Cours C++

0 - Historique

L'objectif du langage C++ était d'ajouter au langage C des possibilités de programmation objet

Langage C

- Créé en 1972
- Dennis Ritchie (laboratoires Bell de A.T.T.)
- Dérive du langage B développé par Ken Thompson
- Programmation structurée
- Normalisation ANSI/ISO
  - 1989
  - 1999

Langage C++

- Créé à partir de 1982
- Bjarne Stroustrup (laboratoires Bell de A.T.T.)
- Dérive du langage C popularisé par Brian Kernighan
- Programmation objet
- Normalisation ANSI/ISO
- 1998
- 2003

1 - Préliminaires

Les grands principes de la Programmation Orientée Objet se résument à :

La classe
généralisation de la notion de type

L'héritage
définition d'une nouvelle classe à partir d'une classe existante

L'encapsulation
abstraction des données

L'objet (instance d'une classe)
association des données et des procédures appelées méthodes

2 - Incompatibilités C/C++ (1)

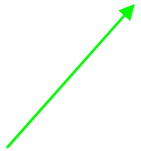
Un source C doit pouvoir être compilé correctement par un compilateur C++

En pratique, il subsiste quelques incompatibilités

Définitions de fonctions en C++

```
int fExemple(x, y)
int x;
float y;
{
 …
}
```

```
int fExemple(int x, float y)
{
 …
}
```



seule la deuxième forme est autorisée en C++

Prototypes en C++
un appel de fonction n'est accepté que si le compilateur connaît le type des arguments et de la valeur de retour

2 - Incompatibilités C/C++ (2)

Arguments et valeur de retour d'une fonction
les différences ne portent que sur la syntaxe des en-têtes et des prototypes des fonctions

Compatibilité entre le type `void *` et les autres pointeurs
seule la conversion d'un pointeur en `void *` est implicite

Qualificateur `const`

portée limitée au fichier source

```
#define N 29
```

peut être remplacé par

```
const int N = 29
```

Utilisable dans une expression

```
const int nbElts = 29
…
int t[nbElts]
```

2 - Incompatibilités C/C++ (3)

Qualificateur `const`

- Ex :

```
const char *ptrSurStrCst = "Une chaîne";

char * const ptrSurStrCst = "Une chaîne";

const char * const ptrSurStrCst = "Une chaîne";
```

1er cas

seule la valeur pointée est constante. On ne peut donc pas écrire

```
ptrSurStrCst[5] = 'c';
```

2ème cas

situation inverse : la chaîne pointée est modifiable, mais l'adresse de début ne l'est pas

3ème cas

ni l'adresse ni le contenu ne peuvent être modifiés

3 - Spécificités C/C++ (1)

Nouvelles possibilités d'E/S

Écriture `cout`

syntaxe : `cout << expr`$_1$ `<< expr`$_2$ `<< … << expr`$_n$

affiche sur le flot `cout` (`stdout`) les valeurs des différentes expressions

types supportés :

- Type de base quelconque
- Chaîne de caractère
- Pointeur

Lecture `cin`

syntaxe : `cin >> lvalue`$_1$ `>> lvalue`$_2$ `>> … >> lvalue`$_n$

lit sur le flot `cin` (`stdin`) les informations et les affecte aux lvalues

types supportés :

- Type de base quelconque
- Chaîne de caractère

3 - Spécificités C/C++ (2)

Nouvelle forme de commentaire

```
cout << "Bonjour\n"  //commentaire de fin de ligne
```

Emplacement libre des déclarations

exemple 1

```
int main() {
int x;
…
x = 5;
…
int y = 2 * x + 1;
}
```

exemple 2

```
for (int i = 0; …; …) {
…
}
```

3 - Spécificités C/C++ (3)

Notion de référence

Transmission des arguments en C

par valeur

```
void echange(int a, int b) {
int tmp;
tmp = a;
a = b;
b = tmp;
}
int main() {
int x = 1;
int y = 2;
echange(x, y);
}
```

L'échange ne peut pas s'effectuer puisque ce sont les valeurs de x et y qui sont transmises à a et b

par adresse

```
void echange(int *a, int *b) {
int tmp;
tmp = *a;
*a = *b;
*b = tmp;
}
int main() {
int x = 1;
int y = 2;
echange(&x, &y);
}
```

L'échange peut s'effectuer puisque ce sont les adresses de x et y qui sont transmises à a et b

3 - Spécificités C/C++ (4)

Notion de référence

Transmission des arguments en C++

par référence

```
void echange(int & a, int & b) {
int tmp;
tmp = a;
a = b;
b = tmp;
}
int main() {
int x = 1;
int y = 2;
echange(x, y);
}
```

Le compilateur prend lui même en charge la transmission des arguments.

Écriture simplifiée
Indépendant de l'utilisateur de la fonction
Risques d'effets de bord

L'échange peut s'effectuer puisque ce sont les adresses de x et y qui sont transmises à a et b

3 - Spécificités C/C++ (5)

Arguments par défaut dans les déclarations de fonctions
la procédure :

```
void p(int a = 29, int b = 8);
```

peut être appelée par :

```
int main() {
     int x = 1, y = 2;
     …
     p(x, y); // appel normal
     p(x);    //  appel possible
     p();     //  appel possible
     p(5, x); //  appel possible
}
```

Les valeurs par défaut doivent obligatoirement être les dernières de la liste

3 - Spécificités C/C++ (6)

Surdéfinition de fonctions
on parle de surdéfinition lorsqu'un même symbole possède plusieurs significations différentes, le choix se faisant en fonction du contexte

```
void sosie(int);
void sosie(float);

int main() {
     int   i = 1;
     float f = 2.5;
     …
     sosie(i);
     sosie(f);
}
```

Le compilateur met correctement en place les appels en tenant compte de la liste des arguments.
La recherche de la fonction surdéfinie se fait en observant certaines règles :

3 - Spécificités C/C++ (7)

Cas des fonctions à un argument

Le compilateur recherche la "meilleure" correspondance possible

- Correspondance exacte
- Correspondance avec promotions numériques
- Correspondance avec conversions dites standards (il peut s'agir de conversions dégradantes)

La recherche s'arrête au premier niveau ayant permis de trouver une correspondance

S'il existe plusieurs correspondances, il y a une erreur de compilation due à l'ambiguïté rencontrée

Cas des fonctions à plusieurs arguments

Le compilateur sélectionne pour chaque argument la ou les fonctions qui réalisent la meilleure correspondance

Si plusieurs fonctions conviennent, il y a là encore une erreur de compilation

3 - Spécificités C/C++ (8)

Opérateurs `new` et `delete`
dans le contexte de la programmation objet, C++ a introduit 2 nouveaux opérateurs, les fonctions de la bibliothèque standard telles `malloc` et `free` restant toujours disponibles

En C l'allocation et la libération mémoire se faisaient de la façon suivante :

```
int *ptr,
    *t;
…
//allocation mémoire
ptr = (int *) malloc(sizeof(int));
t   = (int *) malloc(100 * sizeof(int));
…
//libération mémoire
free(ptr);
free(t);
```

3 - Spécificités C/C++ (9)

Opérateur `new`

```
ptr = new int;
t   = new int[100];
```

syntaxe :

`new` type

`new` type`[n]` avec n entier strictement positif

Opérateur `delete`

```
delete ptr;
delete [] t;
```

syntaxe :

`delete` adresse

Fonction `set_new_handler`

pour gérer les débordement mémoire

3 - Spécificités C/C++ (10)

Fonctions `inline`

Constat sur le préprocesseur

Il crée des expressions avec un comportement inattendu :
une macro définie par

```
#define carre(x) (x * x)
```

et appelée par

```
carre(a++);
```

engendrera l'expression

```
a++ * a++
```

Il ne dispose pas du concept de type

Le préprocesseur est donc considéré comme un endroit à risque

3 - Spécificités C/C++ (11)

C++ présente une amélioration des macros du préprocesseur avec les fonction `inline`

- Elles ont le comportement des fonctions
  (vérification des arguments et de la valeur de renvoi)
- Aucune gestion de code : elles sont stockées dans la table de symboles puis substituées dans le code après vérification

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

Avantage

- Plus rapide qu'une fonction

Inconvénients

- Code incorporé plus important (le code est généré pour chaque appel)
- Pas de compilation séparée

4 - Eléments de base du langage (1)

Types de base

Booléen

bool
2 valeurs possibles : true et false

Caractère

[signed | unsigned] char
Ex : ‘B’

Entier

[signed | unsigned] [long | short] int

[signed | unsigned] {long | short} [int]
Ex : 125

Flottant

{[long] double | float}
Ex : 0.3
2.18E+2

Vide

void
Il ne s'agit pas d'un type à proprement parler. Il sert à déclarer une fonction ne retournant aucun résultat

4 - Eléments de base du langage (2)

Déclarations

[classe_de_mémorisation] [qualifieurs] type identificateur [= valeur];

Classe de mémorisation

Classe de mémorisation	Niveau de déclaration
extern	global ou local
static	global ou local
auto	local
register	local
typedef	global ou local; artifice syntaxique

Qualifieurs

const
indique que le contenu d'une variable ne doit pas changer

volatile
indique au compilateur qu'une variable peut voir son contenu évoluer indépendamment des instructions du programme

4 - Eléments de base du langage (3)

Notions de base

lvalue

Une Left-value (valeur gauche) est un élément de syntaxe C++ pouvant être écrit à gauche d'un = d'affectation

Une left-value se caractérise par :

- Un type précis (et donc une taille en mémoire)
- Une valeur mais...
- Surtout un emplacement de stockage en mémoire

rvalue

Une Right-value (valeur droite) est un élément de syntaxe C++ pouvant être écrit à droite d'un = d'affectation

Une right-value se caractérise par :

- Un type précis (que vous devez savoir identifier) ;
- Une valeur

4 - Eléments de base du langage (3)

Opérateurs

Opérateurs unaires

Opérateurs d'accès aux objets

`[]` indexation

ex :
```
t[i];
```

`&` adresse de

ex :
```
int i, *pi;
pi = &i;
```

`*` contenu de

ex :
```
int t[100];
*pi = 2;
*(t)=0;      //initialise à 0 le 1er élément du tableau
*(t+1)=0;    //initialise à 0 le 2ème élément du tableau
```

`.` sélection

ex :
```
struct (int pi,pr) complexe;
complexe.pr;
```

`->` sélection

ex :
```
struct (int pi,pr) *complexe;
complexe->pr;
```

Opérateurs arithmétiques

`-` moins unaire

`++ --` pré/post incrémentation/décrémentation

ex :
```
int i, j, a, b;
i = 5; j = 10;
a = i++;     //a prend la valeur 5, i prend la valeur 6
b = --j;     //b prend la valeur 9, j prend la valeur 9
```

4 - Eléments de base du langage (4)

Opérateurs

Opérateurs unaires

Opérateurs logiques

`!`	négation logique

Opérateur de bits

`^`	complément à 1 bit à bit

Autres opérateurs unaires

`(type)`	opérateur de caste
`sizeof()`	opérateur de taille

Opérateurs binaires

Opérateurs multiplicatifs

`*`	multiplication
`/`	addition
`%`	modulo

Opérateurs additifs

`+`	addition
`-`	soustraction

Opérateurs de décalage et d'écriture sur un flot

`<<`	décalage gauche/ecriture sur flot
`>>`	décalage droit/lecture sur flot

4 - Eléments de base du langage (5)

Opérateurs

Opérateurs binaires

Opérateurs logiques

`<`	inférieur
`>`	supérieur
`<=`	inférieur ou égal
`>=`	supérieur ou égal
`==`	égalité
`!=`	différence

Opérateurs relationnels

`&&`	et logique
`||`	ou logique

Opérateurs de bits

`&`	et bit à bit
`^`	ou exclusif bit à bit
`|`	ou inclusif bit à bit

4 - Eléments de base du langage (6)

Opérateurs

Opérateur ternaire de condition ?

Son utilisation est

$expression_1$? $expression_2$: $expression_3$;

$expression_1$ est évaluée :

si sa valeur est différente de 0,
l'opérateur ? retourne la valeur $expression_2$,
sinon il retourne la valeur $expression_3$

ex :
```
int x;
…
cout << (x % 2) ? "impair" : "pair";
```

4 - Eléments de base du langage (7)

Opérateurs

Opérateur d'affectation =

Il s'utilise de la manière suivante :

lvalue = expression;

lvalue opérateur_binaire= expression;

lvalue = lvalue opérateur_binaire expression;

ex :
```
x = 62;
x += 5; // équivalent à : x = x + 5
x = x / 2;
```

Une affectation étant encore une right-value, il est possible d'écrire :

```
a = b = c = d;
```

4 - Eléments de base du langage (8)

Opérateurs

Opérateur ,

Cet opérateur constitue une série de rvalues comptant pour une seule rvalue : la dernière de la liste

ex 1 :
```
i = (j = 1, j + 2); //donne à i la valeur 3
```

ex 2 :
```
int i, j;
char s[20];
…
for (i = 0, j = strlen(s) - 1; i < j; i++, j--) {
   char c = s[j]; s[j] = s[i]; s[i] = c;
}
```

4 - Eléments de base du langage (10)

Opérateurs

Opérateur de résolution de portée ::

Il permet, d'une manière générale, de spécifier le bloc auquel l'objet qui le suit appartient

La portée d’une variable : c’est la zone du programme où elle est connue, et peut être utilisée

Si des variables ont le même nom, les plus "éloignées" sont cachées par celles les plus "proches"

4 - Eléments de base du langage (12)

Opérateurs

Opérateur de résolution de portée ::

ex 1 :

```
int i=1; //variable de portée globale
…
int main(void) {
   int i = 3;        //variable homonyme de portée locale
   int j = 2 * ::i; //j vaut à présent 2, et non pas 6
   …
}
```

ex 2 :

```
int valeur() {       //fonction globale
   return 0;
}
struct A {
   int i;
   int valeur() {    //fonction locale
      return i;
   }
   int global_valeur() {
      return ::valeur();//accède à la fonction globale
   }
};
```

4 - Eléments de base du langage (11)

Ordre de précédence des opérateurs

Priorité	Opérateurs	Associativité
16	`::` (résolution de portée)	->
15	`()` `[]` `->` `.`	->
14	`!` `~` `++` `--` `-` `(`type`)` `*` (indirection) `&` (adresse de) `sizeof`	<-
13	`*` (multiplication) `/` (division) `%` (modulo)	->
12	`+` `-`	->
11	`<<` `>>` (décalages et envois sur flots)	->
10	`<` `<=` `>` `>=` (comparaisons)	->
9	`==` `!=` (comparaisons)	->
8	`&`(sur chaîne de bits)	->
7	`^`(ou exclusif sur chaîne de bits)	->
6	`|`(ou inclusif sur chaîne de bits)	->
5	`&&` (et logique)	->
4	`||` (ou logique)	->
3	`?` (opérateur ternaire de condition)	<-
2	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	<-
1	`,`	->

5 - Instructions (1)

Instructions simple et composée

Instruction simple

Une expression devient une instruction lorsqu'elle est suivie d'un point-virgule "`;`"

```
expression;
```

Instruction composée

Une instruction composée, connue aussi sous le nom de bloc, est une suite d'instructions

```
{
 liste_d_instructions;
}
```

Elle peut remplacer n'importe quelle instruction simple

5 - Instructions (2)

Instructions de choix

Instruction if

syntaxe :

```
if (expression) instruction;

if (expression)
    instruction_alors;
else
    instruction_sinon;
```

Une ambiguïté éventuelle sur un else est levée en le rattachant au if le plus proche

3 - Instructions (3)

Schémas de traduction

Langage de description

```
si (a > b )
    alors x := b;
          b := 0;
fsi
```

```
si (a > b)
    alors si (b > c)
              alors x := a;
          fsi
    sinon x := b
fsi
```

Langage C/C++

```
if (a > b ) {
    x = b;
    b = 0;
}
```

```
if (a > b) {
    if (b > c)
        x = a;
}
else
    x = b;
```

5 - Instructions (4)

Instructions de choix

Instruction switch

syntaxe :

```
switch (expression_entière) {
    case constante_scalaire1 : liste_instructions1;
    case constante_scalaire2 : liste_instructions2;
    …
    case constante_scalairen : liste_instructionsn;
    [default : liste_instructions_par_défaut;]
}
```

Attention !

Une fois terminée l'exécution d'une liste d'instructions d'un case, on continue en séquence dans le case suivant.
Si on ne veut pas que ce genre de phénomène se produise, il faut utiliser l'instruction break

5 - Instructions (5)

Instructions de contrôle d'itération

Instruction while

syntaxe :

```
while (expression) instruction;
```

expression doit être soit de type entier, soit de type pointeur

Au début de chaque itération, expression est évaluée. Si celle-ci a une valeur différente de zéro (ou de NULL), l'itération se poursuit, sinon la boucle est terminée.

Cette boucle sera effectuée zéro ou plusieurs fois.

5 - Instructions (6)

Instructions de contrôle d'itération

Instruction do

syntaxe :

```
do {
 liste_instructions;
} while (expression);
```

Comme précédemment, expression doit être soit de type entier, soit de type pointeur.

La suite d'instructions se trouvant entre les symboles do et while est exécutée de manière répétitive, jusqu'à ce que expression produise une valeur égale à zéro (ou à NULL).

Cette suite d'instructions sera exécutée au moins une fois.

