DAUPHINE UNIVERSITÉ PARIS | MIDO MATHEMATIQUES ET INFORMATIQUE DE LA DECISION ET DES ORGANISATIONS

Master Informatique des Organisations

1ère et 2ème années

Spécialité ID/MIAGE-IF/MIAGE-SITN

PSL RESEARCH UNIVERSITY

Le langage C++ (partie I)

Maude Manouvrier

- Qu'est ce que le C++ ?
- Rappels sur la gestion de la mémoire
- Premiers pas en C++ : typage, compilation, structure de contrôle, …
- Classes et objets : définitions et 1er exemple de classe
- Notions de constructeurs et destructeur
- Propriétés des méthodes
- Surcharge des opérateurs
- Objet membre

http://www.lamsade.dauphine.fr/~manouvri/C++/CoursC++_MM.html

MyCourse : M2 MIAGE ID/IF/SITN_2013-2014_C++/Python_Maude Manouvrier

Bibliographie

- ***Le Langage C++ (The C++ Programming Language)***, de Bjarne Stroustrup, Addisson-Wesley – 4ème édition, Mai 2013

- ***How to program – C and introducing C++ and Java*,** de H.M. Deitel et P.J. Deitel, Prentice Hall, 2001 – dernière édition *C++ How To Program* de février 2005

- ***Programmer en langage C++*,** 8ème Édition de Claude Delannoy, Eyrolles, 2011

- ***Exercices en langage C++*,** de Claude Delannoy, Eyrolles, 2007

Merci à

Béatrice Bérard, Bernard Hugueney, Frédéric Darguesse, Olivier Carles et Julien Saunier pour leurs documents!!

Documents en ligne

- *Petit manuel de survie pour C++* de François Laroussinie, 2004-2005, **http://www.lsv.ens-cachan.fr/~fl/Cours/docCpp.pdf**
- *Introduction à C++* de Etienne Alard, revu par Christian Bac, Philippe Lalevée et Chantal Taconet, 2000
  http://www-inf.int-evry.fr/COURS/C++/CoursEA/
- ***Thinking in C++* de Bruce Eckel, 2003**
  http://w2.syronex.com/jmr/eckel/
- **http://www.stroustrup.com/C++.html**
- **http://channel9.msdn.com/Events/GoingNative/2013/Opening-Keynote-Bjarne-Stroustrup**
- **Aide en ligne :**
  http://www.cplusplus.com/doc/tutorial/

Historique du Langage C++

- **3ème langage le plus utilisé au monde (classements TIOBE de Août 2013 *http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*) et LangPop de Avril 2013 *http://langpop.com/*)**
- **JVM (HotSpot) et une partie du noyau de Google Chrome écrits en C++**
- Première version développée par Bjarne Stroustrup de Bell Labs AT&T en 1980
- Appelé à l'origine « Langage C avec classes »
- Devenu une norme ANSI/ISO C++ en juillet 1998 (C++98 - ISO/IEC 14882) – mise à jour en 2003 (C++03)

ANSI : American National Standard Institute

ISO : International Standard Organization

Nouvelle norme C++ : C++11

- C++11 (C++0x) approuvée par l'ISO en 12/08/2011 et disponible depuis sept. 2011 (norme ISO/CEI 14882:2011)
- Quelques sites explicatifs des nouveautés de la norme 2011 :
  - `http://www.siteduzero.com/tutoriel-3-497647-introduction-a-c-2011-c-0x.html`
  - `http://en.wikipedia.org/wiki/C%2B%2B11`
  - `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`
  - `http://www.stroustrup.com/C++11FAQ.html`
  - `http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style`
  - `C++11 improvements over C++03 - http://www.cplusplus.com/articles/EzywvCM9/`

Qu’est-ce que le C++ ?

- D’après Bjarne Stroustrup, conception du langage C++ pour :
  - Être meilleur en C,
  - Permettre les abstractions de données
  - Permettre la programmation orientée-objet
- Compatibilité C/C++ [Alard, 2000] :
  - *C*++ = sur-ensemble de C,
  - C++ $\Rightarrow$ ajout en particulier de l’orienté-objet (classes, héritage, polymorphisme),
  - Cohabitation possible du procédural et de l‘orienté-objet en C++
- Différences C++/Java [Alard, 2000] :
  - C++ : langage compilé / Java : langage interprété par la JVM
  - C/C++ : passif de code existant / Java : JNI (*Java Native Interface*)
  - C++ : pas de machine virtuelle et pas de classe de base / *java.lang.object*
  - C++ : "plus proche de la machine" (gestion de la mémoire)

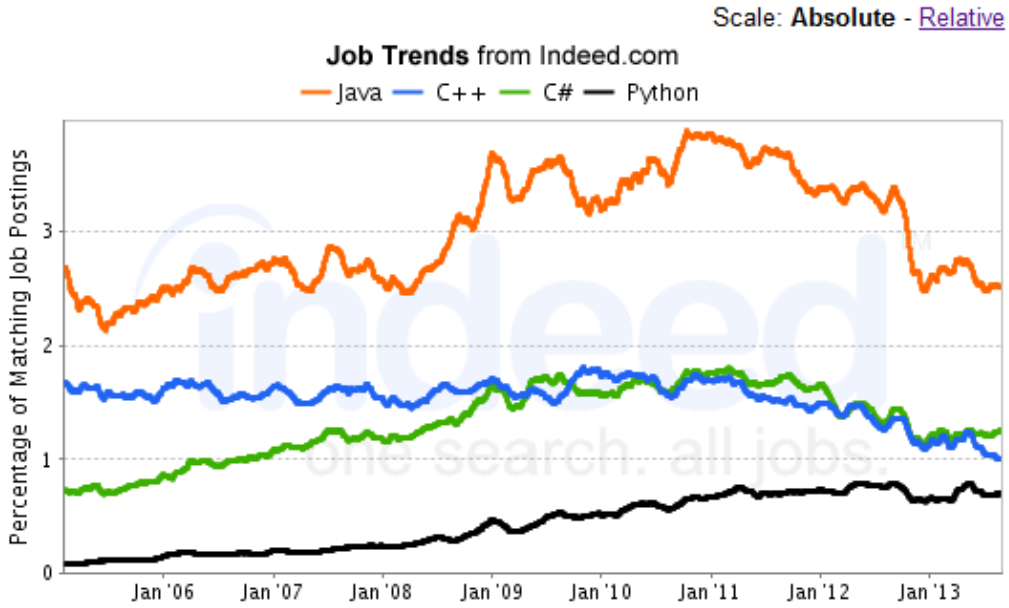
Différences Java et C++

Gestion de la mémoire [Alard, 2000] :

- Java
  - Création des objets par allocation dynamique (*new*)
  - Accès aux objets par références
  - Destruction automatique des objets par le ramasse miettes
- **C++**
  - Allocation des objets en mémoire statique (variables globales), dans la **pile** (variables automatiques) ou dans le **tas** (allocation dynamique),
  - Accès direct aux objets ou par pointeur ou par référence
  - **Libération de la mémoire à la charge du programmeur** dans le cas de l'allocation dynamique
- Autres possibilités offertes par le C++ :

  Variables globales, compilation conditionnelle (préprocesseur), pointeurs, surcharge des opérateurs, patrons de classe *template* et héritage multiple

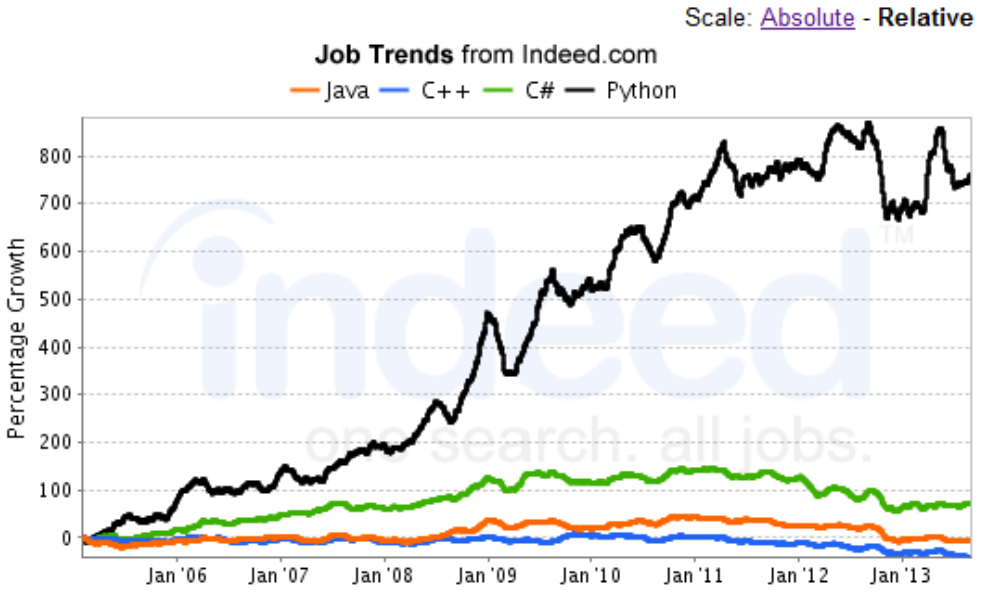
Java, C++, C#, Python Job Trends



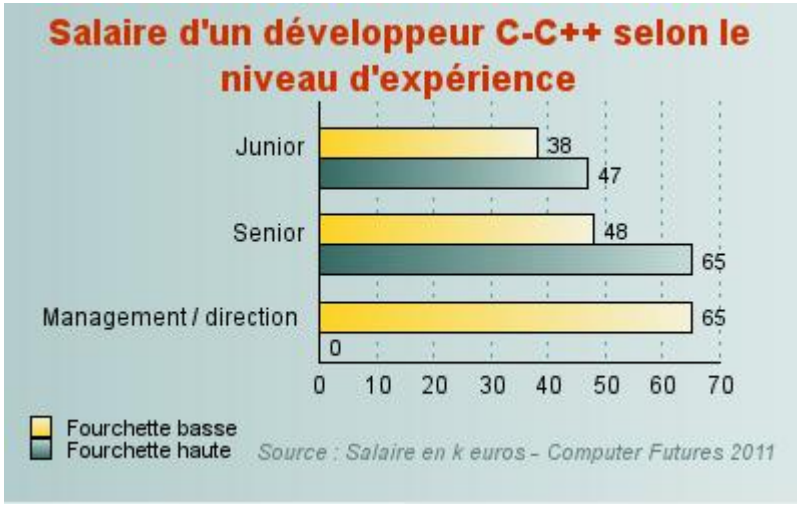
Postes en France :

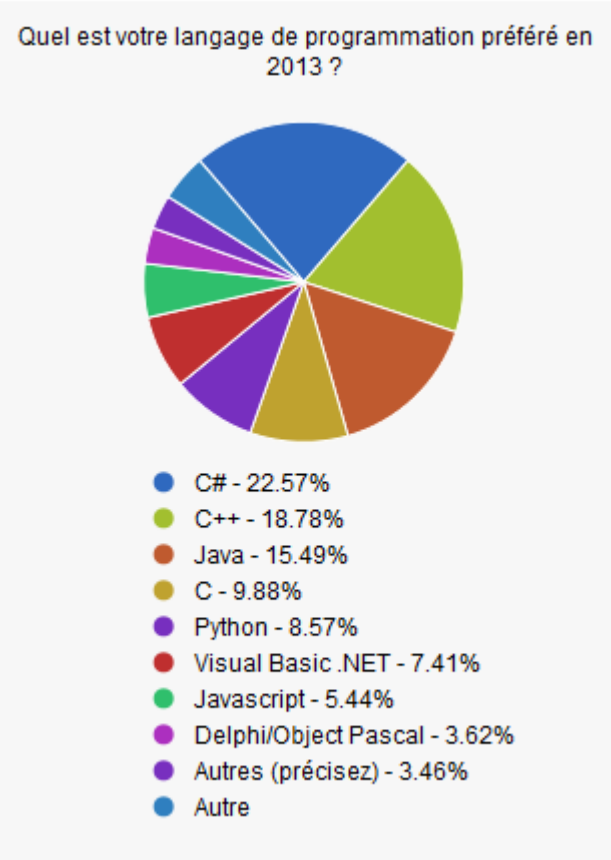
- **3246 en Java**
- **1371 en C++**
- **1292 en C#**
- **670 en Python**

Java, C++, C#, Python Job Trends



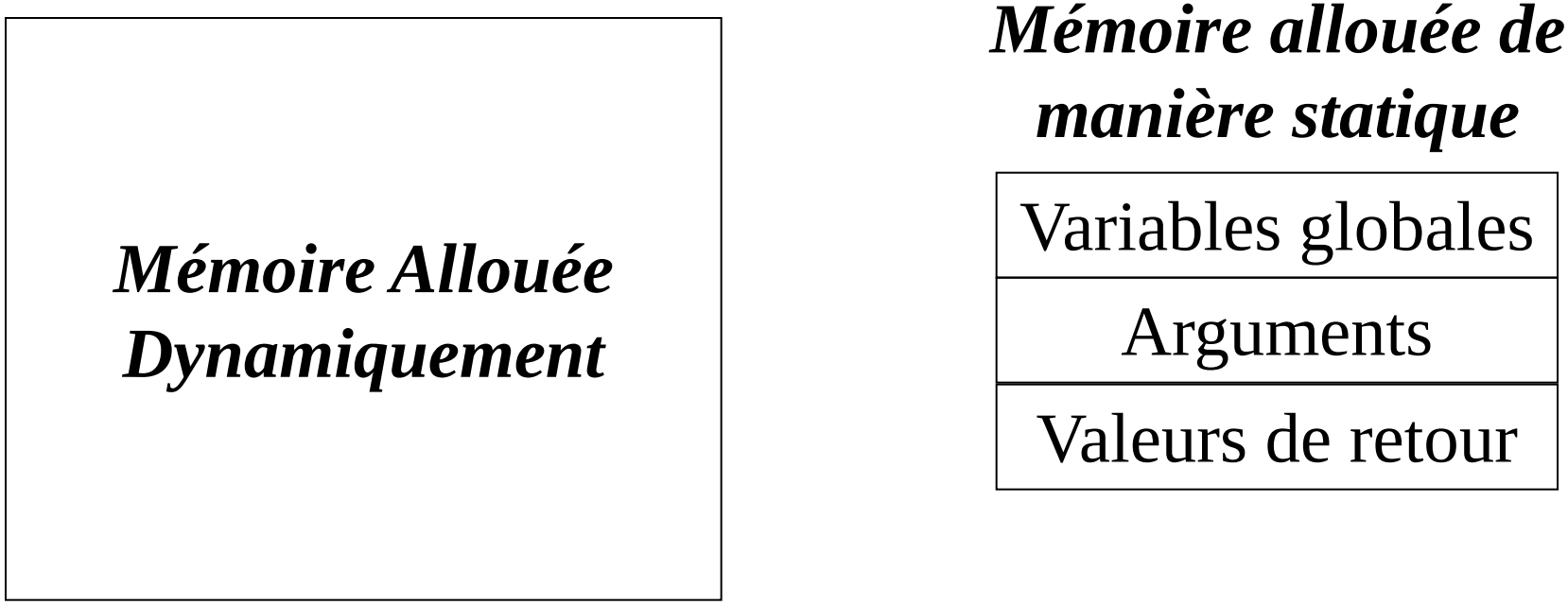
Programmation : le couple C/C++ dominent Java et C# - le couple C/C++, champion toutes catégories des développements de bas niveau (http://www.silicon.fr/langages-de-programmation-basic-fait-de-la-resistance-90859.html - 18/11/13)





http://programmation.developpez.com/actu/56168/Quel-est-votre-langage-de-programmation-prefere-en-2013-Partagez-votre-experience-sur-le-langage-de-votre-choix/ - 607 votant Mai 2013

Rappel : Pile et Tas (1/6)



Rappel : Pile et Tas (2/6)

Exemple de programme en C :

```
/* liste.h */
struct Cell
{
   int valeur;
   struct Cell * suivant;
}

typedef struct Cell Cell;

Cell * ConstruireListe(int taille);
```

Rappel : Pile et Tas (3/6)

Exemple de programme en C :

```
Cell * ConstruireListe(int taille)
{
   int i;
   Cell *cour, *tete;

   tete = NULL;
   for (i=taille; i >= 0; i--)
   {
        cour = (Cell*) malloc (sizeof(Cell));
        cour->valeur = i;
        cour->suivant = tete;
        /* Point d'arrêt 2 - cf transparent 11 */
        tete = cour;
   }
   return tete;
}
```

