

# Programmation avancée : C++

Caroline Larboulette - Mohamed Zouari

Université de Bretagne Sud

Fall 2018

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

## 10 Chapitre 10 - Exceptions

# Variables

- Variables **locales**
  - fonction/bloc
- variables **globales** : extérieures à une classe/fonction
  - si `static` : portée limitée au fichier
  - si `extern` : portée étendue (pour utiliser dans un fichier une variable globale déclarée dans un autre fichier)
- Variables **d'instance** : propres à un objet
  - `public`, `protected`, `private`
- Variables **de classes** : indépendantes des objets
  - `static`

# Entrées || sorties

iostream.h contient les déclarations des classes gérant les entrées/sorties standard.

```
#include <iostream>
```

3 flux de sorties :

- `cout << EXPRESSION1 << ... << EXPn;`
- `cerr << EXPRESSION1 << ... << EXPn;`
- `clog << EXPRESSION1 << ... << EXPn;`

1 flux d'entrée :

- `cin >> VAR1 >> VAR2 >> ... >> VARn ;`

# Espace de noms

- C++ : langage + bibliothèques (fonctions)
- Bibliothèque standard : std

```
#include <iostream>  
std::cout << " Hello world" << std::endl;
```

```
#include <iostream>  
using namespace std;    // espace de noms  
cout << " Hello world" << endl;  
// cout est le flux standard std::cout  
// et pas un flux personnalisé
```

# Espace de noms

- Définition d'un espace de noms

```
namespace nom_espace { type nom_fonction(arguments); ...  
namespace ma_lib {int f1(); ...}
```

- Définition d'une fonction

```
int ma_lib::f1() { ... } ;
```

- Appel d'une fonction

```
using namespace ma_lib;  
f1();
```

- Utile lorsque l'on crée une librairie

## Espace de noms: exemple

```
namespace A
{
    int i, j;           // Déclare A::i et A::j
    void f(int);        // Déclare A::f(int)
}

void g()
{
    int i;              // Déclaration locale de i
    using A::i;          // A::i utilisé sous le nom i
                        // Erreur : i est déjà déclaré
    j=1;                // Erreur : j n'est pas défini
    void f(char);        // Déclaration locale de f(char)
    using A::f;          // Pas d'erreur, surcharge de f
}
```



# Commentaires

- Forme standard en C :

```
cout << valeur << endl;  
/* affiche valeur */
```

- Commentaires sur une ligne : //

```
cout << valeur; // afficher valeur
```

- // invalide /\* et \*/

```
partie1 // part2 /* part3 */ part4
```

- /\* invalide //

```
/* partie1 // partie2 */ partie3
```

# Déclaration

Déclarations n'importe où au sein d'une fonction ou au sein d'un bloc

- Portée limitée au bloc ou à la fonction

```
int main( )  
{  
    int n ;  
    ....  
    int q = 2*n ; ...  
    return 0 ;  
}
```

- Déclaration possible dans une boucle

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

## Résolution de portée

Opérateur "::" permet d'accéder à une variable cachée par une autre variable

- Priorité supérieure à celle de tous les autres opérateurs

```
int i ;  
int foo( )  
{ int i ;    // cache le i global  
  i =:: i ;   ....
```

- :: permet également d'accéder aux champs statiques des structures membre gauche de :: nom de structure

```
struct ma_struct  
    { static int l ; } ;  
main( )  
    { ma_struct::l=1 ; }
```

# Constantes

- Déclaration

```
const TYPE NOM_CONST = VALEUR_CONST;  
const TYPE *NOM_CONST [=VAL_CONST ];  
                // pointeur sur un objet constant  
TYPE *const NOM_CONST = VAL_CONST;  
                // pointeur constant
```

- Exemple

```
const char truc = 'a';  
const char* ptruc; //pointeur sur l'objet constant  
char* const ptruc; //pointeur constant
```

# Constantes

- Une constante ne peut pas être modifiée, elle doit toujours être initialisée

```
const float pi=3.1416 ;  
const int nul=0 ;
```

- Déclaration de paramètres de fonction constants

```
char *strcpy (char* dest , const char* source);
```

# Énumération

Définition d'un sous-type de `int`

- Constantes représentées par des identificateurs
- Début à 0 par défaut
- Rend le code plus lisible

```
enum situation {CELIB, MARIE, DIVORCE, VEUF=-1};  
situation situ = VEUF ; //On peut déclarer des variables  
                        //de type situation  
  
situ = MARIE;  
situ = 2; // erreur: les identificateurs ne sont pas  
          // des lvalue  
MARIE = 3; // erreur  
enum {MASCULIN=1}; // constante symbolique entière  
                  // (type anonyme)  
int sexe = MASCULIN;
```

# Énumération

Possibilité de créer un type énuméré

- Constantes représentées par des identificateurs

```
typedef enum {orange, citron,  
             pamplemousse, tomate} TypFruit;
```

- 2 types énumérés **ne peuvent pas** contenir le même identifiant
- orange dans le cas suivant génère une erreur de symbole

```
typedef enum {orange, citron,  
             pamplemousse, tomate } TypFruit;  
typedef enum {rouge, jaune,  
             orange, vert} TypCouleur;
```

# Type booléen

2 valeurs : true (1) et false (0)

- Conversion implicite des int en bool
  - entier non nul : **true**
  - entier nul : **false**

```
bool plus_grand(int n, int m)
{
    return n > m;
}

bool b = 1024 ; //true
bool x = b + 2 * b ; //true
```



# Conversion implicite

Mise en place par le compilateur

- Dans les affectations : conversion "forcée" dans le type de la variable réceptrice
- Dans les appels de fonctions : conversion "forcée" d'un argument dans le type déclaré dans le prototype
- Conversions systématiques : conversions systématiques : char et short en int, float en double

# Conversion explicite

- Selon syntaxe C : (TYPE) EXPRESSION
- Selon syntaxe C++ : TYPE (EXPRESSION)
- Exemple :

```
struct s1 { int a; };  
typedef pt_s1 struct s1 *;  
pt_s1 pt;  
char *pt2;  
pt = (pt_s1) pt2; <=> pt = pt_s1 (pt2 );
```

# Prototypage

- **Déclaration** (fichier .h)

```
[extern] type fonction(type arg1, ..., type argn);  
int foo(int, char*, double);
```

- **Définition** (fichier .cpp)

```
type fonction(type arg1, ..., type argn)  
{ /* corps de la fonction */ }  
  
int f1(int i, float t) { ... }
```

# Valeurs par défaut

Possibilité d'appel de fonctions avec moins d'arguments que de paramètres.

- Les arguments absents prennent les valeurs par défaut
- Paramètres à valeur par défaut en fin de liste
- Déclaration

```
int foo(int =0, int = 0); //par convention à la déclarati
```

- Utilisation

```
foo(2,3);  
foo(2); //équivalent à foo(2,0);  
foo(int = 0, int); //erreur (uniquement dans la  
//déclaration)  
foo(,6); //erreur (on ne peut supprimer les paramètres  
//que en fin de liste)
```

# Surcharge de fonction

Fonctions de même nom, distinguées par le type et le nombre de leurs paramètres

- Sans correspondance directe, le compilateur tente d'effectuer des conversions implicites
- Déclaration (.h)

```
int  ajoute(int , int ); //1  
char* ajoute(char*, char*); //2  
const char* ajoute(char*, char*); //3
```

- Utilisation (.cpp)

```
int x = ajoute(2, 10); //1  
char *chaine = ajoute("bon", "jour"); //2
```

void

# Type void

- void indique qu'une fonction n'a pas de paramètre

```
int main(void) { return 0 ; }
```

- Pas de variable ou de constante de type void

```
void x ; // erreur
```

- Pas de type implicite int en C++

```
f() {...} ; // erreur  
const c = 7; // erreur
```

## Pointeur de void

- Pointeur (`void*`) utilisé pour pointer sur n'importe quel objet
- `cast` pour affecter la valeur d'un pointeur (`void*`) à un pointeur sur un type précis
- N'importe quel pointeur peut donc être assigné à un (`void*`) sans conversion explicite
- Exemples :

```
void* p ;           // déclaration d'un pointeur
p++ ;               // erreur
float* p_f = p;     // erreur
p_f = (float *) p;  // ok
```

# Fonctions inline

`inline` indique au compilateur que les appels à cette fonction doivent être remplacés par le code de la fonction

- Usage des fonctions inline aux fonctions de petite taille et si optimisation significative
- Pas de changement de contexte, donc plus rapide, mais plusieurs copies

```
inline TYPE NOM_FONCTION( paramètres )  
    {  
        <corps de la fonction> }  
}
```



# Opérateurs new et delete

Opérateurs new et delete permettent l'allocation et la désallocation dynamique d'objets

- Objets alloués par new
  - Non initialisés
  - À libérer explicitement par delete
- Désallocation à la charge du programmeur
- Pas de "garbage collector"

# Gestion mémoire

On distinguera 4 zones mémoire en C++ :

- **Zone des constantes** : allouée et initialisée au démarrage

# Gestion mémoire

On distinguera 4 zones mémoire en C++ :

- **Zone des constantes** : allouée et initialisée au démarrage
- **Zone des variables globales et statiques** : allouée au démarrage, initialisée plus tard

# Gestion mémoire

On distinguera 4 zones mémoire en C++ :

- **Zone des constantes** : allouée et initialisée au démarrage
- **Zone des variables globales et statiques** : allouée au démarrage, initialisée plus tard
- **Pile** : variables automatiques, allouée en début de bloc

# Gestion mémoire

On distinguera 4 zones mémoire en C++ :

- **Zone des constantes** : allouée et initialisée au démarrage
- **Zone des variables globales et statiques** : allouée au démarrage, initialisée plus tard
- **Pile** : variables automatiques, allouée en début de bloc
- **Zone de mémoire dynamique** : variables allouées par `new`

# Opérateurs new et delete

- Syntaxe

```
TYPE *POINTEUR_VARIABLE
```

```
POINTEUR_VARIABLE = new TYPE ;
```

```
TYPE *POINTEUR_TABLEAU ;
```

```
POINTEUR_TABLEAU = new TYPE[DIMENSION];
```

```
delete POINTEUR_VARIABLE ;
```

```
delete [] POINTEUR_TABLEAU ;
```

# Opérateurs new et delete

```
int *ad;  
ad = new int ;  
int *ad = new int ;  
char *ptc = new char[15] ;  
struct s{int a, int b};  
s* pts = new s ;  
delete pts ;  
date *d ;  
d = new date ;  
delete d ;
```

# Opérateurs new et delete

## Exemple de classe

- Déclaration

```
class Particle
{
    public:
    Particle();
    Particle(int , int );
};
```

- Utilisation

```
Particle* p = new Particle; //Pas de () si pas
                             //de parametres
```



# Références

Référence = alias pour un objet

- Opérateur de référence : &

```
type &NOM_REFERENCE=NOM_VARIABLE_INITIAL;
```

- Une référence doit être initialisée
- Une référence est l'alias d'un unique objet

référence  $\neq$  pointeur

# Références

Exemples :

```
int I , J = 10 ;  
int& alias_I = I ;  
I=1 ;                // alias_I vaut 1  
alias_I++ ;          // I vaut 2  
int * p = &alias_I ;  
double & d = 1 ; // Interdit  
alias_I = J ;
```