5 - Instructions (7)

Instructions de contrôle d'itération

Instruction for

syntaxe :

for (expression$_1$; expression$_2$; expression$_3$) instruction;

Elle est ainsi équivalente à l'instruction while suivante

```
expression1;
while (expression2) {
    instruction;
    expression3;
}
```

Cette instruction est une forme singulière et pratique de l'instruction while

Son usage est a réserver aux seuls cas où l'on connaît précisément le nombre d'itérations à effectuer

5 - Instructions (8)

Instructions de rupture de séquence

`break`
Elle cause ainsi l'arrêt de la première instruction while, do, for ou switch englobante

`continue`
Elle provoque l'arrêt de l'itération courante et le passage au début de l'itération suivante dans une boucle contrôlée par une instruction while, do ou for

`return`
Cette instruction provoque le retour chez l'appelant de la fonction courante

syntaxe :

```
return [expression];
```

6 - Entrées/sorties

Trois flots standards existent

définis dans `#include <iostream>`

- `cin` (entrée standard)
- `cout` (sortie standard)
- `cerr` (sortie d'erreur standard)

Formatages de sorties

définis dans `#include <iomanip>`

- `endl` : insère une fin de ligne
- `setw(x)` : la prochaine sortie sera sur une largeur de `x` caractères quitte à rajouter des espaces pour compléter
- `setprecision(p)` : définit la précision des flottants avec `p` décimales
- `setfill(c)` : utilisera `c` comme caractère de remplissage
- `left` : la prochaine sortie sera cadrée à gauche (défaut)
- `right` : la prochaine sortie sera cadrée à droite
- `uppercase` : prochaine sortie en majuscule
- `lowercase` : prochaine sortie en minuscule
- `fixed` : prochaine sortie en virgule fixe, ex : `0.00012`
- `scientific` : prochaine sortie en notation ingénieur, ex : `1.2e-4`
- `flush` : vide le tampon de sortie, en assurant que toutes les opérations précédentes sont effectuées

7 - Les tableaux (1)

Un tableau est un ensemble d'éléments de même type désignés par un identificateur unique. Chaque élément est repéré par une valeur entière appelée indice indiquant sa position dans l'ensemble
Leur indiçage démarre toujours à partir de `0`

syntaxe : type identificateur`[`dimension$_1$`]`…`[`dimension$_n$`]`

Ex :
```
int   t[10],                              //tableau de 10 éléments
      m[2][5] = {2,  6, -4, 8, 11,        //tableau à 2 dimensions
                 3, -1,  0, 9,  2},       //2 lignes, 5 colonnes
      x[5][12][7];                        //tableau à 3 dimensions
                                          //de 420 éléments
float f[20]   = {8.0, 4.5, 2.9};          //tableau de 20 éléments
                                          //dont 3 sont initialisés
…
t[2] = 6;
cout << m[1][3] << endl;                  // affiche la valeur 9
```

Il n'y a pas de vérification des indices de tableaux à l'exécution

7 - Les tableaux (2)

L'identificateur du tableau désigne non pas le tableau dans son ensemble, mais plus précisément l'adresse en mémoire du début du tableau. Ceci implique :

qu'un tableau ne pourra jamais être passé en paramètre fixe (passage var valeur) à une procédure, mais toujours en paramètre modifiable (passage par adresse)

qu'il est impossible d'affecter un tableau à un autre :

Ex:
```
int  a[10],
     b[10];
…
a = b      // cette affectation est interdite
```

La dimension d'un tableau peut être omise dans 2 cas :

le compilateur peut en définir la valeur

l'emplacement mémoire correspondant a été réservé

Ex :
```
int  t[]    =  {18,  9,  38};  //tableau  de  3  éléments
char msg[]   =   "Bonjour";   //chaîne   de   caractères
…                             //la   fonction   fct   admet   en
void fct(int [])              //paramètre un tableau d'entiers
```

7 - Les tableaux (3)

Passer un tableau à une fonction

Ex:

```
int somme(int a[], int n) {
   int som = 0;
   for(int i = 0; i < n; i++)
      som += a[i];
   return som;
}
```

```
int main() {
   int t[]= {11, 33, 55, 77};
   cout << somme(t, 4) << end;
return 0;
}
```

Pour les tableaux multidimensionnels, il faut préciser toutes les dimensions sauf la première, facultative

Ex : `int chose(int a[][4][6], int n);`

8 - Les pointeurs (1)

Un pointeur est une variable désignant l'adresse en mémoire d'un objet ainsi que le type exact de l'objet qui se trouve à cette adresse

Déclaration
Un pointeur d'un type donné est déclaré en ajoutant le signe `*` avant le nom du pointeur

Ex :
```
int  *pi;       //pointeur sur entier
char *chaine;   //pointeur sur caractère (chaîne de caractère)
```

L'emploi de variables dynamiques, manipulées par le biais de pointeurs trouve sa justification dans les situations suivantes :

- lorsqu'on ne connaît pas la taille des variables lors de la compilation, ce qui est le cas d'une table dont on ne connaît pas à priori le nombre d'éléments qu'elle est susceptible de contenir
- lorsqu'une variable locale de taille importante est utilisée dans une fonction pouvant être appelée récursivement, risquant de ce fait un débordement de pile
- lorsqu'on désire gérer la mémoire avec précision en adaptant dynamiquement sa taille aux besoins et en réallouant des zones précédemment libérées

8 - Les pointeurs (2)

Utilisation

L'opérateur * sert à accéder au contenu de la zone mémoire pointée

l'opérateur –> sert à accéder aux champs d'une structure pointée

l'opérateur & sert à obtenir l'adresse de la variable à laquelle il s'applique

Ex :

```
int var  = 5,
    t[5] = {0, 2, 3, 6, 8},
    *pi  = NULL,
    *pt  = NULL
…
pi = &var;           //pi pointe sur var
pt = &t[1];          //pt pointe sur le 2ème élément

cout << *pi << endl; //affiche 5
*pi = 4;             //var est modifié

cout << var << endl; //affiche 4
cout << *pt << endl; //affiche 2
pt += 2;             //pt pointe sur le 4ème élément
cout << *pt << endl; //affiche 6
```

8 - Les pointeurs (3)

Arithmétique sur les pointeurs
Les pointeurs peuvent être incrémentés, décrémentés, additionnés ou soustraits. Dans ce cas, leur nouvelle valeur dépend du type sur lequel ils pointent

Ex :
```
int tab[5],
    *p = tab;
...
*(p+3) = 8;     //équivalent à p[3] = 8;
```

Le nom d'un tableau désigne une constante de type pointeur dont la valeur est l'adresse du premier élément du tableau

```
p = tab;        //équivalent à p = &tab[0];
```

Les pointeurs de type `void *` sont utilisés pour pointer sur quelque chose dont on ne connaît pas le type. Les seuls opérateurs autorisés avec les pointeurs génériques sont

- l'affectation
- la conversion de type

8 - Les pointeurs (4)

Les pointeurs sur les fonctions

Le nom d'une fonction est une constante de type pointeur vers fonction. Il est synonyme de son adresse, exactement comme pour les tableaux.
Il ne faut donc pas utiliser l'opérateur de prise d'adresse `&`.
EX :

Définition d'une fonction

```
int f(int x, int y) {return x + y;}
```

Déclaration d'un pointeur sur une fonction

```
int (*pf)(int, int);      //pointeur vers fonction admettant
                          //2 entiers en paramètres
```

`pf` est donc une variable et pas une fonction

affectation

```
pf = f;
```

Appel

```
cout << (*pf)(3, 5) << endl;     //appel de f
```

La valeur affichée est `8`

8 - Les pointeurs (5)

Les pointeurs sur les fonctions

Un pointeur sur une fonction permet de la passer en paramètre afin qu'elle puisse être appelée

EX :

```
double carre(double x) {return x * x;}
void echant(double a, double b, int n, double (*pf)(double x))
{
     for(int i = 0; i < n; i++) {
          double x = a + (b - a) * i / (n - 1);
          cout << "f(" << x << ") =" << (*pf)(x) << endl;
     }
}
…
echant(1, 3, 3, carre);
echant(0, 3.1415, 10, sin);
```

9 - Etapes de la compilation (1)

Le préprocesseur

- L'inclusion de fichiers (`.h` et parfois autres)
- Substitutions lexicales : les macros

La compilation

- Vérification de la syntaxe
- Traduction dans le langage d'assemblage de la machine cible

L'assemblage

- Traduction finale en code machine
- Production d'un fichier objet (.o ou .obj)

L'édition de liens

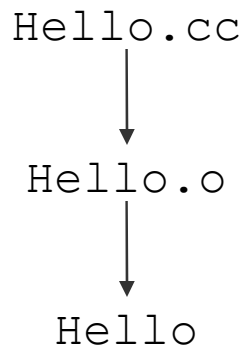
- Unification des symboles internes
- Étude et vérification des symboles externes (.so ou .DLL)

Production de l'exécutable

9 - Etapes de la compilation (2)

Un premier programme `Hello.cc`

```
#include <iostream>
using namespace std;
int main() {
   cout << "Bonjour le monde !" << endl;
   return 0;
}
```



Compilation `g++ -Wall -c Hello.cc`

Edition de liens `g++ -Wall -o Hello Hello.o`

10 - Introduction aux objets (1)

Les techniques de programmation peuvent se regrouper en 4 grandes catégories :

Programmation impérative structurée : Pascal, C, …
Elle est dirigée par les actions

```
typedef struct {
   int wd, ht;
} rect;
int area(rect *r) {
   return r->wd * r->ht;
}
int main() {
   rect r;
   r.wd = 20;
   r.ht = 4;
   printf("Aire: %d\n", area(&r));
}
```

Programmation fonctionnelle : Scheme, Lisp, …

10 - Introduction aux objets (2)

Programmation logique : Prolog, …

Programmation objet : C++, Java, Eiffel, …
Elle est dirigée par les données

```
struct rect {
   int wd, ht;
   void initialiser(int a, int b) {
         wd = a;
         ht = b;
   }
   int area() {
      return wd * ht;
};
int main () {
      rect rec;
      rec.initialiser(20,4);
      cout << rec.area() << endl;
   }
}
```

10 - Introduction aux objets (3)

Encapsulation
L'encapsulation est définie par le regroupement, en une seule entité, des données et des traitements d'un objet.
Les fonctions déclarées ainsi sont des fonctions membres et ne peuvent être appelées qu'en utilisant un élément de type `rect`

Masquage de l'implémentation
Un utilisateur d'un objet n'a pas à connaître la structure interne de cet objet pour l'utiliser. Il a une vue abstraite de l'objet
Un objet peut donc être considéré constitué de 2 parties :

- Une partie privée regroupant l'implémentation de l'objet et inaccessible à l'utilisateur
- Une partie publique représentant l'interface de l'objet avec l'utilisateur

Ces concepts sont réalisés en C++ par la notion de classe

11 - Les classes (1)

Une classe est un type défini par l'utilisateur avec des moyens de protection des données

Elles sont les éléments de base de la programmation C++

Dans une classe on réunit :

Des données :	les données membres
Des fonctions :	les fonctions membres

11 - Les classes (2)

Exemple de déclaration

Fichier `Ratio.h`

```
class Ratio {
   private:
      int num, den;          //numérateur et dénominateur
   public:
      void   affecter(int ,int);
      double valeur_reelle();
      void   ecrire();
      void   inverser();
};
```

11 - Les classes (3)

Accessibilité (droits d'accès aux membres)
Des mécanismes sont proposés permettant de définir précisément l'accessibilité des différents membres.
Par défaut, les membres sont privés

Public

Les éléments qui suivent ce modificateur sont accessibles par n'importe quelle fonction ou opération

Private

Les éléments définis de cette manière dans une classe sont seulement accessibles aux opérations de cette classe

Un objet de cette classe a le droit d'accéder aux membres privés d'un autre objet de la même classe

Protected

Les éléments définis de cette manière dans une classe sont seulement accessibles aux opérations de cette classe et des classes dérivées

Un objet de cette classe a le droit d'accéder aux membres privés d'un autre objet de la même classe

11 - Les classes (4)

Exemple d'implémentation

Fichier `Ratio.cc`

```
#include <iostream>
using namespace std;
#include "Ratio.h"
void Ratio::affecter(int numerateur, int denominateur) {
   num = numerateur;
   den = denominateur;
}
double Ratio::valeur_reelle() {
   return ((double) num ) / den;
}
void Ratio::inverser() {
   int tmp = num;
   num = den;
   den = tmp;
}
void Ratio::ecrire() {
    cout << num << '/' << den;
}
```

11 - Les classes (5)

Où placer les déclarations et les définitions ?

Le plus souvent on place

- Les déclarations dans un fichier `.h`
- Les définitions dans un fichier `.cc`

Plus rarement, et dans le cas de fonctions courtes, on place directement les définitions dans le `.h`

Pourquoi cette habitude ?

- On a plutôt intérêt à garder le fichier `.h` le plus petit possible afin de rendre rapide sa compilation (lors de son inclusion dans les fichiers `.cc`).
- De plus, une fois que le fichier `.cc` a été compilé en un `.o` il n'est pas nécessaire de le recompiler tant qu'on ne modifie pas le `.cc`. On gagne ainsi du temps.
- Permet aussi de ne rendre publique que le `.h` tout en conservant un certain degré de confidentialité dans un `.o` compilé livré au client

11 - Les classes (6)

Instance
Une variable `r` créée avec ce type `Ratio` est appelée instance de la classe

Fichier `testRatio.cc`

```
#include <iostream>
using namespace std;
#include "Ratio.h"

int main() {
   Ratio r;
   r.affecter(3, 4);
   cout << "r = ";
   r.ecrire();
   cout << " = " << r.valeur_reelle() << endl;
   return 0;
}
```

11 - Les classes (7)

Production de l'exécutable

```
//génération d'un objet Ratio.o
g++ -Wall -c Ratio.cc

//génération d'un objet testRatio.o
g++ -Wall -c testRatio.cc

//génération d'un exécutable testRatio
g++ -Wall -o testRatio testRatio.o Ratio.o
```

Exécution

```
./testRatio
```

Résultat produit

```
r = 3/4 = 0.75
```

11 - Les classes (8)

Fichier `Makefile`

```
CFLAGS=-Wall
CC=g++ -c $(CFLAGS)
LD=g++ -o

all: testRatio
testRatio: testRatio.o Ratio.o
	$(LD) $@ $^

testRatio.o: testRatio.cc Ratio.h
	$(CC) $<

Ratio.o: Ratio.cc Ratio.h
	$(CC) $<

clean:
	@rm testRatio *.o *.*~ >& /dev/null
```

11 - Les classes (8)

Constructeurs et destructeurs

En général, il est nécessaire de faire appel à une fonction membre pour attribuer des valeurs aux données d'un objet. C'est ce qui était fait dans l'exemple précédent par appel de la méthode `affecter` :

```
r.affecter(3, 4);
```

Il faut compter sur l'utilisateur pour effectuer l'appel voulu au bon moment. L'absence de procédure d'initialisation peut, dans le cas ou un objet doit s'allouer dynamiquement de la mémoire avant d'être utilisé, conduire à une catastrophe

C++ offre un mécanisme très performant pour traiter ces problèmes : le constructeur.
D'une manière similaire, un objet pourra posséder un destructeur

11 - Les classes (9)

Constructeurs et destructeurs

Constructeur

C'est une fonction spéciale servant à initialiser les objets au moment de leurs créations

- Un constructeur porte toujours le même nom que la classe
- Il peut avoir des paramètres, et éventuellement des valeurs par défaut
- Il n'a jamais de type de retour
- La déclaration a lieu dans le `.h`
- La définition se fait dans le `.cc`

11 - Les classes (10)

Constructeurs et destructeurs

Constructeur

Ex :

Fichier `Ratio.h`

```
class Ratio {
   private:
      int num, den;
   public:
      Ratio(int n, int d);
      Ratio(int n);
      Ratio();
      …
};
```

Fichier `Ratio.cc`

```
#include "ratio.h"
Ratio::Ratio(int n, int d) {
   num = n; den = d;
}
Ratio::Ratio(int n) {
   num = n; den = 1;
}
Ratio::Ratio() {
   num = 0; den = 1;
}
```

Fichier `testRatio.cc`

```
Ratio r1(1, 3);
Ratio r2(4);
Ratio r3;
```

11 - Les classes (11)

Constructeurs et destructeurs

Constructeurs particuliers

Deux constructeurs sont toujours nécessaires dans toute nouvelle classe. Ils sont tellement importants qu'en leur absence, le compilateur tentera de le faire. Et selon un célèbre adage :

”On n'est jamais si bien servi que par soi-même"

Constructeur par défaut

Le constructeur par défaut est un constructeur sans aucun paramètre

Il a pour rôle de créer une instance non initialisée quand aucun autre constructeur fourni n'est applicable

Le constructeur de copie

Le constructeur de copie pour la classe `Ratio` est de la forme :

```
Ratio::Ratio(const Ratio & original) {}
```

Il a pour rôle de créer l'instance courante de façon à ce qu'elle soit une copie identique de celle reçue en paramètre (ici `original`)

Cela sert en particulier lors du passage de paramètre par valeur. Si le constructeur de copie n'est pas défini, il est impossible de passer cette classe en paramètre par valeur

11 - Les classes (12)

Constructeurs et destructeurs

Destructeur

C'est une fonction spéciale s'exécutant au moment de la destruction des objets. Son rôle est de libérer les ressources acquises lors de leurs créations