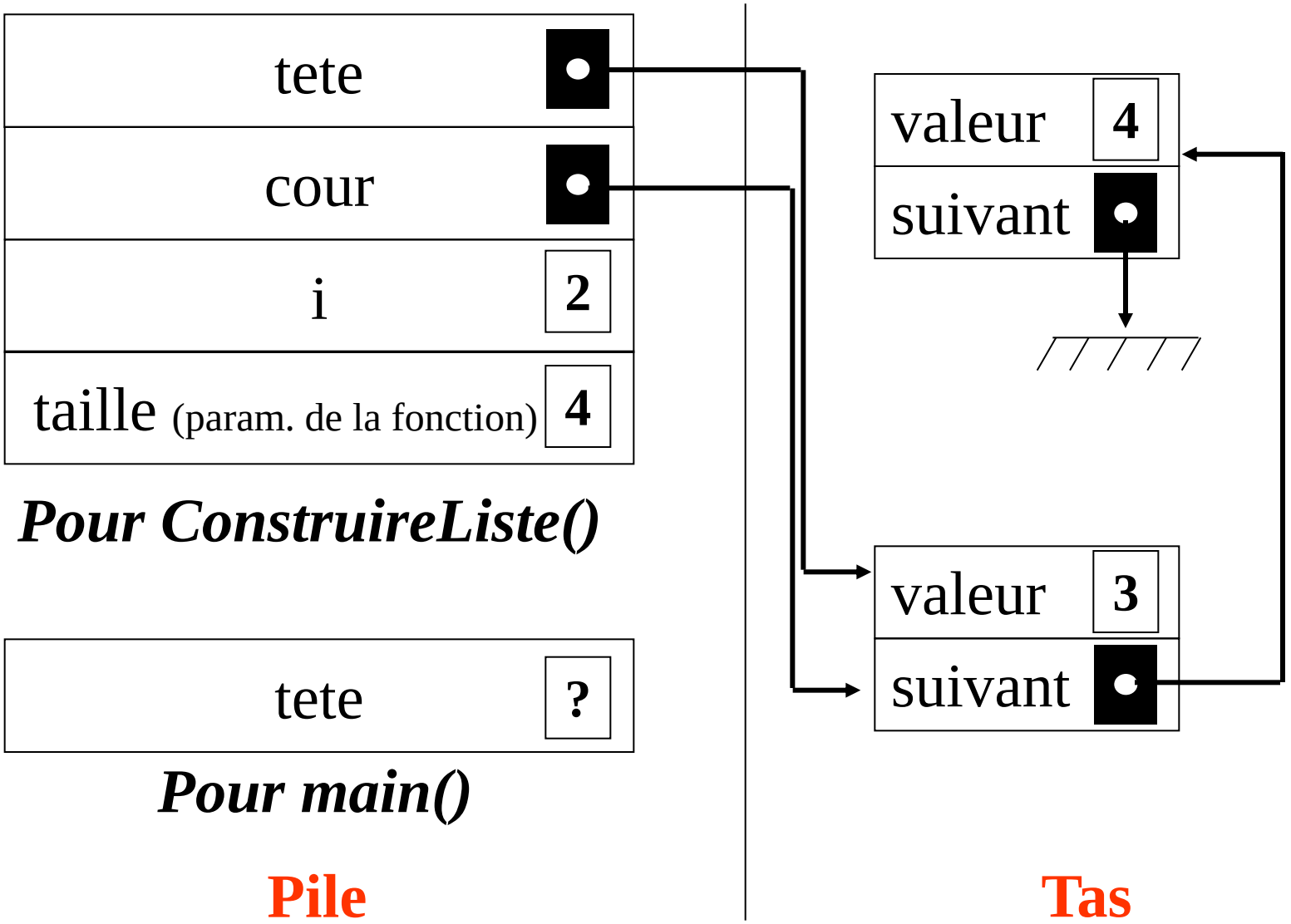
Rappel : Pile et Tas (4/6)

Exemple de programme en C :

```
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include "liste.h"

int main ()
{
   Cell * tete ;
   tete = ConstruireListe(4);
   /* Point d'arrêt 1 - cf transparent 12 */
   …
   return 1;
}
```

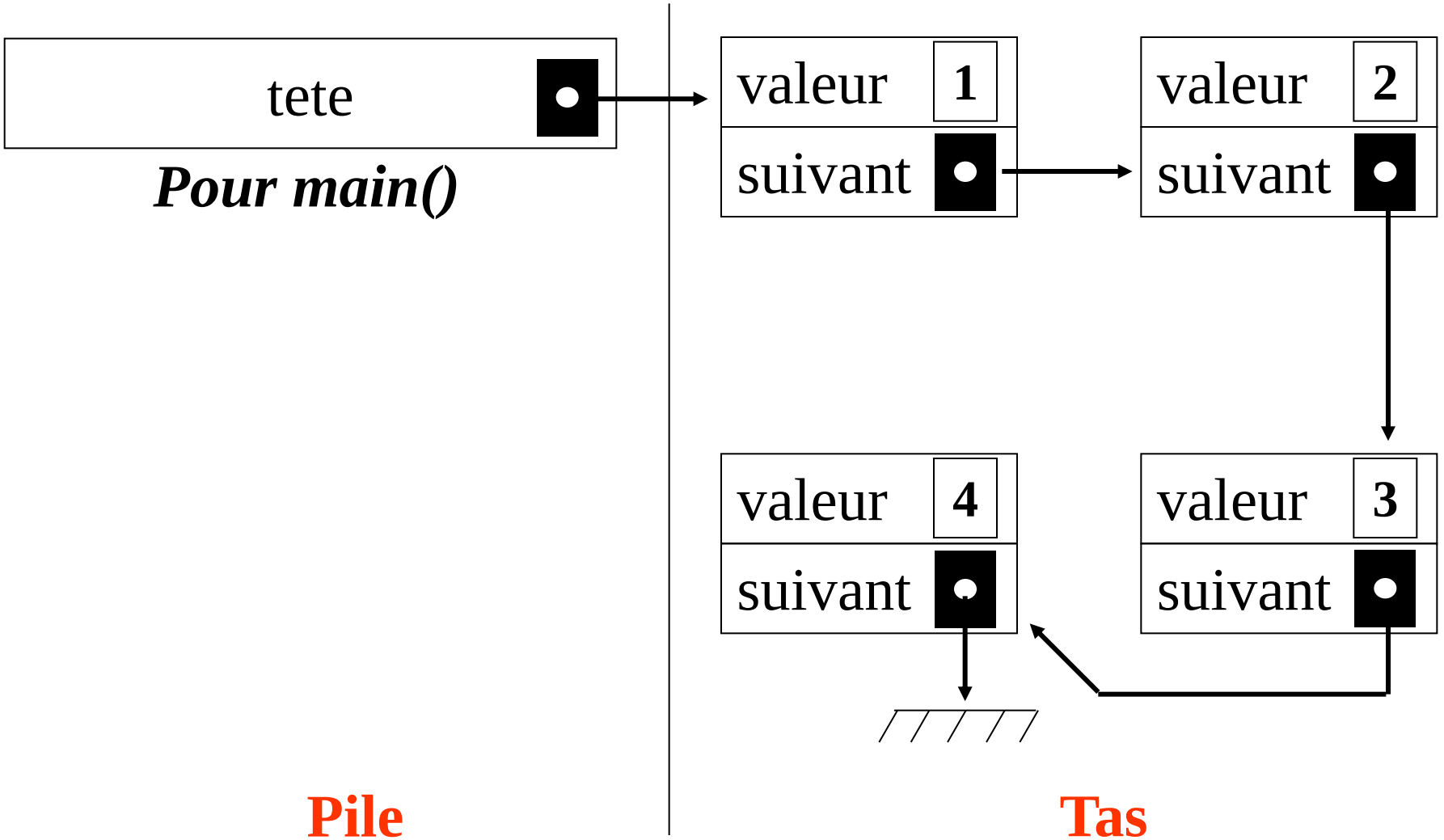
Rappel : Pile et Tas (5/6)

État de la mémoire au point d'arrêt 2 après un 2ème passage dans la boucle



Rappel : Pile et Tas (6/6)

État de la mémoire au point d'arrêt 1



Exemple de programme C++

```
/* Exemple repris du bouquin "How To Program" de Deitel et
Deitel -  page 538 */
// Programme additionnant deux nombres entiers
#include <iostream>
int main()
{
    int iEntier1;
    cout << "Saisir un entier : " << endl; // Affiche à l'écran
    cin >> iEntier1; // Lit un entier
    int iEntier2, iSomme;
    cout << "Saisir un autre entier : " << endl;
    cin >> iEntier2;
    iSomme = iEntier1 + iEntier2;
    cout << "La somme de " << iEntier1 << " et de " << iEntier2
    << " vaut : " << iSomme << endl; // endl = saut de ligne
    return 0;
}
```

`cout` et `cin` (1/2)

Entrées/sorties fournies à travers la librairie *iostream*

- **`cout << expr`$_1$ `<<` … `<< expr`$_n$**
  - Instruction affichant $expr_1$ puis $expr_2$, *etc.*
  - **`cout`** : « flot de sortie » associé à la sortie standard (*stdout*)
  - **`<<`** : opérateur binaire associatif à gauche, de première opérande **`cout`** et de 2ème l'expression à afficher, et de résultat le flot de sortie
  - **`<<`** : opérateur surchargé (ou sur-défini) ⇒ utilisé aussi bien pour les chaînes de caractères, que les entiers, les réels etc.
- **`cin >> var`$_1$ `>>` … `>> var`$_n$**
  - Instruction affectant aux variables var_1, var_2, *etc.* les valeurs lues (au clavier)
  - **`cin`** : « flot d'entrée » associée à l'entrée standard (*stdin)*
  - **`>>`** : opérateur similaire à **`<<`**

cout et cin (2/2)

Possibilité de modifier la façon dont les éléments sont lus ou écrits dans le flot :

`dec`	**lecture/écriture d'un entier en décimal**
`oct`	**lecture/écriture d'un entier en octal**
`hex`	**lecture/écriture d'un entier en hexadécimal**
`endl`	**insère un saut de ligne et vide les tampons**
`setw(int n)`	**affichage de *n* caractères**
`setprecision(int n)`	**affichage de la valeur avec *n* chiffres avec éventuellement un arrondi de la valeur**
`setfill(char)`	**définit le caractère de remplissage**
`flush`	**vide les tampons après écriture**

```
#include <iostream.h>
#include <iomanip.h>  // attention a bien inclure cette librairie

int main() {
  int i=1234;
  float p=12.3456;
  cout << "|" << setw(8) << setfill('*')
       << hex << i << "|" << endl << "|"
       << setw(6) << setprecision(4)
       << p << "|" << endl;
}
```

```
|*****4d2|
|*12.35|
```

Organisation d'un programme C++

- Programme C++ généralement constitué de plusieurs modules, compilés séparément
- Fichier entête – d'extension .h (ou .hh ou .hpp)
  - Contenant les déclarations de types, fonctions, variables et constantes, etc.
  - Inclus via la commande `#include`
- Fichier source – d'extension .cpp ou .C

MonFichierEnTete.h

```
#include <iostream>
extern char* MaChaine;
extern void MaFonction();
```

MonFichier.cpp

```
#include "MonFichierEnTete.h"
void MaFonction()
{
 cout << MaChaine << " \n " ;
}
```

MonProgPirncipal.cpp

```
#include "MonFichierEnTete.h"
char *MaChaine="Chaîne à afficher";
int main()
{
  MaFonction();
}
```

Compilation

- Langage C++ : langage compilé => fichier exécutable produit à partir de **fichiers sources** par **un compilateur**
- Compilation en 3 phases :
  - *Preprocessing* : Suppression des commentaires et traitement des directives de compilation commençant par `#` => code source brut
  - **Compilation** en fichier objet : compilation du source brut => fichier objet (portant souvent l'extension .obj ou .o sans `main`)
  - **Edition de liens** : Combinaison du fichier objet de l'application avec ceux des bibliothèques qu'elle utilise =>fichier exécutable binaire ou une librairie dynamique (.dll sous Windows)
- Compilation => vérification de la syntaxe mais **pas de vérification de la gestion de la mémoire** (erreur d'exécution *segmentation fault*)

Erreurs générées

- Erreurs de compilation

  Erreur de syntaxe, déclaration manquante, parenthèse manquante,...

- Erreur de liens

  Appel a des fonctions dont les bibliothèques sont manquantes

- Erreur d'exécution

  Segmentation fault, *overflow*, division par zéro

- Erreur logique

Compilateur C++

- Compilateurs gratuits (*open-source*) :
  - **Plugin C++ pour Eclipse**

    `http://www.eclipse.org/downloads/moreinfo/c.php`

    Télécharger une version complète pour développer sous Windows :

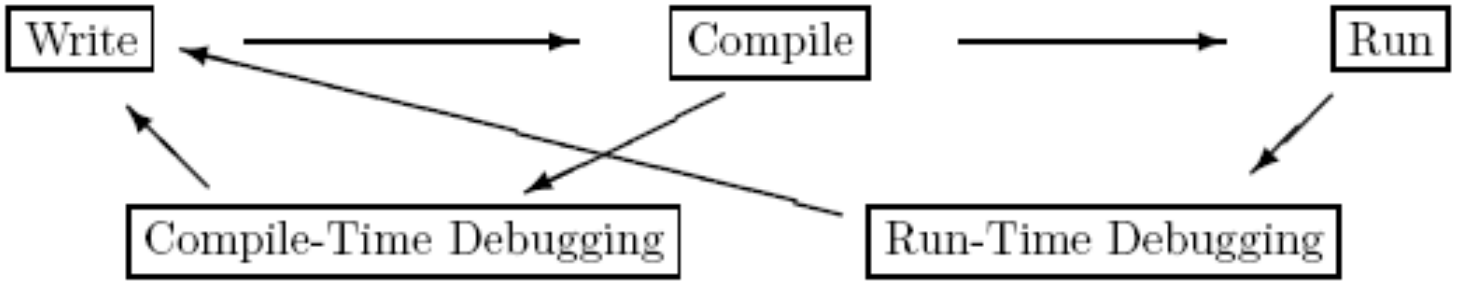
    `http://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/europa/winter/eclipse-cpp-europa-winter-win32.zip`

    Ou depuis Eclipse via "Install New Software..."

    `http://www.eclipse.org/cdt/`
  - **MinGW ou Mingw32 (Minimalist GNU for Windows)**
    `http://www.mingw.org/`
- Compilateurs propriétaires :
  - **Visual C++** (Microsoft – disponible au CRIO INTER-UFR-version gratuite disponible Visual Express mais nécessité de s'inscrire sur le site de Windows : `http://msdn.microsoft.com/fr-fr/express/`)
  - **Borland C++** (version libre téléchargeable à l'adresse : `http://www.codegear.com/downloads/free/cppbuilder`)

Quelques règles de programmation

1. Définir les classes, inclure les librairies etc. dans un fichier d'extension .h
2. Définir le corps des méthodes et des fonctions, le programme **`main`** etc. dans un fichier d'extension .cpp (incluant le fichier .h)
3. Compiler régulièrement
4. Pour déboguer :
   - Penser à utiliser les commentaires et les **`cout`**
   - Utiliser le débogueur



Espaces de noms

- Utilisation d'**espaces de noms** (*namespace*) lors de l'utilisation de nombreuses bibliothèques pour éviter les conflits de noms
- **Espace de noms** : association d'un nom à un ensemble de variable, types ou fonctions

  Ex. Si la fonction *MaFonction()* est définie dans l'espace de noms *MonEspace*, l'appel de la fonction se fait par *MonEspace::MaFonction()*

- Pour être parfaitement correct :

```
std::cin
std::cout
std::endl
```

`::` *opérateur de résolution de portée*

- **Pour éviter l'appel explicite à un espace de noms** : **`using`**

```
using std::cout ; // pour une fonction spécifique
using namespace std; // pour toutes les fonctions
```

Types de base (1/5)

- Héritage des mécanismes de bases du C (pointeurs inclus)

Attention : typage fort en C++!!