# Arguments par référence

- Références utilisées pour passer des arguments (modifiables) par adresse
- Le passage par référence évite la copie des paramètres dans la pile (coûteux)
- Le passage par référence évite la manipulation des pointeurs

# Arguments par référence

En C :

```
void swap(int* x, int* y)
{
    int l = *x ;
    *x = *y ;
    *y = l ;
}
int a=1, b=2 ;
swap(&a, &b) ;
```

En C++ :

```
void swap(int& x, int& y)
{
    int l = x ;
    x = y ;
    y = l ;
}
int a=1, b=2 ;
swap(a, b) ;
```

# Retour par référence

Permet l'écriture d'une fonction dont la valeur retournée peut apparaître en partie gauche d'une affectation (*lvalue*)

Retour par valeur :

```
int x ;  
int valeur_x() {  
    return x;  
}  
...  
int y = valeur_x();
```

Retour par référence :

```
int x ;  
int& valeur_x() {  
    return x;  
}  
...  
int y = -1;  
valeur_x() = y ;
```

# Chaînes

- Type string dans la bibliothèque standard
- A utiliser de préférence à char\*
- Exemple :

```
string s1="Hello"; string s2="world";  
string s3= s1 + ", " + s2;  
if (s1!=s2) s3 += '\n'; // concaténation  
s1[0] = 'B';  
string s4 = 'a';  
string s4 = 1;  
s3 = 'a'; // OK affectation  
string s5; // chaîne vide
```

# Chaînes

## Utilisation des chaînes C

- fonction `c_str()`
- produit une chaîne C à partir d'une string en ajoutant un 0 à la fin
- permet d'utiliser les fonctions définies sur les chaînes C
- Exemple :

```
printf("nom : %s\n", nom.c_str());  
int i = atoi(s.c_str());
```

# Chaînes

- Exemple d'utilisation des string

```
#include <string>
using namespace std;
string str;
getline(cin, str);
cout<<str<<endl;
```

```
> j'aime le C++ et vous ?
j'aime le C++ et vous ?
>
```



# Plan du cours I

1 Chapitre 1 - Bases de C++: types, variables, pointeurs

**2 Chapitre 2 - Classes**

3 Chapitre 3 - Fonctions amies

4 Chapitre 4 - Surdéfinition des opérateurs

5 Chapitre 5 - Héritage

6 Chapitre 6 - Polymorphisme

7 Chapitre 7 - Transtypage

8 Chapitre 8 - Généricité (templates)

9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

## 10 Chapitre 10 - Exceptions

# Notions relatives à un objet C++

- données privées, publiques, protégées
- fonctions privées, publiques, protégées
- fonctions amies
- accesseurs
- opérateurs
- constructeurs/destructeurs

# Structures et classes

une classe peut être définie par les mots clés `struct` et `class`.

- Structure :
  - données : publiques
  - méthodes : publiques
- Classe :
  - données : publiques, protégées ou privées
  - méthodes : publiques, protégées ou privées

## Exemple de structure point

```
struct point
{
    double x, y;
    void initialise(double, double);
    void deplace(double, double);
};
```

...

```
    point a, b ;
```

...

## Exemple de structure personne

```
#include <iostream>
#include <cstring>

struct personne {
    char    nom[20] ;
    int     age ;
    void    afficher() {
        cout << nom << " " << age ;
    }
} ;

main () {
    personne mec ; // objet mec, instance de per
    strcpy(mec.nom, "Durand") ;
    mec.age = 30 ;
    mec.afficher() ;
}
```

# Class

- Nouveau type mais manipulation comme un type "normal"
- Exemples :

```
class Voiture {  
    private : ...           // membres privées  
    protected : ...        // membres protégées  
    public : ...            // membres publiques  
};  
Voiture r7;                // objet type voiture
```

# Encapsulation d'une classe en C++

- Partie privée (**private**)
  - accessible par les méthodes de la classe
  - en principe, **tous** les attributs sont privés
- Partie protégée (**protected**)
  - idem que la partie privée
  - accessible en plus par les classes dérivées
- Partie publique (**public**)
  - accessible par tous = interface
  - opérations sur la structure



## Exemple de classe

```
class exemple
{
    int    a ;
    void f2 (int i) ;
    public :      void f1 (int i) ;
} ; // très important le ; !

void exemple :: f2 (int i)
{
    a = 1 ;          // ok f2 est membre de exemple
    if (i>0) f1(--i) ;    // OK
}
```

## Exemple de classe 2

```
void exemple :: f1 (int i)
{
    a = i ;
    if (i>0) f2(--i) ;
}

main ()
{
    exemple      ex ;
    ex.f1(1) ;
    int i = ex.a ;
    ex.f2(0) ;
}
```

// f1 est memb  
// f1 est membre  
// f1 est publique  
// erreur de compilation  
// erreur de compilation

# Accès aux membres d'une classe

- données relatives à une classe : opérateur de portée ::
- données relatives à une instance : accès par . depuis l'extérieur

```
str.x = 5;                // encapsulation non respectée
initialise(1,0);          // rejet à la compil.
str.initialise(0,1);      // appel initialise
void point::initialise(int abs, int ord)
    { x = abs; y = ord; } // membres x et y
```

# Méthodes d'une classe

- Communes aux différents objets
- Définies à l'intérieur ou à l'extérieur de la structure
- Surchargeables

```
int incremente ( int );  
int incremente () { return (++cpt ); }
```

# Organisation des fichiers

Classe = composant logiciel réutilisable

→ déclaration, définition de classes, programmes dans différents fichiers

- fichier entête (.h) de déclaration d'une classe
- fichier source (.cpp, .cc, .cxx, ...) de définition d'une classe
- fichier objet (.o) de compilation de la définition (générée automatiquement)

## Organisation des fichiers (2)

- Utilisation de la classe point → inclusion du fichier de déclaration `#include "point.h"`
- Recherche du module objet `point.o` à l'édition de liens
- Possibilité de mettre plusieurs classes par fichier
- En pratique, nom fichier = nom classe et une seule classe par fichier (sauf si les classes sont liées par héritage).

## Organisation des fichiers (3)

```
// Fichier point.h
```

```
class point
{
    int x, y;           // membres privées
    public :            // membres publics
        point(int , int) ; // constructeur
        void deplace (int , int) ;
        void affiche() ;
} ;
```

## Organisation des fichiers (4)

```
// Fichier point.cpp

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

// définition des fonctions membres
point::point(int abs, int ord)
{
    x = abs ; y = ord ;
}
void point::deplace(int dx, int dy)
{
    x = x + dx ; y = y + dy ;
}
void point::affiche()
{
    cout<< "x="<< x<< " y="<< y<< endl ;
}
```



# Makefile

```
> more Makefile
```

```
all : test
```

```
test :      Person.o Test_Person.o
```

```
                g++ -o test Person.o Test_Person.o
```

```
Person.o :    Person.cpp Person.h
```

```
                g++ -c Person.cpp
```

```
Test_Person.o : Test_Person.cpp Test_Person.h
```

```
                g++ -c Test_Person.cpp
```

```
clean :      rm Person.o Test_Person.o
```

```
> make
```

# Affectation d'objets

- Fait une affectation des données membres
- Attention aux données déclarées par pointeur, pas de copie profonde ! (Différent de Java)
- Affectation à éviter si l'opérateur “=” n'est pas redéfini

```
class personne {  
    int age;  
    public:  
        ...  
};  
personne mec1, mec2;  
...; // initialisation de mec1  
mec2 = mec1; // encapsulation respectée  
// données privées de mec2 inaccessibles
```

# Constructeur & destructeur

- Problème : initialisation d'un objet au bon vouloir de l'utilisateur (par l'utilisation d'une fonction `init()` par exemple)
- Solution :
  - fonction membre appelée automatiquement à la création d'un objet = **constructeur**
  - fonction membre appelée automatiquement à la destruction d'un objet = **destructeur**

## Constructeur & destructeur (2)

Constructeur et destructeur facultatifs, cependant très utiles pour :

- l'initialisation des données membres de la classe statiques (constructeur)
- la création des variables dynamiques via `new` (constructeur)
- la destruction des variables dynamiques via `delete` (destructeur)
- la fermeture de fichiers, périphériques, etc. utilisés par la classe (destructeur)
- etc.

## Constructeur & destructeur (3)

- Constructeur : même nom que la classe
- Destructeur : même nom que la classe précédé de tilde (~)

```
class point {  
    int x, y;  
public:  
    point(int , int );           // constructeur  
    ~point() ;                  // destructeur  
    void deplace(int , int );  
};  
  
point a(0, 1);                  // argument obligatoire
```

## Exemple de constructeur

```
// Declaration .h
class point
{
    int x, y ;                // décl. membres privés
public :
    point(int , int) ;        // constructeur
    void deplace (int , int) ; // fn publiques
    void affiche() ;
} ;

// Definition .cpp
point::point(int abs , int ord) // définition
{
    x = abs ; y = ord ;      }

// Utilisation
main(){
    point a(0,1), b(1,0) ;
    point c; //Impossible car le constructeur sans argument
              //n'existe pas
    a.deplace(1,-2) ;
    b.affiche() ;           }
```

## Exemple 2 de constructeur

```
//Declaration .h
class point
{
    int x, y ;           // décl. membres privés
    public :
        point() ;        // constructeur sans argument
} ;

//Definition .cpp
point::point() // définition
{
    x = 0 ; y = 0 ; }

//Utilisation
main(){
    point a(); //Incorrect
    point a;   //Correct
}
```

## Règles pour le constructeur

- Quand le constructeur se limite à initialiser des données, le destructeur n'est pas indispensable
- Quand le constructeur alloue dynamiquement de la mémoire, **le destructeur est nécessaire pour libérer l'espace mémoire**
- Le constructeur peut comporter un nombre quelconque d'arguments (0 à n)
- Le constructeur ne renvoie pas de valeur (pas même rien (!!))  
*i.e.* void)
- Un constructeur peut-être surdéfini et/ou posséder des arguments par défaut
- Si aucun constructeur n'est spécifié alors il y a un constructeur par défaut qui est vide



## Règles pour le destructeur

- Le destructeur ne peut pas comporter d'arguments
- Le destructeur ne renvoie pas de valeur

En pratique constructeur et destructeur sont **publics**

- constructeur privé → impossible de créer des objets par déclaration ou allocation (utilisation avec des classes abstraites)
- destructeur privé → pas de conséquence car on ne l'appelle jamais directement

# Objets temporaires

- Objets créés par appel explicite du constructeur

```
point(int , int) ; // constructeur  
point a = point(0,1) ; // appel constructeur explici  
                        // pas de recopie malgre le =  
point a(0,1) ; //meme chose
```

- l'évaluation de `point(0,1)` provoque
  - 1 la création par le compilateur d'un objet temporaire non accessible
  - 2 l'appel du constructeur `point` pour cet objet
  - 3 la recopie de cet objet dans `a`

# Objets dynamiques

- allocation dynamique d'espace mémoire
- Exemples :

```
point * ad_point ;           // déclaration
ad_point = new point ;      // affectation
ad_point->deplace(2,1) ;     // accès fonction
(* ad_point).affiche() ;    // ou encore
ad_point->affiche() ;        // meme chose
ad_point = new point(0,1) ;

//      alloc et construction
delete ad_point ;
```

# Choix de création

- allocation sur la pile

```
{ ... Personne pers1; ... }
```

- plus rapide
- destruction automatique à la fin du bloc

- allocation dynamique sur le tas :

```
{ ... ptr_pers2 = new Personne; ...  
    delete ptr_pers2; ... }
```