- Un destructeur porte toujours le même nom que la classe précédé de `~`
- Aucun paramètre
- Et donc pas de surcharge possible
- Pas de valeur de retour
- La déclaration a lieu dans le `.h`
- La définition se fait dans le `.cc`

Ex :

Fichier `Ratio.h`

```
class Ratio {
   …
   ~Ratio();
   …
};
```

Fichier `Ratio.cc`

```
#include "ratio.h"
Ratio::~Ratio() {
   …
}
```

11 - Les classes (13)

Constructeurs et destructeurs

Visibilité

En théorie la visibilité des constructeurs et destructeurs peut être publique ou privée

En pratique, à moins d'avoir de bonnes raisons de faire le contraire, il vaut mieux les rendre publics

Un destructeur privé ne pourra pas être appelé directement, ce qui est rarement le cas

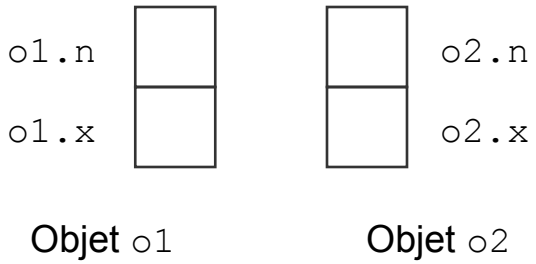
En revanche un constructeur privé rendra impossible la création d'objets de cette classe, que ce soit par une déclaration ou par allocation dynamique.
Ceci peut se justifier dans la cas ou la classe concernée n'est destinée qu'à donner naissance, par héritage, à des classes dérivées

11 - Les classes (14)

Membres statiques

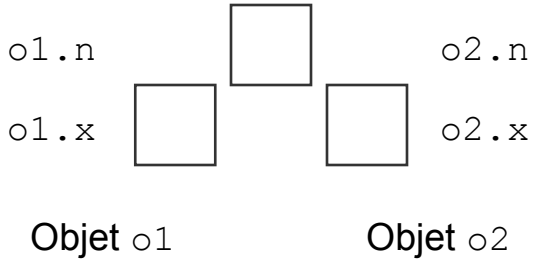
exemple 1 :

```
class ex1 {
   int   n;
   float x;
}
```



exemple 2 :

```
class ex1 {
   static int   n;
   float x;
}
```



11 - Les classes (15)

Membres statiques

Membres données statiques (variables de classe)

Ces variables constituent un espace mémoire partagé par tous les objets de la classe

Le mot `static` précédant la déclaration d'une variable membre désigne celle-ci comme variable de la classe

Membres fonctions statiques

Les fonctions statiques d'une classe constituent l'interface des variables statiques de cette classe.
Elle doivent ne manipuler que des variables de classe

L'appel d'une fonction statique ne nécessite que le nom de la classe à laquelle elle appartient

Une fonction statique peut être appelée même s'il n'existe aucun objet de sa classe

12 - Les paramètres (1)

Passage de paramètre par valeur

C’est le mode de passage le plus simple. La valeur de l’argument réel est recopiée dans l’argument formel

```
void incr(int i) {
   i++;
   cout << "i=" << i << endl;
}

int main() {
   int a = 10;
   incr(a);
   cout << "a=" << a << endl;
   return 0;
}
```

L’exécution du programme produit :

```
i=11
a=10
```

12 - Les paramètres (2)

Passage de paramètre par adresse

L'adresse de l'argument réel est recopiée dans l'argument formel

```
void incr(int *pi) {
   (*pi)++;
   cout << "*pi=" << *pi << endl;
}

int main() {
   int a = 10;
   incr(&a);
   cout << "a=" << a << endl;
   return 0;
}
```

L'exécution du programme produit :

```
*pi=11
a=11
```

12 - Les paramètres (3)

Passage de paramètre par référence

Il met en place un passage de paramètre par adresse
sans que l'on ait à s'en charger.
Il simplifie l'écriture de la fonction concernée et de ses différents appels

```
void incr(int & i) {
   i++;
   cout << "i=" << i << endl;
}

int main() {
   int a = 10;
   incr(a);
   cout << "a=" << a << endl;
   return 0;
}
```

L'exécution du programme produit :

```
i=11
a=11
```

12 - Les paramètres (4)

Passage de paramètre par référence d'un objet constant
Si le paramètre ne doit pas être modifié et que le coût d'une recopie pour un passage par valeur est trop important, on choisit un passage par référence sur un objet constant

```
void truc(const gros_objet & rgo) {
   … = rgo ;     //autorisé
   …
   rgo = … ;     //interdit
}
```

Il devient alors impossible de changer `rgo` mais on peut le consulter librement dans la fonction

13 - Les fonctions membres (1)

Les fonctions membres, comme les fonctions ordinaires peuvent :

Disposer d'arguments par défaut

Ex :
```
int mult(int a = 2, int b = 3) {return a * b;}
```
`mult(3, 4)` délivre 12
`mult(3)` délivre 9
`mult()` délivre 6

Etre surdéfinies
La surdéfinition est la possibilité d'avoir plusieurs fonctions ayant le même nom mais des signatures différentes

Ex :
```
int    plus1(int    i) {return i + 1;}
double plus1(double i) {return i + 1;}
```

La signature d'une fonction est le nombre et le type de chacun de ses arguments

Le prototype d'une fonction est sa signature et le type de sa valeur de retour

13 - Les fonctions membres (2)

Les fonctions membres, comme les fonctions ordinaires peuvent :

- Etre en ligne
  - Pour rendre en ligne une fonction membre on peut
    - Fournir directement la définition de la fonction dans la déclaration même de la classe. Dans ce cas le qualificatif `inline` ne doit pas apparaître

      Ex :
      ```
      class Ratio {
        …
        void inverser() {int tmp = num; num = den; den = tmp;};
      };
      ```

    - Fournir une définition en dehors de la déclaration de classe. Dans ce cas, le qualificatif `inline` doit apparaître à la fois dans la déclaration et dans la définition

      Ex :
      ```
      inline void Ratio::inverser() {
        int tmp = num; num = den; den = tmp;
      }
      ```

  - Cela permet d'économiser le coût d'appel de la fonction en demandant au compilateur de substituer les appels à cette fonction par son code
  - Cela n'a de sens que si la fonction est très courte/rapide
  - Usage classique : les fonctions accesseurs

13 - Les fonctions membres (3)

Les fonctions membres constantes

Dans le cas des objets constants, on doit préciser quelles sont les fonctions membres autorisées à les manipuler

Ex :
```
class Ratio {
   …
   void afficher() const;
   void inverser();
};
…
Ratio       r1;
const Ratio r2;
```

Les instructions suivantes sont licites :
```
r1.afficher();
r2.afficher();
r1.inverser();
```

Celle-ci en revanche est illicite :
```
r2.inverser();
```

Ce mécanisme est parfaitement transposable aux fonctions membres et aux objets `volatile`

14 - Les références (1)

Référence sur une variable

Déclarer une référence `j` sur une variable `i` permet de créer un nouveau nom `j` qui devient synonyme de `i`

La déclaration d'une référence se fait en précisant le type de l'objet référencé, puis le symbole `&`, et le nom de la variable référence qu'on crée

```
int   i = 10; //i est un entier valant 10
int & j = i;  //j est une référence sur i
//à partir d'ici j est synonyme de i, ainsi

j = j + 1; //est équivalent à i = i + 1
```

Attention le = dans la déclaration de la référence n'est pas réellement une affectation puisqu'on ne copie pas la valeur de `i`. En fait, on affirme plutôt le lien entre `i` et `j`. En conséquence

```
int & k = 44; //est donc parfaitement illégal
```

14 - Les références (2)

Auto référence

Toute fonction peut faire référence à l'objet courant par l'intermédiaire du pointeur `this`

EX :
```
Ratio::Ratio(int num, int den) {
    this->num = num;
    this->den = den;
}
```

`this` permet ici de faire la différence entre les membres données et les arguments portant les mêmes noms : `num` et `den`

15 - Les objets (1)

Les objets automatiques et statiques

Les objets automatiques sont créés par une déclaration dans une fonction ou dans un bloc. Ils sont détruits au moment ou l'on quitte la fonction ou le bloc

Les objets statiques sont créés par une déclaration située en dehors de toute fonction avec le mot clé static. Ils sont créés avant l'exécution du main et sont détruits à la fin de son exécution

Les objets temporaires

EX :
```
Ratio r;
r = Ratio(1, 4);
```

L'évalution de l'expression `Ratio(1, 4)` donne lieu à :

la création d'un objet temporaire

l'appel du constructeur `Ratio`

la recopie de cet objet temporaire dans `r`

15 - Les objets (2)

Les objets dynamiques

L'allocation standard des variables, dite allocation statique, donne des variables créées sur la pile et ayant une durée de vie fixée par la portée
L'allocation manuelle de mémoire pour de nouvelles variables, dite allocation dynamique, permet de créer (opérateur `new`) et de détruire (opérateur `delete`) les variables à volonté.
De plus, la zone mémoire d'allocation de telles variables, appelée le tas, permet généralement de stocker des données volumineuses contrairement à la pile.

L'objet existe à partir du `new`
jusqu'à une destruction explicite par `delete`.

EX :
```
Ratio *r;
r = new Ratio(1, 4);
…
(*r).inverser(); //appel de la méthode inverser
r->inverser();   //appel de la méthode inverser
…
delete r;        //destruction de l'objet
```

15 - Les objets (3)

L'initialisation d'objets

Il est très important de distinguer l'initialisation de l'affectation. Il y a initialisation dans 3 cas :

Déclaration d'un objet avec initialiseur

EX :
```
Ratio rA = 1;     // appel du constructeur par recopie
Ratio rB(1, 4);
Ratio rC = rB;    // appel du constructeur par recopie
```

Transmission d'un objet par valeur

Retour par valeur d'un objet

L'initialisation d'un objet provoque toujours l'appel d'un constructeur particulier : le contructeur par recopie

15 - Les objets (4)

Le constructeur par recopie (suite)

EX :
```
class Vecteur {
   private:
      int nElts,
          *t;
   …
   public:
      Vecteur(int nElts) {t = new int[this->nElts = nElts];}
   …
};
…
Vecteur vA(3);
```

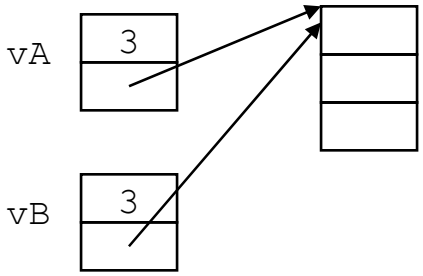
Des problèmes peuvent se poser dès lors qu'un objet contient des pointeurs sur des parties dynamiques. Il y a 2 possibilités.

15 - Les objets (5)

Le constructeur par recopie (suite)

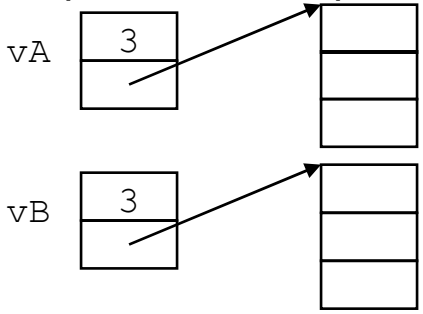
Le constructeur par recopie n'a pas été défini

La déclaration `Vecteur vB = vA;` conduit à la situation suivante :



Le constructeur par recopie est défini

C'est maintenant à lui de prévoir la recopie de tous les membres de l'objet



Dans le cas d'une affectation il faudra surdéfinir l'opérateur =

15 - Les objets (6)

EX :

```
class Vecteur {
      private:
         int  nElts,
              *t;
         …
      public:
         …
         Vecteur(int nElts)       {t = new int[this->nElts = nElts];}

         Vecteur(const Vecteur &v) {
            t = new int[nElts = v.nElts];
            for (int i = 0; i < nElts; i++) t[i] = v.t[i];
         }
         ~Vecteur()               {delete [] t;}
         …
};
```

15 - Les objets (7)

Les objets membres

EX :

```
class Point {
   private:
      int x, y;
   public:
      Point(int, int);
}
class PointEnCouleur {
   private:
      Point p;
      int   couleur;
   public:
      PointEnCouleur(int, int, int);
};
PointEnCouleur::PointEnCouleur(int x, int y, int c):p(x, y) {
    couleur = c;
}
```

Les constructeurs seront appelés dans l'ordre `Point`, `PointEnCouleur` et les destructeurs dans l'ordre inverse

15 - Les objets (8)

Liste d'initialisation
Dans le cas d'objets comportant plusieurs objets membres, la sélection des arguments destinés aux différents constructeurs se fait en séparant chaque liste par une virgule

EX :

15 - Les objets (10)

Liste d'initialisation
Elle peut s'appliquer à n'importe quel membre

EX :
```
class Ratio {
   private:
      int num, den;
   public:
      Ratio(int n, int d):num(n), den(d);
   …
};
```

L'appel du constructeur Ratio provoquera l'initialisation des membres `num` et `den` avec respectivement les valeur `n` et `d`

Cette possibilité peut devenir indispensable pour initialiser un membre donnée qui est une référence.
On ne peut qu'initialiser une référence, jamais lui affecter une nouvelle valeur.

16 - Les chaînes de caractères (1)

Le type caractère `char`, codé sur un octet, permet le stockage d'une valeur numérique entre 0 et 255 représentant le code ASCII d'une lettre

Il existe une convention de représentation des chaînes qui consiste à les considérer comme une suite de caractères terminée par un octet nul (`'\0'`)

La création d'une chaîne suppose de réserver en mémoire un emplacement. Cette réservation peut se faire de 2 façons :

```
char str1[50];              //allocation statique
char *str2 = new char[50];  //allocation dynamique
```

Initialisation d'une chaîne à sa réservation

```
char str3[20] = "Bonjour",
     *str4    = "Le monde";
```

Modification d'une chaîne à sa réservation

```
str1[5]    ='A'; //modifie le 6ème caractère de la chaîne str1
*(str1 + 1)='B'; //modifie le 2ème caractère de la chaîne str1
*(str1 + 2)='C'; //modifie le 3ème caractère de la chaîne str1
```

16 - Les chaînes de caractères (2)

Lecture et écriture simplement avec >> et <<

```
char str[200];
cin >> str;  //lit une suite de lettres jusqu'à un séparateur,
             //les place dans les cases str,
             //puis rajoute un '\0' final
cout << str; //affiche le contenu de str à l'écran
             //et s'arrête grâce au '\0'
```

Il impossible de savoir à l'avance combien de caractères seront saisis, ce qui peut entraîner de graves problèmes.

Il faut donc bien penser à dimensionner, éventuellement surdimensionner correctement la chaîne réceptrice.

```
char str1[4]="pipo";  // incorrect : 4 cases
char str2[5]="pipo";  // correct : 4+1 cases
char str3[]="pipo";   // le plus simple
```

16 - Les chaînes de caractères (3)

Fonctions de manipulation de chaînes

Rajouter `#include <cstring>` dans l'en-tête du fichier

Fonctions de copie

```
char *strcpy(char *dest, const char *src)
char *strncpy(char *dest, const char *src, size_t longueur)
void *memcpy(void *dest, const void *src, size_t longueur)
void *memmove(void * dest, const void *src, size_t longueur)
```

Fonctions de concaténation

```
char *strcat(char *dest, const char *src)
char *strncat(char *dest, const char *src, size_t longueur)
```

Fonctions de comparaison

```
int strcmp(const char *chaine1, const char *chaine2)
int strncmp(const char *chaine1, const char *chaine2, size_t n)
int memcmp(const char *zone1, const char *zone2, size_t n)
```

16 - Les chaînes de caractères (4)

Fonctions de manipulation de chaînes

Fonctions de recherche

```
char *strchr(const void *chaine, int c)
char *strrchr(const char *chaine1, int c)
size-t *strspn(const char *chaine1, const char *chaine2)
size-t *strcspn(const char *chaine1, const char *chaine2)
char *strpbrk(const char *chaine1, const char *chaine2)
char *strstr(const char *chaine1, const char *chaine2)
char *strok(const char *chaine, const char *delimiteurs)
void *memchr(const void *zone, int c, size_t n)
```

Fonctions diverses

```
size-t *strlen(const char *chaine)
void *memset(void *zone, int c, size_t n)
char *strerror(int numErreur)
```

16 - Les chaînes de caractères (5)

Fonctions membres des instances prédéfinies cin et cout
Rajouter `#include <iostream>` dans l'en-tête du fichier

`bool cin.get(char& c)`
lit un seul caractère, le place dans `c` et renvoie `true` si la lecture a réussi

`cin.getline(char *s, int nmax)`
lit une ligne de moins de (`nmax-1`) lettres, et la place dans `s`.
Si la ligne est plus longue, ce qui dépasse est perdu (mais ne provoque pas de faute de mémoire)

```
cout.put(c);
```
écrit le caractère c

```
cin.putback(c);
```
remet le caractère `c` dans `cin`. Il sera donc le prochain caractère lu par un `cin.get(c);`

16 - Les chaînes de caractères (6)

Les chaînes C++ : la classe string
Elle évite les problèmes de débordement de tableau et d'octet nul, mais avec un coup de traitement (en temps et en mémoire) plus élevé.
Rajouter `#include <string>` dans l'en-tête de fichier

Déclarations triviales :

```
string s1;               //s1 contient 0 caractère
string s2 = "New York";  //s2 contient 8 caractères
string s3(60, '*');      //s3 contient 60 astérisques
string s4 = s3;          //s4 contient 60 astérisques
string s5(s2, 4, 2);     //s5 contient «Yo»
```

