

```
class Individu {
  public:
    Individu (char* Name, int j, int
          m, int a);
    Individu (char* Name, Date& d);
    void afficher();
  private :
    char Nom[100];
    Date DN;
};
```

```
int main () {
   Individu I1("Jean", 5,12,2006);
   Date d(16,6,1971);
   Individu I2("Paul", d);
   I2.afficher();
   return 0;
}
```

- Constructeurs de `Individu` incorrect !

```
Individu::Individu (char* Name, int j, int m, int a)
{
    ...
}
 Individu::Individu (char* Name, Date& d)
{
    ...
}
```

⇨ Problème Le constructeur de `Date` n'a pas de valeur par défaut ce qui interdit de construire `DN`.

- Solution - liste d'initialisation avec appel au constructeur de recopie

```
Individu::Individu (char* Name, int j, int m, int a) : DN(j,m,a)
{   strcpy(Nom, Name); }

Individu::Individu (char* Name, Date& d) : DN(d)
{   strcpy(Nom, Name); }
```

Hiérarchie des classes et héritage

- Objectifs de l'héritage - Tr. 72
- Principe de la dérivation - Tr. 75
- Constructeurs, destructeur et opérateurs de recopie - Tr. 78
- Notion sur l'héritage multiple - Tr. 81
- Méthodes virtuelles et polymorphisme - Tr. 83

Objectifs de l'héritage

➽ L'héritage est l'un des principes fondamentaux de la P.O. Il a pour objectif de hiérarchiser les classes et les objets. L'héritage est mis en oeuvre par la construction de classes dérivées.

➽ Une classe dérivée :

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

➽ Avantages :

- Une classe dérivée modélise un cas particulier de la classe de base ;
- Une hiérarchie de classes facilite la solution de problèmes complexes ;
- Facilite la maintenance, le développement et les extensions ;
- Permet d'ajouter de nouvelles méthodes ;

➽ Inconvénient :

- Introduit une forme de rigidité (à user avec modération) ;

Héritage et dérivation

- La définition d'une classe dérivée est :

```
class classe_derivee : protection
classe_de_base {...}
```

- Les types de protection sont `public`, `protected` ou `private`. On parle de dérivation *publique*, *protégée* ou *privée*. `Protected` et `Private` permettent de réduire l'accès à la classe de base par la classe dérivée. Nous utiliserons uniquement la dérivation `publique`.

- Contrôle des accès : le statut `protected`.

```
class point {
  public:
    ...
  protected :
    ...
  private :
    ...
};
```

- Les attributs ou méthodes déclarés `protected`
  - restent inaccessibles pour les utilisateurs de la classe, ils apparaissent comme membres `private`,
  - deviennent accessibles pour les membres d'une éventuelle classe dérivée, ils apparaissent comme membres `public`.

Principe de la dérivation

- Exemple

```
class point {
  public :
    void init(int x1, int y1)
      { x = x1; y = y1; }
    void deplace(int dx, int dy)
      { x += dx; y += dy; }
    void affiche ()
      { cout << x << "," << y;}
  protected :
    int x, y;
};
```

```
#include"point.h"
class pointcol : public point {
  public :
    void colore(short cl)
      {couleur = cl; };
   private :
    short couleur;
};
```

- Chaque objet de type `pointcol` peut faire appel :
  - aux méthodes publiques de `pointcol` (ici `colore(...)`),
  - aux méthodes publiques de la classe de base `point`.

```
void main () {
  pointcol p;
  p.init(10,20); p.colore(5); p.affiche();
  p.deplace(2,4); p.affiche();
}
```

➽ Lacune de cette implémentation : lorsqu'on nous appelons `affiche()`, nous n'obtenons pas d'information sur la couleur du point coloré.

➽ Idée : On peut redéfinir la méthode `affiche()` de la classe mère (on parle de *masquage* et non de *surcharge*)

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

➽ Attention : Pour que cette solution fonctionne, il faut que les attributs de la classe mère soient déclarés `protected` et non `private`. Ainsi, la classe dérivée a accès aux attributs internes de sa classe mère.