- création par `new`
- manipulation par pointeur
- destruction manuelle par `delete`

# Initialisation d'un objet

- Valeur d'initialisation transmise en argument au constructeur
- Schéma de conversion implicite

```
class    personne
{
    int    age ; char    nom[20] ;
    public :
    personne (char * pseudo)
        { strcpy(nom, pseudo) ; }
} ;
```

```
personne    mec ;
mec = "Martin" ;           // appel constructeur
```

## Initialisation d'un objet - exemple 2

```
//Declaration .h
class Point
{
    int x,y;
    public :
    Point (int abs)
        { x = abs; y=0; }
} ;

//Utilisation
Point a(3); //conversion implicite d'un entier
//vers un Point
Point a = 3;
```

# Constructeur de copie

- Cas où la valeur d'initialisation est un objet de la même classe que l'objet initialisé
- Exemple :

```
Point a;  
Point b = a ;
```

- Si aucun constructeur par copie n'est prévu :
  - existence/traitement par défaut
  - recopie des valeurs des membres de a dans b
  - problèmes avec les pointeurs

## Exemple de problème sur une mauvaise initialisation

- avec une donnée membre dynamique :

```
class tableau {  
    int taille ;  
    int * tab ;  
public :  
    tableau (int n = 10) {  
        taille = n ;  
        tab = new int[n] ;}  
    ~tableau () { delete [] tab ;}  
    ...  
} ;  
    ...  
  
tableau t1(5) ;  
tableau t2 = t1 ;           // création du tableau t2
```



## Constructeur de copie (suite)

- Syntaxe spécifique :

```
X::X(const X&);  
Point::Point(const Point &);  
tableau::tableau(tableau & t);
```

- L'objet est passé en paramètre via **une référence** (sinon appel récursif infini)
- `const` n'est pas obligatoire mais souhaitable

## Solution du problème précédent :

- Constructeur par copie !

```
tableau::tableau(const tableau& t)
{
    tab = new int[ taille = t.taille ]; //alloc nouveau ta
    for (int i=0; i<taille; i++)
        tab[i] = t.tab[i]; //recopie de toutes les
                           //valeurs du tableau
}
```

- Autre solution possible avec un opérateur d'affectation...  
(chapitre 4)

```
tableau& tableau::operator=(const tableau& t)
```

# Initialisations

- Quelles sont les différences ?

```
class Point;  
Point a; Point b;  
Point a(); //Incorrect , () est une fonction  
Point a(b); //Recopie par copy constructeur  
Point a = b; //Recopie par =  
Point a (Point(b));  
Point a (Point(2));
```

# Tableaux d'objets

- Tableau dont les éléments sont de type classe

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

main (){
    point courbe[10] ; //appel 10 fois du constructeur
    ... ;
    int    i = 3;
    courbe[i].affiche() ;
}
```

# Objets d'objets

- Donnée membre de type classe
- Exemple :

```
class courbe{  
    point p[80] ;  
    public :  
    void trace() ;  
} ;
```

```
courbe cb ;
```

- point doit posséder un constructeur par défaut !

# Liste d'initialisation

- Initialisation des attributs à la construction

```
class Point {  
    int i ; Toto t ;  
    public : ... } ;
```

```
Point::Point(int arg1 , Toto arg2)  
{  
    i=arg1 ; t=arg2 ;  
}
```

- est équivalent à

```
Point::Point(int arg1 , Toto arg2)  
:  
    i(arg1) ,  
    t(arg2)  
{}
```

# Liste d'initialisation

- Utile pour initialiser des membres constants

```
class Point {  
    const int n;  
    public :  
    Point();  
} ;
```

```
Point::Point(){n=12;} //interdit
```

```
Point::Point()  
:  
n(12)  
{}  
//ok
```

## Donnée membre statique

- Donnée partagée par tous les objets ( $\neq$  propres à chaque objet)
- Exemple :

```
class point{
    int    x, y ;
    static int compteur ;    // compteur du nombre de points
public :
    point(int , int) ;      // constructeur
    ~point() ;              // destructeur
} ;

point::point(int abs , int ord) : x(abs), y(ord){
    cout << "Création du " << ++compteur
        << "eme point" << endl ;
}

point::~~point(){
    cout<< "Reste " << --compteur << " points";
}
```



## Donnée membre statique (2)

- Existence des membres statiques indépendantes des objets
- A déclarer à l'extérieur de la classe

```
int point::compteur = 0 ;
```

- Pas d'initialisation lors de la définition dans la classe !

# Fonction membre statique (1)

- Fonction qui accède aux membres statiques
- Ne pas s'appliquer à un objet en particulier
- Peut être appelée sans instanciation de la classe
- En général n'accède pas aux membres non statiques
- Déclaration :

```
static type membre();
```

- Appel :

```
classe::membre();  
ou objet.membre();
```

## Fonction membre statique (2)

- Exemple :

```
class livre {  
    static int getnblivre()  
        {return nblivre;}  
  
    ...  
  
    private :  
    static int nblivre; ...  
};  
  
...  
cout << "nblivre :" << livre::getnblivre() << endl;  
livre l1("C++");  
cout << "nblivre :" << l1.getnblivre();
```

# Fonction membre constante

- Fonction qui ne peut pas :
  - modifier les attributs des objets de la classe
  - retourner une référence ou un pointeur non constant sur une donnée membre

- Syntaxe :

```
type classe::methode const ;  
type classe::methode const { ... }
```

- renforce les contrôles du compilateur !

## Fonction membre constante (2)

- Exemple :

```
class pile    { ...  
    bool isEmpty() const {  
        return (tete==0 ? 1 : 0) ;  
    }  
} ; ...
```

```
class String  { ...  
    int longueur(void) const {  
        return strlen(chaine) ;  
    }  
} ;
```

# Accesseur

- Accesseur en consultation

```
int get_x() const {return x ;}  
const int get_x() const {return x ;}
```

- Accesseur en consultation/modification

```
int& get_x() {return x ;}  
point A ;  
A.get_x()++ ;  
const int& get_y() {return y ;}  
A.get_y()++ ; // erreur
```

# Surcharge des méthodes

- Même nom à différentes fonctions
- résolution par le compilateur en fonction du nombre et de la nature des arguments  $\Rightarrow$  pas d'ambiguïté

```
point::point()                // constructeur 1
{
    x = 0 ; y = 0 ; }
point::point(int abs)         // constructeur 2
{
    x = abs ; y = 0 ; }
point::point(int abs, int ord) // constructeur 3
{
    x = abs ; y = ord ; }
```

- utiliser les arguments par défaut !

# Protection des objets

Toute fonction d'une classe peut accéder à n'importe quel membre de n'importe quel objet de la classe

3 types de passage d'objets en paramètre

- transmission des objets par valeur
- transmission des objets par pointeur
- transmission des objets par référence



# Problème de transmission

- Par valeur
  - objet possédant des pointeurs sur des zones allouées dynamiquement ; recopie des pointeurs (pas des zones pointées)
- Par référence ou "adresse" d'un objet
  - Objet modifiable
  - Non modifiable si const

# Objet en retour

Même cas que pour un objet en argument :

- transmission par valeur
- transmission par pointeur
- transmission par référence

## Objet en retour (2)

- Problème de transmission d'adresse ou de référence dans le cas d'un objet local à la fonction !
- pointe sur un objet qui n'existe plus !

```
point point::inverse() // inverse d'un point
// DANGER : point & point::inverse() ;
{
    point inv ;
    inv.x = y ; inv.y = x ;
    return inv ;}
```

# Auto-référence

- Rappel : une fonction membre a accès à l'objet appelant
- Auto-référence = accès à l'adresse de l'objet appelant
- `this`
  - variable définie implicitement
  - pointeur de type `X*` (classe `X`)
  - `*this` = objet appelant
- **Intérêt** : enchaînement des opérations !

## Auto-référence (exemple 1)

```
class point ;  
...  
void point::affiche()  
{  
    cout << "Adresse du point: " << this << endl;  
    cout << "Valeurs : " << x << "," << y ;  
}  
  
int point::identité(const point & p)  
{  
    return ((this->x == p.x) && (this->y == p.y)) ;  
}
```

## Auto-référence (exemple 2)

```
class liste {  
    liste* precedent , * suivant ;  
public:  
    void insere(liste*) ;  
} ;  
  
void liste::insere(liste* l)    // insere &l  
{  
    l->suivant = suivant ;    // dans liste  
    l->precedent = this ;    // en this  
    suivant->precedent = l ;  
    suivant = l ;  
}
```

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies**
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

10

Chapitre 10 - Exceptions



## Fonctions amies

- Approche objet : principe d'encapsulation
- On peut être amené à vouloir assouplir l'encapsulation dans certaines configurations particulières
- Exemple :
  - classe vecteur/classe matrice
  - concevoir une fonction de calcul du produit vecteur par matrice
  - accesseurs ou fonction `friend`
- But de la fonction amie: autoriser à une fonction membre d'une classe (e.g., vecteur) d'accéder aux données privées d'une autre classe (e.g., matrice)

## Fonctions friend

- Fonction extérieure à la classe
  - Accès aux données privées de la classe
  - Déclaration d'amitié dans la déclaration de la classe
- Plusieurs types d'amis
  - fonction extérieure (indépendante), amie d'une classe
  - fonction membre d'une classe et amie d'une autre classe
  - fonction amie de plusieurs classes
  - toutes les fonctions membres d'une classe sont amies d'une autre classe

## Fonction extérieure amie d'une classe

```
//Declaration point.h
class point
{
    int    x, y ;
    public :
        point(int abs = 0, int ord = 0)
        { x = abs ; y = ord ; }
        friend bool egal(point, point) ;
} ;
//Implementation hors d'une classe
bool egal(point p, point q)
    { return ((p.x == q.x) && (p.y == q.y)) ;      }
//Utilisation
point    a, b ;
egal(a, b) ;
```

## Méthode membre d'une classe amie d'une autre classe

- Il faut préciser dans la déclaration de la classe de la fonction par l'opérateur ::

```
friend bool point::egal(point) ;
```

- accès autorisé aux objets privées de la classe amie

```
class A
{
    ...
    friend int B::fn(int , A) ; // fn est amie de A
} ;
```

```
class B
{
    ...
    int fn(int , A) ;          // fn est membre de B
} ;
```

```
int      B::fn(int x, A objetA)
{
    ... }

```

## Amie de plusieurs classes

```
class A
{
    ...
    friend int fn(A, B)    //    fn est amie de A
};

class B
{
    ...
    friend int fn(A, B)    //    fn est amie de B
};

int    fn(A objetA , B objetB)
{    ...    }
```

## Toutes les méthodes amies

- Déclaration globale d'amitié !