Lecture et écriture triviales :

```
string s;
cin >> s;
cout << s;
```

Affectation, concaténation, et comparaison de chaînes directement entre instances avec les opérateurs standards
=, +, <, >, <=, ==, etc…

Longueur de la chaîne : méthode `length();`

17 - Les flots (1)

Rajouter `#include <iostream>` dans l'en-tête du fichier

Un flot peut être considéré comme un canal :

- Recevant de l'information, dans le cas d'un flot de sortie
- Fournissant de l'information, dans le cas d'un flot en entrée

L'opérateur `<<` est surchargé dans la classe `ostream` pour prendre en compte l'écriture dans tous les types de base
Il existe 3 instances de la classe `ostream` :

- `cout` : flot de sortie standard
- `cerr` : flot de sortie erreur
- `clog` : identique à `cerr` mais utilisant un tampon intermédiaire

L'opérateur `>>` est surchargé dans la classe `istream` pour prendre en compte la lecture dans tous les types de base
Il existe une instance de la classe `istream` :

- `cin` : entrée standard

17 - Les flots (2)

La classe `ostream` : fonctions membres

`ostream& put(char c);`
écrit un caractère `c`

`ostream& write(const char* s , streamsize n);`
écrit les n premiers caractères de la chaîne `s`

`streampos tellp();`
retourne la position d'écriture

`ostream& seekp(streampos pos);`
définit la position d'écriture à `pos`

`ostream& seekp(streamoff off, ios_base::seekdir dir);`
définit la position d'écriture à `off`, mais en prenant pour référence `dir` :

`ios_base::beg`	début
`bufferios_base::cur`	position courante
`bufferios_base::end`	fin

`ostream& flush();`
vide le tampon de sortie

17 - Les flots (3)

La classe `istream` : fonctions membres

```
streamsize  gcount() const;
```
retourne le nombre de caractères lu par la dernière opération

```
int get();
```
lit un caractère et le retourne

```
istream& get(char& c);
```
lit un caractère et le range dans `c`

```
istream& get(char* s, streamsize n);
```
lit `n` caractères et les range dans `s`

```
istream& get(char* s, streamsize n, char delim);
```
lit `n` caractères jusqu'au délimiteur `delim` et les range dans `s`

```
istream& get(streambuf& sb);
```
lit le flot et range les caractères dans le tampon `sb`

```
istream& get(streambuf& sb, char delim);
```
lit le flot jusqu'à `delim` et range les caractères dans `sb`

```
istream& getline(char* s, streamsize n);
```
lit une ligne d'au maximum `n` caractères

```
istream& getline(char* s, streamsize n, char delim);
```
lit une ligne d'au maximum `n` caractères jusqu'au délimiteur `delim`

17 - Les flots (4)

La classe `istream` : fonctions membres

`istream& ignore(streamsize n = 1, int delim = EOF);`
lit les `n` prochains caractères jusqu'au délimiteur `delim`

`int peek();`
lit un caractère sans l'extraire

`istream& read(char* s, streamsize n);`
lit `n` caractères

`streamsize readsome(char* s, streamsize n);`
lit `n` caractères présents dans le tampon de lecture

`istream& putback(char c);`
replace dans le flot le caractère `c`

`istream& unget();`
replace dans le flot le caractère précédemment lu

`streampos tellg();`
retourne la position de lecture

`istream& seekg(streampos pos);`
définit la position d'écriture à `pos`

`istream& seekg(streamoff off, ios_base::seekdir dir);`
définit la position d'écriture à `off`, mais en prenant pour référence `dir`

17 - Les flots (5)

Statut d'erreur sur un flot

A chaque flot est associé un ensemble de bits formant le statut d'erreur

- `eofbit` : fin de fichier
- `failbit` : la prochaine opération d'entrée/sortir ne peut aboutir
- `badbit` : le flot est dans un état irrécupérable
- `goodbit` : aucun bit d'erreur n'est activé

Il permet de rendre compte du bon ou du mauvais déroulement des opérations sur un flot

Lorsqu'un flot est dans un état d'erreur, aucune opération ne peut aboutir tant que :

- La condition d'erreur n'a pas été corrigée
- Le bit d'erreur correspondant n'a pas été réinitialisé

Une opération d'entrée/sortie a réussi lorsqu'un des bits suivants est positionné :

- `goodbit`
- `eofbit`

17 - Les flots (6)

La classe `ios` : fonctions membres

Accès aux bits d'erreur

Ces fonctions retournent 1 (vrai) en cas de

`bool eof() const;`	fin de fichier
`bool bad() const;`	altération du flot
`bool fail() const;`	échec d'une opération d'e/s
`bool good() const;`	aucun des bits du statut d'erreur n'est activé

`iostate rdstate() const;`
retourne le statut d'erreur

Modification du statut d'erreur

`void clear(iostate state = goodbit);`
active les bits d'erreur correspondant à la valeur `state`

`void setstate(iostate state);`
modifie le bit du statut d'erreur correspondant à la valeur `state`, équivalent à :
`clear(rdstate() | state );`

17 - Les flots (7)

Gestion du formatage

Rajouter `#include <iomanip>` dans l'en-tête du fichier

Chaque flot conserve en permanence son statut de formatage.
Il comporte :

- Un mot d'état
- Des valeurs numériques définissant :
  - Le gabarit
    largeur du champ d'impression
  - La précision numérique
    nombre de chiffres affichés après le point décimal et
    nombre de chiffres significatifs dans le cas de notation exponentielle
  - Le caractère de remplissage
    caractère employé pour compléter un gabarit

17 - Les flots (8)

Gestion du formatage : description du statut de formatage

Nom du champ	Nom du bit	Signification
	`ios::skipws`	saut des espaces (en entrée)
`ios::adjustfield`	`ios::left`	cadrage à gauche (en sortie)
	`ios::right`	cadrage à droite (en sortie)
	`ios::internal`	remplissage après signe ou base
`ios::basefield`	`ios::dec`	conversion décimale
	`ios::oct`	conversion octale
	`ios::hex`	conversion hexadécimale
	`ios::showbase`	affiche indicateur de base (en sortie)
	`ios::showpoint`	affiche le point décimal (en sortie)
	`ios::uppercase`	affiche en majuscule les chiffres hexa (en sortie)
	`ios::showpos`	affiche les nombres positifs précédés de `+` (en sortie)
`ios::floatfield`	`ios::scientific`	notation scientifique
	`ios::fixed`	notation virgule fixe
	`ios::unitbuf`	vide les tampons à chaque écriture
	`ios::stdio`	vide les tampons à chaque écriture sur `stdout` ou `stderr`

17 - Les flots (9)

Gestion du formatage : actions sur le statut de formatage

Manipulateurs non paramétriques

Syntaxe :

```
flot >> manipulateur
flot << manipulateur
```

Manipulateur	Utilisation	Action
`dec`	entrée/sortie	Active le bit de conversion décimale
`hex`	entrée/sortie	Active le bit de conversion hexadécimale
`oct`	entrée/sortie	Active le bit de conversion octale
`ws`	entrée	Active le bit de saut des espaces
`endl`	sortie	Insère un saut de ligne et vide le tampon
`ends`	sortie	Insère un caractère de fin de chaîne
`flush`	sortie	Vide le tampon

17 - Les flots (10)

Gestion du formatage : actions sur le statut de formatage

Manipulateurs paramétriques

Ces manipulateurs sont des fonctions dont le prototype est de la forme :

```
istream & manipulateur(argument)
ostream & manipulateur(argument)
```

Manipulateur	Utilisation	Action
`setbase(int)`	entrée/sortie	Définit la base de conversion
`resetiosflags(long)`	entrée/sortie	Réinitialise tous les bits désignés par l'argument
`setiosflags(long)`	entrée/sortie	Active tous les bits désignés par l'argument
`setfill(int)`	entrée/sortie	Définit le caractère de remplissage
`setprecision(int)`	entrée/sortie	Définit la précision des nombres flottants
`setw(int)`	entrée/sortie	Définit le gabarit

17 - Les flots (10)

Gestion du formatage : actions sur le statut de formatage

Fonctions membres

`setf(long bits)`
active les bits spécifiés par `bits`
`setf(long bits, long champ_de_ bits)`
active les bits spécifiés par `bits` au sein de `champ_de_bits`

`char fill()`
retourne le caractère de remplissage
`char fill(char)`
définit à `c` le caractère de remplissage et retourne l'ancien

`intprecision()`
retourne la précision numérique
`int precision(int p)`
définit à `p` la précision numérique retourne l'ancienne

`int width()`
retourne le gabarit actuel
`int width(int largeur)`
définit à `largeur` le gabarit et retourne l'ancien

17 - Les flots (11)

Connexion d'un flot à un fichier
Rajouter `#include <fstream>` dans l'en-tête du fichier

Les constructeurs des classes `ifstream` et `ofstream` sont :

```
ifstream(const char *filename,
         ios_base::openmode mode = ios_base::in)
ofstream(const char *filename,
         ios_base::openmode mode = ios_base::out)
```

Ecriture :

Déclarer une instance de la classe `ofstream` (output file stream) :
Ex : `ofstream ofile("fichier.txt");`

S'en servir à la place de `cout` :
Ex :
```
char *str = "Hello";
ofile << str << endl;
```

Lecture :

Déclarer une instance de la classe `ifstream` (input file stream) :
Ex : `ifstream ifile("fichier.txt");`

S'en servir à la place de `cin` pour lire (par exemple) des entiers et calculer leur somme :
Ex :
```
int som = 0, i;
while(ifile >> i) som += i;
```

17 - Les flots (12)

Exemples :

Création d'un fichier texte et écriture d'un compte à rebours dedans :

```
{
 ofstream ofile("decompte.txt");
 for(int i = 10; i > 0; i--) ofile << i << endl;
 ofile << "boum!" << endl;
}
//la fermeture du fichier se fait toute seule
//lors de la destruction de ofile
```

Recopie caractère par caractère d'un fichier dans un autre :

```
{
 char c;
 ifstream ifile("source.txt");
 ofstream ofile("destination.txt");
 // tant qu'on arrive à lire un char on le copie dans ofile
 while(ifile.get(&c)) ofile.put(c);
}
```

Il est important de refermer le fichier aussitôt que possible en provoquant la destruction de l'instance de `ifstream` ou de `ofstream`.

17 - Les flots (13)

Modes d'ouverture d'un fichier

Le mode d'ouverture est défini par un état dans lequel chaque bit possède une signification particulière.

Bit de mode d'ouverture	Signification
`ios::in`	Ouverture en lecture (obligatoire pour `ifstream`)
`ios::out`	Ouverture en écriture (obligatoire pour `ofstream`)
`ios::app`	Ouverture en ajout
`ios::ate`	Positionnement en fin de fichier à l'ouverture
`ios::trunc`	Si le fichier existe, son contenu est perdu
`ios::nocreate`	Le fichier doit exister
`ios::noreplace`	Le fichier ne doit pas exister

17 - Les flots (14)

Accès direct dans un fichier

Des fonctions membres permettent l'accès direct :

`streampos tellg()`
retourne la position actuelle dans un fichier `ifstream`

`streampos tellp()`
retourne la position actuelle dans un fichier `ofstream`

`istream& seekg(streampos pos)`
positionne le pointeur de fichier `ifstream` à `pos`

`istream& seekg(streamoff off, ios_base::seekdir dir)`
positionne le pointeur de fichier `ifstream` à `off` par rapport à `dir`

`ostream& seekp(streampos pos)`
positionne le pointeur de fichier `ofstream` à `pos`

`ostream& seekp(streamoff off, ios_base::seekdir dir)`
positionne le pointeur de fichier `ofstream` à `off` par rapport à `dir`

Une constante entière détermine la position servant de référence :

`ios::beg`	début du fichier
`ios::cur`	position courante
`ios::end`	fin de fichier

17 - Les flots (15)

Flux de chaînes : `istringstream` et `ostringstream`

Compromis entre un `string` et un `istream` ou `ostream`

Lecture et écriture dedans avec >> et <<
comme avec `istream` ou `ostream`

Obtention d'un `string` C++ pour une manipulation ultérieure
Utiliser la méthode `str()` pour obtenir le `string` résultant

Exemple :

```
#include <iostream>
#include <sstream> // stringstream !
#include <string>
…
{
  string s("Chaîne"); int n = 10; float pi = 3.14;
  ostringstream oss;
  oss << s << ',' << n << ',' << pi;
  cout << "Résultat : " << oss.str() << endl;
}
```

produit l'affichage : `"Résultat : Chaîne,10,3.14`

18 - Les fonctions amies (1)

L'unité de protection en C++ est la classe.

Une fonction membre peut accéder à tous les membres de n'importe quel objet de sa classe

En revanche, le principe d'encapsulation interdit à une fonction membre d'une classe d'accéder à des données privées d'une autre classe

Exemple :

Soient les classes suivantes :

```
class Vecteur4 {
   private:
      int v[4];
   public:
      int elem(int);
}
int Vecteur4::elem(int i) {
   return v[i];
}
```

```
class Matrice4 {
   private:
      int m[4][4];
   public:
      int elem(int, int);
}
int Vecteur4::elem(int i, int j) {
   return v[i][j];
}
```

18 - Les fonctions amies (2)

On désire pouvoir vérifier si un `Vecteur4` particulier correspond à une ligne d'une `Matrice4` particulière :
on utilise une fonction extérieure compare

```
int compare(const Matrice & m, const Vecteur4 & v) {
   int trouve = 0, discord;
   for (int i = 0; (i < 4) && (! trouve); i++) {
      discord = 0;
      for (int j = 0; (j < 4) && (! Discord); j++)
         discord = (v.elem(j) ! = m.elem(i, j));
      trouve = !discord;
   }
   return trouve;
}
```

- L'inconvénient d'une telle solution est qu'elle nécessite de très nombreux appels de petites fonctions d'accès
- La solution au problème précédent est d'avoir une fonction compare qui soit bien extérieure aux 2 classes mais qui ait des droits spéciaux d'accès aux parties privées des 2 classes
- Une telle situation, pour une fonction existe en C++. C'est une fonction non membre des 2 classes mais considérée comme amies par ces 2 classes

18 - Les fonctions amies (3)

On distingue plusieurs situations d'amitié :

Une fonction indépendante, amie d'une classe

Une fonction indépendante, amie de plusieurs classes

Ex :

```
class Matrice4;
class Vecteur4 {
   …
   public:
      friend int compare(const Matrice4 &, const Vecteur4 &);
}
class Matrice4 {
   …
   public:
      friend int compare(const Matrice4 &, const Vecteur4 &);
}
int compare(const Matrice4 & m, const Vecteur4 & v) {
   …
}
```

18 - Les fonctions amies (4)

Une fonction membre d'une classe, amie d'une autre classe

Ex :

```
class Matrice4;

class Vecteur4 {
   …
   public:
      int compare(const Matrice4 &);
}

class Matrice4 {
   …
   public:
      friend int Vecteur4::compare(const Matrice4 &);
}

int Vecteur4::compare(const Matrice4 & m) {
   …
}
```

18 - Les fonctions amies (5)

Toutes les fonctions membres d'une classe, amies d'une autre classe

Ex :

```
class Vecteur4 {
   friend class Matrice4;
   …
}
```

Dans ce cas, toutes les fonctions membres de la classe `Matrice4` deviennent des amies de la classe `Vecteur4`

19 - La surcharge d'opérateurs (1)

Ex :

```
int     iA, iB = 5, iC = 10;
double  dA, dB = 18.09 dC = 7,11;
iA = iB + iC; // + a pour opérandes 2 entiers iB et iC
dA = dB + dC; // + a pour opérandes 2 réels dB et dC
```

L'opérateur + est utilisé de la même manière pour une addition d'entiers et pour une addition de réels.
On dit qu'il est surchargé (surdéfini)

En C++ ce processus de surcharge d'un opérateur peut-être étendu par l'utilisateur pour effectuer un traitement spécifique dès lors qu'il porte sur au moins un objet

Il existe 45 opérations : `+ - * / = += -= *= /= [] () == …`
Presque toutes ces opérations peuvent être (re)définies sur les nouvelles classes afin d'étendre les possibilités du langage

Seuls `::` , `.` et `.*` ne peuvent être redéfinis

19 - La surcharge d'opérateurs (2)

Règles pour la surcharge d'opérateurs

Le nom de la fonction à surcharger est

`type operator symbole(…)`

où symbole est le symbole représentant l'opérateur

Les opérateurs `=`, `[]`, `()` et `->` doivent être surchargés par une fonction membre

On ne peut changer ni l'arité (unaire, binaire), ni la priorité, ni l'associativité d'un opérateur

Il ne faut pas faire d'hypothèse sur la signification d'un opérateur
EX : la surcharge de `+` et de `=` n'implique pas la surcharge de `+=`

Les opérateurs ne peuvent être surchargés qu'utilisés avec des classes

La surcharge d'un opérateur est liée à la classe d'un de ses opérandes

L'objectif de la surcharge d'opérateur n'est pas de simplifier la vie du programmeur mais bien celle de l'utilisateur de la classe

19 - La surcharge d'opérateurs (3)

Surcharge au moyen d'une fonction amie
Les opérateurs surchargés au moyen d'une fonction amie prennent un argument de type référence sur classe

EX :

```
class Complexe {
 private:
   double r, i;
 public:
   …
   friend Complexe operator + (const Complexe &, const Complexe &);
};

Complexe operator + (const Complexe & c1, const Complexe & c2) {
   Complexe c;
   c.r = c1.r + c2.r;
   c.i = c2.i + c2.i;
   return c;
}
```