- Déclaration et initialisation de variables :

```
bool this_is_true = true; // variable boléenne
int i = 0; // entier
long j = 123456789; // entier long
float f = 3.1; // réel
// réel à double précision
double pi = 3.14159265358979323846264643;
char c='a'; // caractère
```

- « Initialisation à la mode objet » :

```
int i(0) ;
long  j(123456789);
```

…

Types de base (2/5)

Le type d'une donnée détermine :

- La place mémoire (**`sizeof()`**)
- Les opérations légales
- Les bornes

Name	Description	Size*	Range*
`char`	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
`short int (short)`	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int (long)`	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean value. It can take one of two values: true or false.	1byte	true or false
`float`	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
`double`	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`long double`	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
`wchar_t`	Wide character.	2 *or* 4 bytes	1 wide character

Types de base (3/5) : réel

- Représenté par un nombre à virgule flottante :
  - Position de la virgule repérée par une partie de ses bits (exposant)
  - Reste des bits permettant de coder le nombre sans virgule (mantisse)
- Nombre de bits pour le type **`float`** (32 bits) : 23 bits pour la mantisse, 8 bits pour l'exposant, 1 bit pour le signe
- Nombre de bits pour le type **`double`** (64 bits) : 52 bits pour la mantisse, 11 bits pour l'exposant, 1 bit pour le signe
- Nombre de bits pour le type **`long double`** (80 bits) : 64 bits pour la mantisse, 15 bits pour l'exposant, 1 bit pour le signe
- Précision des nombres réels approchée, dépendant du nombre de positions décimales, d'au moins :
  - 6 chiffres après la virgule pour le type **`float`**
  - 15 chiffres après la virgule pour le type **`double`**
  - 17 chiffres après la virgule pour le type **`long double`**

Types de base (4/5) : caractère

- Deux types pour les caractères, codés sur 8 bits/1octets
  - **`char`** (-128 à 127)
  - **`unsigned char`** (0 à 255)

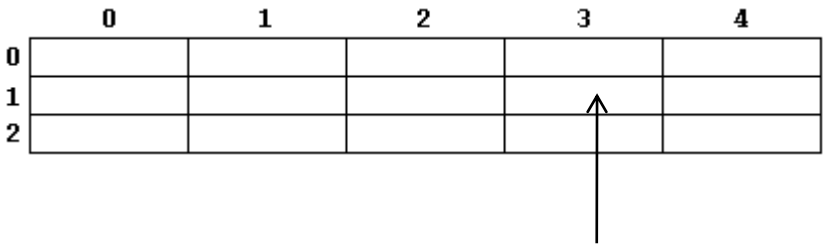
  Exemple: **`'a' 'c' '$' '\n' '\t'`**
- Les caractères imprimables sont toujours positifs
- Caractères spéciaux :

  `\n` (nouvelle ligne)
  `\t` (tabulation horizontale)
  `\f` (nouvelle page)
  `\b` (*backspace*)
  `EOF`, …

Types de base (5/5) : Tableau

```
int tab1[5] ; // Déclaration d'un tableau de 5 entiers
// Déclaration et initialisation d'un tableau de 3 entiers
int tab2 [] = {1,2,3} ; // Les indices commencent à zéro
```

```
int tab_a_2dim[3][5];
```



`tab_a_2dim[1][3]`

```
char chaine[]= "Ceci est une chaîne de caractères";
// Attention, le dernier caractère d'une chaîne est '\0'
char ch[]= "abc" ; // ⇔ char ch[]= {'a','b','c', '\0'};
```

Déclaration, règles d'identification et portée des variables

- Toute variable doit être déclarée avant d'être utilisée
- Constante symbolique : `const int taille = 1000;`

```
// Impossible de modifier taille dans la suite du programme
```

- La portée (visibilité) d'une variable commence à la fin de sa déclaration jusqu'à la fin du bloc de sa déclaration

```
// une fonction nommée f de paramètre i
void f (int i) {  int j; // variable locale
                    j=i;
                 }
```

- Toute double déclaration de variable est interdite dans le même bloc

```
int i,j,m; // variable globale
void f(int i) {
                    int j,k; // variable locale
                    char k;  // erreur de compilation
                    j=i;
                    m=0;
                 }
```

Opérations mathématiques de base

```
int i = 100 + 50;
int j = 100 - 50;
int n = 100 * 2;
int m = 100 / 2; // division entière
int k= 100 % 2; // modulo - reste de la division entière

i = i+1;
i = i-1;

j++; // équivalent à j = j+1;
j--; // équivalent à j = j-1;

n += m; // équivalent à n = n+m;
m -= 5; // équivalent à m = m-5;
j /= i; // équivalent à j = j/i;
j *= i+1; // équivalent à j = j*(i+1);

int a, b=3, c, d=3;
a=++b; // équivalent à b++; puis a=b; => a=b=4
c=d++; // équivalent à c=d; puis d++; => c=3 et d=4
```

A utiliser avec parcimonie – car code vite illisible!!

Opérateurs de comparaison

```
int i,j;
…
if(i==j) // évalué à vrai (true ou !=0) si i égal j
{
  … // instructions exécutées si la condition est vraie
}

if(i!=j) // évalué à vrai (true ou !=0) si i est différent de j

if(i>j) // ou (i<j) ou (i<=j) ou (i>=j)


if(i) // toujours évalué à faux si i==0 et vrai si i!=0
if(false) // toujours évalué à faux
if(true) // toujours évalué à vrai
```

Ne pas confondre = (affectation) et == (test d'égalité)

```
if (i=1) // toujours vrai et i vaut 1 après
```

Opérations sur les chaînes de caractères

- Sur les tableaux de caractères : fonctions de la librairie C **`string.h`**

  Voir documentation :
  `http://www.cplusplus.com/reference/clibrary/cstring/`

```
#include <stdio.h>
#include <string.h>
int main ()
{
  char source[]="chaîne exemple",destination[20];
  strcpy (destination,source); // copie la chaîne source dans la
                                  chaîne destination
}
```

- Sur la classe **`string`** : méthodes appliquées aux objets de la classe `string`

  Voir documentation : **`http://www.cplusplus.com/reference/string/string/`**

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
    string str ("chaîne test");
    cout << " str contient " << str.length() << " caractères s.\n";
    return 0;
}
```

On reviendra sur les notions de fonctions et de méthodes!!

Structures de contrôles (1/4)

```
x = 10;
y = x > 9 ? 100 : 200; // équivalent à
                       // if(x>9) y=100;
                       // else y=200;

int main()
{
   float a;

   cout << "Entrer un réel :";
   cin >> a;

   if(a > 0) cout << a << " est positif\n";
   else
    if(a == 0) cout << a << " est nul\n";
    else cout << a << " est négatif\n";
}

// Mettre des {} pour les blocs d'instructions des if/else pour
// éviter les ambiguïtés et lorsqu'il y a plusieurs instructions
```

Structures de contrôles (2/4)

```
for(initialisation; condition; incrémentation)
   instruction; // entre {} si plusieurs instructions
```

Exemples :

```
for(int i=1; i <= 10; i++)
  cout << i << " " << i*i << "\n"; // Affiche les entiers de
                                   // 1 à 10  et leur carré

int main()
{
  int i,j;
  for(i=1, j=20; i < j; i++, j-=5)
   {
     cout << i << " " << j << "\n";
   }
}
```

Résultat affiché :

```
1 20
2 15
3 10
4 5
```

Structures de contrôles (3/4)

```
int main()
{ char ch;
  double x=5.0, y=10.0;
  cout << " 1. Afficher la valeur de x\n";
  cout << " 2. Afficher la valeur de y\n";
  cout << " 3. Afficher la valeur de xy\n";
  cin >> ch;

  switch(ch)
  {
    case '1': cout << x << "\n";
              break; // pour sortir du switch
                     // et ne pas exécuter les commandes suivantes
    case '2': cout << y << "\n";
              break;
    case '3': cout << x*y << "\n";
              break;
    default: cout << « Option non reconnue \n";

  } \\ Fin du switch

} \\ Fin du main
```

Structures de contrôles (4/4)

```
int main ()
{
    int n=8;
    while (n>0)
    { cout << n << " ";
       --n;
    }
     return 0;
 }
```

Instructions exécutées tant que *n* est supérieur à zéro

Résultat affiché :

```
8 7 6 5 4 3 2 1
```

```
int main ()
{
    int n=0;
    do
    { cout << n << " ";
       --n;
    }
    while (n>0);
    return 0;
 }
```

Instructions exécutées une fois puis une tant que *n* est supérieur à zéro

Résultat affiché :

```
0
```

Type référence (&) et déréférencement automatique

- **Possibilité de définir une variable de type *référence***

```
int i = 5;
int & j = i; // j reçoit i
   // i et j désignent le même emplacement mémoire
```

Impossible de définir une référence sans l'initialiser

- **Déréférencement automatique** :

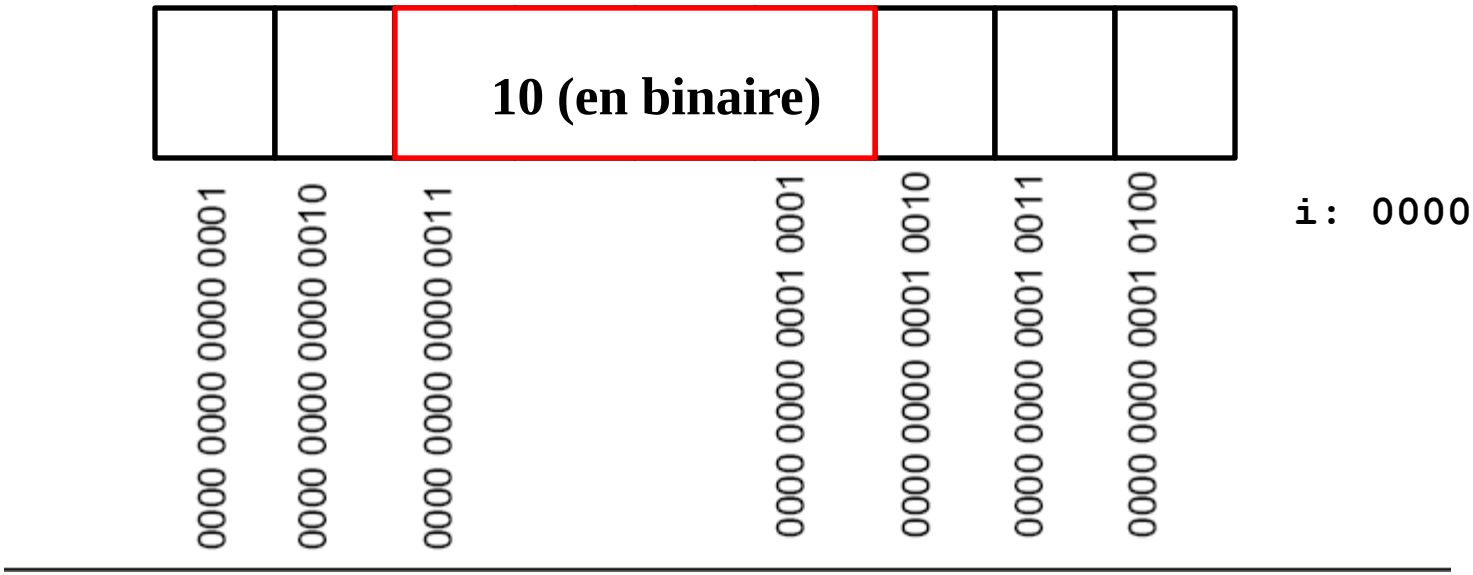
  Application automatique de l'opérateur d'indirection * à chaque utilisation de la référence

```
int i = 5;
int & j = i;    // j reçoit i
int k = j;      // k reçoit 5
j += 2;  // i reçoit i+2 (=7)
j = k;   // i reçoit k (=5)
```

Une fois initialisée, une référence ne peut plus être modifiée – elle correspond au même emplacement mémoire

Pointeurs (1/8)

Mémoire décomposée en "cases" (1 octet) consécutives numérotées (ayant une adresse) que l'on peut manipuler individuellement ou par groupe de cases contigües



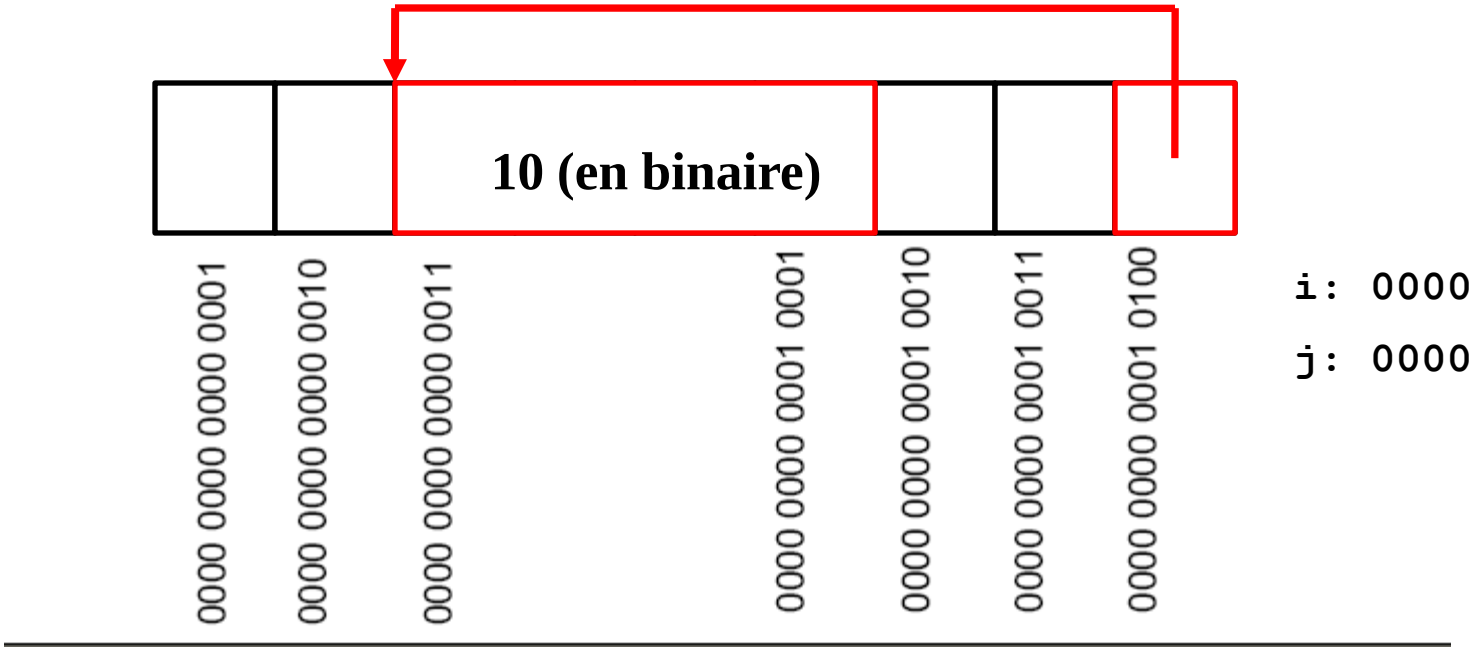

```
int i=10 ;
```

1. Réservation d'une zone mémoire de 4 octets (la 1ère libre)
2. Association du nom i à l'adresse du début de la zone
3. Copie de la valeur en binaire dans la zone mémoire

`&i` correspond à l'adresse du début de la zone mémoire où est stockée la valeur de `i`

Pointeurs (2/8)

Pointeur = variable contenant une adresse en mémoire




```
i: 0000 0000 0000 0011
j: 0000 0000 0001 0100
```

```
int i=10;
int *j=&i;
```

Pointeurs (3/8)

- 2 opérateurs : **`new`** et **`delete`**

```
float *PointeurSurReel = new float;
// Équivalent en C :
//   PointeurSurReel = (float *) malloc(sizeof(float));
int *PointeurSurEntier = new int[20];
// Équivalent en C :
// PointeurSurEntier = (int *) malloc(20 * sizeof(int));
delete PointeurSurReel; // Équivalent en C : free(pf);
delete [] PointeurSurEntier; // Équivalent en C :  free(pi);
```

- **`new type`** : définition et allocation d'un pointeur de type *`type*`*
- **`new type [n]`** : définition d'un pointeur de type *`type*`* sur un tableau de *n* éléments de type *`type`*
- En cas d'échec de l'allocation, `new` déclenche une exception du type `bad_alloc`
- Possibilité d'utiliser `new (nothrow)` ou `set_new_handler`

Pointeurs (4/8)

```
// Programme repris de [Delannoy,2004, Page 52]
#include <cstdlib>      //  ancien <stdlib.h> pour exit
#include <iostream>
using namespace std ;
int main()
{   long taille ;
    int * adr ;
    int nbloc ;
    cout << "Taille souhaitee ? " ;
    cin >> taille ;
    for (nbloc=1 ; ; nbloc++)
    { adr = new (nothrow) int [taille] ;
      if (adr==0) { cout << "**** manque de memoire ****\n" ;
                    exit (-1) ;
                  }
      cout << "Allocation bloc numero : " << nbloc << "\n" ;
    }
    return 0;
}
```

/ Pour que `new` retourne un pointeur nul en cas d'échec */*

`new (nothrow)` non reconnu par certaines versions du compilateur GNU g++

Pointeurs (5/8)

```
#include <iostream>    // Programme repris de [Delannoy,2004, Page 53]
#include <cstdlib>     // ancien <stdlib.h> pour exit
#include <new>         // pour set_new_handler (parfois <new.h>)
using namespace std ;
void deborde () ; // prototype - déclaration de la fonction
                  // fonction appelée en cas de manque de mémoire
int main()
{
   set_new_handler (deborde) ;
   long taille ;
   int * adr, nbloc ;
   cout << "Taille de bloc souhaitee (en entiers) ? " ; cin >> taille ;
   for (nbloc=1 ; ; nbloc++)
      { adr = new int [taille] ;
        cout << "Allocation bloc numero : " << nbloc << "\n" ;
      }
   return 0;
}
void deborde ()        // fonction appelée en cas de manque de mémoire
{  cout << "Memoire insuffisante\n" ;
   cout << "Abandon de l'execution\n" ; exit (-1) ;
}
```

`set_new_handler` non reconnu par le compilateur Visual C++

Pointeurs (6/8)

- Manipulation de la valeur pointée :