```
class A;  
  
class B  
{  
    ...  
};  
  
class A  
{  
    ...  
    friend class B ;  
};
```

## Règles de l'amitié

- L'amitié n'est pas transitive
  - si une classe A est amie d'une classe B, elle-même amie d'une classe C, alors A n'est pas amie de la classe C par défaut.
- L'amitié n'est pas héritée
  - si une classe A est amie d'une classe B et que la classe C est une classe fille de la classe B, alors A n'est pas amie de la classe C par défaut.

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs**
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)



# Plan du cours II

## 10 Chapitre 10 - Exceptions

## Surdéfinition d'opérateurs - Surcharge

- Définition (rappel): attribuer le même nom à des fonctions différentes. Lors d'un appel, le choix de la bonne fonction est effectué par le compilateur suivant le nombre et le type des arguments.
- Cas particulier: surdéfinition des opérateurs
  - en C++, une classe est un type à part entière → opérateurs intégrés: les objets se comportent comme les types de base ( $a+b$ ,  $a-b$ , ...)
  - notation : `operator+` ou `operator +` pour l'opération `+`
- Remarques
  - `operator+()` est une fonction
  - Utilisation possible avec la notation fonctionnelle

## Forme canonique d'une classe

- Méthodes prédéfinies pour chaque classe X

```
X::X() ; // constructeur
```

```
X::~~X() ; // destructeur
```

```
X::X(const X&) ; // constructeur par copie
```

```
X& X::operator=(const X&) ; //opérateur d'affectation
```

```
X& X::operator&(const X&) ; //opérateur d'adressage
```

```
X& X::operator , (const X&) ; //opérateur de séquençement
```

- Attention, pour le copy constructor et l'operator=, il faut les redéfinir si on doit recopier des pointeurs

# Surdéfinition des opérateur

- 2 façons de faire
  - par fonction amie
  - par fonction membre
- Exemple: addition de 2 points

## Surdéfinition par fonction amie

- Fonction extérieure, amie de la classe point, prend 2 points en argument

- `point operator+(point&, point&) ;`

```
class    point
{
    ...
    friend point operator+(point&, point&);
    ...
};
//Implementation
point operator+(point a, point b)
{
    point p;
    p.x = a.x + b.x;
    p.y = a.y + b.y;
    return p;
}
```

- `p+q` est équivalent à `operator+(p,q)`

## Surdéfinition par une méthode

- Premier opérande implicite (objet appelant la fonction membre), un seul paramètre

```
class    point
{
    ...
    public:
        point operator+(point &);
};
//Implementation .cpp
point point::operator+(point a)
{
    point p;
    p.x = x + a.x;
    p.y = y + a.y;
    return p;
}
```

- Dissymétrie entre les 2 opérandes:  $p+q$  équivaut à  $p.operator+(q)$

## Exemple d'opérateur : les flux d'affichage

- Redéfinir « pour un type d'objet donné
  - comportement à l'affichage

```
#include <iostream> // pour les flux de sortie

ostream& operator<<(ostream & os, const X & o) {
    os << "je dispose de " << o.x << "Tutut";
    return os;
}
```

- Déclarée en friend dans la classe concernée
- Notez l'usage des références qui permettent de mettre en cascade ces opérateurs (lvalue)

## Règles de surdéfinition

- Se limiter aux opérateurs existants: impossible d'inventer de nouveaux symboles
- Conservation de la pluralité (unaire, binaire...) de l'opérateur initial
- Conservation de la priorité et de l'associativité (\* avant +...)
- Pas d'hypothèse sur la commutativité
- Tous les opérateurs sont surdéfinissables, sauf `.` `.*` `::` `?:`  
`sizeof` `typeid` `xxx_cast<>()`
- y compris `cast`, `new` et `delete` !

### Règle des trois

si le constructeur, ou le constructeur par défaut ou le destructeur est redéfini alors les trois méthodes doivent être redéfinies



## Règles de surdéfinition (suite)

- On ne peut surdéfinir un opérateur que s'il comporte au moins un argument de type classe
- L'opérateur doit être **membre** ou pour les fonctions amies comporter **au moins un** opérande de type classe
  - sinon surcharge pour des types de base (ce qui est interdit)
- Les opérateurs `=`, `[]`, `()`, `->`, `new`, `delete`, `new[]`, `delete[]` doivent être définis comme **membre**
  - assure que le 1er opérande est une l-value
  - pas de surdéfinition sur les types de base
- Liberté totale de la signification ! Ne soyez pas trop tordus...

# Opérateur new

- surdéfinition par une fonction membre

```
void* operator new(size_t) ;
```

- 1er argument de type `size_t` : taille de l'objet à allouer
- valeur de retour de type `void*` : adresse de l'emplacement alloué pour l'objet

- Exemple :

```
#include <cstdint> // pour size_t
```

```
void* operator new(size_t sz) {  
    cpt++;  
    cout << "Il y a " << cpt << "objets";  
    return ::new char[sz]; // appel new predefini  
}
```

# Opérateur delete

- surdéfinition d'une fonction membre

```
void operator delete (void *);
```

- 1er argument de type pointeur sur la classe correspondante  
void\*
- 2e paramètre : taille de la zone à restituer
- aucune valeur de retour (void)

```
void operator delete (void * dp)
{
    cpt--;
    ::delete (dp);
}
```

## Opérateurs \* & ->

- Redéfinir \*
  - si les objets peuvent être utilisés dans des expressions manipulant des pointeurs
- Redéfinir &
  - si l'objet doit renvoyer une adresse autre que la sienne
- Redéfinir ->
  - si la classe encapsule d'autres classes
  - retour d'un type où -> est encore applicable

# Opérateur fonctionnel()

- Opérateur n-aire
- Exemple matrice

```
double& matrice::operator()  
(unsigned short int i, unsigned short int j)  
{  
    return data[i][j];  
}
```

```
double matrice::operator()  
(unsigned short int i, unsigned short int j) const  
{  
    return data[i][j];  
}
```

```
matrice m;  
m(2,3)=10.0;
```

# Opérateurs de conversion

- Conversions d'un type en un autre type  $X \rightarrow Y$
- 2 possibilités :

- constructeur de Y à 1 argument de type X

```
matrice(const vecteur &)
```

- opérateur de transtypage Y() dans X

```
class vecteur{  
    ...  
    operator matrice(); //pas de type de retour  
    ...  
};
```

# Conversion en type de base

- marche aussi pour les types de base
- Surdéfinition d'opérateur unaire classique

```
classe :: operator type_base()
```

- Exemples :

```
complex :: operator double();  
chaine :: operator char const *() const;
```

- pas de valeur de retour pour cast (nécessairement le type de retour)

# Conversion de type de base

```
// Definition
class Point
{ public:
    Point(int abs) {x = abs; y = 0;}
};
// Utilisation
Point a;
a = Point(3); //conversion implicite de int en point
```



## Conversion de type de base

Pour avoir l'inverse, il faut redéfinir l'opérateur `int()` dans la classe `point`

```
// Definition
class Point
{
    operator int ();
}
//Implementation
Point::operator int ()
{ return x; }
// Utilisation
Point a(3,4);
int n1;
n1 = int (a); //la conversion est explicite avec un cast
n1 = a; //conversion implicite
a+3; //appel implicite
```

## Exemples de conversion (1)

```
int n1, n2 ;  
point p1, p2 ;    ...  
n1 = int(p1) ;  
n1 = p2 ;  
n1 = p1 + 3 ;  
n2 = p2 + 1.25 ;  
void fn(int) ;  
fn(p1) ;  
double z ;  
z = p1 + 1.25; // x + 1.25: point -> int -> double
```

# Conversion par l'opérateur =

```
operator =      (int abs = 0 , int ord = 0)
{
    x = abs ; y = ord ;
}
...
p1 = 3 ;
```

- pas de conversion par appel au constructeur
- appel de operator=

# Constructeur comme opérateur

```
Utilisation du constructeur comme opérateur (conversion t
//Fonction membre
point operator + (point a) ;
...
point    p1, p2 ;
int      n ;      ...
p1 = 5 + p2 ;      //      erreur (5.operator+(p2))
p1 = p2 + 5;      // OK
//Fonction amie
friend   point    operator +      (point a, point b) ;
...
point    p1, p2 ;
int      n ;      ...
p1 = 5 + p2 ;      //      OK
p1 = p2 + 5 ;      //OK
```

# Conversion entre classes

```
class point{
    double x, y ;
    ...
    operator complexe(){ // operateur de cast
        complexe c ;
        c.reel = x ; c.imag = y ;
        return c ;
    }
};

class complexe{
    ...
    // constructeur
    complexe (point & p)
    {    reel = p.x ; imag = p.y ;}
};
```

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage**
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

10 Chapitre 10 - Exceptions

# L'héritage en C++

- Rappel sur la notion d'héritage
  - Possibilité de dériver une classe fille (**sous-type**)
  - à partir d'une classe mère (**sur-type**)
  - en ajoutant des données et/ou des méthodes
- Chaque instance de la classe dérivée contient les données de sa classe de base par héritage, sans devoir les redéfinir



# Notion d'héritage

- L'héritage permet
  - la structuration d'une conception
  - la réutilisation du code déjà écrit
  - la redéfinition de fonctions (**spécialisation**)
  - l'ajout de nouvelles fonctionnalités (**extension**)
  - la mise en facteur
  - d'imposer d'une interface à une classe
- Exemples
  - *Voiture, Bus, Camion* : sous-ensembles du type Véhicule
  - *Carré, Cercle, Triangle* : sous-ensembles du type Figure

# Types d'héritage en C++

- **Trois** types d'héritage :
  - héritage simple
  - héritage multiple
  - héritage virtuel
- L'héritage ne se borne pas à un seul niveau
- Toute classe peut servir de classe de base

# Modes d'héritage

- 3 qualifications possibles de l'héritage :
  - classe de base **privée**
  - classe de base **publique** (le plus courant)
  - classe de base **protégée**
- gère les différents droits d'accès aux membres
- Le mode d'accès est facultatif
  - par défaut : héritage privé

```
class <sous-classe>:[mode] <sur-classe>
```

Exemple:

```
class Triangle : public Polygone {};
```

## Modes d'héritage (suite)

- Quel que soit le mode d'héritage, la **partie privée** de la sur-classe est **inaccessible** par les fonctions membres de la sous-classe (ce qui est privé dans Polygone est inaccessible dans Triangle, mais les variables existent)
- La déclaration du destructeur en privé interdit toute dérivation car destruction d'objets impossible
- Les constructeurs ne sont pas hérités

## Modes d'héritage (suite)

- Lorsqu'un membre est redéfini dans une classe dérivée, il reste toujours possible d'accéder aux membres de même nom de la base en utilisant l'opérateur de résolution de portée (::)
  - si l'accès est autorisé ...
- Exemple (Moto hérite de Véhicule) :

```
class Vehicule { void affiche(); };  
class Moto : public Vehicule { void affiche(); };  
  
void Moto::affiche()  
{  
    cout << "Moto :" << endl;  
    Vehicule::affiche();  
}
```

# Classe dérivée de base privée

- Syntaxe :

```
class <sous-class>:[ private ] <sur-class>
```

- La partie **public** et **protected** de la classe de base deviennent **private**, les fonctions membres de la sous-classe et les amis y ont donc accès (mais pas les classes extérieures)
- La partie **private** de la sur-classe est présente en tant que zone de stockage sur les instances de la sous-classe mais les fonctions membres spécifiques à la sous-classe *ne peuvent pas y accéder*

## Classe dérivée de base privée (suite)

- Conséquences de la dérivation privée :
  - l'héritage privé est moins fréquent que l'héritage public
  - interdit à un utilisateur d'une classe dérivée l'accès à l'interface de sa classe de base
    - héritage d'un détail de mise en œuvre sans nécessité conceptuelle
  - 2 cas précis :
    - lorsque toutes les fonctions utiles de la classe de base sont redéfinies dans la classe dérivée
    - lorsque l'on souhaite adapter l'interface d'une classe

## Exemple d'héritage privé