19 - La surcharge d'opérateurs (4)

Surcharge au moyen d'une fonction membre
Le premier argument de la fonction précédente se trouve être transmis implicitement : ce sera l'objet ayant appelé la fonction membre

EX :

```
class Complexe {
 private:
   double r, i;
 public:
   …
   Complexe operator + (const Complexe &);
};

Complexe Complexe::operator + (const Complexe & c2) {
   Complexe c;
   c.r = r + c2.r;
   c.i = i + c2.i;
   return c;
}
```

19 - La surcharge d'opérateurs (5)

Surcharge des opérateurs `++` et `--`

Il faut distinguer la pré/post incrémentation/décrémentation

Version préfixée

syntaxe : classe operator opérateur`()`

Version postfixée

syntaxe : classe operator opérateur`(int)`

19 - La surcharge d'opérateurs (6)

Surcharge des opérateurs ++ et −−
avec une fonction amie
EX :

```
class Complexe {
 private:
   double r, i;
 public:
   …
   friend Complexe operator ++ (Complexe &));
   friend Complexe operator ++ (Complexe &, int);
};

Complexe operator ++ (Complexe & c) {
   c.r++; c.i++;
   return c;
}

Complexe operator ++ (Complexe & c, int n) {
   Complexe cTmp = c;
   c.r++; c.i++;
   return cTmp;
}
```

19 - La surcharge d'opérateurs (7)

Surcharge des opérateurs ++ et −−
avec une fonction membre
EX :

```
class Complexe {
 private:
   double r, i;
 public:
   …
   Complexe operator ++ ();
   Complexe operator ++ (int);
};

Complexe Complexe::operator ++ () {
   r++; i++;
   return *this;
}

Complexe Complexe::operator ++ (int n) {
   Complexe cTmp = *this;
   r++; i++;
   return cTmp;
}
```

19 - La surcharge d'opérateurs (8)

Surcharge de l'opérateur = **avec une fonction membre**

Le problème est voisin de celui de la construction par recopie.
Quelques nuances sont cependant à prendre en considération.
Dans le cas d'une affectation

```
a = b
```

il faut :

Libération mémoire de l'emplacement pointé par `a`

Création dynamique d'un nouvel emplacement dans lequel sont recopiés les valeurs de l'emplacement pointé par `b`

Mise en place des valeurs des membres données de `a`

Toujours `return *this;` à la fin des différents opérateurs d'affectations , car il faut renvoyer une référence sur l'instance qu'on vient de modifier, c'est-à dire `*this`

19 - La surcharge d'opérateurs (9)

Surcharge de l'opérateur = **avec une fonction membre**

EX :

```
class Vecteur {
 private:
   int n, *t;
 public:
   …
   Vecteur & operator = (const Vecteur & v);
};

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

19 - La surcharge d'opérateurs (10)

Surcharge de l'opérateur `[]` **avec une fonction membre**

EX 1 :

```
class Vecteur {
 private:
   int n, *t;
 public:
   …
   int & operator [] (int);
};

int & Vecteur::operator [] (int i) {
   return t[i];
}
…
Vecteur v;
```

Pour que `v[i]` soit une lvalue, il est donc nécessaire que la valeur de retour soit transmise par référence

19 - La surcharge d'opérateurs (11)

Surcharge de l'opérateur `[]` **avec une fonction membre**

EX 2 :

```
class Vecteur {
 private:
   int n, *t;
 public:
   …
   int operator [] (int) const;
};

int Vecteur::operator [] (int i) {
   return t[i];
}
…
Vecteur v;
```

`v[i]` est ici une rvalue, ce qui interdit toute modification

19 - La surcharge d'opérateurs (12)

Surcharge des opérateurs << et >> **avec une fonction amie** pour les types définis par l'utilisateur

Ces opérateurs doivent recevoir un flot en premier argument, ce qui empêche d'en faire des fonctions membres

La valeur de retour est obligatoirement la référence au flot concerné

Le prototype de ces fonctions est de la forme :

```
ostream & operator << (ostream &, Classe);
ostream & operator << (ostream &, const Classe &);
istream & operator >> (istream &, Classe &);
```

EX :

```
class Complexe {
   private:
      double r, i;
   public:
      …
      friend istream & operator >> (istream &, Complexe &);
      friend ostream & operator << (ostream &, const Complexe &);
};
```

19 - La surcharge d'opérateurs (13)

Surcharge des opérateurs << et >> **avec une fonction amie**

```
istream & operator >> (istream & is, Complexe & cx) {
   // lecture d'un nombre sous la forme a+ib
   char   c;
   is >> cx.r;
   cx.i = 0;
   if ((c = is.get()) == '+')
      if ((c = is.get()) == 'i')
         is >> cx.i;
      else {
         is.putback(c);
         is.clear(ios::badbit);
      }
   else {
         is.putback(c);
         if (c != '\n') is.clear(ios::badbit);
   }
   return is;
}
```

19 - La surcharge d'opérateurs (14)

Surcharge des opérateurs << et >> **avec une fonction amie**

```
ostream & operator << (ostream & os, const Complexe & cx) {
   os << cx.r;
   if (cx.i > 0) os << "+i" << cx.i;
   return os;
}

int main() {
   Complexe c;

   cin >> c;
   if (cin.bad())
      cout << "ERREUR DE LECTURE" << endl;
   else
      cout << c << endl;

   return 0;
}
```

19 - La surcharge d'opérateurs (15)

Forme canonique d'une classe

Dès lors qu'une classe dispose de pointeurs sur des parties dynamiques, le constructeur par recopie par défaut ainsi que l'opérateur d'affectation se montrent insatisfaisants

Il devient nécessaire de munir la classe d'au moins les 4 fonctions membres suivantes :

EX :

```
class Ratio {
   …
 public:
   Ratio(…);                                  //constructeur

   ~Ratio();                                  //destructeur

   Ratio(const Ratio &);                      //constructeur par recopie

   Ratio & operator = (const Ratio &);        //opérateur d'affectation
};
```

20 - Les conversions (1)

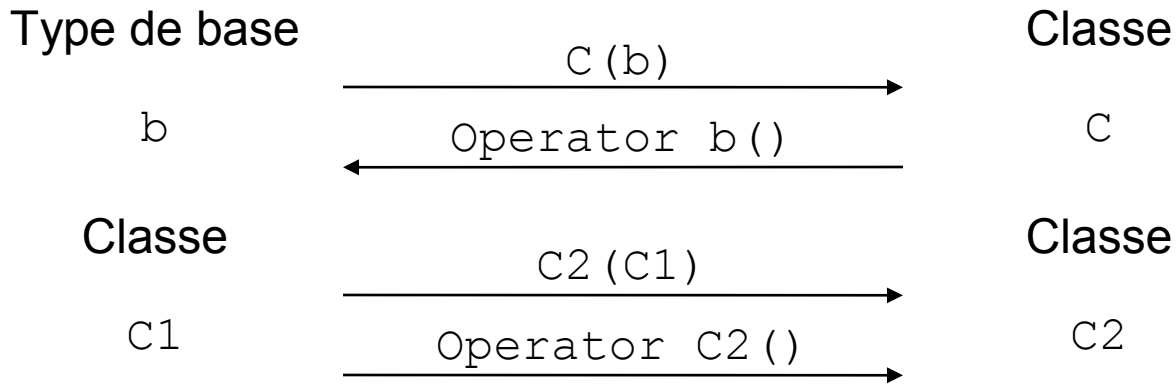
On distingue deux types de conversions :

Les conversions explicites lorsqu'on fait appel à l'opérateur de transtypage `()`

Les conversions implicites mises en place par le compilateur.
On les rencontre dans :

- Les affectations
- Les appels de fonctions
- Les expressions

Les différentes possibilités de conversions définies par l'utilisateur se résument à :



20 - Les conversions (2)

L'opérateur de transtypage (cast) doit toujours être défini comme une fonction membre et le type de la valeur de retour ne doit pas être mentionné

Les règles de conversions employées par le compilateur dans les expressions sont :

- La recherche de la surdéfinition de l'opérateur d'affectation est d'abord effectuée
- En son absence, les opérandes sont converties
- Les conversions mises en place par l'utilisateur ne sont mises en œuvre que lorsque c'est nécessaire
- Une erreur est générée en cas d'ambiguïté
- Une conversion dégradante n'est jamais mise en œuvre de manière implicite

20 - Les conversions (3)

Conversion d'un type classe en un type de base

Fichier `Ratio.h`

```
class Ratio {
   …
   public:
      …
      operator double();
};
```

Fichier `Ratio.cc`

```
#include "ratio.h"
Ratio::operator double() {
   return (double) num / den;
}
```

Fichier `testRatio.cc`

```
Ratio  r1(1, 3), r2(2, 5);
double d1, d2;
…
d1 = (double) r1;  // appel explicite de double()
d1 = double (r1);  // autre écriture possible
d2 = r2;           // appel implicite de double()
```

20 - Les conversions (4)

Conversion d'un type de base en un type classe

Fichier `Ratio.h`

```
class Ratio {
  …
  public:
     …
     Ratio(int = 0, int = 1);
     Ratio & operator = (int);
};
```

Fichier `Ratio.cc`

```
#include "Ratio.h"
Ratio::Ratio(int num, int den) {
   this->num = num;
   this->den = den ? den : 1;
}
Ratio & Ratio::operator = (int num) {
   this->num = num;
   this->den = 1;
   return *this;
}
```

Fichier `testRatio.cc`

```
Ratio r(1);   // appel du constructeur à 1 argument
…
r = 4;        // appel de l'opérateur d'affectation
              // ou appel du constructeur à 1 argument si l'opérateur
              // = n'avait pas été surdéfini
```

20 - Les conversions (5)

Conversion d'un type classe en un autre type classe

Opérateur de transtypage

Fichier `Classes.h`

```
class Ratio;
class Point {
  private:
   int x, y;
  …
  public:
   operator Ratio ();
};
class Ratio {
  …
  public:
   …
   friend Point::operator Ratio();
};
```

20 - Les conversions (6)

Conversion d'un type classe en un autre type classe

Opérateur de transtypage

Fichier `Classes.cc`

```
#include "Classes.h"

Point::operator Ratio() {
  Ratio r;
  r.num = x;
  r.den = y;
  return r;
}
```

Fichier `testConversions.cc`

```
Point p1(18, 9), p2(7, 11);
Ratio r;
…
r = (Ratio) p1;  // conversion explicite
r = p2;          // conversion implicite
```

20 - Les conversions (7)

Conversion d'un type classe en un autre type classe

Opérateur d'affectation

Fichier `Classes.h`

```
class Point;
class Ratio {
  private:
   int x, y;
  …
  public:
   …
   Ratio & operator = (const Point &);
};
class Point {
  …
public:
   …
   friend Ratio & Ratio::operator = (const Point &);
};
```

20 - Les conversions (8)

Conversion d'un type classe en un autre type classe

Opérateur d'affectation

Fichier `Classes.cc`

```
#include "Classes.h"

Ratio & Ratio::operator = (const Point &) {
  return Ratio(x , y);
}
```

Fichier `testConversions.cc`

```
Point p1(18, 9);
Ratio r;
…
r = p2;          // appel de l'opérateur d'affectation
```

21 - Les exceptions (1)

Un programme quel qu'il soit peut soulever des erreurs. Il existe alors deux stratégies pour les gérer :

Localement (comme en C) :
On injecte du code de détection/traitement d'erreur là où c'est nécessaire
Ex :

```
double inverse(double x) {
   if (x == 0) {
      fprintf(stderr, "Division par zero !!!\n");
      exit(1); // ARRET du programme en force !
   }
   return 1 / x;
}
```

Pas satisfaisant car provoque l'arrêt du programme et ne permet pas de contourner l'erreur

L'idéal serait de délocaliser le traitement de l'erreur dans la fonction qui appelle `inverse()`

Séparation des préoccupations : gestion d'exceptions
On dissocie du code applicatif, le code lié aux erreurs qu'il est susceptible d'engendrer (signalement, gestion). Il devient possible de traiter une erreur non pas là où elle survient, mais plus tard (dans le code appelant)

21 - Les exceptions (2)

Les langages récents (C++, Java, etc) apportent une façon élégante de gérer l'apparition puis le traitement des erreurs qui surviennent au moment de l'exécution des programmes

Constatation : le lieu du programme où se produit l'erreur n'est généralement pas celui où il est possible de prendre des mesures pour la traiter

Une exception est un signal qui indique que quelque chose d'anormal s'est produit. Le traitement d'erreur est découpé en deux parties :

- Son déclenchement : instruction `throw`
  - Lancer une exception consiste à signaler une anomalie
- Son traitement : deux instructions inséparables `try` et `catch`
  Capturer consiste à signaler que l'anomalie va être gérée
  (que l'on va essayer de rattraper la "casse")

Ce mécanisme est désigné sous le nom de "gestion des exceptions".
Il permet de réaliser proprement des tentatives successives visant à éviter l'erreur avant d'abandonner.
Par exemple, si c'est une forme d'allocation de mémoire qui provoque l'échec, il est possible de tenter de libérer certains objets peu utiles avant de recommencer l'allocation, et si cela ne suffit pas, alors seulement provoquer un échec.

21 - Les exceptions (3)

Une fonction qui détecte une erreur peut se contenter de lancer une exception

Cette exception interrompt l'exécution de la fonction, et un gestionnaire d'exception approprié est recherché

La recherche du gestionnaire suit le même chemin que celui utilisé lors de la remontée des erreurs : à savoir la liste des appelants

La première fonction appelante qui contient un gestionnaire d'exception approprié prend donc le contrôle, et effectue le traitement de l'erreur

Si le traitement est complet, le programme reprend son exécution normale

Dans le cas contraire, le gestionnaire d'exception peut relancer l'exception (auquel cas le gestionnaire d'exception suivant est recherché) ou terminer le programme

21 - Les exceptions (4)

Le mécanisme des exceptions du C++ garantit que tous les objets de classe de stockage automatique définis dans les blocs `try` sont détruits lorsque l'exception qui remonte sort de leurs portées

Ainsi, si toutes les ressources sont encapsulées dans des classes disposant d'un destructeur capable de les détruire ou de les ramener dans un état cohérent, la remontée des exceptions effectue automatiquement le ménage

De plus, les exceptions peuvent être typées, et caractériser ainsi la nature de l'erreur qui s'est produite. Ce mécanisme est donc strictement équivalent en termes de fonctionnalités aux codes d'erreurs utilisés en C

21 - Les exceptions (5)

Lancement/interception d'une exception

L'objet construit pour lancer l'exception est également détruit lorsqu'on quitte la fonction qui l'a lancée à l'aide de `throw`. Le compilateur effectue donc une copie de cet objet pour le transférer au premier bloc `catch` capable de le recevoir. Cela implique qu'il y ait un constructeur de copie

Il faut décrire l'erreur à l'aide d'une classe quelconque

Ex :

```
class Erreur {
   public:
      string cause; //chaîne spécifiant la cause de l'exception
      // Le constructeur
      Erreur(string c) : cause(c) {}
      // Le constructeur de copie
      // Il est utilisé par le mécanisme des exceptions
      Erreur(const erreur &source) : cause(source.cause) {}
};
```

21 - Les exceptions (6)

Lancement/interception d'une exception

Il faut lancer l'exception à l'aide de throw

```
double inverse(double x) throw (Erreur) {
   if (x == 0)
      throw Erreur("Division par zero !!!\n");
   else
      return 1 / x;
}
```

Il faut intercepter l'exception à l'aide de try et catch

```
void exemple() {
   double x;
   cout << "Donnez un réel";
   cin >> x;
   try {
      cout << "1/" << x << "=" << inverse(x) << endl;
   }
   catch(Erreur & e) {
      cerr << "ERREUR: "<< e.cause << endl;
   }
}
```

21 - Les exceptions (7)

Remontée des exceptions

Si, lorsqu'une exception se produit dans un bloc `try`, il est impossible de trouver le bloc `catch` correspondant à la classe de cette exception, il se produit une erreur d'exécution

La fonction prédéfinie `std::terminate` est alors appelée. Elle se contente d'appeler une fonction de traitement de l'erreur, qui elle-même appelle la fonction `abort` de la librairie C. Cette fonction termine en catastrophe l'exécution du programme fautif en générant une faute (les ressources allouées par le programme ne sont donc pas libérées, et des données peuvent être perdues)

Ce n'est généralement pas le comportement désiré

Il est ainsi possible de le modifier en changeant la fonction appelée par `std::terminate`

21 - Les exceptions (8)

Remontée des exceptions

Ex :