```
int *p = new int; // p est de type int*
(*p)=3; // (*p) est de type int
int *tab = new int [20]; // tab est de type int*
// tab correspond à l'adresse du début de la zone mémoire
// allouée pour contenir 20 entiers
(*tab)=3; // équivalent à tab[0]=3
```

- Manipulation de pointeur :

```
tab++; // décalage de tab d'une zone mémoire de taille sizeof(int)
       // tab pointe sur la zone mémoire devant contenir le 2ème
       // élément du tableau
(*tab)=4; // équivalent à tab[1]=4
          // car on a décalé tab à l'instruction précédente
*(tab+1)=5; // équivalent à tab[2]=5 sans décaler tab
tab-=2 ; // tab est décalée de 2*sizeof(int) octets
```

- Libération de la mémoire allouée :

```
delete p;
delete [] tab;
```

Ne pas oublier de libérer la mémoire allouée!!

Pointeurs (7/8)

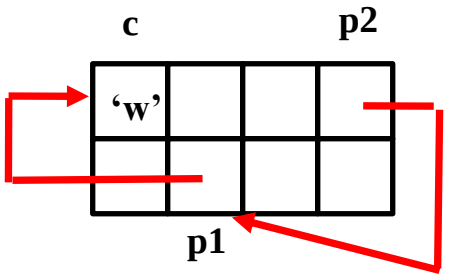
Manipulation des pointeurs et des valeurs pointées (suite)

```
int *p1 = new int;
int *p2 = p1 ; // p2 pointe sur la même zone mémoire que p1
*p2=11; //=> *p1=11; car p1 et p2 pointe sur la même zone mémoire
int *tab = new int [20];
*tab++=3; // équivalent à *(tab++)=tab[1]=3
          // car ++ a une priorité plus forte que *
          // ⇔ tab++; *tab=3;
(*tab)++; // => tab[1]=4
int i=12;
p2=&i; // p2 pointe sur la zone mémoire où est stockée i
*p2=13; // => i=13
p2=tab; // p2 pointe comme tab sur le 2ème élément du tableau
p2++; // p2 pointe sur le 3ème élément (d'indice 2)
*p2=5; // => tab[2]=5 mais tab pointe toujours sur le 2ème élément
p1=p2; // => p1 pointe sur la même zone que p2
       // ATTENTION : l'adresse de la zone allouée par new pour p1
       //             est perdue!!
```

Pointeurs (8/8)

- Pointeurs de pointeur :

```
char c='w';
char *p1=&c; // p1 a pour valeur l'adresse de c
             // (*p1) est de type char
char **p2=&p1; // p2 a pour valeur l'adresse de p1
               // *p2 est de type char*
               // **p2 est de type char
```




```
cout << c ;
cout << *p1;
cout << **p2;
```
// 3 instructions équivalentes qui affiche 'w'

- Précautions à prendre lors de la manipulation des pointeurs :
  - Allouer de la mémoire (**new**) ou affecter l'adresse d'une zone mémoire utilisée (**&**) avant de manipuler la valeur pointée
  - Libérer (**delete**) la mémoire allouée par **new**
  - Ne pas perdre l'adresse d'une zone allouée par **new**

Fonctions

- Appel de fonction toujours précédé de la déclaration de la fonction sous la forme de prototype (signature) 
- Une et une seule définition d'une fonction donnée mais autant de déclaration que nécessaire
- **Passage des paramètres par valeur** (comme en C) ou **par référence**
- **Possibilité de surcharger ou sur-définir une fonction**

```
int racine_carree (int x) {return x * x;}
double racine_carree  (double y) {return y * y;}
```

- **Possibilité d'attribuer des valeurs par défaut aux arguments**

```
void MaFonction(int i=3, int j=5); // Déclaration
int x =10, y=20;
MaFonction(x,y);
MaFonction(x);
MaFonction();
```

Les arguments concernés doivent obligatoirement être les derniers de la liste

A fixer dans la déclaration de la fonction pas dans sa définition

Passage des paramètres par valeur (1/2)

```
#include <iostream>
void echange(int,int); // Déclaration (prototype) de la fonction
                       // A mettre avant tout appel de la fonction

int main()
{   int n=10, p=20;
    cout << "avant appel: " << n << " " << p << endl;
    echange(n,p); // Appel de la fonction
    cout << "apres appel: " << n << " " << p << endl;
}

void echange(int a, int b) // Définition de la fonction
{   int c;
    cout << "debut echange : " << a << " " << b << endl;
    c=a; a=b; b=c;
    cout << "fin echange : " << a << " " << b << endl;
}
```

Lors de l'appel **`echange(n,p)`** : **`a`** prend la valeur de **`n`** et **`b`** prend la valeur de **`p`**
Mais après l'appel (à la sortie de la fonction), les valeurs de **`n`** et **`p`** restent inchangées

```
avant appel: 10 20        fin echange: 20 10
debut echange: 10 20      apres appel: 10 20
```

Passage des paramètres par valeur (2/2)

```
#include <iostream>

void echange(int*,int*); // Modification de la signature
                         // Utilisation de pointeurs

int main()
{  int n=10, p=20;
   cout << "avant appel: " << n << " " << p << endl;
   echange(&n,&p);
   cout << "apres appel: " << n << " " << p << endl;
}

void echange(int* a, int* b)
{  int c;
   cout << "debut echange : " << *a << " " << *b << endl;
   c=*a; *a=*b; *b=c;
   cout << "fin echange : " << *a << " " << *b << endl;
}
```

Lors de l'appel **`echange(&n,&p)`** : **`a`** pointe sur **`n`** et **`b`** pointe sur **`p`**
Donc après l'appel (à la sortie de la fonction), les valeurs de **`n`** et **`p`** ont été modifiées

```
avant appel: 10 20      fin echange: 20 10
debut echange: 10 20    apres appel: 20 10
```

Passage des paramètres par référence

Si on surcharge la fonction en incluant la fonction prenant en paramètre des entiers => ambigüité pour le compilateur lors de l'appel de la fonction!!

```
#include <iostream>

void echange(int&,int&);

int main()
{   int n=10, p=20;
    cout << "avant appel: " << n << " " << p << endl;
    echange(n,p);   // attention, ici pas de &n et &p
    cout << "apres appel: " << n << " " << p << endl;
}

void echange(int& a, int& b)
{   int c;
    cout << "debut echange : " << a << " " << b << endl;
    c=a; a=b; b=c;
    cout << "fin echange : " << a << " " << b << endl;
}
```

Lors de l'appel **`echange(n,p)`** : **`a`** et **`n`** correspondent au même emplacement mémoire, de même pour **`b`** et **`p`**

Donc après l'appel (à la sortie de la fonction), les valeurs de **`n`** et **`p`** sont modifiées

```
avant appel: 10 20          fin echange: 20 10
debut echange: 10 20        apres appel: 20 10
```

const (1/2)

- **Constante symbolique** : `const int taille = 1000;`

```
// Impossible de modifier taille dans la suite du programme
```

`const` **définit une expression constante = calculée à la compilation**

- **Utilisation de `const` avec des pointeurs**
  - **Donnée pointée constante :**

```
const char* ptr1 = "QWERTY" ;
ptr1++; // OK
*ptr1= 'A'; // KO - assignment to const type
```

  - **Pointeur constant :**

```
char* const ptr1 = "QWERTY" ;
ptr1++; // KO - increment of const type
*ptr1= 'A'; // OK
```

  - **Pointeur et donnée pointée constants :**

```
const char* const ptr1 = "QWERTY" ;
ptr1++; // KO - increment of const type
*ptr1= 'A'; // KO - assignment to const type
```

const (2/2)

```
void f (int* p2)
{
   *p2=7; // si p1==p2, alors on change également *p1
}

int main ()
{
   int x=0;
   const int *p1= &x;
   int y=*p1;
   f(&x);
   if (*p1!=y) cout << "La valeur de *p1 a été modifiée";
   return 0;
}
```

`const int* p1` indique que la donnée pointée par p1 ne pourra par être modifiée par l'intermédiaire de p1, pas qu'elle ne pourra jamais être modifiée

STL

Librairie STL (*Standard Template Library*) : incluse dans la norme C++ ISO/IEC 14882 et développée à Hewlett Packard (Alexander Stepanov et Meng Lee) - définition de conteneurs (liste, vecteur, file etc.)

- ```
  #include <string> // Pour utiliser les chaînes de caractères
  #include <iostream>

  using namespace std ;
  int main()
  { string MaChaine="ceci est une chaine";
     cout << "La Chaine de caractères \""<< MaChaine
          << "\" a pour taille " << MaChaine.size() << "."
          << endl;
     string AutreChaine("!!");
     cout << "Concaténation des deux chaines : \""
          << MaChaine + AutreChaine<<"\".«  << endl ;
     return 0;
   }
  ```

- ```
  #include <vector> // patron de classes vecteur
  #include <list> // patron de classes liste

  vector<int> v1(4, 99) ; // vecteur de 4 entiers égaux à 99
  vector<int> v2(7) ;   // vecteur de 7 entiers
  list<char> lc2 ; // Liste de caractères
  ```

Classes et objets (1/6) : définitions

- **Classe :**
  - Regroupement de données (attributs ou champs) et de méthodes (fonctions membres)
  - Extension des structures (`struct`) avec différents niveaux de visibilité (`protected`, `private` et `public`)
- En programmation orientée-objet pure : encapsulation des données et accès unique des données à travers les méthodes
- **Objet** : instance de classe
  - Attributs et méthodes communs à tous les objets d'une classe
  - Valeurs des attributs propres à chaque objet
- **Encapsulation**
  - Caractérisation d'un objet par les spécifications de ses méthodes : interface
  - Indépendance vis à vis de l'implémentation

Classes et objets (2/6) : 1er exemple de classe

```
// Exemple de classe repris de [Deitel et Deitel, 2001]

class Horaire{

  private: // déclaration des membres privés
           // private: est optionnel (privé par défaut)
     int heure;       // de 0 à 24
     int minute;      // de 0 à 59
     int seconde;     // de 0 à 59

  public: // déclaration des membres publics
     Horaire(); // Constructeur
     void SetHoraire(int, int, int);
     void AfficherMode12h();
     void AfficherMode24h();
};
```

Interface de la classe

Classes et objets (3/6) : 1er exemple de classe

Constructeur : Méthode appelée automatiquement à la création d'un objet

```
Horaire::Horaire() {heure = minute = seconde = 0;}
```

Définition d'un constructeur ⇒ Création d'un objet en passant le nombre de paramètres requis par le constructeur

```
int main()
{ Horaire h;  // Appel du constructeur qui n'a pas de paramètre
  …
}
```

Si on avait indiqué dans la définition de la classe :

```
Horaire (int = 0, int = 0, int = 0);
```

- **Définition du constructeur :**

```
Horaire:: Horaire (int h, int m, int s)
 { SetHoraire(h,m,s);}
```

- **Déclaration des objets :**

```
Horaire h1, h2(8), h3 (8,30), h4 (8,30,45);
```

Classes et objets (4/6) : 1er exemple de classe

```
// Exemple repris de [Deitel et Deitel, 2001]

void Horaire::SetHoraire(int h, int m, int s)
{
   heure = (h >= 0 && h < 24) ? h : 0 ;
   minute = (m >= 0 && m < 59) ? m : 0 ;
   seconde = (s >= 0 && s < 59) ? s : 0 ;
}

void Horaire::AfficherMode12h()
{
   cout << (heure < 10 ? "0"  : "" ) << heure << ":"
    << (minute < 10 ? "0"  : "" ) << minute;
}

void Horaire::AfficherMode24h()
{
   cout << ((heure == 0 || heure == 12) ? 12 : heure %12)
    << ":"  << (minute < 10 ? "0"  : "" << minute
    << ":"  << (seconde < 10 ? "0"  : "" << seconde
    <<  (heure < 12 ? " AM"  : " PM"  );
}
```

Classes et objets (5/6) : 1er exemple de classe

```
// Exemple repris de [Deitel et Deitel, 2001]

#include "Horaire.h"

int main()

{
   Horaire MonHoraire;

   // Erreur : l'attribut Horaire::heure est privé
   MonHoraire.heure = 7;

   // Erreur : l'attribut Horaire::minute est privé
   cout << "Minute = " << MonHoraire.minute ;

   return 0;
}
```

Résultat de la compilation avec g++ sous Linux

```
g++ -o Horaire Horaire.cpp Horaire_main.cpp
Horaire_main.cpp: In function `int main()':
Horaire.h:9: `int Horaire::heure' is private
Horaire_main.cpp:9: within this context
Horaire.h:10: `int Horaire::minute' is private
Horaire_main.cpp:11: within this context
```

Classes et objets (6/6) : 1er exemple de classe

```
// Exemple de classe repris de [Deitel et Deitel, 2001]

class Horaire{
   private : // déclaration des membres privés
        int heure;      // de 0 à 24
        int minute;     // de 0 à 59
        int seconde;    // de 0 à 59
   public : // déclaration des membres publics
        Horaire();       // Constructeur
        void SetHoraire(int, int, int);
        void SetHeure(int);
        void SetMinute(int);
        void SetSeconde(int);
        int GetHeure();
        int GetMinute();
        int GetSeconde();
        void AfficherMode12h();
        void AfficherMode24h();
};

void Horaire::SetHeure(int h)
  {heure = ((h >=0) && (h<24)) ? h : 0;}

int Horaire:: GetHeure() {return heure;}
```

Pour affecter des valeurs aux attributs privés

Pour accéder aux valeurs des attributs privés

Quelques règles de programmation

1. **Définir les classes, inclure les librairies etc. dans un fichier d'extension .h**
2. **Définir le corps des méthodes, le programme main etc. dans un fichier d'extension .cpp (incluant le fichier .h)**
3. **Compiler régulièrement**
4. **Pour déboguer :**
   - **Penser à utiliser les commentaires et les `cout`**
   - **Utiliser le débogueur**

Utilisation des constantes (1/4)

```
const int MAX = 10;
…
int tableau[MAX];
cout << MAX ;
…
class MaClasse
{
  int tableau[MAX];
}
```

En C++ : on peut utiliser une constante n'importe où après sa définition

Par convention : les constantes sont notées en majuscules

Utilisation des constantes (2/4)

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

// déclaration d'une constante
const int max=10;

class MaClasse
{
  int tableau[max]; // Utilisation de la constante dans une classe
  public:
    MaClasse() { cout << "constructeur" << endl ;
                 for(int i=0; i<max; i++) tableau[i]=i;
               }
    void afficher()
     { for(int i=0; i<max; i++)
       cout << "tableau[" << i << "]=" << tableau[i] << endl;
     }
};

int main()
{ cout << "constante : " << max << endl;
  MaClasse c;
  c.afficher();
}
```

Attention au nom des constantes

Il existe une fonction max :

```
/usr/include/c++/3.2.2/bits/stl_algobase.h:207:
also declared as `std::max'
```

(de gcc 3.2.2)

Compile sans problème avec g++ 1.1.2 (sous linux) ou sous Visual C++ 6.0 (sous windows)

Erreur de compilation sous Eclipse 3;1.0 et gcc 3.2.2!!