➽ Autre possibilité : La classe dérivée peut accéder aux méthodes publiques de sa classe mère :

```
void pointcol::affiche() {
  point::affiche(); // méthode de la classe mere
  cout << "," << couleur;
}
```

Exemple récapitulatif

```
// Fichier pointcol.h
#include "point.h"

class pointcol : public point {
  public :
    void colore(short cl){couleur =
         cl; };
    void affiche();
    void init(int x1, int y1, short
         cl);
  private :
    short couleur;
};
```

```
#include "pointcol.h"
void pointcol::affiche() {
  point::affiche();
  cout << "," << couleur;}
void pointcol::init(int x1,
     int y1, short cl){
  point::init(x1,y1);
  couleur = cl; }

void main() {
  pointcol p;
  p.init(10, 20, 5); p.affiche();
  p.point::affiche();
  p.deplace(2,4); p.affiche();
  p.colore(2);    p.affiche();
}
```

Constructeur, destructeur et op. de recopie

➽ Appel aux constructeurs : Lorsqu'on construit un objet d'une classe `B` héritée d'une classe `A`, le C++ se charge de prévoir dans l'appel du constructeur de `B` l'appel au constructeur de `A`.

➽ Comment faire lorsque le constructeur de `A` nécessite des arguments ? Exemple :

```
class point {
  ...
  public :
    point(int x, int y);
  ...
};
```

```
class pointcol : public point {
  ...
  public :
    pointcol(int x, int y, short cl);
  ...
};
```

➽ Solution : On écrit l'en-tête du constructeur de la manière suivante :

```
pointcol::pointcol(int x1,int y1,short cl) : point(x1,y1) {
  couleur = cl;
}
```

- La déclaration `pointcol(10,15,3)` entraînera :
  - l'appel au constructeur de la classe `point` avec les arguments `10` et `15` ;
  - la recopie de `3` dans l'attribut `couleur` de la classe dérivée ;

- Destructeurs : L'appel au destructeur de la classe dérivée entraîne automatiquement l'appel au destructeur de la classe mère. Ordre d'appel :(1) destructeur classe fille, puis (2) destructeur classe mère.

- Constructeur de recopie : Le principe énoncé ci-dessous fonctionne aussi lorsque l'on souhaite surcharger le constructeur de recopie par défaut :

```
pointcol::pointcol(pointcol& A) : point(A) {
   couleur=A.couleur;
}
```

- Cependant il faut considérer le cas où la classe fille n'a pas de constructeur de recopie. Dans ce cas, il y a (i) appel au constructeur de recopie par défaut, lequel fait appel au constructeur de recopie de la classe mère (celui qui a été défini ou celui par défaut), et (ii) initialisation des attributs spécifiques de la classe fille.

➽ Opérateur d'affectation = : À vous de faire explicitement appel aux méthodes de la classe mère.

```
pointcol& pointcol::operator=(pointcol& A) {
    this->point::operator=(A);
    couleur = A.couleur;
    return *this;
}
```

Remarque : Dans cet exemple, les opérateurs d'affectation de la classe fille et de la classe mère auraient été suffisants.

Notion sur l'héritage multiple

- Une classe peut hériter de plusieurs classes. Les classes mères sont alors énumérées dans la définition : `class D : public B1, public B2 ... ;`.

- Les modalités d'accès aux attributs des classes mères restent les mêmes. L'opérateur de résolution de portée `::` est utilisé (i) soit lorsque l'on veut accéder à un membre d'une des classes mères si celui-ci a été redéfini dans la classe dérivée, (ii) soit lorsqu'un membre de deux classes mères porte le même nom et qu'il faut préciser celui qui nous intéresse.

- Appels aux constructeurs et destructeurs : Les constructeurs des classes mères sont appelés dans l'ordre avec lequel ils sont cités dans la déclaration de la classe fille. Les destructeurs sont appelés dans l'ordre inverse. Pour les constructeurs,nous pourrons avoir `D::D(...) : B1(...), B2(...) ...`

- Les classes virtuelles : La cascade d'héritages suivants :

```
class B : public A ...;,
class C : public A ...; et
class D : public A, public B ...;,
```

conduit à dupliquer les attributs de la classe `A` dans la classe `D`. Pour résoudre cela, on fait apparaître le mot `virtual` dans les définitions des classes `B` e `C`.