```
void mon_gestionnaire(void) {
   cout << "Exception non gérée reçue !" << endl;
   cout << "Je termine le programme proprement..." << endl;
   exit(-1);
}

int lance_exception(void) {
   throw 2;
}

int main(void) {
   set_terminate(mon_gestionnaire);
   try {
      lance_exception();
   }
   catch (double d) {
      cout << "Exception de type double reçue : " << d << endl;
   }
   return 0;
}
```

21 - Les exceptions (9)

Au moment où on constate l'erreur :

- On crée une instance d'un type choisi, si possible dérivée de `std::exception` ou de l'une de ses sous-classes
- On initialise cette instance de manière à donner le plus d'informations possibles sur l'erreur qui s'est produite
- On envoie cette instance avec `throw`
- L'instance remonte la pile des fonctions appelantes, jusqu'à trouver un `catch` qui intercepte ce type choisi
- Le bloc correspondant du `catch` est exécuté
- Si aucun bloc `catch` n'est rencontré alors c'est le gestionnaire d'exception par défaut qui sera utilisé (celui là existe toujours!). En général, il provoque la fin du programme avec un message d'erreur signalant qu'une exception est survenue (mais sans plus de détail puisqu'il n'a aucune raison de savoir comment traiter cette exception).

Il est possible de mettre plusieurs `catch` pour des types différents sur un même `try`

21 - Les exceptions (10)

Un bon programme devrait dériver ses propres classes d'exceptions de la classe `std::exception` ou d'une des ses classes dérivées…
Hélas l'information sur les classes prédéfinies est difficile à trouver.
Le plus simple consiste à aller lire les fichiers include du compilateur

Attention à bien utiliser un passage par référence pour les instances de `std::exception` et de ses dérivées sinon la fonction virtuelle `what()` ne sera pas appelée sur le bon type !

Il est possible (si cela a un sens) de redéclencher la même exception dans un `catch` après avoir réalisé une action :

```
try {…}
catch(type t) {action; throw;}
```

Dans ce cas, elle continue sa remontée dans la pile jusqu'à trouver un nouveau `catch` valable

Il est aussi possible d'attraper n'importe quelle exception en indiquant "…" à la place du type d'exception à attraper :

```
catch(…) { }
```

21 - Les exceptions (11)

Lancement d'exception dans un constructeur

Le lancement d'une exception dans un constructeur est la seule solution pour signaler une erreur lors de la construction d'un objet

Lorsqu'une exception est lancée à partir d'un constructeur, la construction de l'objet échoue. Ce comportement soulève le problème des objets partiellement initialisés, pour lesquels il est nécessaire de faire un peu de nettoyage à la suite du lancement de l'exception

Le C++ dispose donc d'une syntaxe particulière pour les constructeurs des objets susceptibles de lancer des exceptions.

- Elle permet simplement d'utiliser un bloc `try` pour le corps de fonction des constructeurs
- Les blocs `catch` suivent alors la définition du constructeur, et effectuent la libération des ressources que le constructeur aurait pu allouer avant que l'exception ne se produise

21 - Les exceptions (12)

Lancement d'exception dans un constructeur

Le comportement du bloc `catch` des constructeurs avec bloc `try` est différent de celui des blocs `catch` classiques.
Les exceptions ne sont normalement pas relancées une fois qu'elles ont été traitées

Il faut utiliser explicitement le mot-clé `throw` pour relancer une exception à l'issue de son traitement

Dans le cas des constructeurs avec un bloc `try` cependant, l'exception est systématiquement relancée

Le bloc `catch` du constructeur ne doit donc prendre en charge que la destruction des données membres partiellement construites, et il faut toujours capter l'exception au niveau du programme qui a cherché à créer l'objet

21 - Les exceptions (13)

Lancement d'exception dans un constructeur

Ex :

Fichier `Ratio.h`

```
#include <exception>
#include <string>

using namespace std;

class Ratio {
   …
   public:
      Ratio(int num = 0, int den = 1) throw (Erreur);
   …
}


class Erreur: public exception {
   private:
      string cause;
   public:
      Erreur(string c) throw() : cause(c) {}
      ~Erreur() throw() {}
      const char* what() const throw() {return cause.c_str();}
};
```

21 - Les exceptions (14)

Lancement d'exception dans un constructeur

Ex :

Fichier `Ratio.cc`

```
Ratio::Ratio(int num, int den) throw (Erreur)
try {
   if (den == 0) throw Erreur("Numérateur nul");
   this->num = num;
   this->den = den;
   reduire();
}
catch (Erreur & e) {
   cout << "ERREUR : " << e.what() << endl;
}
```

Fichier `testRatio.cc`

```
…
try {
   r = Ratio(4, 0);
}
catch (...) {
   cout << "Exception interceptée" << endl;
}
…
```

22 - La généricité (1)

La généricité est un mécanisme qui permet d'écrire un modèle de traitement

Le code source comporte alors une ou plusieurs inconnues, correspondant à un ou plusieurs types de données

Le code est compilé lors de l'instanciation du modèle

Le principe est donc très simple, la syntaxe un peu moins

Dans le cas des classes, une variable générique se comporte comme tout autre variable

La généricité peut porter sur

- Les fonctions
  Elles peuvent être surchargées et surdéfinies
- Les classes
  Elles ne peuvent être que surdéfinies

22 - La généricité (2)

Considérons la fonction suivante :

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

en supposant qu'on veuille comparer des flottants, on doit écrire une nouvelle fonction :

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

Ces 2 fonctions effectuent le même traitement, mais sur des types de paramètres et de résultats différents

Cette double écriture est une perte de temps, une source d'erreur, et révèle que la fonction `min` ne dépend pas du type de ce que l'on compare

Il est ainsi possible d'écrire en C++ une seule fois la fonction min de manière à ce qu'elle soit utilisable pour comparer toute paire de variables d'un même type

A l'aide d'une unique définition, comportant des paramètres de type, on décrit une famille de fonctions et de classes. Ce concept porte le nom de patrons, modèles ou encore templates

22 - La généricité (3)

Fonctions génériques (patrons de fonctions)

L'exemple précédent peut s'écrire :

Définition du modèle

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

Utilisation du modèle

```
int   a, b;
float c, d;
…
cout << "Minimum de " << a << " et " << b << " = " << min(a, b);
cout << "Minimum de " << c << " et " << d << " = " << min(c, d);
```

22 - La généricité (4)

Fonctions génériques (patrons de fonctions)

Un patron de fonction peut comporter plusieurs paramètres de type séparés par une virgule, chacun étant précédé du mot `class`

Les définitions (et pas seulement les déclarations) des patrons se font dans les fichiers `.h`

L'instanciation se fait dans les fichiers `.cc`

Le compilateur crée (instancie) autant de fonctions différentes que nécessaire. Dans l'exemple précédent, il crée 2 fonctions `min()` différentes :
`min(int, int)` et `min(float, float)`

Limitations des fonctions génériques

Les opérateurs utilisés dans une fonction générique doivent être définis pour les types classes manipulés. Ainsi pour pouvoir utiliser `min(r1, r2)`, `r1` et `r2` étant des objets `Ratio`, il est indispensable que l'opérateur `<` soit surdéfini pour la classe `Ratio`

Pour imposer qu'un paramètre de type corresponde à un pointeur, on le fait suivre de `*`

Ex :
```
template <class T> T min (T *a, T *b) {
        return (*a < *b) a : b;
    }
```

22 - La généricité (5)

Surdéfinition, spécialisation et fonctions génériques

On peut surdéfinir un patron de fonction de même que le spécialiser

EX :

```
template <class T> T min (T a, T b) {
   return (a < b) ? a : b;
}
template <class T> T min (T a, T b, T c) { //surdéfinition
   return min(min(a, b), c);
}
char *min (char *a, char *b) {              //spécialisation
   return (strcmp(a, b) <= 0) ? a : b;
}
```

La seule règle à respecter est :
L'appel d'une fonction ne doit pas conduire à une ambiguïté. Un seul patron ou fonction doit pouvoir être utilisé à chaque fois

Lorsqu'une fonction et un patron peuvent convenir, la priorité est donnée à la fonction

22 - La généricité (6)

Les classes génériques (patrons de classes)
Elles permettent au compilateur, en écrivant une seule fois la définition de la classe, de l'adapter automatiquement à différents types
EX :
fichier `Vecteur.h`

```
template <class T, int n> class VectN {
   private:
      T t[n];
…
};
```

fichier `testVecteur.cc`

```
#include "Vecteur.h"
…
const int max = 100;
int max2 = 20;
VectN <int, 10> t;
VectN <char, max> tc;
VectN <Ratio, max2> tt;  //illégal car max2 n'est pas constant
```

22 - La généricité (7)

Les classes génériques (patrons de classes)
La définition des fonctions membres diffère selon qu'elles sont en ligne ou non

Définition en ligne dans le `.h`
Ex :

```
T & operator[](int i) {return t[i];};
const T & operator[](int i) const {return t[i];};
```

Définition directement après la classe mais toujours dans le `.h` en spécifiant le contexte d'application avec
classe<paramètres_du_patron>::fontion()
Ex :

```
template<class T> T & VectN<T>::operator[](int i){
   return t[i];
};
template<class T> const T & VectN<T>::operator[](int i)const{
   return t[i];
};
```

Il n'est ainsi pas possible de livrer à un utilisateur une classe patron compilée puisqu'il faudra lui fournir le `.h` contenant les définitions

22 - La généricité (8)

Les classes génériques (patrons de classes)

Définition du patron dans le .h et instanciation dans les .cc

Le compilateur crée/instancie autant de classes différentes que nécessaire

Ici aussi, la partie `<…>` fait partie du nom de la classe. En particulier, on retiendra que `VectN` n'est pas un nom de classe, mais que par contre `VectN<int,5>` en est un

Les paramètres de types doivent être connus au moment de la compilation.

Les paramètres expressions qui ne sont pas des types (ici `n`) doivent être des expressions constantes dont la valeur est connue au moment de la compilation

Il est possible de spécialiser un patron de classe :

- En spécialisant pour les valeurs de tous les paramètres
  EX : `template <Ratio, 10> class VectN`
- En spécialisant une fonction membre ou une classe
  EX : `class VectN <Ratio>`

22 - La généricité (9)

Les classes génériques (patrons de classes)

Spécialisation d'un patron de classe :

EX :

```
// Définition d'un patron de classe
template <class T1, class T2, int I> class C {};

// Spécialisation n°1 de la classe
template <class T, int I> class C<T, T*, I> {…};
// Spécialisation n°2 de la classe
template <class T1, class T2, int I> class C<T1*, T2, I> {…};
// Spécialisation n°3 de la classe :
template <class T> class C<int, T*, 5> {…};
// Spécialisation n°4 de la classe
template <class T3, class T4, int I> class C<T3, T4*, I> {…};

//instanciations
C <int, float, 10>     c;

C <char, char *, 5>    c1;
C <int *, float, 10>   c2;
C <int, double *, 10>  c3;
C <float, int * , 10>  c4;
```

22 - La généricité (10)

Les classes génériques (patrons de classes)
Les fonctions amies permettent d'accéder à la partie privée et protégée des classes définies à partir du patron
On distingue 3 déclarations d'amitié :

Amitié avec des fonctions ou des classes ordinaires
La déclaration et la définition se font comme précédemment

Amitié avec un patron de classe ou un patron de fonction

```
template <class T> ma_classe {
   …
   template <class T> friend class une_classe <T>;
   template <class T> friend type fonction(une_classe <T>);
};
```

Il faut procéder à une déclaration préalable, faute de quoi l'édition de liens ne pourra pas s'effectuer correctement

22 - La généricité (11)

Les classes génériques (patrons de classes)

Amitié avec un patron de classe ou un patron de fonction

EX : la déclaration et la définition doivent se trouver dans le `.h`

```
template <class T> class PVecteur;
template <class T> PVecteur<T> operator - (const PVecteur<T> &);
…
template <class T> class Pvecteur {
   …
   friend PVecteur<T> operator -<> (const PVecteur<T> & v);
   …
}
…
template <class T> PVecteur<T> operator - (const PVecteur<T> & v) {
   PVecteur<T> vTmp = v;
   for (int i = 0; i < vTmp.n; i++) vTmp.t[i] *= -1;
   return vTmp;
}
```

22 - La généricité (12)

Les classes génériques (patrons de classes)

Amitié avec une instance de patron

```
template <class T> ma_classe {…};
```

Deux cas sont possibles

Amitié avec une instance sur un type connu de la classe l'autorisant

```
friend class une_classe <int>;
friend type une_fonction(float);
```

Amitié avec une instance sur un type générique de la classe l'autorisant

```
friend class une_classe <T>;
friend type une_fonction(T);
```

23 - L’héritage (1)

Quand une classe est testée et fonctionne, elle fonctionnera toujours. On peut donc la réutiliser

Une classe qui fonctionne est généralement compilée, ce qui garantit qu’elle ne sera pas modifiée afin de ne pas perturber les applications actuelles utilisatrices de cette classe

Une classe compilée est fermée car on ne peut pas remettre en cause son code pour prendre en compte de nouvelles applications

Une classe est ouverte en ce sens qu’elle peut servir de base à la construction de nouvelles classes

Des remarques précédentes on peut conclure que la programmation par objets permet d’avoir à la fois un logiciel ouvert et fermé

23 - L'héritage (2)

Réutilisation de classes par composition
Dans une classe, un membre peut être

- Une fonction
- Une donnée d'un type de base
- Un objet instance d'une autre classe

La technique consiste à construire une classe par composition (encapsulation ou agrégation) à partir d'autres classes en définissant des membres instances de classes existantes
Ex :

```
class Jeu {…}
class Damiers {…}
   …

class JeuADamiers {
   private:
      Jeu    jeu;              // membre instance de Jeu
      Damier *damier;          // membre instance de Damier
      …
}
```

23 - L'héritage (3)

Réutilisation de classes par héritage
La technique consiste à ajouter des propriétés à une classe existante pour la spécialiser et obtenir une nouvelle classe réalisant une extension de comportement

Ex :

```
class Jeu {
   …
   public:
      Jeu(int);
      void unCoup();
};
class JeuADamiers : public Jeu {
   …
   public:
      JeuADamiers(int);
      void unCoup(int);
};
class Echecs : public JeuADamiers {
   …
   public:
      Echecs();
      void unCoup(int, int);
};
```

23 - L'héritage (4)

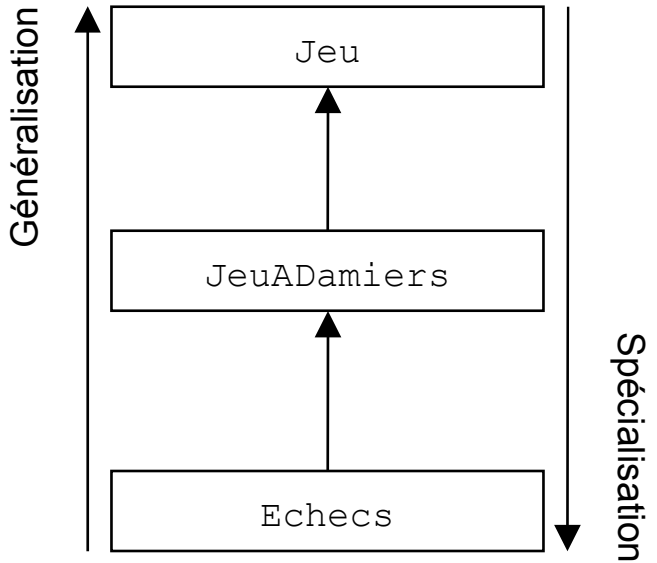
Réutilisation de classes par héritage

Ex :

```
Jeu::Jeu(int i) {…}
Jeu::unCoup() {…}


JeuADamiers::JeuADamiers(int i) : Jeu(i) {…}
JeuADamiers::unCoup(int i) {
   Jeu::unCoup();
   …
}

Echecs::Echecs() : JeuADamiers(8) {…}
Echecs::unCoup(int i, int j) {
   JeuADamiers::unCoup(i);
   …
}
```

23 - L'héritage (5)

Réutilisation de classes par héritage



On dit que :

`Jeu` est la classe parente (superclass) de `JeuADamiers` et de `Echecs`

`JeuADamiers` et `Echecs` sont les classes filles/dérivées de `Jeu`

`JeuADamiers` et `Echecs` hérite/descend de `Jeu`

L'utilisation répétée de l'héritage permet de constituer une hiérarchie de classes représentant une relation de spécialisation entre ces classes. Une classe peut aussi hériter de plusieurs classes parentes

23 - L'héritage (6)

Visibilité ou droits d'accès

- Les constructeurs, le destructeur, de même que l'opérateur = de la classe de base ne sont pas hérités dans la classe dérivée
- Si on définit une classe dérivée, sans mention particulière ou avec la mention `private` (héritage privé), les données publiques de la classe de base deviennent privées au niveau de la classe dérivée
- Si on définit une classe dérivée d'une classe de base avec la mention `public` (héritage public), les données publiques de la classe de base sont accessibles par les utilisateurs des objets instances de la classe dérivée
- Les données privées de la classe de base ne sont accessibles que par les membres de la classe de base
- Pour avoir des données privées dans la classe de base, inaccessibles aux utilisateurs des objets de cette classe et des classes dérivées, mais accessibles par les membres des classes dérivées, on fait précéder leur déclaration de la mention `protected`

23 - L'héritage (7)

Réutilisation de classes par héritage

Ex :

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

class B : public A {
   public:
      void g();
      void h();
};

// un objet B est un objet A avec des "choses" en plus
…
A a;
B b;
a.f();    // légal
a.g();    // illégal : a n'a pas de g()
b.f();    // légal
b.g();    // légal
b.A::h()  // légal : appel de la méthode h de A
```

23 - L'héritage (8)

Visibilité ou droits d'accès

```
Class Base {
 private:
  int  i1B;
 public:
  int  i2B;
  void fB();
 protected:
  int  i3B;
}
```

```
Class Derive : Base {
 public:
  void fD() {
   i1B = 1;      //ERREUR
   i2B = 2;
   i3B = 3;      //ERREUR
   fB();
  }
}
```

```
Class Derive : public Base {
 public:
  void fD() {
   i1B = 1;      //ERREUR
   i2B = 2;
   i3B = 3;
   fB();
  }
}
```

```
int main() {
 Base   b;
 Derive d;
 d.i2B = 2;    //ERREUR
 d.i3B = 3;    //ERREUR                    //ERREUR
 d.fB();       //ERREUR
 d.fD();
 b.fD();       //ERREUR                    //ERREUR
}
```

Une troisième forme de dérivation existe : `protected`. Dans ce cas, les membres publics de la classe de base seront considéré protégés dans toutes les dérivations ultérieures.