Utilisation des constantes (3/4)

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

// déclaration d'une constante dans un espace de nom
namespace MonEspace{const int max=10;}

class MaClasse
{
  int tableau[MonEspace::max];
  public:
    MaClasse() { cout << "constructeur" << endl ;
                 for(int i=0; i< MonEspace::max; i++)
                     tableau[i]=i;
               }
    void afficher()
     { for(int i=0; i< MonEspace::max; i++)
        cout << "tableau[" << i << "]="
          << tableau[i] << endl;
     }
};

int main()
{ cout << "constante : " << MonEspace:: max << endl;
  MaClasse c;
  c.afficher();
}
```

Possibilité de déclarer la constante `max` dans un espace de noms => pas de bug de compil. sous Eclipse 3.1.0

Utilisation des constantes (4/4)

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

// déclaration d'une constante
const int MAX=10;

class MaClasse
{
  int tableau[MAX];
  public:
    MaClasse() { cout << "constructeur" << endl ;
                 for(int i=0; i< MAX; i++)
                     tableau[i]=i;
               }
    void afficher()
     { for(int i=0; i< MAX; i++)
        cout << "tableau[" << i << "]="
          << tableau[i] << endl;
     }
};

int main()
{ cout << "constante : " << MAX << endl;
  MaClasse c;
  c.afficher();
  cout << max(10,15);
}
```

Par convention : les constantes sont notées en majuscules

Notions de constructeurs et destructeur (1/7)

- **Constructeurs**
  - De même nom que le nom de la classe
  - Définition de l'initialisation d'une instance de la classe
  - Appelé implicitement à toute création d'instance de la classe
  - Méthode non typée, pouvant être surchargée
- **Destructeur**
  - De même nom que la classe mais précédé d'un tilde (~)
  - Définition de la désinitialisation d'une instance de la classe
  - Appelé implicitement à toute disparition d'instance de la classe
  - Méthode non typée et sans paramètre
  - Ne pouvant pas être surchargé

Notions de constructeurs et destructeur (2/7)

```
// Programme repris de [Delannoy, 2004] - pour montrer
   les appels du constructeur et du destructeur

class Exemple
{
    public :
      int attribut;
      Exemple(int); // Déclaration du constructeur
      ~Exemple();   // Déclaration du destructeur
} ;

Exemple::Exemple (int i)  // Définition du constructeur
{ attribut = i;
  cout << "** Appel du constructeur - valeur de
   l'attribut = " << attribut << "\n";
}

Exemple::~Exemple()  // Définition du destructeur
{ cout << "** Appel du destructeur - valeur de l'attribut
   = " << attribut << "\n";
}
```


Notions de constructeurs et destructeur (3/7)

```
void MaFonction(int); // Déclaration d'une fonction

int main()
{
   Exemple e(1); // Déclaration d'un objet Exemple
   for(int i=1;i<=2;i++) MaFonction(i);
   return 0;
}
void MaFonction (int i) // Définition d'une fonction
{
   Exemple e2(2*i);
}
```

Résultat de l'exécution du programme :

```
** Appel du constructeur - valeur de l'attribut = 1
** Appel du constructeur - valeur de l'attribut = 2
** Appel du destructeur - valeur de l'attribut = 2
** Appel du constructeur - valeur de l'attribut = 4
** Appel du destructeur - valeur de l'attribut = 4
** Appel du destructeur - valeur de l'attribut = 1
```

Notions de constructeurs et destructeur (4/7)

```
// Exemple de constructeur effectuant une allocation
// dynamique - repris de [Delannoy, 2004]
class TableauDEntiers
{
        int nbElements;
        int * pointeurSurTableau;
   public:
        TableauDEntiers(int, int);   // Constructeur
        ~ TableauDEntiers();         // Destructeur
        …
}

// Constructeur allouant dynamiquement de la mémoire pour nb entiers
TableauDEntiers::TableauDEntiers (int nb, int max)
{ pointeurSurTableau = new int [nbElements=nb] ;
  for (int i=0; i<nb; i++) // nb entiers tirés au hasard
    pointeurSurTableau[i]= double(rand())/ RAND_MAX*max;
} // rand() fournit un entier entre 0 et RAND_MAX

TableauDEntiers::~TableauDEntiers ()
{ delete [] pointeurSurTableau ; // désallocation de la mémoire
}
```

Notions de constructeurs et destructeur (5/7)

Constructeur par recopie (*copy constructor*) :

- Constructeur créé par défaut mais pouvant être redéfini
- Appelé lors de l'**initialisation d'un objet par recopie** d'un autre objet, lors du **passage par valeur d'un objet** en argument de fonction ou en **retour d'un objet** comme retour de fonction

```
MaClasse c1;
MaClasse c2=c1; // Appel du constructeur par recopie
```

- Possibilité de définir explicitement un constructeur par copie si nécessaire :
  - Un seul argument de type de la classe
  - Transmission de l'argument par référence

```
MaClasse(MaClasse &);
MaClasse(const MaClasse &) ;
```

Notions de constructeurs et destructeur (6/7)

```
// Reprise de la classe Exemple
class Exemple
{
    public :
      int attribut;
      Exemple(int); // Déclaration du constructeur
      ~Exemple();   // Déclaration du destructeur
} ;

int main()
{  Exemple e(1); // Déclaration d'un objet Exemple
   Exemple e2=e; // Initialisation d'un objet par recopie
   return 0;
}
```

Résultat de l'exécution du programme avant la définition explicite du constructeur par recopie :

```
** Appel du constructeur - valeur de l'attribut = 1
** Appel du destructeur - valeur de l'attribut = 1
** Appel du destructeur - valeur de l'attribut = 1
```

Notions de constructeurs et destructeur (7/7)

```
// Reprise de la classe Exemple
class Exemple
{ public :
        int attribut;
        Exemple(int);
      // Déclaration du constructeur par recopie
      Exemple(const Exemple &);
      ~Exemple();
} ;

// Définition du constructeur par recopie
Exemple::Exemple (const Exemple & e)
{ cout << "** Appel du constructeur par recopie ";
  attribut = e.attribut;  <-------  // Recopie champ à champ
  cout << " - valeur de l'attribut après recopie = " << attribut <<
endl;
}
```

Résultat de l'exécution du programme avant la définition explicite du constructeur par recopie :

```
** Appel du constructeur - valeur de l'attribut = 1
** Appel du constructeur par recopie  - valeur de l'attribut après recopie= 1
** Appel du destructeur - valeur de l'attribut = 1
** Appel du destructeur - valeur de l'attribut = 1
```

Méthodes de base d'une classe

- **Constructeur**
- **Destructeur**
- **Constructeur par copie**
- **Opérateur d'affectation (=)**

Attention aux implémentations par défaut fournies par le compilateur

Si une fonctionnalité ne doit pas être utilisée alors en interdire son accès en la déclarant `private`

Propriétés des méthodes (1/4)

- **Surcharge des méthodes**

```
MaClasse();                 Afficher();
MaClasse(int);              Afficher(char* message);
```

- Possibilité de définir des **arguments par défaut**

```
MaClasse(int = 0);          Afficher(char* = "" );
```

- Possibilité de définir des **méthodes en ligne**

```
inline MaClasse::MaClasse() {corps court};

class MaClasse
{ …
 MaClasse() {corps court};
}
```

Définition de la méthode dans la déclaration même de la classe

Incorporation des instructions correspondantes (en langage machine) dans le programme ⇒ plus de gestion d'appel

Propriétés des méthodes (2/4)

Passage des paramètres objets

- **Transmission par valeur**

```
bool Horaire::Egal(Horaire h)
  { return ((heure == h.heure) && (minute == h.minute)
   && (seconde == h.seconde)) ;
  }
    // NB : Pas de violation du principe d'encapsulation
int main()
{ Horaire h1, h2;
  …
   if (h1.Egal(h2)==true)
        // h2 est recopié dans un emplacement
        // local à Egal nommé h
        // Appel du constructeur par recopie
}
```

- **Transmission par référence**

```
bool Egal(const Horaire & h)
```

Propriétés des méthodes (3/4)

Méthode retournant un objet

- **Transmission par valeur**

```
Horaire Horaire::HeureSuivante()
 { Horaire h;
    if (heure<24) h.SetHeure(heure+1);
       else h.SetHeure(0);
    h.SetMinute(minute); h.SetSeconde(seconde);
    return h; // Appel du constructeur par recopie
 }
```

- **Transmission par référence**

```
Horaire & Horaire::HeureSuivante()
```

La variable locale à la méthode est détruite à la sortie de la méthode – Appel automatique du destructeur!

Propriétés des méthodes (4/4)

Méthode constante

- Utilisable pour un objet déclaré constant
- Pour les méthodes ne modifiant pas la valeur des objets

```
class Horaire
{ …
  void Afficher() const ;
  void AfficherMode12h();
}
```

NB : Possibilité de surcharger la méthode et d'ajouter à la classe :

```
Afficher();
```

Fonction main

```
const Horaire h;
h.Afficher(); // OK
h.AfficherMode12h() // KO
          // h est const mais pas la méthode!!
Horaire h2;
h2.Afficher(); //OK
```

Auto-référence

Auto-référence : pointeur `this`

- Pointeur sur l'objet (i.e. l'adresse de l'objet) ayant appelé
- Uniquement utilisable au sein des méthodes de la classe

```
Horaire::AfficheAdresse()
{ cout << "Adresse : " << this ;
}
```

Qualificatif statique : `static` (1/2)

- Applicable aux attributs et aux méthodes
- Définition de propriété indépendante de tout objet de la classe ⇔ **propriété de la classe**

```
class ClasseTestStatic
{
  static int NbObjets; // Attribut statique
  public:
    // constructeur inline
    ClasseTestStatic() {NbObjets++;};
    // Affichage du membre statique inline
    void AfficherNbObjets ()
    { cout << "Le nombre d'objets instances de la
                classe  est : " << NbObjets << endl;
    };
  static int GetNbObjets() {return NbObjets;};
};
```

Qualificatif statique : static (2/2)

```
// initialisation de membre statique
int ClasseTestStatic::NbObjets=0;

int main ()
{
→ cout << "Nombre d'objets de la classe :"
          << ClasseTestStatic ::GetNbObjets() << endl;

   ClasseTestStatic a; a.AfficherNbObjets();
   ClasseTestStatic b, c;
   b.AfficherNbObjets(); c.AfficherNbObjets();
   ClasseTestStatic d;
   d.AfficherNbObjets(); a.AfficherNbObjets();
};
```

```
Nombre d'objets de la classe : 0
Le nombre d'objets instances de la classe est : 1
Le nombre d'objets instances de la classe est : 3
Le nombre d'objets instances de la classe est : 3
Le nombre d'objets instances de la classe est : 4
Le nombre d'objets instances de la classe est : 4
```

Surcharge d'opérateurs (1/5)

Notions de **méthode amie** : **`friend`**

- Fonction extérieure à la classe ayant accès aux données privées de la classes
- Contraire à la P.O.O. mais utile dans certain cas
- Plusieurs situations d'« amitié » [Delannoy, 2001] :
  - Une fonction indépendante, amie d'une classe
  - Une méthode d'une classe, amie d'une autre classe
  - Une fonction amie de plusieurs classes
  - Toutes les méthodes d'une classe amies d'une autre classe

```
friend type_retour NomFonction (arguments) ;
// A déclarer dans la classe amie
```

Surcharge d'opérateurs (2/5)

Possibilité en C++ de redéfinir n'importe quel opérateur unaire ou binaire : =, ==, +, -, *, \, [], (), <<, >>, ++, -- , +=, -=, *=, /= , & etc.

```
class Horaire
{ …
  bool operator== (const Horaire &);
}

bool Horaire::operator==(const Horaire& h)
{
  return((heure==h.heure) && (minute ==
  h.minute) && (seconde == h.seconde));
}
```

Fonction main

```
Horaire h1, h2;
…
if (h1==h2) …
```

Surcharge d'opérateurs (3/5)

```
class Horaire
{ …
  friend bool operator== (const Horaire &, const
  Horaire &); // fonction amie
}

bool operator==(const Horaire& h1, const
  Horaire& h2)
{
  return((h1.heure==h2.heure) && (h1.minute ==
  h2.minute) && (h1.seconde == h2.seconde));
}
```

Fonction main

```
Horaire h1, h2;
…
if (h1==h2) …
```

Un opérateur binaire peut être défini comme :

- **une méthode à un argument**
- **une fonction `friend` à 2 arguments**
- **jamais les deux à la fois**

Surcharge d'opérateurs (4/5)

```
class Horaire
{ … // Pas de fonction friend ici pour l'opérateur ==
}

// Fonction extérieure à la classe
bool operator==(const Horaire& h1, const Horaire& h2)
{
  return((h1.GetHeure()==h2.GetHeure()) &&
  (h1.GetMinute() == h2.GetMinute()) &&
  (h1.GetSeconde() == h2.GetSeconde()) );
}
```

Fonction main

```
Horaire h1, h2;
…
if (h1==h2) …
```

Surcharge d'opérateurs (5/5)

```
class Horaire
{ …
  const Horaire& operator= (const Horaire &);
}

const Horaire& Horaire::operator=(const Horaire& h)
{
  heure=h.heure;
  minute = h.minute;
  seconde= h.seconde;
  return *this;
}
```

Fonction main

```
Horaire h1(23,16,56),h2;
…
h2=h1; …
```

Copy constructeur vs. Opérateur d'affectation

```
MaClasse c;
MaClasse c1=c; // Appel au copy constructeur!
MaClasse c2;
c2=c1; // Appel de l'opérateur d'affectation!

// Si la méthode Egal a été définie par :
// bool Egal(MaClasse c);
if(c1.Egal(c2)) …; // Appel du copy constructeur!
                   // c2 est recopié dans c
// Si l'opérateur == a été surchargé par :
// bool operator==(MaClasse c);
if(c1==c2) …; // Appel du copy constructeur!
              // ⇔ c1.operator==(c2)
              // c2 est recopié dans c
```

Objet membre (1/4)

Possibilité de créer une classe avec un membre de type objet d'une classe

```
// exemple repris de [Delannoy, 2004]
class point
 {
   int abs, ord ;
   public :
    point(int,int);
 };

 class cercle
 {
   point centre; // membre instance de la classe point
   int rayon;
   public :
    cercle (int, int, int);
 };
```

Objet membre (2/4)

```
#include "ObjetMembre.h"

point::point(int x=0, int y=0)
{
   abs=x; ord=y;
   cout << "Constr. point " << x << " " << y << endl;
}

cercle::cercle(int abs, int ord, int ray) : centre(abs,ord)
{
   rayon=ray;
   cout << "Constr. cercle " << rayon << endl;
}

int main()
{
   point p;
   cercle c (3,5,7);
}
```

Affichage :

```
Constr. point 0 0
Constr. point 3 5
Constr. cercle 7
```

Objet membre (3/4)

```
// Autre manière d'écrire le constructeur de la classe cercle
cercle::cercle(int abs, int ord, int ray)
{
    rayon=ray;
    // Attention : Création d'un objet temporaire point
    // et Appel de l'opérateur =
    centre = point(abs,ord);
    cout << "Constr. cercle " << rayon << endl;
}
int main()
{
    point p = point(3,4); // ⇔ point p (3,4);
    // ici pas de création d'objet temporaire
    cercle c (3,5,7);
}
```

Affichage :

```
Constr. point 3 4
Constr. point 0 0
Constr. point 3 5
Constr. cercle 7
```

Objet membre (4/4)

- **Possibilité d'utiliser la même syntaxe de transmission des arguments à un objet membre pour n'importe quel membre (ex. des attributs de type entier) :**

```
class point
{  int abs, ord ;
   public :
    // Initialisation des membres abs et ord avec
    // les valeurs de x et y
    point (int x=0, int y=0) : abs(x), ord(y) {};
};
```

- **Syntaxe indispensable en cas de membre donnée constant ou de membre donnée de type référence :**

```
class Exemple
{  const int n;
   public :
    Exemple();
};
// Impossible de faire n=3; dans le corps du constructeur
// n est un membre (attribut) constant!!
Exemple::Exemple() : n(3) {…}
```

UNIVERSITE PARIS DAUPHINE

Master Mathématiques, Informatique, Décision, Organisation (MIDO)

1ère année

Le langage C++ (partie II)

Maude Manouvrier

- Héritage simple
- Héritage simple et constructeurs
- Héritage simple et constructeurs par copie
- Contrôle des accès
- Héritage simple et redéfinition/sur-définition de méthodes et d'attributs
- Héritage simple et amitié
- Compatibilité entre classe de base et classe dérivée
- Héritage simple et opérateur d'affectation

Héritage simple (1/3)

- **Héritage** [Delannoy, 2004] :
  - Un des fondements de la P.O.O
  - A la base des possibilités de réutilisation de composants logiciels
  - Autorisant la définition de nouvelles classes « dérivées » à partir d'une classe existante « de base »
- **Super-classe** ou classe mère
- **Sous-classe** ou classe fille : spécialisation de la super-classe - héritage des propriétés de la super-classe
- Possibilité d'héritage multiple en C++

Héritage simple (2/3)

```
class CompteBanque
{
    long ident;
    float solde;
    public:
     CompteBanque(long id, float so = 0);
     void deposer(float);
     void retirer(float);
     float getSolde();
};

class ComptePrelevementAuto : public CompteBanque
{
    float prelev;
    public:
     void prelever();
     ComptePrelevementAuto(long id, float pr, float so);
 };
```

Héritage simple (3/3)

```
void transfert(CompteBanque cpt1, ComptePrelevementAuto cpt2)
{
    if (cpt2.getSolde() > 100.00)
    {
        cpt2.retirer(100.00);
        cpt1.deposer(100.00);
    }
}

void ComptePrelevementAuto::prelever()
{
    if (getSolde() > 100.00)
    {
      // La sous-classe a accès aux méthodes publiques de
      // sa super-classe - sans avoir à préciser à quel objet
      // elle s'applique
    }
}
```

Une sous-classe n'a pas accès aux membres privés de sa super-classe!!