```
class Personne {  
    string nom;  
    public :  
        void affiche();  
};  
  
class Societaire : private Personne {  
    int cotisation;  
    public :  
        void affiche();  
};  
  
int main() {  
    Societaire un_soc;  
    un_soc.affiche();  
}
```



# Classe dérivée de base publique

- Syntaxe :

```
class <sous-class> : public <sur-class>
```

- La partie **public** de la classe de base s'ajoute à l'interface de la sous-classe (héritage d'interface)
- Les membres de la classe de base conservent leur statut dans la classe dérivée
- C'est la forme d'héritage la plus courante, mais c'est aussi celle qui lie le plus fortement les classes
- Conséquences de la dérivation publique
  - Conversion implicite dans la classe de base ; un pointeur sur une classe dérivée peut être implicitement converti en un pointeur sur une classe de base publique (pas le cas avec l'héritage privé)
  - Les fonctions membres de la sous-classe ne peuvent pas accéder à la partie privée héritée

## Héritage public : partie protégée

- La partie protégée (**protected**) d'une classe permet de distinguer entre les classes clientes d'une classe et les classes qui ont un rapport privilégié avec elle (ses sous-classes et ses classes amies)
- les membres protégés se comportent comme
  - des membres privés vis à vis des clients
  - des membres publics vis à vis des dérivées et amies

# Classe dérivée de base protégée

- Syntaxe

```
class <sous-class>: protected <sur-class>
```

- Rare en pratique
- les membres publics de la classe de base deviennent membres protégés de la classe dérivée
- les autres membres conservent leur statut (protected reste protected et private reste private)

## Modes d'accès

- Il est possible de modifier individuellement le mode de protection d'un membre d'une classe de base
- La modification du mode de protection d'un membre ne peut **en aucun cas** augmenter la visibilité d'un membre de la classe de base

```
class A {  
    protected:    int p, q;  
    public:        int v, w;  
};
```

```
class B: public A {  
    public :  
        A::q; // erreur q ne peut devenir public  
    protected :  
        A::p; // p reste protégé dans B  
    private :  
        A::v; // devient privé dans B  
};
```

## Exemple de classe dérivée

```
class Personne{  
    string nom;  
    int age;  
public :  
    void afficher( );  
    void init(string , int );  
};
```

```
void Personne::afficher( ){  
    cout<<"nom: "<<nom<<endl;  
    cout<<"age: "<<age<<endl;  
}
```

```
void Personne::init(string __nom, int __age){  
    nom = __nom;  
    age = __age;  
}
```

## Exemple de classe dérivée (suite)

```
class Chercheur : public Personne{  
    string labo;  
    public:  
    void afficher( );  
    void toutinit(string , int , string );  
};
```

```
void Chercheur::afficher( ){  
    Personne::afficher( );  
    cout<<"Labo : "<<labo<<endl;  
}
```

```
void Chercheur::toutinit(string _nom, int _age,  
                           string _labo){  
    init(_nom, _age);  
    labo=_labo;  
}
```

## Exemple de classe dérivée (fin)

```
void main( ){
    Personne *p_p; Personne per;
    per.init("Dupont", 30);
    Chercheur *p_c; Chercheur ch;
    ch.init("Einstein", 50);
    ch.toutinit("Lavoisier", 20, "Academie");
    ch.afficher( );
    per = ch;                // OK
    ch = per;                // Non, manque d'infos
    p_c = &ch;
    p_p = p_c;               // OK (héritage public)
    p_c = &per;              // Erreur, une personne
                             // n'est pas un chercheur
}
```

# Dérivation et constructeur

- Si la classe de base n'a pas de constructeur la classe dérivée n'a d'obligation de fournir un constructeur que pour ses propres attributs
- Si la classe de base a un constructeur sans argument, c'est lui qui sera appelé si la classe dérivée n'en fournit pas
- Si les constructeurs de la classe de base ont des arguments, alors la classe dérivée doit fournir un constructeur pour le passage des arguments
- Ordre de construction
  - d'abord la base
  - ensuite la classe dérivée
- Destructeurs dans l'ordre inverse



# Liste d'initialisation

- Exemple :

```
class Base
{
    int b;
public :
    Base(int arg1) : b(arg1)
        {...} // b=arg1
};
```

```
class Dérivée : public Base
{
    int d;
public :
    Dérivée(int arg1 , int arg2): Base(arg1), d(arg2)
        {...}
};
```

# Héritage multiple

- La syntaxe est dérivée de l'héritage simple, les classes de base sont séparées par des virgules

```
class ma_classe:           public base1 , public base2  
{ ... };
```

- L'ordre des classes de base est important !
  - les bases seront initialisées dans l'ordre où elles figurent dans la liste d'initialisation du constructeur

## Exemple d'héritage multiple

```
class b1{
private:
    char* c;
public:
    b1(char* x) : c(x) {}
    void voir(){
        cout<<"c: " << c << endl;}
};
```

```
class b2{
private:
    char* j;
public:
    b2(char* x) {j=x;}
    void voir() {
        cout<<"j: " << j << endl;}
};
```

## Exemple d'héritage multiple (2)

```
class db1b2 : public b1, public b2{
private:
    int j;
public:
    db1b2(char*, char*, int);
    void voir( );
};

db1b2::db1b2(char* x, char* y, int z) :
    b1(x), b2(y), j(z)    { }

void db1b2::voir( ){
    b1::voir( ); b2::voir( );
    cout<<" "<< j << endl;
}
```

## Exemple d'héritage multiple (3)

```
main( ){  
    char* s1="Hello World";  
    char* s2="Vive C++";  
    char* s3="Bon Courage";  
    char* s4="STROUSTRUP \oe uvre pour C++";  
    b1 x(s3);  
    x.voir( ); //affiche Bon courage  
    b2 y(s4);  
    y.voir( ); //affiche STROUSTRUP \oe uvre pour C++  
    db1b2 d(s1,s2,421);  
    d.voir( ); //affiche Hello World Vive C++ 421  
}
```

# Héritage virtuel

- Ne pas confondre avec les fonctions virtuelles (chapitre suivant !)
- Capacité à hériter d'une classe qui est partagée par les dérivées en cas d'héritage multiple
  - Schéma d'héritage en diamant
- Permet qu'un objet de la classe dérivée ne contienne qu'un seul sous-objet partagé
- Mise en œuvre :
  - déclaration des classes de base en tant que classe virtuelle dans la spécification de l'héritage

## Héritage virtuel (suite)

- Exemple

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

- Héritage en diamant
- *A apparaît* deux fois dans D
- Utilisation de *virtual* sur les classes B et C pour que le contenu de A n'apparaisse qu'une fois dans D.
- A doit disposer d'un constructeur sans argument (ou pas de constructeur du tout)

## Héritage virtuel (suite)

- Exemple

```
class A { public: int a; void f(); };  
class B : public virtual class A {int b; ...};  
class C : virtual public class A {int c; ...};  
class D : public class B, public class C {int d; ...};
```

- la classe D ne contiendra qu'une seule occurrence de la classe A
- les références aux membres de A dans D ne sont donc pas ambiguës (si non virtuel, il faut distinguer B::a de C::a)
- Pas d'info pour A dans les constructeurs de B et C
- Le constructeur d'une classe virtuelle est appelé avant les autres: A, B, C, D



# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme**
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

## 10 Chapitre 10 - Exceptions

# Notion de polymorphisme

## Définition

faculté d'une méthode à pouvoir s'appliquer à des objets de classes différentes permettant ainsi des implémentations plus abstraites et générales

- Son principe repose sur l'héritage :
  - Possibilité, Y dérivant de X (Y est un X), d'utiliser Y où on peut utiliser X
  - Un pointeur sur un type d'objet peut recevoir l'adresse de n'importe quel objet descendant

# Polymorphisme en C++

- Réalisation : redéfinition (**spécialisation**) d'une méthode héritée pour tenir compte des propriétés locales de la classe

```
class Employé {  
    int prime() ; ... }
```

```
class Ingénieur{  
    int prime();...}
```

```
class Chercheur{  
    int prime();...}
```

- On souhaite utiliser la bonne version de `prime()` en fonction de l'objet appelant

# Liaison dynamique

- Liaison : mécanisme qui associe le code d'une méthode à exécuter et l'appel de la méthode
- **Liaison dynamique** :
  - liaison résolue à l'exécution
  - repose sur l'héritage et le polymorphisme
  - C++ : liaison tardive (sous-type du type statique)
- Mécanisme **RTTI**
  - Run-Time Type Identification
  - Identification du type à l'exécution

## Exemple de liaison statique

- Liaisons statiques

```
Polygone* p = new Polygone ;  
Rectangle* r = new Rectangle ;  
p->affiche() ; // affiche() de Polygone  
r->affiche() ; // affiche() de Rectangle  
p = r ;  
p->affiche() ; // affiche() de Polygone  
Rectangle r2 = *p ; // erreur  
Rectangle* r3 = p ; // erreur
```

- Ne fonctionne qu'avec des pointeurs ou des références

# Fonction virtuelle

- Permet en C++ la liaison tardive
- choix de la fonction appelée à l'exécution
- appel de la fonction appropriée pour une classe dérivée de façon transparente à l'utilisateur
  - Permet d'encapsuler les détails d'implémentation dans les classes dérivées
  - la base ne déclare que l'interface (virtuelle pure)

## Fonction virtuelle (suite)

- Fonction membre particulière
- appelée au moyen d'un pointeur ou d'une référence sur une classe de base
- résolution par le compilateur selon le type de l'objet appelant (**RTTI**)
- Mot clef : `virtual`  
`virtual void f();`
- seulement nécessaire dans la déclaration de la classe



## Fonction virtuelle (suite)

- Un constructeur ne peut pas être virtuel
- Un destructeur peut être virtuel
- **Conseils** :
  - si une classe est dérivable alors déclarer le destructeur comme virtuel
  - déclarer virtuelle les fonctions qui peuvent être redéfinies dans les classes dérivées
  - ne pas mettre toutes les fonctions virtuelles car plus lent (indirection supplémentaire par le pointeur virtuel et la table virtuelle)

# Fonction virtuelle pure

- Une fonction virtuelle qui est déclarée avec "= 0" (ou NULL) après la liste des arguments, est une fonction virtuelle pure

```
class C {  
    virtual void f() const = 0; };
```

- Toute classe qui déclare ou hérite une virtuelle pure est une classe abstraite
  - Classe non instanciable
  - Obligation de redéfinir f dans la classe héritée

# Classe abstraite

Une classe de base abstraite est utilisée pour déclarer une interface sans avoir à donner toute la définition de cette interface

- cette déclaration spécifie les opérations abstraites supportées par tous les objets des classes dérivées
- permet de définir une classe de généralisation
  - Concept abstrait (Vehicule, Forme ...)
- Une classe dérivée qui hérite mais ne redéfinit pas une fonction virtuelle pure est aussi abstraite
  - Il appartient aux classes dérivées de fournir les définitions des opérations abstraites

# Exemple

```
class vehicule {  
public:  
    virtual double accelerer(double) = 0;  
    virtual double vitesse() = 0;  
};
```

```
vehicule v;    // Erreur à la compilation
```

```
class voiture : public vehicule {  
public:  
    virtual double accelerer(double);  
    virtual double vitesse();  
};
```

## Exemple 2