23 - L'héritage (9)

Propriété issue de l'héritage : conversions automatiques

Si `B` hérite de `A`, alors toutes les instances de `B` sont aussi des instances de `A`, et il est donc possible d'écrire :

```
A a; B b; a = b;
```

Propriété conservée lorsqu'on utilise des pointeurs :

```
A *pa; B *pb = …; pa = pb;
```

car pointer sur un `B` c'est avant tout pointer sur un `A`

Evidement, l'inverse n'est pas vrai :

```
A a; B b; b = a;                //ERREUR
```

Pareil pour les pointeurs :

```
A *pa = …; B *pb; pb = pa;   //ERREUR
```

car pointer sur un `A` n'est pas pointer sur un `B`.

23 - L'héritage (10)

Héritage et modifications

Une classe dérivée est définie relativement à une classe de base

```
class Base {
   public:
      …
      void f();
}
```

en introduisant 3 types de différences :

Ajouts de membres

```
class Derive : public Base {
   public:
      void g();
}
```

Modification de membres

```
class Derive : public Base {
   public:
      void f();
}
```

Suppression de membres

```
class Derive : private Base {
…
}
```

23 - L'héritage (11)

Appel de constructeurs

```
class Base {
   private:
      int x, y;
   public:
      Base(int a, int b) {x = a; y = b;}
}
class Derive : public Base {
   private:
      int z;
   public:
      Derive(int a, int b, int c) : Base(a, b) {z = c;}
}
```

Comme une instance de `Derive` est avant tout une instance de `Base`, dans le constructeur de `Derive`, on explicite la façon de créer l'instance `Base` dans la liste d'initialisation

Le constructeur de `Base` est appelé avant le constructeur de `Derive`

Les destructeurs sont appelés dans l'ordre inverse

23 - L'héritage (12)

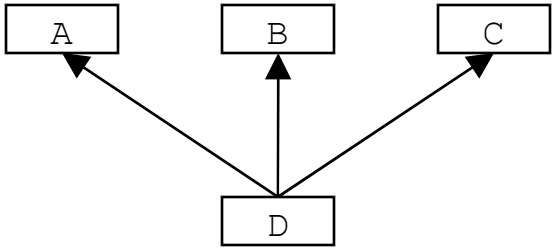
Exemple :

```
class Base {
   public:
      void f() {cout << "Base::f()" << endl;}
}
class Derive : public Base {
   public:
      void f() {cout << "Derive::f()" << endl;}
}
int main() {
   Base   b, *p;
   Derive d;
   p = &b; p->f(); // affiche Base::f()
   p = &d; p->f(); // affiche Base::f()
}
```

Le choix de la fonction est conditionné par le type de `p` connu au moment de la compilation :
puisque `p` est un `Base*` alors il utilise `Base::f()`
Il existe cependant un moyen pour que le deuxième appel affiche `Derive::f()` (déclaration de `f` virtuelle dans `Base`)

23 - L'héritage multiple (1)

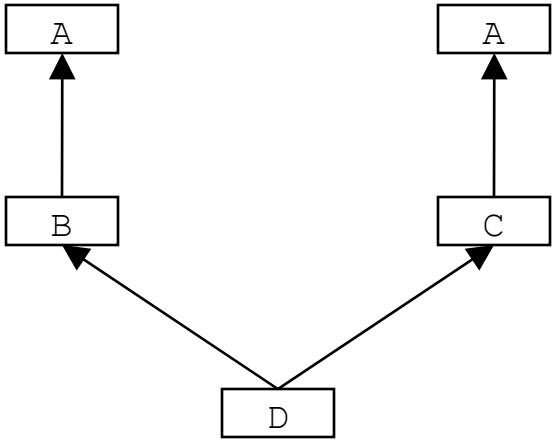
Une classe peut être dérivée à partir de plusieurs classes de bases



Ex :

```
class A {
   …
   public int x, y;
   …
};
```

Mais elle peut aussi être dérivée plusieurs fois de la même classe de base d'une manière indirecte

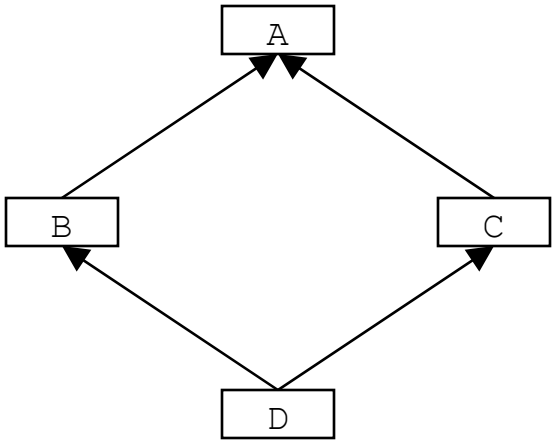



```
class B : public A {…};
class C : public A {…};
class D : public B, public C {…};
```

La classe `D` hérite de 2 objets distincts de la classe `A`. Les membres de `A` vont donc apparaître 2 fois dans `D`
Il faut utiliser l'opérateur de résolution de portée `::` pour distinguer les 2 jeux de membres de `D` qui sont `B::A` et `C::A`
L'appel des constructeurs se fait dans l'ordre `A`, `B`, `A`, `C`, `D`

23 - L'héritage multiple (2)

Afin de ne faire apparaître les membres de `A` qu'en un seul exemplaire dans `D`, il faut préciser dans les déclarations de `B` et de `C` que la classe `A` est virtuelle



Ex :

```
class A {
   …
   public int x, y;
   …
};
class B : virtual public A {…};
class C : virtual public A {…};
class D : public B, public C {…};
```

L'appel des constructeurs se fait dans l'ordre `A`, `B`, `C`, `D` (le constructeur d'une classe virtuelle étant toujours appelé en premier)

Le choix des informations à fournir au constructeur de `A` se fait non plus dans `B` ou dans `C`, mais dans `D`

`D(`paramètres`) : B(`paramètres`) : A(`paramètres`)`

Il est aussi possible de combiner les deux modes d'héritage

23 - L'héritage multiple (3)

Exemple : ambiguïté d'identification

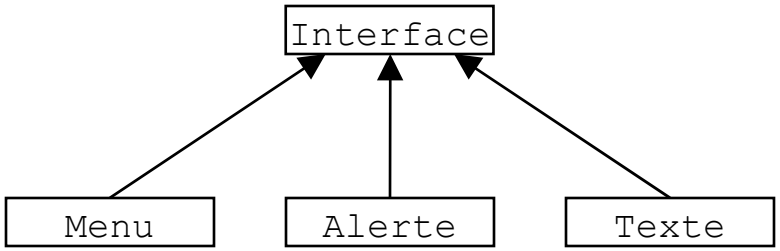
```
class A {                              class B {
   protected:    int i, j;                 private:      int i;
   public:       void f() {…}              protected:    int j;
};                                           public:       void f() {…}
                                       };

class C : public A, public B {
   private: int i;
   public:  void g() {
            i = 0;     // valide
            A::i = 0;  // valide
            B::i = 0;  // invalide : B::i est privé
            j = 0;     // ambigu : A::j ou B::j
            }
};
int main() {
   C c;
   c.f();              // ambigu : A::f() ou B::f()
   c.B::f();           // valide
}
```

24 - Le polymorphisme (1)

Le polymorphisme est un moyen de manipuler des objets hétéroclites de la même manière, pourvu qu'ils disposent d'une interface commune

Il permet donc de simplifier les traitements communs applicables à différents types d'objets, en les rendant communs à tous ces objets

Un objet polymorphe est un objet susceptible de prendre plusieurs formes pendant l'exécution



Les classes `Menu`, `Alerte` et `Texte` sont dérivées de la classe `Interface` et sont donc des "sortes" d'`Interface`

Un objet instance de la classe `Interface` peut, durant l'exécution, prendre la forme d'un `Menu`, d'une `Alerte` ou d'un `Texte` (par affectation)

24 - Le polymorphisme (2)

Exemple 1 :

```
class A {
   …
   public:
      …
      void f() {
         cout << "A::f()" << endl;
   }
};
```

```
class B : public A {
   …
   public:
      …
      void f() {
         cout << "B::f()" << endl;
};
```

```
int main() {
   A a; B b; A *p;
   p = &a; p->f(); // affiche A::f()
   p = &b; p->f(); // affiche A::f()
}
```

Le deuxième appel `p->f()` invoque la fonction `f()` de la classe `A` bien que `p` pointe sur une instance de la classe `B`

Ce mauvais choix de fonction membre est dû à la liaison statique effectuée lors de la compilation (early binding).
`p` est lié définitivement à la classe `A` même si, à l'exécution, `p` pointe sur `B`

24 - Le polymorphisme (3)

Exemple 1 (suite) :

```
class A {
   …
   public:
      …
      virtual void f() {
         cout << "A::f()" << endl;
   }
};
```

```
class B : public A {
   …
   public:
      …
      void f() {
         cout << "B::f()" << endl;
};
```

```
int main() {
   A a; B b; A *p;
   p = &a; p->f(); // affiche A::f()
   p = &b; p->f(); // affiche B::f()
}
```

Le choix de la fonction est maintenant conditionné par le type exact de l'objet pointé par `p` connu au moment de l'exécution (late binding).
Le deuxième appel `p->f()` invoque ici la fonction `f()` de la classe `B`

Les inconvénients sont :

- Les instances de `A` et de `B` prennent plus de place en mémoire (taille d'un pointeur, au moins)
- L'appel à `f()` est plus lent (le temps d'une indirection de pointeur supplémentaire, au moins)

24 - Le polymorphisme (4)

Exemple 2 :

```
class A {
   private:  int *p;
   public:   A()  {p = new int[4]; cout << "A()";}
             ~A() {delete [] p; cout<< " ~A()" << endl;}
};
class B : public A {
   private:  int *q;
   public:   B()  {q = new int[64]; cout<< "B() et q=" << q;}
             ~B() {delete [] q; cout<< " ~B()";}
};
int main() {
   for(int i = 0; i < 4; i++) {
      A *pA = new B(); delete pA;
   }
}
```

```
A()B() et q=0x100170 ~A()
A()B() et q=0x100270 ~A()
A()B() et q=0x100370 ~A()
A()B() et q=0x100470 ~A()
```

L'affichage met en évidence un problème de "fuite" de mémoire
Le destructeur de `B` n'est jamais appelé

24 - Le polymorphisme (5)

Exemple 2 (suite) :

```
class A {
   private:  int *p;
   public:   A()  {p = new int[4]; cout << "A()";}
             virtual ~A() {delete [] p; cout<< " ~A()" << endl;}
};
class B : public A {
   private:  int *q;
   public:   B()  {q = new int[64]; cout<< "B() et q=" << q;}
             ~B() {delete [] q; cout<< " ~B()";}
};
int main() {
   for(int i = 0; i < 4; i++) {
      A *pA = new B(); delete pA;
   }
}
```

```
A()B() et q=0x1001170 ~B() ~A()
A()B() et q=0x1001170 ~B() ~A()
A()B() et q=0x1001170 ~B() ~A()
A()B() et q=0x1001170 ~B() ~A()
```

Avec la déclaration du destructeur de `A` en virtuel, le destructeur de `B` est correctement appelé

24 - Le polymorphisme (6)

Le C++ permet, par le polymorphisme, que des objets de types différents répondent différemment à un même appel de fonction

En pratique, il faut déclarer `virtual` la fonction concernée dans la classe la plus générale de la hiérarchie d'héritage (`A` dans l'exemple précédant)

Les fonctions virtuelles réalisent une liaison dynamique à l'exécution entre un objet et une fonction membre (liaison tardive)

Toutes les classes dérivées qui apportent une nouvelle version de `f()` utiliseront leur propre fonction

Il reste évidemment possible d'appeler la fonction `f()` de `A` en spécifiant `p->A::f()`

25 - Classes abstraites (1)

Une classe est dite abstraite si elle contient au moins une fonction virtuelle pure

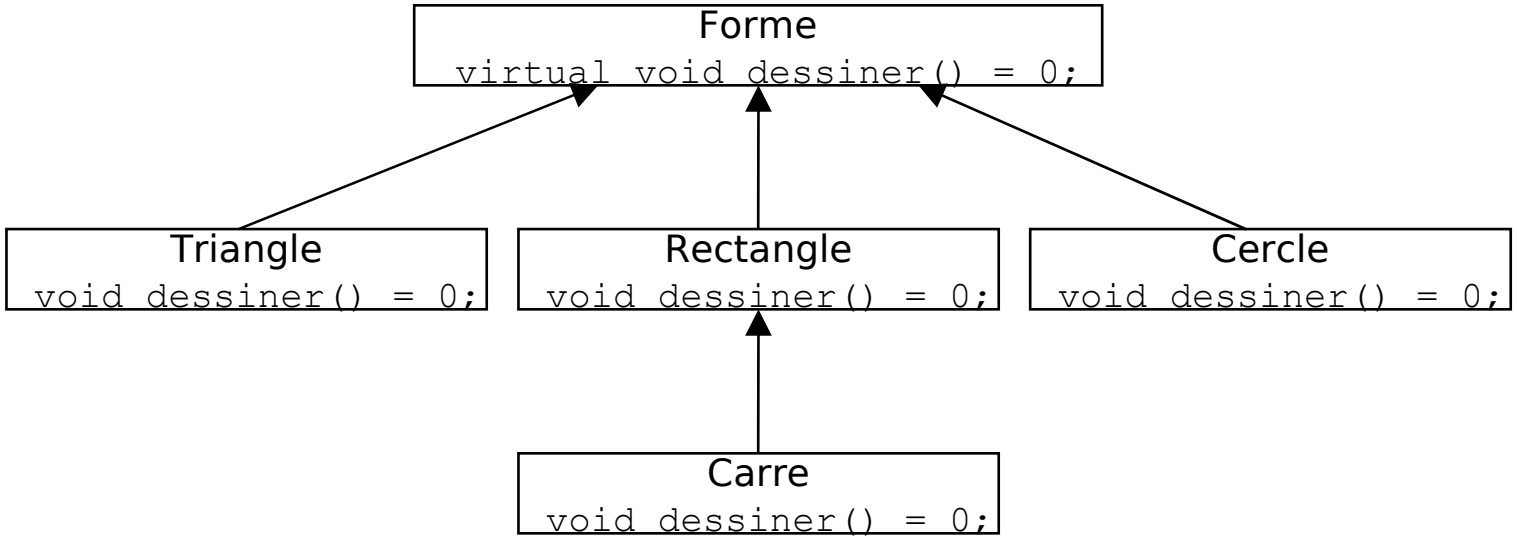
Une fonction membre est dite virtuelle pure losqu'elle est définie de la façon suivante :

```
virtual type nomFonction(paramètres) = 0;
```

Il est obligatoire d'avoir une définition pour les fonctions virtuelles pures au niveau des classes dérivées

Une classe abstraite permet d'introduire certaines fonctions virtuelles dont on ne peut encore donner aucune définition

25 - Classes abstraites (2)



Dans cet exemple, on ne sait pas programmer ce que doit faire la fonction `dessiner()` dans le contexte de `Forme`.
On veux juste s'assurer de sa présence dans toutes les classes filles, sans devoir la définir dans la classe parente.