Héritage simple et constructeurs (1/4)

```
class Base
{
    int a;
    public:
     Base() : a(0) {}          // ⇔ Base() { a=0; }
     Base(int A) : a(A) {}     // ⇔ Base(int A) { a=A; }
};

class Derived : public Base
{
    int b;
    public:
     Derived() : b(0) {} // appel implicite à Base()
     Derived(int i, int j) : Base(i), b(j) {} // appel explicite
};
```

`Derived obj;` ⇒ « construction » d'un objet de la classe `Base` puis d'un objet de la classe `Derived`

Destruction de `obj` ⇒ appel automatique au destructeur de la classe `Derived` puis à celui de la classe `Base` (ordre inverse des constructeurs)

Héritage simple et constructeurs (2/4)

```
// Exemple repris de [Delannoy, 2004] page 254
#include <iostream>
using namespace std ;

// ************ classe point **********************
class point
{
   int x, y ;
   public :
    // constructeur de point ("inline")
    point (int abs=0, int ord=0)
     { cout << "++ constr. point : " << abs << " " << ord << endl;
       x = abs ; y =ord ;
     }
    ~point () // destructeur de point ("inline")
     { cout << "-- destr.  point :    " << x << " " << y << endl ;
     }
} ;
```

Héritage simple et constructeurs (3/4)

```
// ************ classe pointcol ******************
class pointcol : public point
{
   short couleur ;
   public :
    pointcol (int, int, short) ;  // déclaration constructeur pointcol
   ~pointcol ()                   // destructeur de pointcol ("inline")
     { cout << "-- dest. pointcol - couleur : " << couleur << endl ;
     }
} ;

pointcol::pointcol (int abs=0, int ord=0, short cl=1) : point (abs, ord)
{
    cout << "++ constr. pointcol : " << abs << " " << ord << " " << cl
      << endl ;
    couleur = cl ;
}
```

Héritage simple et constructeurs (4/4)

```
// ************ programme d'essai ****************
int main()
{
    pointcol a(10,15,3) ;              // objets non dynamiques
    pointcol b (2,3) ;
    pointcol c (12) ;
    pointcol * adr ;
    adr = new pointcol (12,25) ;       // objet dynamique
    delete adr ;
}
```

Résultat :

```
++ constr. point :    10 15
++ constr. pointcol : 10 15 3
++ constr. point :    2 3
++ constr. pointcol : 2 3 1
++ constr. point :    12 0
++ constr. pointcol : 12 0 1
++ constr. point :    12 25
++ constr. pointcol : 12 25 1
-- dest. pointcol - couleur : 1
-- destr.  point :    12 25
-- dest. pointcol - couleur : 1
-- destr.  point :    12 0
-- dest. pointcol - couleur : 1
-- destr.  point :    2 3
-- dest. pointcol - couleur : 3
-- destr.  point :    10 15
```

Héritage simple et constructeurs par copie (1/7)

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

class point
{
   int x, y ;
   public :
    point (int abs=0, int ord=0)  // constructeur usuel
      { x = abs ; y = ord ;
        cout << "++ point    " << x << " " << y << endl ;
      }
    point (point & p)  // constructeur de recopie
      { x = p.x ; y = p.y ;
        cout << "CR point    " << x << " " << y << endl ;
      }
} ;
```

Rappel : appel du constructeur par copie lors

- **de l'initialisation d'un objet par un objet de même type**
- **de la transmission de la valeur d'un objet en argument ou en retour de fonction**

Héritage simple et constructeurs par copie (2/7)

```
class pointcol : public point
{
   int coul ;
   public :
    // constructeur usuel
    pointcol (int abs=0, int ord=0, int cl=1) : point (abs, ord)
      {
        coul = cl ;
        cout << "++ pointcol " << coul << endl ;
      }
    // constructeur de recopie
    // il y aura conversion implicite de p dans le type point
     pointcol (pointcol & p) : point (p)
       {
        coul = p.coul ;
        cout << "CR pointcol " << coul << endl ;
      }
} ;
```

Si pas de constructeur par copie défini dans la sous-classe ⇒ Appel du constructeur par copie par défaut de la sous-classe et donc du constructeur par copie de la super-classe

Héritage simple et constructeurs par copie (3/7)

```
void fct (pointcol pc)
{
   cout << "*** entree dans fct ***"  << endl ;
}

int main()
{
   void fct (pointcol) ; // Déclaration de f
   pointcol a (2,3,4) ;
   fct (a) ; // appel de fct avec a transmis par valeur
}
```

Résultat :

```
++ point     2 3
++ pointcol  4
CR point     2 3
CR pointcol  4
*** entree dans fct ***
```

pointcol a (2,3,4) ;

fct (a) ;

Héritage simple et constructeurs par copie (4/7)

Soit une classe B, dérivant d'une classe A :

```
B b0;
B b1 (b0); // Appel du constructeur par copie de B
B b2 = b1 ; // Appel du constructeur par copie de B
```

- Si aucun constructeur par copie défini dans B :

  ⇒ Appel du constructeur par copie par défaut faisant une copie membre à membre

  ⇒ Traitement de la partie de b1 héritée de la classe A comme d'un membre du type A ⇒ Appel du constructeur par copie de A

- Si un constructeur par copie défini dans B :

  - **B ([const] B&)**

  ⇒ Appel du constructeur de A sans argument ou dont tous les arguments possède une valeur par défaut

  - **B ([const] B& x) : A (x)**

  ⇒ Appel du constructeur par copie de A

Le constructeur par copie de la classe dérivée doit prendre en charge l'intégralité de la recopie de l'objet et également de sa partie héritée

Héritage simple et constructeurs par copie (5/7)

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

// Exemple repris et adapté de "C++ - Testez-vous"
// de A. Zerdouk, Ellipses, 2001

class Classe1
{  public :
   Classe1(){ cout << "Classe1::Classe1()" << endl;}

   Classe1(const Classe1 & obj)
    { cout << "Classe1::Classe1(const Classe1&)" << endl;}
};

class Classe2 : public Classe1
{ public:
  Classe2() { cout << "Classe2::Classe2()" << endl;}

  Classe2(const Classe2 & obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl;}
};
```

Héritage simple et constructeurs par copie (6/7)

```
int main()
{
    Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1()
Classe2::Classe2(const Classe2&)
```

Appel du constructeur de la classe mère car pas d'appel explicite au copy const. de la classe mère dans le copy const. de la classe fille

Héritage simple et constructeurs par copie (7/7)

Si le constructeur par recopie de la `Classe2` défini comme suit :

```
// Appel explicite au copy const. de la classe mère
Classe2(const Classe2 & obj) : Classe1(obj)
    { cout << "Classe2::Classe2(const Classe2&)" << endl;}

int main()
{
    Classe2 obj1;
    Classe2 obj2(obj1); // Classe2 obj2=obj1;
}
```

Résultat :

```
Classe1::Classe1()
Classe2::Classe2()
Classe1::Classe1(const Classe1&)
Classe2::Classe2(const Classe2&)
```

Contrôle des accès (1/9)

Trois qualificatifs pour les membres d'une classe : `public, private` et `protected`

- **`Public`** : membre accessible non seulement aux fonctions membres (méthodes) ou aux fonctions amies mais également aux clients de la classe
- **`Private`** : membre accessible uniquement aux fonctions membres (publiques ou privées) et aux fonctions amies de la classe
- **`Protected`** : comme **`private`** mais membre accessible par une classe dérivée

Contrôle des accès (2/9)

```
class Point
{
    protected: // attributs protégés
       int _x;
       int _y;
    public:
       Point (...);
       affiche();
       ...
};

class Pointcol : public Point
{
    short _couleur;
    public:
       void affiche()
       { // Possibilité d'accéder aux attributs protégés
         // _x et _y de la super-classe dans la sous-classe
        cout << "je suis en " << _x << " " <<  _y << endl;
        cout << " et ma couleur est " << _couleur << endl;
       }
};
```

Contrôle des accès (3/9)

Membre protégé d'une classe :

- Équivalent à un membre privé pour les utilisateurs de la classe
- Comparable à un membre public pour le concepteur d'une classe dérivée
- Mais comparable à un membre privé pour les utilisateurs de la classe dérivée

Possibilité de violer l'encapsulation des données

Contrôle des accès (4/9)

Plusieurs modes de dérivation de classe :

- Possibilité d'utiliser **`public`**, **`protected`** ou **`private`** pour spécifier le mode de dérivation d'une classe
- Détermination, par le mode de dérivation, des membres de la super-classe accessibles dans la sous-classe
- Dérivation privée par défaut

Contrôle des accès (5/9)

Dérivation publique :

- Conservation du statut des membres publics et protégés de la classe de base dans la classe dérivée
- Forme la plus courante d'héritage modélisant : « une classe dérivée est une spécialisation de la classe de base »

```
class Base
{
  public:
    void méthodePublique1();
  protected:
    void méthodeProtégée();
  private:
    void MéthodePrivée();
};
```

```
class Derivee : public Base
{
  public:
    int MéthodePublic2()
        {
          méthodePublique1(); // OK
          méthodeProtégée();  // OK
          MéthodePrivée();    // KO
        }
};
```

Contrôle des accès (6/9)

Dérivation publique :

statut dans la classe de base	accès aux fonctions membres et amies de la classe dérivée	accès à un utilisateur de la classe dérivée	nouveau statut dans la classe dérivée
`public`	oui	oui	`public`
`protected`	oui	non	`protected`
`private`	non	non	`private`

Les fonctions ou classes amies de la classe de base ont accès à tous les membres de la classe de base qu'ils soient définis comme `public`, `private` ou `protected`

Contrôle des accès (7/9)

Dérivation privée :

- Transformation du statut des membres publics et protégés de la classe de base en statut privé dans la classe dérivée
- Pour ne pas accéder aux anciens membres de la classe de base lorsqu'ils ont été redéfinis dans la classe dérivée
- Pour adapter l'interface d'une classe, la classe dérivée n'apportant rien de plus que la classe de base (pas de nouvelles propriétés) mais offrant une utilisation différente des membres

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

```
class Derivee : private Base
{
   void méthodePrivée()
    {
      ...
      méthodePublique(); // OK
    }
};
```

```
Derivee obj;
obj.méthodePublique(); // KO
```

```
Base*  b= &obj; // KO
```

Contrôle des accès (8/9)

Dérivation privée (suite)

Possibilité de laisser un membre de la classe de base public dans la classe dérivée

- Redéclaration explicite dans la classe dérivée
- Utilisation de **`using`**

```
class Derivee : private Base // dérivation privée
{
  …
  public:
    Base::méthodePublique1(); // La méthode publique
                              // de la classe de base
                              // devient publique dans
                              // la classe dérivée
    using Base::méthodePublique2(); //idem
};
```

Contrôle des accès (9/9)

Dérivation protégée : Transformation du statut des membres publics et protégés de la classe de base en statut protégé dans la classe dérivée

classe de base			dérivée publique		dérivée protégée		dérivée privée	
statut initial	accès fonct. membres /amies	accès utilis.	nouveau statut	accès utilis.	nouveau statut	accès utilis.	nouveau statut	accès utilis.
public	oui	oui	public	oui	protégé	non	privé	non
protégé	oui	non	protégé	non	protégé	non	privé	non
privé	oui	non	privé	non	privé	non	privé	non

Ne pas confondre le mode de dérivation et le statut des membres d'une classe

Héritage simple

constructeurs/destructeurs/constructeurs par copie

- Pas d'héritage des constructeurs et destructeurs $\Rightarrow$ il faut les redéfinir
- Appel implicite des constructeurs par défaut des classes de base (super-classe) avant le constructeur de la classe dérivée (sous-classe)
- Possibilité de passage de paramètres aux constructeurs de la classe de base dans le constructeur de la classe dérivée par appel explicite
- Appel automatique des destructeurs dans l'ordre inverse des constructeurs
- Pas d'héritage des constructeurs de copie et des opérateurs d'affectation

Héritage simple et redéfinition/sur-définition (1/2)

```
class Base
 {
    protected :
    int a;
    char b;
    public :
     …
     void affiche();
 };
```

```
void Base::affiche()
{ cout << a << b << endl; }

void Derivee::affiche()
{ // appel de affiche
  // de la super-classe
  Base::affiche();
  cout << "a est un réel";
}
```

```
class Derivee : public Base
 {
    float a; // redéfinition de l'attribut a
    public :
     …
     void affiche(); // redéfinition de la méthode affiche
     float GetADelaClasseDerivee(){return a;} ;
     int GetADeLaClasseDeBase() {return Base::a;}
 };
```

Héritage simple et redéfinition/sur-définition (2/2)

```
class A
 {   …
     public :
      void f (int);
      void f (char);
      void g (int);
      void g (char);
      …
 };
class B : public A
 {   …
     public :
      void f (int);
      void f (float);
         …
 };
```

```
int main()
 {
     int n;
     float x;
     char c;
     B b;
     …
     b.f(n);// appel de B::f(int)
     b.f(x);// appel de B::f(float)
     b.f(c);// appel de B::f(int)
     …// avec conversion de c en int
     …// pas d'appel à A::f(int)
     …// ni d'appel à A::f(char)
     …
     b.g(n);// appel de A::g(int)
     b.g(x);// appel de A::g(int)
        // conversion de x en int
     b.g(c);// appel de A::g(char)

 }
```

Héritage simple et amitié (1/2)

- **Mêmes autorisations d'accès pour les fonctions amies d'une classe dérivée que pour ses méthodes**

Pas d'héritage au niveau des déclarations d'amitié

```
class A
{    friend class ClasseAmie;
     public:
       A(int n=0) : attributDeA(n) {}
     private:
       int attributDeA;
 };
class ClasseAmie
{    public:
       ClasseAmie(int n=0) : objetMembre(n) {}
       void affiche1() {cout << objetMembre.attributDeA << endl;}
                          // OK: Cette classe est est amie de A
     private:
       A objetMembre;
};
```

Héritage simple et amitié (2/2)

```
class A
{    friend class ClasseAmie;
     public:
       A(int n=0): attributDeA(n) {}
     private:
       int attributDeA;
 };
class ClasseAmie
{  public:
       ClasseAmie(int n=0): objetMembre(n) {}
       void affiche1() {cout << objetMembre.attributDeA << endl;}
                          // OK: Cette classe est est amie de A
     private:
       A objetMembre;
};
class ClasseDérivée: public ClasseAmie
{      public:
         ClasseDérivée(int x=0,int y=0): ClasseAmie(x), objetMembre2(y) {}
         void Ecrit() { cout << objetMembre2.attributDeA << endl; }
                        // ERREUR: ClasseDérivée n'est pas amie de A
                        // error: `int A::attributDeA' is private
       private:
         A objetMembre2;
 };
```

Compatibilité entre classe de base et classe dérivée (1/2)

- Possibilité de **convertir implicitement une instance d'une classe dérivée en une instance de la classe de base**, si l'héritage est public

L'inverse n'est pas possible : impossibilité de convertir une instance de la classe de base en instance de la classe dérivée

```
ClasseDeBase a;
ClasseDérivée b;
a=b; // légal et appel de l'opérateur = de la classe de Base
     // s'il a été redéfini ou de l'opérateur par défaut sinon
b=a; // illégal
     // error: no match for 'operator=' in `b = a'
```

Compatibilité entre classe de base et classe dérivée (2/2)

- Possibilité de convertir un pointeur sur une instance de la classe dérivée en un pointeur sur une instance de la classe de base, si l'héritage est public

```
ClasseDeBase o1, * p1=NULL;
ClasseDérivée o2, * p2=NULL;

p1=&o1; p2=&o2;

p1->affiche() ; // Appel de ClasseDeBase::affiche();
p2->affiche() ; // Appel de ClasseDérivée::affiche();

p1=p2; // legale: ClasseDérivée* vers ClasseDeBase*

p1->affiche() ; // Appel de ClasseDeBase::affiche();
                // et non de ClasseDérivée::affiche();

p2=p1; // erreur sauf si on fait un cast explicite
// error: invalid conversion from `ClasseDeBase*' to `ClasseDérivéé*'
p2= (ClasseDérivée*) p1; // Possible mais Attention les attributs membres
                         // de la classe dérivée n'auront pas de valeur
```

Héritage simple et opérateur d'affectation (1/6)

- Si pas de redéfinition de l'opérateur = dans la classe dérivée :

  ⇒Affectation membre à membre

  ⇒Appel implicite à l'opérateur = sur-défini ou par défaut de la classe de base pour l'affectation de la partie héritée

- Si redéfinition de l'opérateur = dans la classe dérivée :

  ⇒Prise en charge totale de l'affectation par l'opérateur = de la classe dérivée

Héritage simple et opérateur d'affectation (2/6)

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

class point
{  protected :
     int x, y ;
   public :

     point (int abs=0, int ord=0)
       { x=abs ; y=ord ;}

     point & operator = (const point & a)
      { x = a.x ; y = a.y ;
        cout << "operateur = de point" << endl;
        return * this ;
      }
} ;
```

Héritage simple et opérateur d'affectation (3/6)