Dans ce cas, un constructeur de la classe `D` aura l'allure suivante :

```
D::D(..) : B(...), C(...), A(...) ....
```

Dans ce cas, les constructeurs de `B` et `C` n'auront plus à spécifier d'information pour le constructeur de `A`.

- On se sert beaucoup de l'héritage multiple dans la programmation d'interfaces graphiques ...

Méthodes virtuelles et polymorphisme

- Le mécanisme des méthodes virtuelles permet de mettre en oeuvre la ligature dynamique et le polymorphisme. Concept qui permet d'appliquer une même opération à des objets de type différents, sans qu'il soit nécessaire d'en connaître le type exact.

- Typage statique des objets

```
class A {
  ...
  public :
    void fct (...);
    ...
};
```

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

Supposons `A* pta;` et `B* ptb;`. Alors l'instruction `pta=ptb;` est autorisée. Cependant, quelque soit l'objet pointé par `pta`, `pta->fct(...)` appelle toujours la méthode de la classe `A`. C'est le *typage statique des objets*.

➽ Typage dynamique des objets Lorsqu'une fonction est déclarée *virtuelle*, l'appel à cette fonction ou à ses re-définitions dans des classes dérivées est résolu au moment de l'exécution, selon le type de l'objet concerné : c'est le typage dynamique des objets (ou ligature dynamique des fonctions).

```
class A {
  ...
  public :
    virtual void fct (...);
    ...
};
```

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

Dans cet exemple, `pta->fct(...)` appellera effectivement la méthode `fct` de la classe correspondant réellement au type de l'objet pointé par `pta`.

➽ Fonction virtuelle pure Une classe comportant au moins une fonction virtuelle pure est un *classe abstraite*, c'est à dire qu'il n'est pas possible de créer des objets de ce type. Une telle fonction se déclare avec une initialisation à 0 : `virtual void affiche() = 0;`

Gestion des exceptions

➽ Le mécanisme des exceptions est destiné à permettre aux fonctions profondes d'une bibliothèque de notifier la survenue d'une erreur aux fonctions hautes qui utilisent la bibliothèque. Mots-clés : `throw`, `try` et `catch`.

➽ Une **exception** est une rupture de séquence déclenchée par un programme à l'aide de l'instruction `throw`. Les points-clés de ce mécanisme sont les suivants :

- la fonction qui détecte un événement exceptionnel construit une exception et la lance (**`throw`**) vers la fonction qui l'a appelée (grâce à **`try`**) ;
- l'exception est un nombre, une chaîne ou un objet d'une classe héritée de la classe **`exception`** comportant des informations pour la caractérisation de l'événement à signaler ;
- une fois lancée, l'exception traverse la fonction qui l'a lancée, la cascade de ses fonctions appellantes, jusqu'à atteindre une fonction qui a prévu d'attraper (**`catch`**) ce type d'exception ;
- si une exception arrive à traverser toutes les fonctions actives, car aucune de ces fonctions n'a prévu de l'attraper, alors elle produit la terminaison du programme.

Syntaxe

```
try {
  instructions susceptibles de provoquer, soit directement
       soit dans des fonctions appelées, le lancement d''une
       exception
}
catch(declarationParametre1) {
  instructions pour traiter les exceptions correspondant au
       type de parametre1
}
catch(declarationParametre2) {
  instructions pour traiter les exceptions, non attrapées
       par le gestionnaire précédent, correspondant au type
       de parametre2
}
catch(...) {
  instructions pour traiter toutes les exceptions non
       attrapées par les gestionnaires précédents
}
```

Exemple 1

```
#include <cmath>
#include <iostream>
using namespace std;

void racine(double d) {
    if (d<0)
      throw 1;
    double h=sqrt(d);
    cout << "Racine␣("<<d<<")="<<h;
    cout << endl;
}
```

```
int main() {
    double d;
    cout << "Entrez␣un␣reel:" << endl;
    cin >> d;
    try {
      racine(d);
    }
    catch(int n) {
      if (n==1) {
        cout << "Nbre␣neg!"<<endl;
        racine(-d);
      }
    }
    return 0;
}
```