25 - Classes abstraites (3)

La déclaration `virtual` oblige tous les descendants à contenir une méthode `dessiner()`

`Forme::dessiner()` n'est pas définie.
Elle est dite virtuelle pure

La classe `Forme` ne peut pas être instanciée.
Elle est dite abstraite

Une classe fille peut ne pas apporter de définition à `dessiner()`.
Elle est alors aussi abstraite (non-instanciable) à son tour, etc…

Une classe dans laquelle il n'y a plus une seule fonction virtuelle pure est dite concrète et devient instanciable

26 - Les espaces de noms (1)

Les espaces de nommage sont des zones de déclaration qui permettent de délimiter la recherche des noms des identificateurs par le compilateur

Leur but est essentiellement de regrouper les identificateurs logiquement et d'éviter les conflits de noms entre plusieurs parties d'un même projet

Comme on utilise quasiment tout le temps les fonctions de la bibliothèque standard, on utilise presque tout le temps `using namespace std;` pour se simplifier la vie

Syntaxe :

```
namespace nom { déclarations | définitions }
```

Lorsqu'un espace de nommage porte un nom très compliqué, il peut être avantageux de définir un alias pour ce nom :

```
namespace nom_alias = nom;
```

26 - Les espaces de noms (2)

Pour, par exemple, faire cohabiter une fonction `sin` très rapide mais faisant un calcul approché, avec la fonction `sin` de la bibliothèque standard qui effectue un calcul précis mais (relativement) lent :

- En C, la solution, consiste à donner un nom avec un préfixe :
  `fast_sin` et `sin` peuvent alors cohabiter dans le même programme
- Le C++ introduit la notion d'espace de nom pour améliorer le procédé :
  `sin` est déclaré dans la bibliothèque standard de C++, et son "vrai nom" est donc `std::sin`.
  Nous plaçons notre fonction rapide dans un espace de nom que nous choisissons, par exemple `fast_maths`. Le "vrai nom" de cette fonction sera alors `fast_maths::sin`
- Pour éviter de donner systématiquement le nom complet quand on ne se sert que d'une seule des deux fonctions, on peut écrire :

  ```
  using fast_maths::sin;
  ```
  `sin` signifiera toujours `fast_maths::sin`

  ```
  using namespace fast_maths;
  ```
  Ainsi dès que C++ ne connaît pas un symbole, il le cherche dans `fast_maths`

26 - Les espaces de noms (3)

Exemple :
Placement et utilisation d'un symbole C++ dans un espace de nom

Fichier `fast_maths.h`

```
namespace fast_maths {
   int i, j
   double sin(double);
}
```

Fichier `fast_maths.cc`

```
#include "fast_maths.h"
double fast_maths::sin(double d) {…}
```

Fichier `main.cc`

```
#include "fast_maths.h"
using fast_maths::i;
using fast_maths::sin;
…
i = 0;                          //équivalent à fast_maths::i = 0;
j = 0;                          //ERREUR j n'est pas défini
double y = sin(3.14);           //équivalent à fast_maths::sin
```

On peut si nécessaire créer des espaces de noms imbriqués les uns dans les autres…

26 - Les espaces de noms (4)

Exemple :

Déclaration `using` dans une classe

```
namespace A {
   float f;
}

class Base {
   private:  int i;
   public:   int j;
};

class Derivee : public Base {
   using A::f;         //Illégal
                       //f n'est pas dans une classe de Base
   using Base::i;      //Interdit
                       //Derivee n'a pas le droit d'utiliser Base::i
   public:
      using Base::j;  //Légal
};
```

27 - La Standard Template Library (1)

La plupart des programmes sont amenés, à un moment ou à un autre, à conserver et à manipuler un nombre arbitraire de données en mémoire

En général, les structures de données et les algorithmes mis en œuvre apportent des solutions à des classes de problèmes parfaitement identifiés

Ces structures de données sont communément appelées des conteneurs en raison de leur capacité à contenir des données de nature variée

Afin d'éviter aux programmeurs la réécriture des structures de données et de leurs algorithmes associés les plus classiques, la bibliothèque standard définit un certain nombre de classes patrons pour les conteneurs les plus courants

Ces classes sont paramétrées par le type des données des conteneurs et peuvent donc être utilisées virtuellement pour toutes les situations qui se présentent

27 - La Standard Template Library (2)

La STL est une bibliothèque de classes conteneurs, d'algorithmes et d'itérateurs. Elle fournit beaucoup des algorithmes et des structures de données de base

Les conteneurs de la bibliothèque standard ne sont pas définis par les algorithmes qu'ils utilisent, mais plutôt par l'interface qui peut être utilisée par les programmes clients

La bibliothèque standard impose également des contraintes de performances sur ces interfaces en termes de complexité

En réalité, ces contraintes sont tout simplement les plus fortes qui soient, ce qui garantit aux programmes qui les utilisent qu'ils auront les meilleures performances possibles

La STL contient une série de patrons prêts à l'emploi :

27 - La Standard Template Library (3)

Séquences :
Une séquence est un conteneur capable de stocker ses éléments de manière séquentielle, les uns à la suite des autres. Les éléments sont donc parfaitement identifiés par leur position dans la séquence, et leur ordre relatif est donc important

- `vector<>` : tableau homogène auto-redimensionnable
- `list<>` : listes
- `deque<>` : piles et files d'attente (tampon circulaire dynamique)

Conteneurs associatifs :
Les conteneurs associatifs, en revanche, manipulent leurs données au moyen de valeurs qui les identifient indirectement. Ces identifiants sont appelées des clefs par analogie avec la terminologie utilisée dans les bases de données

- `set<>` : ensemble non ordonné
- `multiset<>` : comme `set<>` mais avec doublons
- `map<>` (et `multimap<>`) : dictionnaires/tableaux associatifs

Conteneurs spéciaux :

`basic_string<>` : base du type `string`
en fait, on a `typedef basic_string<char> string`

27 - La Standard Template Library (4)

Exemple :

Tableau homogène autoredimensionnable :

```
vector<int> tab(8); // semblable à int tab[8]
```

Comme pour un tableau, accès possible avec `[]` :

```
tab[2] = 10;
```

Accès au nombre d'éléments avec `size()` :

```
cout << tab.size() << endl;
```

Accès au premier et dernier éléments :

```
int & premier = tab.front();
int & dernier = tab.back();
```

Ajout en fin avec éventuel ajout de nouvelles cases :

```
tab.push_back(11);
```

Suppression du dernier élément :

```
tab.pop_back();
```

27 - La Standard Template Library (5)

Abstraction des pointeurs : les itérateurs

Un itérateur n'est rien d'autre qu'un objet permettant d'accéder à tous les objets d'un conteneur donné, souvent séquentiellement, selon une interface standardisée

La dénomination d'itérateur provient donc du fait que les itérateurs permettent d'itérer sur les objets d'un conteneur, c'est-à-dire d'en parcourir le contenu en passant par tous ses objets

Comme les itérateurs sont des objets permettant d'accéder à d'autres objets, ils ne représentent pas eux-mêmes ces objets, mais plutôt le moyen de les atteindre. Ils sont donc comparables aux pointeurs, dont ils ont exactement la même sémantique

Chaque classe conteneur apporte l'implémentation d'un itérateur pour son parcours

Il existe 5 concepts d'itérateur :

- les itérateurs de la catégorie Output
- les itérateurs de la catégorie Input
- les itérateurs de la catégorie Forward
- les itérateurs de la catégorie Bidirectionnal
- les itérateurs de la catégorie RandomAccess sont les plus puissants. Ils fournissent tout

27 - La Standard Template Library (6)

Les itérateurs

Tous les conteneurs de la bibliothèque standard disposent d'itérateurs

Ils permettent de parcourir tous les éléments d'un conteneur séquentiellement à l'aide de

- l'opérateur de déréférencement `*` et de
- l'opérateur d'incrémentation `++`

Les conteneurs définissent donc tous un type `iterator` et un type `const_iterator`

Le type d'itérateur `const_iterator` est défini pour accéder aux éléments d'un conteneur en les considérant comme des constantes. Ainsi, si le type des éléments stockés dans le conteneur est `T`, le déréférencement d'un `const_iterator` renverra un objet de type `const T`

Afin de permettre l'initialisation de leurs itérateurs, les conteneurs fournissent deux méthodes

`begin()`, `rbegin()`	retourne un itérateur référençant le premier/dernier élément
`end()` `rend()`	retourne la valeur de fin/début de l'itérateur lorsqu'il a passé le dernier/premier élément du conteneur

27 - La Standard Template Library (7)

La bibliothèque standard définit tout un jeu de fonctions patrons extérieures aux conteneurs et dont le but est de réaliser des opérations de haut niveau

Ces fonctions sont appelées algorithmes

Les algorithmes ne dérogent pas à la règle de généricité que la bibliothèque standard C++ s'impose

La STL fournit ainsi un grand nombre d'algorithmes sur des conteneurs abstraits, et utilisant la notion d'itérateurs :

- Algorithmes de tri (efficaces et complexes, genre QuickSort)
- Algorithmes de recherche (dichotomie, …)
- etc…

Etudier la documentation de la STL pour en savoir plus !

27 - La Standard Template Library (8)

Exemple :

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <iterator>
using namespace std;

int main () {
   int nombreOcc;
   int t[] = {10, 20, 30, 30, 20, 10, 10, 20}; //8 éléments

   nombreOcc = (int) count (t, t + 8, 10);
   cout << "10 apparaît " << nombreOcc << " fois" << endl;

   vector<int> v (t, t + 8);
   nombreOcc = (int) count (v.begin(), v.end(), 20);
   cout << "20 apparaît " << nombreOcc << " fois" << endl;

   //affichage du contenu de v
   for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
      cout << " " << *it;
   cout << endl;
   return 0;
}
```

28 - Spécificateur `extern` (1)

On sait que les variables globales sont communes à tout un programme. Idem, pour les fonctions…

Mais, elles ne sont connues que dans les fichiers "`.cc`" dans lesquelles elles ont été définies et/ou déclarées

Problème : Comment faire connaître dans un "`.cc`" l'existence d'une variable globale définie dans un autre fichier "`.cc`" ?

Réponse : Il suffit de rajouter une déclaration (et PAS une définition) de cette variable dans l'en-tête du deuxième fichier "`.cc`"

En général, on écrit une bonne fois pour toute ces déclarations dans un fichier "`.h`" que l'ont peut ainsi facilement inclure dans l'en-tête

Les déclarations de fonctions sont juste constituées des prototypes de fonctions (leur première ligne de définition)

Les déclarations de variables sont constituées de leur définition précédée du mot clé `extern` qui permet de ne pas créer la variable, mais juste de dire que cet objet existe déjà ailleurs

28 - Spécificateur `extern` (2)

Exemple :
On veut créer deux variables (`var1` et `var2`) et deux fonctions (`fct1` et `fct2`) utilisables n'importe où dans les fichiers `fichier1.cc` et `fichier2.cc`

Fichier `declarations.h`

```
extern int var1;          // Déclaration de var1
extern double var2;       // Déclaration de var2
int fct1(int);            // Déclaration de fct1
double fct2(double);      // Déclaration de fct2
```

Fichier `fichier1.h`

```
#include "declarations.h"

// Définition de var1
int var1 = 10;

// Définition de fct1
int fct1(int a) {
   var1 += a;
   var2 += var1;
   return var1;
}
```

Fichier `fichier2.h`

```
#include "declarations.h"

// Définition de var2
double var2 = 11;

// Définition de fct2
double fct2(double b) {
   var2 += 2.3
   var1 += var2;
   return var2;
}
```

28 - Spécificateur `static` (1)

Malgré le nom, aucun rapport avec l'allocation statique

Plusieurs notions suivant le contexte d'application

Il peut s'appliquer à

- une variable,
- une fonction,
- une donnée membre,
- une fonction membre

28 - Spécificateur `static` (2)

`static` en dehors d'une classe ou d'une fonction

Exemple :

Fichier `truc.cc`

```
static int a = 10;
static int f() {…}
```

Fichier `machin.cc`

```
int main() {
   a = 10;        // impossible
   int i = f();   // impossible
}
```

Sert à limiter la visibilité d'un identificateur au seul fichier où il est défini

Dans l'exemple, la variable globale `a` et la fonction `f()` ne peuvent être utilisés que dans le fichier `truc.cc`

Même avec `extern`, impossible d'accéder à `a` ou à `f`

Il devient possible de créer dans ce fichier une autre variable/fonction globale ayant le même nom

28 - Spécificateur `static` (3)

`static` dans une fonction

Exemple :

```
int compte() {
   static int n = 0;
   return n++;
}

int main() {
   cout << compte() << endl; // affiche 0
   cout << compte() << endl; // affiche 1
   cout << compte() << endl; // affiche 2
}
```

Une des utilisations les plus fréquentes

La valeur persiste entre deux appels à la fonction

L'allocation mémoire de la variable se fait non pas dans la pile mais dans le tas

L'initialisation de la variable est donc très importante

La variable se comporte un peu comme une variable globale qui ne serait visible que dans sa fonction

28 - Spécificateur `static` (4)

`static` dans une classe

Exemple :

Fichier `A.h`

```
class A {
   private:
      static int compte;
   public:
      A();
      int combien();
};

int main() {
   A a1, a2, a3;
   cout << a1.combien(); // affiche 3
}
```

Fichier `A.cc`

```
int A::compte = 0;

A::A() {++compte;}

int A::combien() {return compte;}
```

Une autre utilisation fréquente

La donnée est attachée à la classe elle-même et pas aux instances : il n'y a qu'un seul `compte` pour toutes les instances de `A`

Importance de l'initialisation, qui se fait dans le `.cc` grâce au spécificateur de portée

28 - Spécificateur `static` (5)

`static` dans une classe

Exemple :

Fichier `A.h`

```
class A {
   private:
     static int compte;
   public:
     A();
     int combien();
     static void reset(int n = 0);
};

int main() {
   A a1, a2, a3;
   cout << a1.combien(); // affiche 3
   A::reset();
   A a4;
   cout << a1.combien(); // affiche 1
}
```

Fichier `A.cc`

```
int A::compte = 0;

A::A() {++compte;}

int A::combien() {return compte;}

void A::reset(int n) {compte = n;}
```

La fonction `reset`, bien que faisant partie de la classe, ne peut pas être appelée sur une instance de cette classe

Elle est appelée globalement, avec le spécificateur de portée

29 - Identification dynamique de type (1)

Opérateur `typeid`

Syntaxe : typeid (expression)

Le résultat de l'opérateur typeid est une référence sur un objet constant de classe `type_info`

Quand typeid est appliqué à une expression dont le type est polymorphique le résultat est le type de la classe la plus dérivée

```
…
#include <typeinfo>
class CBase {virtual void f(){}};
class CDerivee : public CBase {};
int main () {
 int  *p = NULL, i = 0;
 CBase *b = new CBase;
 CBase *d = new CDerivee;

 cout << typeid(p).name() << '\n';
 cout << typeid(i).name() << '\n';
 cout << typeid(b).name() << '\n';

 cout << typeid(d).name() << '\n';
 cout << typeid(*b).name()<< '\n';

 cout << typeid(*d).name()<< '\n';
 return 0;
}
```

```
int *
int
class Cbase *
class Cbase *
class Cbase
class CDerivee
```

29 - Identification dynamique de type (2)

Transtypage

Bien qu'il soit interdit d'utiliser un pointeur sur une classe de base pour initialiser un pointeur sur une classe dérivée, cette opération peut être légale, si le programmeur sait que le pointeur pointe bien sur un objet de la classe dérivée

Le langage exige cependant un transtypage explicite

Le mécanisme d'identification dynamique des types peut être alors utilisé pour vérifier, à l'exécution, si le transtypage est légal.
S'il ne l'est pas, un traitement particulier doit être effectué, mais s'il l'est, le programme peut se poursuivre normalement

Le C++ fournit un jeu d'opérateurs de transtypage qui permet de faire ces vérifications dynamiques

`static_cast`	Opérateur de transtypage à tout faire. Ne permet pas de supprimer le caractère const ou volatile
`const_cast`	Opérateur spécialisé et limité au traitement des caractères const et volatile
`dynamic_cast`	Opérateur spécialisé et limité au traitement des "downcast"
`reinterpret_cast`	Opérateur spécialisé dans le traitement des conversions de pointeurs peu portables

29 - Identification dynamique de type (3)

Transtypage

Syntaxe :
op_cast<expression_type>(expression_à_transtyper)

où op_cast prend l'une des valeurs : `static_cast`, `const_cast`, `dynamic_cast` `reinterpret_cast`

Exemple :

```
class A {public: virtual ~A() {}
};
class B: public A;
class C: public A;

A *mystere() {return new B;}
int main() {
   A *chose = mystere();
   B *b = dynamic_cast<B*>(chose);
   C *c = dynamic_cast<C*>(chose);
   if ( b != NULL )     cout << "C'est un B" << endl;
   else if (c != NULL ) cout << "C'est un C" << endl;
        else    cout << "Mais qu'est-ce que c'est ?" << endl;
   return 0;
}
```

29 - Identification dynamique de type (4)

Transtypage

Le `dynamic_cast` essaie de stocker une instance de classe donnée (entre parenthèses) dans une variable d'un type descendant (entre chevrons)

Si l'objet pointé ou référencé est bien du type indiqué pour le transtypage, l'opération se déroule correctement

En revanche, s'il n'est pas du bon type, `dynamic_cast` n'effectue pas le transtypage

Si le type cible est un pointeur, le pointeur `NULL` est renvoyé.
Si en revanche l'expression caractérise un objet ou une référence d'objet, une exception de type `bad_cast` est lancée

Pour que ce mécanisme fonctionne, il faut au moins une méthode virtuelle dans la classe de base, ceci pour indiquer au compilateur de créer des tables de virtualité

30 - Arguments de la ligne de commande

Si un programme C désire accéder aux arguments de la ligne de commande, cette ligne de commande est disponible au travers de deux variables :

`argc`
cette variable `argc` désigne le nombre total d'arguments passés

`argv`
la variable `argv` est un tableau de pointeurs sur des chaînes de caractères représentant les arguments

Ces deux variables doivent être spécifiées comme arguments formels à la fonction principale

Exemple :
Affichage du nom du programme et des arguments transmis

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
   for(int i = 0; i< argc ; i++)
      cout << "argv[" << i << "]=" << argv[i] << endl;
   return 0;
}
```

31 - Définition de synonyme

Le mot réservé typedef est un artifice syntaxique permettant simplement la définition de synonyme de type qui peut ensuite être utilisé à la place d'un nom de type

Syntaxe : typedef <type> <nom>;

Exemple :

```
typedef int entier;
entier i = 10;
typedef entier *pointeur_sur_entier;
pointeur_sur_entier pi = &i;
typedef entier tab10entiers[10];
tab10entiers tab = {1, 2, 3, …};
typedef double (*fct)(double);
fct f1 = sin, f2 = fabs;
// f1 est l'adresse de sinus, f2 de valeur absolue
typedef fct (*f)(entier,fct []);
// Compliqué ! f est l'adresse d'une fonction qui prend un entier
// et un tableau de pointeur sur des fonctions de R dans R,
// et qui renvoie l'adresse d'une fonction de R dans R
```

FIN