```
class pointcol : public point
{
  int couleur ;
  public :
  pointcol (int abs=0, int ord=0, int c=0);
  // Pas de redéfinition de l'opérateur = dans la classe dérivée
};

pointcol::pointcol(int abs, int ord, int c) : point(abs,ord)
{ couleur=c;}

int main()
{
   pointcol c, d;
   d=c;
}
```

operateur = de point

Héritage simple et opérateur d'affectation (4/6)

```
class pointcol : public point
{
  int couleur ;
  public :
  pointcol (int abs=0, int ord=0, int c=0);
  // Redéfinition de l'opérateur = dans la classe dérivée
  pointcol & operator = (const pointcol & a)
     {  couleur=a.couleur;
        cout << "operateur = de pointcol" << endl;
        return * this ;
      }
};
pointcol::pointcol(int abs, int ord, int c) : point(abs,ord)
{ couleur=c;}

int main()
{  pointcol c, d;
   d=c;
}
```

```
operateur = de pointcol
```

Héritage simple et opérateur d'affectation (5/6)

```
// Redéfinition de l'opérateur = dans la classe dérivée
// Avec appel explicite à l'opérateur = de la classe de base
// en utilisant des conversions de pointeurs
pointcol & pointcol::operator = (const pointcol & a)
 {  point * p1;
    p1=this; // conversion d'un pointeur sur pointcol
             // en pointeur sur point
    const point *p2= &a;  // idem
    *p1=*p2; // affectation de la partie « point » de a
    couleur=a.couleur;
    cout << "operateur = de pointcol" << endl;
    return * this ;
 }

int main()
{  pointcol c, d;
   d=c;
}
```

```
operateur = de point
operateur = de pointcol
```

Héritage simple et opérateur d'affectation (6/6)

```
// Redéfinition de l'opérateur = dans la classe dérivée
// Avec appel explicite à l'opérateur =
// de la classe de base
pointcol & pointcol::operator = (const pointcol & a)
 {  // Appel explicite à l'opérateur = de point
    this->point::operator=(a);
    couleur=a.couleur;
    cout << "operateur = de pointcol" << endl;
    return * this ;
 }

int main()
{
   pointcol c, d;
    d=c;
}
```

```
operateur = de point
operateur = de pointcol
```

UNIVERSITE PARIS DAUPHINE

Master Mathématiques, Informatique, Décision, Organisation (MIDO)

1ère année

Le langage C++ (partie III)

Maude Manouvrier

- Héritage multiple
- Héritage virtuel
- Fonction / méthode virtuelle et typage dynamique
- Fonction virtuelle pure et classe abstraite
- Patrons de fonctions
- Patrons de classes

Héritage multiple (1/5)

- Possibilité de créer des classes dérivées à partir de plusieurs classes de base
- Pour chaque classe de base : possibilité de définir le mode d'héritage
- **Appel des constructeurs dans l'ordre de déclaration de l'héritage**
- **Appel des destructeurs dans l'ordre inverse de celui des constructeurs**

Héritage multiple (2/5)

```
class Point
{
  int x;
  int y;
  public:
   Point(...){...}
   ~Point(){...}
   void affiche(){...}
};
```

```
class Couleur
{
  int coul;
  public:
    Couleur(...){...}
    ~Couleur(){...}
    void affiche(){...}
};
```

```
// classe dérivée de deux autres classes
class PointCouleur : public Point, public Couleur
{
  ...
   // Constructeur
   PointCouleur (...) : Point(...), Couleur(...)
   void affiche(){Point::affiche(); Couleur::affiche(); }
};
```

Héritage multiple (3/5)

```
int main()
{   PointCouleur p(1,2,3);
    cout << endl;
    p.affiche(); // Appel de affiche() de PointCouleur
    cout << endl;
    // Appel "forcé" de affiche() de Point
    p.Point::affiche();
    cout << endl;
    // Appel "forcé" de affiche() de Couleur
    p.Couleur::affiche();
}
```

```
** Point::Point(int,int)
** Couleur::Couleur(int)
** PointCouleur::PointCouleur(int,int ,int)
Coordonnées : 1 2
Couleur : 3
Coordonnées : 1 2
Couleur : 3
** PointCouleur::~PointCouleur()
** Couleur::~Couleur()
** Point::~Point()
```

Si `affiche()` n'a pas été redéfinie dans `PointCouleur` :

```
error: request for
member `affiche' is
ambiguous

error: candidates are:
void Couleur::affiche()
void Point::affiche()
```

Héritage multiple (4/5)

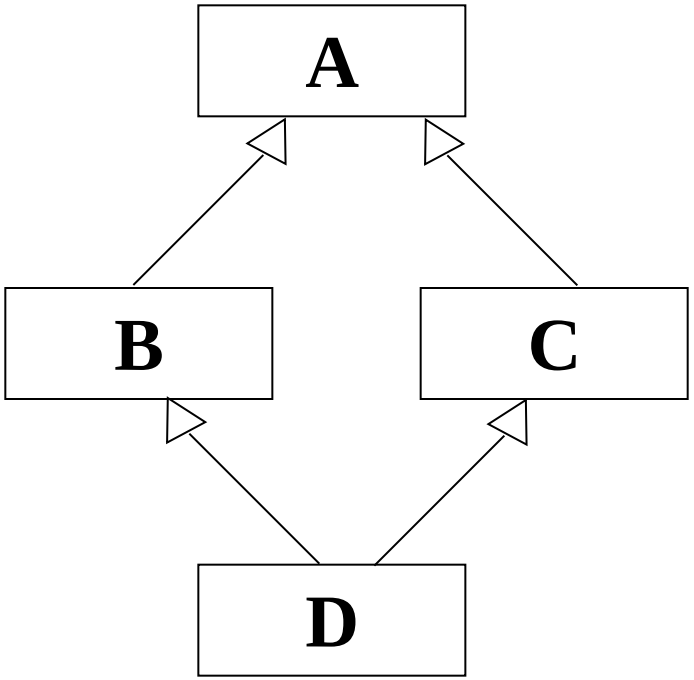
```
class A
{ public:
    A(int n=0) { /* ... */ }
  // ...
};
```

```
class B
{ public:
    B(int n=0) { /* ... */ }
  // ...
};
```

```
class C: public B, public A
{ //      ^^^^^^^^^^^^^^^^^^
  //      ordre d'appel des constructeurs des classes de base
  public:
    C(int i, int j) : A(i) , B(j) // Attention l'ordre ici ne
     { /* ... */ }                // correspond pas à l'ordre
                                  // des appels
   // ...
};

int main()

{
  C objet_c;
  // appel des constructeurs B(), A() et C()
  // ...
}
```

Héritage multiple (5/5)

```
class A
{ int x, y;
  ....
};

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

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



- **Duplication des membres données de A dans tous les objets de la classe D**
- Possibilité de les distinguer les copies par `A::B::x` et `A::C::x` ou `B::x` et `C::x` (si pas de membre `x` pour `B` et `C`)
- **Pour éviter la duplication : héritage virtuel**

Héritage virtuel (1/6)

- Possibilité de déclarer une classe « virtuelle » au niveau de l'héritage **pour préciser au compilateur les classes à ne pas dupliquer**
- Placement du mot-clé virtuel avant ou après le mode de dérivation de la classe

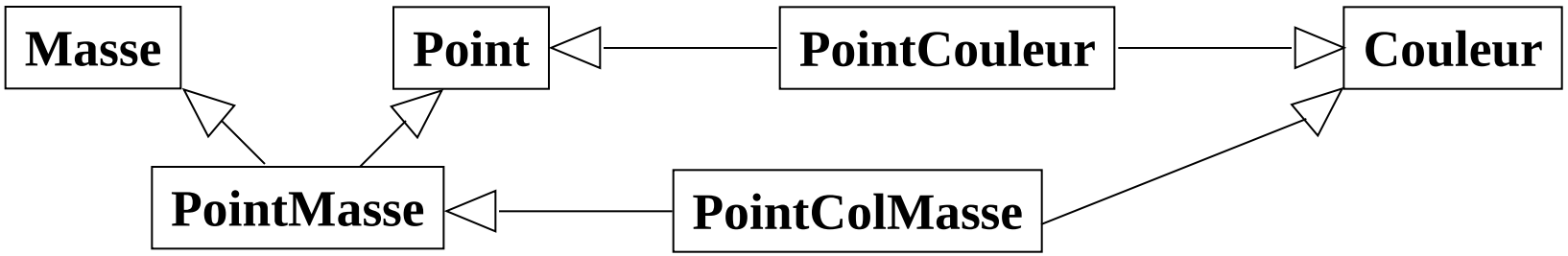
```
// La classe A ne sera introduite qu'une seule fois dans les
// descendants de B - aucun effet sur la classe B elle-même
class B : public virtual A {....};

// La classe A ne sera introduite qu'une seule fois dans les
// descendants de C - aucun effet sur la classe C elle-même
class C : public virtual A {....};

// Ici le mot-clé virtual ne doit pas apparaître!!
class D : public B, public C {....};
```

Héritage virtuel (2/6)

Interdiction de préciser des informations à transmettre au constructeur de la classe A dans les classes B et C

- Mais indication, dans le constructeur de la classe D de quels arguments transmettre au constructeur de la classe A

Nécessité d'avoir un constructeur sans argument dans la classe A



Héritage virtuel (3/6)

```
class point
{ int x, y ;
  public :
   point (int abs, int ord)
     { cout << "++ Constr. point " << abs << " " << ord << endl ;
       x=abs ; y=ord ;
     }
   point ()  // constr. par défaut nécessaire pour dérivations virtuelles
     { cout << "++ Constr. defaut point \n" ; x=0 ; y=0 ; }
   void affiche ()
    { cout << "Coordonnees : " << x << " " << y << endl ;}
} ;
class coul
{  short couleur ;
  public :
   coul (short cl)
     { cout << "++ Constr. coul " << cl <<  endl ;
       couleur = cl ;
     }
   void affiche ()
     { cout << "Couleur : " << couleur << endl ;
     }
} ;
```

Héritage virtuel (4/6)

```
// Exemple repris de [Delannoy, 2004]
class pointcoul : public virtual point, public coul
{ public :
    pointcoul (int abs, int ord, int cl) : coul (cl)
      // pas d'info pour point car dérivation virtuelle
    { cout << "++++ Constr. pointcoul "
           << abs << " " << ord << " «   << cl << endl ;
    }
    void affiche ()
     { point::affiche () ; coul::affiche () ;
     }
} ;
class masse
{  int mas ;
  public :
   masse (int m)
     { cout << "++ Constr. masse " << m << endl ;
       mas = m ;
     }
   void affiche ()
     { cout << "Masse    : " << mas << endl ;
     }
 } ;
```

Héritage virtuel (5/6)

```
class pointmasse : public virtual point, public masse
{ public :
    pointmasse (int abs, int ord, int m) : masse (m)
     // pas d'info pour point car dérivation virtuelle
    { cout << "+++ Constr. pointmasse " << abs << " "
           << ord << " " << m << "\n" ;
    }
    void affiche ()
     { point::affiche () ; masse::affiche () ;
     }
} ;
class pointcolmasse : public pointcoul, public pointmasse
{ public :
    pointcolmasse (int abs, int ord, short c, int m) : point (abs, ord),
      pointcoul (abs, ord, c), pointmasse (abs, ord, m)
      // infos abs ord en fait inutiles pour pointcol et pointmasse
    { cout << "+++ Constr. pointcolmasse " << abs + " " << ord << " "
      << c << " " << m << endl ;
    }
    void affiche ()
     { point::affiche () ; coul::affiche() ; masse::affiche () ;
     }
} ;
```

Héritage virtuel (6/6)

```
int main()
{  pointcoul p(3,9,2) ;
```

```
++ Constr. defaut point
++ Constr. coul 2
++++ Constr. pointcoul 3 9 2
```

```
   p.affiche () ; // appel de affiche de pointcoul
```

```
Coordonnees : 0 0
Couleur : 2
```

```
   pointmasse pm(12, 25, 100) ;
```

```
++ Constr. defaut point
++ Constr. masse 100
++++ Constr. pointmasse 12 25 100
```

```
   pm.affiche () ;
```

```
Coordonnees : 0 0
Masse   : 100
```

```
   pointcolmasse pcm (2, 5, 10, 20);
```

```
++ Constr. point 2 5
++ Constr. coul 10
++++ Constr. pointcoul 2 5 10
++ Constr. masse 20
++++ Constr. pointmasse 2 5 20
++++ Constr. pointcolmasse ++++ Constr.
```

```
   pcm.affiche () ;
}
```

```
pointcolmasse 5 10 20
Coordonnees : 2 5
Couleur : 10
Masse   : 20
```

Fonction/Méthode virtuelle et typage dynamique (1/9)

Ne pas confondre héritage virtuel et le statut virtuel des fonctions et des méthodes

- **Liaison statique** : **type de l'objet pointé déterminé au moment de la compilation.**

  De même pour les méthodes à invoquer sur cet objet

  ⇒Appel des méthodes correspondant au type du pointeur et non pas au type effectif de l'objet pointé

- **Polymorphisme** ⇒ possibilité de choisir dynamiquement (à l'exécution) une méthode en fonction de la classe effective de l'objet sur lequel elle s'applique – **liaison dynamique**

- **Liaison dynamique obtenue en définissant des méthodes virtuelles**

Fonction/Méthode virtuelle et typage dynamique (2/9)

```
class Personne
{
  string nom;
  string prenom;
  int age;
  char sexe;

  public :
   // Constructeur
   Personne(string n, string p, int a, char s)
   { nom=n; prenom=p; age=a; sexe=s;
     cout << "Personne::Personne("<<nom<< "," <<
       prenom << "," << age << ","  <<  sexe << ")"  << endl;
   }

  // Affichage
  void Affiche()
  { if (sexe == 'M') cout << "Monsieur "
     else cout << "Madame/Mademoiselle " ;
    cout << prenom  << " " << nom << " agée de " <<
           age << " ans." << endl;
  }

  // Destructeur
  ~Personne() {cout << "Personne::~Personne()" << endl;}
};
```

Fonction/Méthode virtuelle et typage dynamique (3/9)

```
class Etudiant : public Personne
{ int note;
  public:
  // Constructeur
  Etudiant(string nm, string p, int a, char s, int n):Personne(nm,p,a,s)
  { note = n;
   cout << "Etudiant::Etudiant(" <<GetNom()<< "," << GetPrenom() <<
    ","<< GetAge() << "," << GetSexe() << "," << note << ")" << endl;
  }
  void Affiche() // Affichage
  { Personne::Affiche();
  cout << "Il s'agit d'un étudiant ayant pour note :" << note << "." <<
   endl;
  }
  // Destructeur
  ~Etudiant() {cout << "Etudiant::~Etudiant()" << endl;}
};

int main()
{ Personne * p1 = new Etudiant("GAMOTTE","Albert",34,'M',13);
  p1->Affiche();
  delete p1;
}
```

```
// Pas d'appel de la méthode Affiche() ou du
// destructeur de la classe Etudiant
```

```
Personne::Personne(GAMOTTE,Albert,34,M)
Etudiant::Etudiant(GAMOTTE,Albert,34,M,13)
Monsieur Albert GAMOTTE agé de 34 ans.
Personne::~Personne()
```

Fonction/Méthode virtuelle et typage dynamique (4/9)

```
class Personne
{
  string nom;
  string prenom;
  int age;
  char sexe;

  public :
   // Constructeur
   Personne(string n, string p, int a, char s)
   { nom=n; prenom=p; age=a; sexe=s;
     cout << "Personne::Personne("<<nom<< "," <<
       prenom << "," << age << ","  <<  sexe << ")"  << endl;
   }

  virtual void Affiche() // Affichage
  { if (sexe == 'M') cout << "Monsieur "
     else cout << "Madame/Mademoiselle " ;
    cout << prenom  << " " << nom << " agée de " <<
           age << " ans." << endl;
  }

  // Destructeur
  virtual ~Personne() {cout << "Personne::~Personne()" << endl;}
};
```

Fonction/Méthode virtuelle et typage dynamique (5/9)

```
int main()
{
  Personne * p1 = new Etudiant("GAMOTTE","Albert",34,'M',13);

  // Appel de la méthode Affiche() de la classe Etudiant
  // La méthode étant virtuelle dans la classe Personne
  p1->Affiche();

  // Appel du destructeur de la classe Etudiant
  // qui appelle celui de la classe Personne
  delete p1;
}
```

```
Personne::Personne(GAMOTTE,Albert,34,M)
Etudiant::Etudiant(GAMOTTE,Albert,34,M,13)
Monsieur Albert GAMOTTE agé de 34 ans.
Il s'agit d'un étudiant ayant pour note :13.
Etudiant::~Etudiant()
Personne::~Personne()
```

Fonction/Méthode virtuelle et typage dynamique (6/9)