```
void polymorphe() {  
    point p1;  
    point tabpoint[]={point(), point(3,5), point(1,-1)};  
    ObjetGraphique* tab[2];  
    tab[0] = new Rectangle(p1,1,2,Vert);  
    tab[1] = new Polygone(3,tabpoint,Bleu);  
    for (int i=0;i<2;i++) {  
        tab[i]->surface(); delete tab[i];  
    }  
}
```

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage**
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

## 10 Chapitre 10 - Exceptions

## Transtypage

Opération de conversion d'un type à un autre basée en général sur le polymorphisme

- spécificité du C++
- Avantages
  - plus sûr que le transtypage à la C
  - repose sur l'identification dynamique des types
  - opérateurs non redéfinissables
- 4 opérateurs de transtypage :
  - transtypage statique
  - transtypage dynamique
  - transtypage de constante
  - transtypage de réinterprétation



# Transtypage statique

- Pas de vérification des types dynamiques lors du transtypage
- **conversion contrôlée à la compilation**
- si certain de la validité du transtypage (l'expression est du type cible), conversion entre type de même famille
- syntaxe :

**static\_cast** <type\_cible> (expression )

## Exemple de static\_cast

```
int i ;
double d;
long n ;
char * p_c;
i = n ; // OK
// écriture ANSI
i = static_cast<int> (n);
i = d; // warning
i = (int) n ; // à éviter
p_c = i; // warning ==> erreur a l'exécution
p_c = static_cast<char *> (i); // erreur
```

# Transtypage dynamique

- Convertir une expression en un pointeur ou une référence d'une classe
- **conversion contrôlée à l'exécution**
- permet de tester des égalités de type
- syntaxe :

`dynamic_cast<type_cible>(expression)`

```
// La classe Cercle hérite de la classe ObjetGraphique
ObjetGraphique *g;
Cercle * moncercle = new Cercle;
g = moncercle;
Cercle *c = dynamic_cast<Cercle *>(g);
```

# Transtypage dynamique

- Condition nécessaire :
  - il existe une méthode virtuelle dans la définition de la classe (RTTI)
- Conversion **réussie** à l'exécution si :
  - la source est de même type ou d'un type dérivé de la cible
  - la cible est de type `void*`
- Sinon la conversion **échoue** et :
  - rend le pointeur `NULL`
  - déclenche l'exception `bad_cast`

## Exemple de dynamic\_cast

```
class Employe { ... virtual ... };
class Directeur : public Employe { ... };

int main() {
    Employe *tab_emploi[2];
    tab_emploi[0] = new Employe("Dupont");
    tab_emploi[1] = new Directeur("J2M");
    ...
    Directeur *direc = dynamic_cast<Directeur*> (tab_emploi[i])
    if (direc != NULL) direc->dirige();
    ...
}
```

## Exemple dynamic\_cast (suite)

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

---

```
A a ; B b ; C c ; A* pA ; A* pA2 ; B* pB ; C* pC ;  
pA      = &a ;  
pB      = dynamic_cast<B *>(pA) ;           // ???  
pA      = &b ;  
pB      = dynamic_cast<B *>(pA) ;           // ???  
pA2     = dynamic_cast<A *>(pA) ;           // ???  
pC      = dynamic_cast<C *>(pA) ;           // ???  
pB      = &b ;  
pA2     = dynamic_cast<A *>(pB) ;           // ???  
pC      = dynamic_cast<C *>(pB) ;           // ???
```

## RTTI et classe `type_info`

```
class type_info {  
    public :  
        // opérateurs de comparaison de type  
        int operator ==(const type_info &) const;  
        int operator !=(const type_info &) const;  
        // fonction renvoyant le nom de la classe  
        const char * name() const;  
        // fonction permettant de définir un ordre sur les objets  
        bool before (const type_info &) const;  
        ...  
};  
  
// type de l'objet appelant  
cout << typeid(*this).name();
```

- `typeid()` est un opérateur !

# Opérateur type\_id

```
#include <typeinfo>
```

```
class Point {virtual void affiche() {};}
class Pointcol : public point {void affiche() {};}

```

```


```

```
int main() {
    Point p; Pointcol pc;
    Point *adp;
    adp = &p;
    cout << "type adp : " << typeid(adp).name();
    cout << "type *adp : " << typeid(*adp).name();
    adp = &pc;
    cout << "type adp : " << typeid(adp).name();
    cout << "type *adp : " << typeid(*adp).name();
}
```



# Transtypage de constante

- Transtypages dont le type destination est moins contraint que le type source vis-à-vis des mots-clés `const` et `volatile`
- pour des références et des pointeurs
- ajoute ou supprime `const` ou `volatile`
- ne permet pas les transtypages classiques

`const_cast<type>(expression)`

## Exemple de const\_cast

```
int n = 2 ;  
const int * pa1 = &n ;  
int* pa2 ;  
pa2 = pa1 ; // ???  
pa2 = (int *) pa1 ; // à éviter  
  
// écriture ANSI  
pa2 = const_cast<int *> (pa1) ;  
pa1 = const_cast<const int *> (pa2) ;
```

# Transtypage de réinterprétation

- Réinterpréter les données d'un type en un autre type
- conversion la plus dangereuse
- Tout n'est pas possible !
  - la valeur cible doit contenir au moins autant de bits que la source
- Syntaxe :

```
reinterpret_cast<type_cible>(expression)
```

# Transtypage de réinterprétation

```
// conversion de int en pointeur de char
int i;
char * ptr = reinterpret_cast<char*>( i );

// conversion de int en référence sur char
char & ref = reinterpret_cast<char&>( i );

// conversion pointeur de char -> pointeur de double
double * ptr2 = reinterpret_cast<double*>( ptr );
```

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)**
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

## 10 Chapitre 10 - Exceptions

# La généricité

Un gabarit (patron, template, modèle, type générique) est une fonction ou une classe associée à des **paramètres formels** qui peuvent être des **types**

- Lorsque le compilateur analyse l'instanciation d'un template, il remplace le type formel par le type réel et **génère le code associé**
- **avantages des patrons** :
  - Permet de définir une fonction générique s'adaptant à n'importe quel type
  - Permet la mise en œuvre d'un nombre "infini" de surcharges d'une certaine fonction
  - Évite de dupliquer le code afin de créer un nouveau type

# Patrons de fonctions

- C'est une extension de la surdéfinition de fonctions
  - écriture unique de la fonction
- Plus restrictif que la surdéfinition de fonctions
  - code unique,
  - substitution de paramètres
- éventuellement déclarée comme `friend`



## Rappel : surcharge de fonction

- Surcharge de fonctions :

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

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

- Version template :

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

# Analyse du compilateur

- Instanciation du template :
  - Fonction correspondant exactement (même nom, nombre et type de paramètres)
  - Fonction correspondante après des conversions implicites

```
main( )  
{  
    int n=4, p=12;  
    float x=2.5, y=3.25;  
    cout << "min(n,p) = " << min(n, p) << endl;  
    cout << "min(x,y) = " << min(x, y) << endl;  
}
```

# Patron de fonction de type class

```
• class vect{
    int x, y;
public:
    vect(int abs=0, int ord=0);
    ostream & operator<<(ostream &, const vect &);
    friend bool operator < (vect, vect);
};

bool operator< (vect a, vect b){
    return (a.x*a.x + a.y*a.y) < (b.x*b.x + b.y*b.y);
}

void main(){
    vect u(3, 2), v(4, 1), w;
    w = min(u, v) ;
    cout << "min(u, v) = "; w.affiche();
} // Résultat : min(u, v) = 3 2
```

## Paramètres de type d'un patron

- Les paramètres peuvent intervenir dans :
  - l'en-tête
  - des déclarations de variables locales
  - toutes les instructions exécutables...

```
template <class T, class U> fct(T a, T *b, U c){  
    T x;  
    U *adr;                                ...  
    adr = new U[10];                        ...  
    n = sizeof (U);                         ...  
}
```

- Mot clef *typename* et *class* sont interchangeables, SAUF lors de la déclaration de template dans un template: obligation d'utiliser *class*

# Instanciation d'un patron

- Correspondance exacte des types obligatoire

```
int n; char c;  
unsigned int q; const int r = 10;  
int t[10]; int *adi; ...  
min(n, c)           // erreur  
min(n, q)           // erreur  
min(n, r)           // erreur en version 3  
                      // min(const int, const int) en  
min(t, adi) // erreur en version 3  
                      // min(int *, int *) en ANSI
```

# Instanciación d'un patron

```
template <class T, class U> T fct(T x, U y, T z)
{
    return x + y + z;
}
```

```
main( ) {
    int n = 1, p = 2, q = 3;
    float x = 2.5, y = 5.0;
    cout << fct(n, x, p) << endl;
        // affiche (int) 5
    cout << fct(x, n, y) << endl;
        // affiche (float) 8.5
    cout << fct(n, p, q) << endl;
        // affiche (int) 6;
    cout << fct(n, p, x) << endl;
        // erreur
}
```

# Type formel pointeur

- Exemple

```
#include <iostream>
template <class T> int compte(T *tab, int n=5){
    int nz = 0;
    for(int i=0; i<n; i++)
        if(!tab[i]) nz++;
    return nz;
}

main( ){
    int t[] = { 5, 2, 0, 2, 0, -1, 0};
    char c[5] = { 0, 12, 0, 0, 0};
    cout << "compte(t)=" << compte(t, 7) << endl;
    cout << "compte(c)=" << compte(c) << endl;
}

// résultat : compte(t) = 3
//             compte(c) = 4
```

# Conversion dans les patrons

```
template <class T, class U, class V> V min(T a, U b, V bidon)
{ if (a < b) return a; // a converti dans le type V
  else return b;      // b converti dans le type V
}
```

```
int n;
char c, c1;
unsigned int q;
min(n, c, n); // résultat de type int
min(c, q, n); // résultat de type int
min(c, c1, q); // résultat de type unsigned int
```



# Surdéfinition de patron

```
#include <iostream>

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){
    return (min(min(a, b), c));
}

main(){
    int n=12, p=15, q=2;
    float x=3.5, y=4.25, z=0.25;
    cout << min(n, p) << endl ;
    cout << min(1, 2.0) << endl ; // erreur
    cout << min<int>(1, 2.0) << endl ;
    cout << min(n, p, q) << endl ;
    cout << min(x, y, z) << endl ;
}
```

## Spécialisation de fonctions de patrons

- Définition d'une ou plusieurs fonctions particulières qui seront utilisées en lieu et place de celle instanciée par un patron

```
// patron de fonctions
template <class T> T min (T a, T b) { ... }

// version spécialisée pour le type char *
char * min (char * cha, char * chb) { ... }

int n, p;
char * adr1, * adr2 ;
// appelle la fonction instanciée par le patron général
// soit ici : int min (int, int)
min (n, p)
// appelle la fonction spécialisée
// char * min (char *, char *)
min (adr1, adr2)
```

# Patron de classe

- Déclaration

```
template <typename monType> class | struct | union nom;
```

- Si les méthodes de la classe ne sont pas définies dans la déclaration de la classe, elles devront elles aussi être déclarées template

- `// monType représente le paramètre template de la classe`  

```
template <typename monType>  
type classe <monType>::nom(paramètres_méthode) {...}
```

## Exemple de patrons de classe

```
template <class T> class point{
    T x; T y;
public :
    point(T abs = 0, T ord = 0);
    void affiche();
};
```