Exemple 2

```
#include<iostream>
using namespace std;

class Vecteur
{
  public:
    Vecteur(int size);
    ~Vecteur();
    void SetVal(int i, double v);
  private:
    double *tab;
    int N;
};
```

```
#include "Vecteur.h"

void IndiceValide(int i, int n) {
  if (i < 0 || i >= n)
    throw("indice hors bornes");
}

Vecteur::Vecteur(int size) {
  N=size;
  tab = new double[N];
}

Vecteur::~Vecteur() {
  if (tab != NULL) {
    delete [] tab;
    tab = NULL;
  }
}

void Vecteur::SetVal(int i, double v) {
  IndiceValide(i, N);
  tab[i] = v;
}
```

Exemple 2 (suite)

```
#include "Vecteur.h"

int main() {
  int n, i;
  double x;
  try {
    cout << "n ? "; cin >> n;
    Vecteur t(n);
    cout << "i, t[i] ? "; cin >> i >> x;
    t.SetValue(i,x);
    cout << "t[" << i << "] = " << x << endl;
  }
  catch (const char* message) { // origine: IndiceValide
    cout << "Probleme: " << message << endl;
  }
  catch (bad_alloc) { // origine: new (exception std)
    cout << "Probleme allocation memoire";
  }
  catch (...) { // toutes les autres exceptions
    cout << "Probleme indetermine";
  }
  cout << " - Termine";
  return 0;
}
```

La classe `exception`

- Les exceptions lancées par les fonctions de la bibliothèque standard sont toutes des objets de classes dérivées d'une classe définie spécialement à cet effet, la **`classe exception`**.
- Au minimum la classe `exception` contient les membres suivants :

```
class exception {
  public:
    exception() throw();
    exception(const exception &e) throw();
    exception &operator=(const exception &e) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

- La méthode `what` renvoie une chaîne de caractères qui est une description informelle de l'exception.

Les `exception` de la bibliothèque standard

- Les principales classes dérivées directement ou indirectement de `exception` :
  - **bad_exception** : signale l'émission par une fonction d'une exception qui n'est pas déclarée dans sa clause `throw`.
  - **bad_cast** : signale l'exécution d'une expression `dynamic cast` invalide.
  - **bad_typeid** : indique la présence d'un pointeur `p` nul dans une expression `typeid(*p)`
  - **logic_error** : elles signalent des erreurs provenant de la structure logique interne du programme
    - **domain_error** : erreur de domaine,
    - **invalid_argument** : argument invalide,
    - **length_error** : tentative de création d'un objet de taille supérieure à la taille maximum autorisée,
    - **out_of_range** : arguments en dehors des bornes.
  - **bad_alloc** : correspond à l'échec d'une allocation de mémoire.
  - **runtime_error** : signalent des erreurs, autres que des erreurs d'allocation de mémoire, qui ne peuvent être détectées que durant l'exécution du programme :
    - **range_error** : erreur de rang,
    - **overflow_error** : débordement arithmétique (par le haut),
    - **underflow_error** : débordement arithmétique (par le bas).

Conclusion

➽ Ce que l'on n'a pas vu ...

- Les conversions de type : opérateurs de `cast` comme `i = (int) 3.5;` mais avec des classes ;
- Le qualificatif `static` pour les membres de classe.
- La bibliothèque standard (conteneur, itérateur et algorithme) ;

➽ Vous êtes maintenant suffisamment équipés pour apprendre seul ... Et notamment savoir utiliser les très nombreuses librairies `C++` que l'on trouve sur Internet (n'essayer pas de « ré-inventer la roue »), aussi bien dans le domaine des interfaces graphiques (fenêtre, menus, ...) que pour réaliser des calculs scientifiques (EDP, calculs matriciels, générateurs de nombres aléatoires, ...).

➽ Regardons maintenant comment faire des interfaces graphiques avec QT...