```
template <class T> void point <T> :: affiche(){
    cout <<"coord : " <<x<< " " <<y<< endl;
}
```

```
main(){
    point<int> a_int(3, 5);
    a_int.affiche();
    point<char> a_char('d', 'y');
    a_char.affiche();
    point<double> a_double(3.5, 2.3);
    a_double.affiche();
}
```

## Patron à valeur par défaut

```
template<class T = char> class Chaine;
```

```
// instantiation explicite de Chaine<char>  
template Chaine<>;
```

```
template<class T, class U=int> class A;  
// déclaration d'un objet A<double,int>  
A<double> a;  
// déclaration d'un objet A<double,double>  
A<double, double> aa;
```

# Paramètres de patron de classe

```
template <class T, class U, class V>
class essai{
    T x;
    U t[5];
    ...
    V fonction(int, U);
    ...
};
```

- Instanciation d'une classe patron

```
class essai <int, float, int> ce1;
class essai <int, int*, double> ce2;
class essai <char*, int, obj> ce3;
class essai <float, point<int>, double> ce4;
class essai <point<int>, point<float>, char *> ce5;
```

# Spécialisation d'un patron de classes

- Spécialiser tout ou une partie des fonctions membre
- Exemple

```
template <class T> class point { ..... } ;  
  
// fournir une version spécialisée pour  
// le cas où T est le type char  
class point <char>  
{  
    // nouvelle définition de la classe point  
    // pour les caractères  
} ;
```

# Le patron du patron

- classe template en tant que type générique !

- syntaxe :

```
template <template <class Type> class Classe [ ,... ] >
```

- type : type générique utilisé dans la déclaration de la classe template Classe
- template : paramètres template template
- un paramètre template template peut avoir une valeur par défaut (classe template déclarée avant)



## Exemple de patron de patron

```
template <class T> class Tableau  
{ // Définition de la classe template Tableau  
};
```

```
template <class U, class V,  
        template <class T> class C=Tableau>  
    class Dictionnaire  
{  
    C<U> Clef;  
    C<V> Valeur;  
    ...  
};
```

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)**

# Plan du cours II

## 10 Chapitre 10 - Exceptions

# STL : STandard Template Library

- Bibliothèque de classes/adaptateurs/algorithms reposant sur la généricité
- Motivations :
  - la flemme, la réutilisation du code
  - la facilité d'écriture
  - la lisibilité du code

## Philosophie de la STL

Le principe de la STL repose sur l'indépendance des

- données
- structures
- algorithmes

# STL : présentation

La STL contient :

- des containers :
  - de séquence
  - associatifs
- des algorithmes :
  - sort, find, search, ...
  - copy, swap, for\_each, ...
- des itérateurs :
  - permettent le parcours des containers

## STL : présentation (2)

- Les différents types et fonctions de la STL sont éprouvés et fiables :
  - Gain de temps au développement
  - Minimisation des erreurs potentielles
  - réutilisation

Avant de coder un nouvel algorithme, toujours se demander si son équivalent n'existe pas dans la STL

# STL - Les containers de séquences

- Vector
  - extensible et relocalisable
  - accès aléatoire par index en  $O(1)$
  - insertions et retraits à la fin en  $O(1)$
  - insertions et retraits lents au milieu
- List (liste circulaire doublement chaînée)
  - insertions et retraits n'importe où en  $O(1)$
  - accès au début et à la fin en  $O(1)$
  - accès lent aux autres éléments
- Deque (*Double Ended Queue* = tampon circulaire dynamique)
  - exactement pareil que Vector pour la complexité
  - pas d'insertion au milieu mais à l'avant/arrière
  - bon compromis vitesse d'accès / souplesse du remplissage

# Les Vecteurs

```
#include <vector>
```

- Vecteur = tableau généralisé qui contient des éléments de même type
- Permet d'accéder à un élément du tableau via un index compris entre 0 et n-1, avec n la taille du vecteur
- modification dynamique de la taille **automatique**
- méthodes importantes :

```
size(), empty(), clear(), resize(taille)  
front(), back()  
Push_front(val), push_back(val)  
Pop_front(), pop_back()  
Operator : =, ==, <
```



## Les vecteurs (2)

- Accès aux éléments :
  - avec des [] comme un tableau - non sécurisé
  - avec at(i) - vérifie les débordements, lance des exceptions, plus sûr
- construction :

```
vector<int> nums;  
vector<double> vals(20);  
vector<Figure *> dessin;
```

# Les listes

```
#include <list>
```

- Liste doublement chaînée
- avantages :
  - Ajout/suppression en  $O(1)$
- inconvénients :
  - Accès aux éléments:  $O(n)$

- méthodes importantes :

`size()`, `empty()`, `clear()`, `resize(taille)`

`front()`, `back()`

`Push_front(val)`, `push_back(val)`

`Pop_front()`, `pop_back()`, `remove(val)`

Operator : `=`, `==`, `<`

Spécialisations: `sort()`, `unique()`, `reverse()`

- Insertions/suppressions au milieu : nécessite les itérateurs

# Les Deque (Double Ended Queue)

```
#include <deque>
```

- Se comporte comme un vecteur
- Cependant, l'ajout et la suppression d'éléments au début se fait en  $O(1)$
- Parfois plus avantageux que `vector`, surtout dans le cas où l'on a besoin d'un accès aux éléments efficaces avec ajout/suppression aux extrémités

# STL - Les containers associatifs

- Set ( liste triée )
  - Chaque élément doit être unique ( pas de doublons )
  - accès aléatoire par clé ( l'objet lui-même est la clé )
- Multiset
  - Un set qui permet les doublons
- Map ( association clé / élément )
  - une clé pour un élément ( pas de doublons )
  - accès aléatoire par clé
- Mulimap
  - Une map où les éléments peuvent être multiples pour une clé

# Les ensembles

```
#include <set>
```

- Chaque valeur d'un ensemble est unique
- Les opérations sont toutes en  $O(n)$  (ajout, suppression, accès)
- En interne, les éléments sont toujours triés par ordre croissant
- méthodes importantes :

```
Insert(val), erase(val)
```

```
Find(val) (renvoie un itérateur)
```

```
Clear()
```

```
Size(), empty()
```

```
Operator : =, ==, <
```

# Les dictionnaires (map)

```
#include <map>
```

- Association clé/valeur (clés uniques)

- méthodes importantes :

```
Insert ( val ), erase ( val ), Find ( val )  
                                     (renvoie un itérateur)
```

```
Size ( ), empty ( ), Clear ( )
```

```
Operator : =, ==, <, []
```

- [] permet d'accéder à un élément par sa clé

# STL : les itérateurs

- L'itérateur est une généralisation des pointeurs. Il permet de parcourir en séquence les éléments d'un conteneur.
- Chaque container fournit un type d'itérateur
- exemple :

```
vector<int>::iterator
```

- Chaque conteneur fournit deux itérateurs accessibles par les méthodes :
  - Begin() : it sur le premier élément
  - End() : it sur le premier élément qui n'appartient pas au container
  - utilisé comme condition d'arrêt pour un parcours par exemple

# Exemple

- un petit exemple de copie

```
void main() {  
    list<int> l(4,0); // l vaut 0::0::0::0::[]  
    vector<int> v(10,1); // v vaut [1,1,1,1,1,1,1,1,1,1]  
    vector<int>::iterator i =  
        copy(l.begin(), l.end(), v.begin());  
    copy(v.begin(), i, ostream_iterator<int>(cout, " "));  
    copy(i, v.end(), ostream_iterator<int>(cout, " "));  
}  
  
// affiche  
// 0 0 0 0 1 1 1 1 1 1
```



## Exemple (2)

- parcours inverse

```
void main(){
    int arr[] = {1,4,9,16,25};
    list<int> liste(arr, arr+5);
    list<int>::reverse_iterator revit;
    revit = liste.rbegin();
    while (revit != liste.rend()) cout << *revit++ << " ";
}

// affiche
// 25 16 9 4 1
```

## Exemple (3)

- *back inserter*

```
void main(){
    int t1[] = { 1, 3, 5, 7, 9};
    deque<int> dq1(t1, t1+5);
    int t2[] = { 2, 4, 6};
    deque<int> dq2(t2, t2+3);

    copy(dq1.begin(), dq1.end(), back_inserter(dq2));
    for (int i=0 ; i<dq2.size() ; i++)
        cout << dq2[i] << " ";
}

// affiche
// 2 4 6 1 3 5 7 9
```

# STL : l'adaptateur

- Adaptateur de container

Adaptateur < Container < Truc > >

- Stack (pile)

- Peut être implémentée avec Vector, List, Deque
- Principe LIFO : push et pop à la même extrémité

- Queue (file)

- Peut être implémentée avec List, Deque
- Principe FIFO

- Priority\_queue (file avec priorité)

- Dépend d'une fonction de comparaison:
- `priority_queue< Container<Truc> , fcomp<Truc> >`

# STL : les foncteurs

- Abstraction de la notion de fonction
- Foncteur = Objet dont la classe définit l'opérateur () , l'opérateur de fonction
- Permettent d'appliquer des fonctions sur les objets manipulés
- il existe déjà plusieurs foncteurs dans la STL
  - opérations arithmétiques, booléens, logiques, etc...

```
#include <iostream>
#include <functional>
```

```
template <class T, class F>
    T applique(T i, T j, F foncteur){
        return foncteur(i, j);
    }
void main(void){
    plus<int> foncteur_plus;
    cout << applique(2, 3, foncteur_plus) << endl;
}
```

# STL : algorithmes

- Algorithmes non-modifiants
  - Parcours et recherche, ne modifient pas les éléments du container
- Algorithmes modifiants
  - Modifient les éléments d'un container en réarrangeant, enlevant, modifiant les valeurs contenues
- Algorithmes de mutation
  - Change l'ordre des éléments du container sans changer les valeurs
- Algorithmes numériques :
  - Réalise un calcul sur les valeurs contenues

# STL - petit exemple : le tri

```
#include <iostream>
#include <list>
#include <algorithm>

List<char> initC(char*c){
    List<char> majax;
    while (*c != '\0') majax.push_back(*c++);
    return majax;
}

void main() {
    List<char> gerard = initC("le plus super des magiciens");
    gerard.sort();
    gerard.unique();
    for (List<char>::iterator i = gerard.begin() ; i != gerard.
}
```

# STL : difficultés

- Les erreurs Template peuvent être illisibles !
- soit la fonction :

```
template <class T>
void printReverse(const vector<T> &vect) {
    for (vector<T>::reverse_iterator curr = vect.rbegin();
         curr != vect.rend(); ++curr)
        cout << *curr << ", ";
}
```

## STL : difficultés (2)

```
Error E2034 c:\Borland\Bcc55\include\rw/iterator.h 442:
Cannot convert 'const int *' to 'int *' in function
reverse_iterator<int *>::reverse_iterator(const reverse_itera
Error E2094 vect.cpp 19: 'operator!=' not implemented intype
'reverse_iterator<int *>' for arguments of type
'reverse_iterator<const int *>' in function
printReverse<int>(const vector<int , allocator<int> > &)
Error E2034 c:\Borland\Bcc55\include\rw/iterator.h 442:
Cannot convert 'const int *' to 'int *' in function
reverse_iterator<int *>::reverse_iterator(const reverse_itera
Warning W8057 c:\Borland\Bcc55\include\rw/iterator.h 442:
Parameter 'x' is never used in function
reverse_iterator<int *>::reverse_iterator(const reverse_itera
*** 3 errors in Compile ***
```

- une fois compilé...
- Pourquoi ? : reverse\_iterator différent de  
const\_reverse\_iterator



## Exemple de vecteur complet

```
template <class T>
ostream& op<<(ostream& out, const vector<T> &vect) {
    out << "(";
    for (int i = 0; i < vect.size(); i++)
        out << vect[i] << ",";
    out << ")";
    return out;
}
```

```
template <class T>
void printReverse(const vector<T> &vect) {
    cout << "(";
    for (vector<T>::const_reverse_iterator
        curr = vect.rbegin();
        curr != vect.rend(); curr++)
        cout << *curr << ",";
    cout << ")";
}
```

## Exemple : suite

```
void main() {
    srand(time(NULL));
    vector<int> ints;
    cout << "Initial size ==" << ints.size()
          << "\n Initial capacity ==" << ints.capacity() << endl;
    for (int i = 0; i < 5; i++)
        ints.push_back(rand() % 20);
    cout << "\n Now, size == " << ints.size()
          << "\n Capacity == " << ints.capacity();
    cout << "\n vector: " << ints << "\n vector reversed: ";
    printReverse(ints);
    try {
        ints.at(100) = 20;
    } catch (out_of_range oor) {
        cout << "\n Tried to set mem 100,";
        cout << "\n but caught exception: " << oor.what();
    }
    sort(ints.begin(), ints.end());
    cout << "After sort, vect: " << ints;
```

## Exemple : suite

Produira :

```
//Initial size == 0
//Initial capacity == 0
//Now, size == 5 Capacity == 256
//vector: (7,3,16,14,17,)
//vector reversed: (17,14,16,3,7,)
//Tried to set mem 100 to 20,
//but caught exception: index out of range in function:
//      vector:: at(size_t) index: 100 is
//greater than max_index: 5
//After sort, vector: (3,7,14,16,17,)
```

# Les enfants de la STL

- De nombreuses librairies template existent sur le net :
  - Boost Lib
  - GTL (Graph Template Library)
  - VTK (Vizualisation Toolkit)
  - Lapack++,
  - etc.

# Plan du cours I

- 1 Chapitre 1 - Bases de C++: types, variables, pointeurs
- 2 Chapitre 2 - Classes
- 3 Chapitre 3 - Fonctions amies
- 4 Chapitre 4 - Surdéfinition des opérateurs
- 5 Chapitre 5 - Héritage
- 6 Chapitre 6 - Polymorphisme
- 7 Chapitre 7 - Transtypage
- 8 Chapitre 8 - Généricité (templates)
- 9 Chapitre 9 - STL (Standard Template Library)

# Plan du cours II

## 10 Chapitre 10 - Exceptions

# Définition

Les exceptions permettent une gestion propre des erreurs à **l'exécution** en C++.

Principes :

- détecter des conditions "exceptionnelles" pouvant être rencontrées au cours de l'exécution d'un programme (problèmes d'allocation mémoire, mauvaise manipulation de l'utilisateur, etc.)
- traiter ces incidents
- dissocier **erreurs/traitements**

C++ offre un mécanisme puissant de gestion des exceptions

# Principe général

- Exception = rupture de séquence dans l'exécution du programme lancée par **throw**
- **throw**(expression avec un type donné)
  - le type de l'expression permet l'identification de l'exception
  - branchement à un gestionnaire d'exception
- Le choix du gestionnaire est déterminé par le type de l'exception : **catch**(expression d'un type donné)
- pour pouvoir être détectée, une exception doit se trouver au sein d'un bloc **try** qui est immédiatement suivi des gestionnaires d'exceptions



## Un premier exemple

```
//déclaration d'une classe pour un type d'exception (vide)
class vect_depassement{};

//Classe vecteur d'entier
class vect{
    int nb_elem;
    int * tab ,
public :
    vect(int){tab = new int[nb_elem = n];}
    ~vect(){ delete [] tab;}

    int & operator [] (int) {
        if (i<0 || i>nb_elem){
            vect_depassement l;
            throw (l);
        }
        return tab[i];
    }
};
```

## Un premier exemple (2)

```
// Interception d'une exception vect_depassement
int main(){
    try
    {
        vect v(3);
        v[4] = 1; //dépassement d'indice
    }
    catch(vect_depassement l)
    {
        cout << "Exception : depassement d'indice" << endl;
        exit(-1);
    }
}

// Sortie du programme :
// Exception : depassement d'indice
```

## Un premier exemple (3)

- Dans l'exemple précédent :
  - un seul type d'exception
  - un seul gestionnaire d'exception
  - pas d'information transmise au gestionnaire (classe `vect_depassement` vide)
- Le gestionnaire est défini indépendamment des fonctions pouvant déclencher les exceptions : l'utilisateur de la classe `vect` peut définir des gestionnaires différents suivant les utilisations
- si le bloc `try` n'était pas présent, l'exception lancée par `[]` provoquerait un arrêt du programme.

## Reprise de l'exécution

- Le gestionnaire d'exception n'est pas obligé de mettre fin au programme
- Dans ce cas, l'exécution du programme se poursuit à **la suite du bloc try concerné**

destruction des objets définis au sein du bloc try et non détruits au moment où l'exception est lancée ?

## Deuxième exemple

```
// On définit deux nouvelles classes d'exception
```

```
class vect_depassement{  
public:  
    int out; //indice de depassement  
    vect_depassement(int i) { out = i; }    //constructeur  
};
```

```
class vect_bad_size{  
public:  
    int nb;          //taille demandee  
    vect_bad_size(int n) { nb = n; }    //constructeur  
};
```

## Deuxième exemple (2)

```
vect::vect(int n){ //constructeur
    if (n <= 0){
        vect_bad_size c(n); //taille non admissible
        throw c;
    }
    tab = new int[nb_elem = n]; //construction normale
}

int & vect::operator [] (int i){
    if (i<0 || i>nb_elem){
        vect_depassement l(i); //depassement indice
        throw (l);
    }
    return tab[i];
}
```

## Deuxième exemple (3)

```
int main(){
    try{
        vect v(-1); //provoque exception vect_bad_size
        v[5] = 3; //provoquerait exception vect_depassement
    }
    catch(vect_depassement l){
        cout << "Exception depassement indice : "
              << l.out << endl;
        exit(-1);
    }
    catch(vect_bad_size c){
        cout << "Exception vect_bad_size : "
              << c.nb << endl;
        exit(-1);
    }
    return 0;
}
// sortie du prog:
// Exception vect_bad_size : -1
```

## Remarques

- La seconde exception n'est pas lancée : la fin du bloc try n'est pas exécutée
- Les membres des classes d'exceptions permettent de transmettre des informations au gestionnaires
- Le choix du gestionnaire se fait **en fonction du type** de l'exception



# Reprise de l'exécution

- Lorsque le gestionnaire d'exception ne met pas fin à l'exécution :
  - Exécution du code du gestionnaire
  - Reprise de l'exécution à la suite du bloc try concerné
- Le mécanisme de traitement des exceptions **supprime toutes les variables automatiques** des blocs dont on provoque la sortie
  - mais ne s'applique pas aux données dynamiques !

# Règles de choix du gestionnaire

- Le gestionnaire d'exception reçoit toujours **une copie** de l'expression mentionnée à `throw`
- règles de sélection du gestionnaire
  - Règle 1 : recherche d'un gestionnaire correspondant au type exact mentionné dans `throw`

```
catch (A a)
catch (A & a)
catch (const A a)
catch (const A & a)
```

- comme le type est passé **par valeur**, chacune des expressions ci-dessus conviennent

## Règles de choix du gestionnaire (2)

- règles de sélection du gestionnaire
  - Règle 2 : recherche d'un gestionnaire correspondant à **une classe de base** du type mentionné dans `throw`
  - permet un regroupement plus ou moins fin du traitement des exceptions

```
class vect_erreur {...};  
class vect_depassement : public vect_erreur {...};  
class vect_bad_size : public vect_erreur {...};
```

- permet :

```
try {...} catch (vect_erreur e) {...}
```

- ou

```
try {...}  
  catch (vect_depassement l) {...}  
  catch (vect_bad_size c) {...}
```

## Règles de choix du gestionnaire (3)

- règles de sélection du gestionnaire
  - Règle 3 : recherche d'un gestionnaire correspondant à un pointeur sur une classe dérivée type mentionné dans `throw` (si le type est lui-même un pointeur).
  - Règle 4 : Recherche d'un gestionnaire correspondant à un type quelconque représenté par `catch (...)`

```
catch (exception_type1)
{ //traitement }
catch (...)
{ //traitement }
catch (exception_type2)
{ //traitement }
```

# Cheminement des exceptions

- Quand une exception est levée par une fonction :
  - On cherche d'abord un gestionnaire dans le bloc try (éventuel) associé à cette fonction selon les règles précédentes
  - En cas d'échec (on en trouve pas ou pas de bloc try), on cherche dans un bloc try (éventuel) d'une fonction appelante, etc.
  - Si aucun gestionnaire n'est trouvé : appel de la fonction `terminate` qui par défaut met fin à l'exécution
    - dans tous les cas, dès qu'un gestionnaire qui convient a été trouvé, l'exception est traitée

## Redéclenchement d'une exception

- l'instruction `throw` sans argument retransmet l'exception au niveau englobant.
  - permet de compléter un traitement standard par un traitement spécifique
- remarque : lorsque le gestionnaire se trouve dans un constructeur ou un destructeur l'exception est toujours retransmise au niveau englobant

# Exemple

```
void f(){
    try{
        int n=2;
        throw n;
    }
    catch (int){
        cout << "exception int dans f" << endl;
        throw;
    }
}

int main(){
    try{f();}
    catch (int){
        cout << "exception int dans main"<<endl;
        exit(-1);
    }
    return 0;
}
```

## Spécification d'interface de fonction

- Une fonction peut spécifier les exceptions qu'elle est susceptible de provoquer (elle-même ou dans les fonctions appelées)

```
class A {  
void f1 (); // n'importe quel type d'exception  
void f2 () throw (); // ne peut pas lever d'exception  
void f3 () throw (int); // peut lever des exceptions  
                        // de type int  
void f4 () throw (int, double, X); // peut lever 3 types  
                                // d'exceptions  
};
```

- Dans ce cas, toute exception non prévue entraîne l'appel de la fonction `unexpected()`



# Exceptions de base

- déclarées dans le fichier en-tête `<stdexcept>`
- dérivent toutes d'une classe de base : `exception`
  - il est possible d'en hériter
  - fonction membre virtuelle pure `what` qui renvoie un pointeur sur une chaîne de caractères (précisant la nature de l'exception)
  - toutes ces classes possèdent un constructeur dont l'argument est une chaîne (éventuellement renvoyée par `what`)

## Exemple complet

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

class exception1 : public exception
{
public:
    exception1() {}
    const char * what() const throw() {return "exception1";}
};

class exception2 : public exception
{
public:
    exception2() {}
    const char * what() const throw() {return "exception2";}
};
```

## Exemple complet (2)

```
int main(){
    try{
        cout << "bloc try 1" << endl;
        throw exception1();
    }
    catch (exception & e){
        cout << "exception : " << e.what() << endl;
    }
    try{
        cout << "bloc try 2 \n";
        throw exception2();
    }
    catch (exception & e){
        cout << "exception : " << e.what() << endl;
    }
    return 0;
}
```