Toujours déclarer virtuel le destructeur d'une classe de base destinée à être dérivée pour s'assurer que une libération complète de la mémoire

- Pas d'obligation de redéfinir une méthode virtuelle dans les classes dérivées
- Possibilité de redéfinir une méthode virtuelle d'une classe de base, par une méthode virtuelle ou non virtuelle dans une classe dérivée

Nécessité de respecter le prototype de la méthode virtuelle redéfinie dans une classe dérivée (même argument et même type retour)

Fonction/Méthode virtuelle et typage dynamique (7/9)

- Possibilité d'identifier et de comparer à l'exécution le type d'un objets désigné par un pointeur ou une référence
- **Opérateur `typeid`** permettant de récupérer les informations de type des expressions
- Informations de type enregistrées dans des objets de la **classe `type_info`**
  - Classe prédéfinie dans l'espace de nommage `std`
  - Classe offrant des opérateurs de comparaison de type (== et `!=`) et une méthode `name()` retournant une chaîne de caractères représentant le nom du type

Format des noms de types pouvant varier d'une implémentation à une autre (pas de norme)

Fonction/Méthode virtuelle et typage dynamique (8/9)

```
// Exemple repris de [Delannoy, 2004]
#include <iostream>
#include <typeinfo>      // pour typeid
using namespace std ;

class point
{ public :
     virtual void affiche () { } // ici vide
                                 // utile pour le polymorphisme
} ;

class pointcol : public point
{ public :
     void affiche () { } // ici vide
} ;

int main()
{ point p ; pointcol pc ;
  point * adp ;
  adp = &p ;
  cout << "type de adp  : " << typeid (adp).name()  << endl ;
  cout << "type de *adp : " << typeid (*adp).name() << endl ;
  adp = &pc ;
  cout << "type de adp  : " << typeid (adp).name()  << endl ;
  cout << "type de *adp : " << typeid (*adp).name() << endl ;
}
```

```
type de adp  : P5point
type de *adp : 5point
type de adp  : P5point
type de *adp : 8pointcol
```

Fonction/Méthode virtuelle et typage dynamique (9/9)

```
// Exemple repris de [Delannoy, 2004]
int main()
{
  point p1, p2 ;
  pointcol pc ;
  point * adp1, * adp2 ;
  adp1 = &p1 ; adp2 = &p2 ;
  cout << "En A : les objets pointes par adp1 et adp2
          sont de " ;
  if (typeid(*adp1) == typeid (*adp2))
            cout << "meme type" << endl ;
       else cout << "type different" << endl;
  adp1 = &p1 ; adp2 = &pc ;
  cout << "En B : les objets pointes par
          adp1 et adp2 sont de " ;
  if (typeid(*adp1) == typeid (*adp2))
            cout << "meme type" << endl ;
      else cout << "type different" << endl;
}
```

```
En A : les objets pointes par adp1 et adp2 sont de meme type
En B : les objets pointes par adp1 et adp2 sont de type
different
```

Fonction virtuelle pure et classe abstraite (1/2)

- **Classe abstraite** : classe sans instance, destinée uniquement à être dérivée par d'autres classes
- **Fonction virtuelle pure** : fonction virtuelle déclarée sans définition dans une classe abstraite et devant être redéfinie dans les classes dérivées

```
class MaClasseAbstraite
{ …
  public :
  // Définition d'une fonction virtuelle pure
  // =0 signifie qu'elle n'a pas de définition
  // Attention, c'est différent d'un corps vide : { }
  virtual void FonctionVirtuellePure() = 0;
  …
}
```

Fonction virtuelle pure et classe abstraite (2/2)

- **Toute classe comportant au moins une fonction virtuelle pure est abstraite**
- Toute fonction virtuelle pure doit
  - Être redéfinie dans les classes dérivées
  - Ou être déclarée à nouveau virtuelle pure
    $\Rightarrow$ Classe dérivée abstraite
- Pas de possibilité de définir des instances d'une classe abstraite
- Mais **possibilité de définir des pointeurs et des références sur une classe abstraite**

Patrons de fonctions (1/8)

Patron de fonctions : fonction générique exécutable pour n'importe quel type de données

```
int minimum(int a, int b)
{ if(a<b)return a;
   else return b;
}
```

```
float minimum(float a, float b)
{ if(a<b)return a;
   else return b;
}
```

```
// création d'un patron de fonctions
template <typename T> T minimum (T a, T b)
{   if (a < b) return a ;
     else return b ; // ou return a < b ? a : b ;
}
```

Patrons de fonctions (2/8)

- **Définition d'un patron : `template <typename T>` ou `template <class T>`**
- **Paramètre de type quelconque : `T`**

```
// Exemple d'utilisation du patron de fonctions minimum
// repris de [Delannoy, 2004]
int main()
{
   int n=4, p=12 ;
   float x=2.5, y=3.25 ;
   cout << "minimum (n, p) = " << minimum (n, p) << endl ;
   cout << "minimum (x, y) = " << minimum (x, y) << endl ;

   char * adr1 = "monsieur", * adr2 = "bonjour" ;
   cout << "minimum (adr1, adr2) = " << minimum (adr1, adr2)
        << endl; ;
}
```

```
minimum (n, p) = 4
minimum (x, y) = 2.5
minimum (adr1, adr2) = monsieur
```

Patrons de fonctions (3/8)

```
// Exemple d'utilisation du patron de fonctions minimum
// repris de [Delannoy, 2004]
class vect
{  int x, y ;
  public :
    vect (int abs=0, int ord=0) { x=abs ; y=ord; }
    void affiche () { cout << x << " " << y ; }
    friend int operator < (vect&, vect&) ;
} ;
int operator < (vect& a, vect& b)
{  return a.x*a.x + a.y*a.y < b.x*b.x + b.y*b.y ;}
int main()
{
    vect u (3, 2), v (4, 1), w ;
    w = minimum (u, v) ;
    cout << "minimum (u, v) = " ; w.affiche() ;
}
```

```
minimum (u, v) = 3 2
```

Patrons de fonctions (4/8)

- **Mécanisme des patrons :**

  $\Rightarrow$ Instructions utilisées par le compilateur pour fabriquer à chaque fois que nécessaire les instructions correspondant à la fonction requise

  $\Leftrightarrow$ « des déclarations »

- **En pratique, placement des définitions de patron dans un fichier approprié d'extension .h**

- **Possibilité d'avoir plusieurs paramètres de classes différentes** dans l'en-tête, dans des déclarations de variables locales ou dans les instructions exécutables

- Mais **nécessité que chaque paramètre de type apparaisse au moins une fois dans l'en-tête du patron** pour que le compilateur soit en mesure d'instancier la fonction nécessaire

Patrons de fonctions (5/8)

```
// Exemple repris de [Delannoy, 2004]
template <typename T, typename U>
 void fct(T a, T* b, U c)
{
   T x; // variable locale x de type T
   U* adr; // variable locale adr de type pointeur sur U
   ...
   adr = new U[10]; // Allocation dynamique
   ...
   int n=sizeof(T);
   ...
}
```

Nécessité de passer un ou plusieurs arguments au constructeur des objets déclarés dans le corps des patrons de fonctions

```
template <typename T> void fct (T a)
{
  T x(3); // Si on appelle fct avec un paramètre int
   ....    // le compilateur sait exécuter int x(3);
}
```

Patrons de fonctions (6/8)

Possibilité de redéfinir les patrons de fonctions

```
// Exemple repris de [Delannoy, 2004]
#include <iostream.h>

template <typename T> T minimum (T a, T b)    // patron I
{
   if(a<b) return a;
    else return b;
}

template <typename T> T minimum (T a, T b, T c) // patron II
{ return minimum(minimum(a,b),c);}

int main()
{
   int n=12, p=15, q=2;
   float x=3.5, y=4.25, z=0.25;
   cout << minimum(n,p) << endl;       // patron I
   cout << minimum(n,p,q) << endl;     // patron II
   cout << minimum(x,y,z) << endl;     // patron II
}
```

```
12
2
0.25
```

```
cout << minimum (n, x) << endl ;
// => BUG car error: no matching function for
// call to `minimum(int&,float&)
```

Patrons de fonctions (7/8)

Possibilité de redéfinir les patrons de fonctions

```
// Exemple repris de [Delannoy, 2004]

// patron numéro I
template <typename T> T minimum (T a, T b)
{  if (a < b) return a ;
      else return b ;
}

// patron numéro II
template <typename T> T minimum (T * a, T b)
{  if (*a < b) return *a ;
      else return b ;
}

// patron numéro III
template <typename T> T minimum (T a, T * b)
{  if (a < *b) return a ;
      else return *b ;
}
```

Patrons de fonctions (8/8)

Possibilité de redéfinir les patrons de fonctions

```
// Exemple repris de [Delannoy, 2004]
int main()
{
  int n=12, p=15 ;
  float x=2.5, y=5.2 ;

  // patron numéro I    int minimum (int&, int&)
  cout << minimum (n, p) << endl ;
  // patron numéro II   int minimum (int *, int&)
  cout << minimum (&n, p) << endl ;
  // patron numéro III  float minimum (float&, float *)
  cout << minimum (x, &y) << endl ;
  // patron numéro I    int * minimum (int *, int *)
  cout << minimum (&n, &p) << endl ;
}
```

```
12
12
2.5
0x22eeb0
```

Ne pas introduire d'ambiguïté

```
// Ambiguité avec le premier template
// pour minimum (&n,&p)
template <typename T> T minimum (T* a, T * b)
{  if (*a < *b) return *a ;
      else return *b ;
}
```

Patrons de classes (1/12)

Patron de classes : Définition générique d'une classe permettant au compilateur d'adapter automatiquement la classe à différents types

```
// Définition d'un patron de classes
template <class T> class Point
{ T x;
  T y;
  public:
    Point (T abs=0, T ord=0){x=abs; y=ord; }
    void affiche();
};

// Corps de la méthode affiche()
template <class T> void Point<T>::affiche()
{ cout << "Coordonnées: " << x << " " << y << endl;}
```

Patrons de classes (2/12)

```
int main()
{
  // Déclaration d'un objet
  Point<int> p1(1,3);
  // => Instanciation par le compilateur de la
  // définition d'une classe Point dans laquelle le
  // paramètre T prend la valeur int

  p1.afficher();

  // Déclaration d'un objet
  Point <double> p2 (3.5, 2.3) ;
  // => Instanciation par le compilateur de la
  // définition d'une classe Point dans laquelle le
  // paramètre T prend la valeur double

  p2.affiche () ;
}
```

Patrons de classes (3/12)

Contraintes d'utilisation des patrons :

- Définition de patrons (de fonctions ou de classes) utilisée par la compilateur pour instancier (fabriquer) chaque fois que nécessaire les instructions requises
- **Impossibilité de livrer à un utilisateur un patron de fonction ou de classe compilé**
- En pratique, **placement des définitions de patrons (de fonctions ou de classes) dans un fichier approprié d'extension .h**
- **Rappel** : pour les classes ordinaires, possibilité de livrer la déclaration des classes (.h) et un module objet correspondant aux fonction membres

Patrons de classes (4/12)

Possibilité d'avoir un nombre quelconque de paramètres génériques :

```
template <class T, class U, class V> class Essai
{
   T x; // Membre attribut x de type T
   U t[5]; // Membre attribut t de type tableau *
           // de 5 éléments de type U
   ...
   V fml(int, U); // Méthode à deux arguments,
                  // un de type entier et l'autre
                  // de type U et retournant
                  // un résultat de type V
   ...
};

Essai <int,float,int> ce1;
Essai <int, int*,double> ce2;
Essai <float, Point<int>, double> ce3;
Essai <Point<int>,Point<float>,char*> ce4;
```

Patrons de classes (5/12)

Remarques :

- Possibilité pour un patron de classes de comporter des membres (donnée ou fonction) statiques

  Attention: «Association de la notion statique au niveau de l'instance et non au niveau du patron » => un jeu de membres statiques par instance

- Possibilité d'avoir un argument formel pour une fonction patron de type patron de classe

```
template <class T> void MaFonction (Point<T>)
{ …
}
```

⇒Instanciation du type `T` par le compilateur y compris pour le patrons de classe `Point<T>`

Patrons de classes (6/12)

Patrons de classes avec paramètres d'expression :

```
template <class T, int n> class tableau
{
  T tab [n] ;
  public :
   tableau () { cout << "construction tableau" << endl ; }
   T & operator [] (int i) { return tab[i] ;}
} ;

class point
{
  int x, y ;
  public :
   point (int abs=1, int ord=1 ) : abs(x), ord(y)
     {
       cout << "constr point " << x << " " << y << endl ;
     }
   void affiche ()
     { cout << "Coordonnees : " << x << " " << y << endl ; }
} ;
```

Patrons de classes (7/12)

Patrons de classes avec paramètres d'expression :

```
int main()
{   tableau <int,4> ti ;
    int i ;
    for (i=0 ; i<4 ; i++) ti[i] = i ;
    cout << "ti : " ;
    for (i=0 ; i<4 ; i++) cout << ti[i] << " " ;
    cout << endl ;
    tableau <point,3> tp ;
    for (i=0 ; i<3 ; i++) tp[i].affiche() ;
}
```

```
construction tableau
ti : 0 1 2 3
constr point 1 1
constr point 1 1
constr point 1 1
construction tableau
Coordonnees : 1 1
Coordonnees : 1 1
Coordonnees : 1 1
```

Patrons de classes (8/12)

Spécialisation des méthodes d'un patron de classes :

```
#include <iostream>
using namespace std ;
#include <iomanip.h>  // Bibliothèque à inclure pour setprecision

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

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

// Méthode affiche() spécialisée pour les réels
void point<double>::affiche ()
{ cout << "Coordonnees : " << setprecision(2) << x << " " <<
   setprecision(2) << y << endl ; }

int main ()
{ point <int> ai (3, 5) ; ai.affiche () ;
  point <double> ad (3.55, 2.33) ; ad.affiche () ;
}
```

```
Coordonnees : 3 5
Coordonnees : 3.5 2.3
```

Patrons de classes (9/12)

Spécialisation de patron de classes :

```
template <class T> class point // Patron de classes
{ T x ; T y ;
  public :
   point (T abs=0, T ord=0) : abs(x), ord(y)
   { cout << "Constructeur du patron template <class T> class point«
          << endl;}

   void affiche () {cout << "Coordonnees : " << x << " " << y << endl;}
} ;

template <> class point<double> // Spécialisation du patron
{ double x ; double y ;
  public :
   point<double> (double abs=0, double ord=0)
   { cout << "Constructeur de template <> class point<double> " << endl;
     x = abs ; y = ord ;}
   void affiche ()
    {cout << "Coordonnees : " << setprecision(2) << x
      << " " << setprecision(2) << y << endl ;}
} ;

int main ()
{  point <int> ai (3, 5) ; ai.affiche () ;
   point <double> ad (3.55, 2.33) ; ad.affiche () ;
}
```

```
constructeur du patron template <class T> class point
Coordonnees : 3 5
constructeur de template <> class point<double>
Coordonnees : 3.5 2.3
```

Patrons de classes (10/12)

Transmission de paramètres par défaut à un patron :

```
// Patron de classes avec un 1er paramètre de valeur par défaut 3
// et de 2ème paramètre de valeur par défaut point
// classe point préalablement définie
template <int n=3, class T=point> class tableau
{  T tab [n] ;
  public :
   tableau () { cout << "construction tableau "; }
   T & operator [] (int i) { return tab[i] ;}
} ;

int main()
{  tableau <4,int> ti ; int i ; for (i=0 ; i<4 ; i++) ti[i] = i ;
   cout << "ti : " ; for (i=0 ; i<4 ; i++)
   cout << ti[i] << " " ; cout << endl ;
   tableau <2> tp ; // ⇔ tableau <2,point> tp ;
   tableau <> tp2; // ⇔ tableau <3,point> tp2 ;
}
```

```
construction tableau ti : 0 1 2 3
constr point 1 1
constr point 1 1
construction tableau
constr point 1 1
constr point 1 1
constr point 1 1
construction tableau
```

Patrons de classes (11/12)

Patron et relation d'amitié :

```
template <class T> class essai
{ int x;

  public :

  // classe amie de toutes les instances de essai
  friend class point;

  // fonction amie de toutes les instances de essai
  friend int Mafonction(int);

  // classe amie, instance d'un patron
  friend class tableau <4,int>;

  // classe patron amie de toutes les instances de essai
  friend class tableau <3,T>;
};
```

NB : Couplage entre le patron généré et les déclarations d'amitié correspondante

Pour l'instance **`essai <point>`**
⇒ déclaration d'amitié avec **`tableau <3,point>`**

Patrons de classes (12/12)

Exemple de déclaration de variables :

```
template <int n=3, class T=point> class tableau
{ T tab [n] ;
  public :
   tableau ()
    { cout << "construction tableau à" << n  << " éléments" << endl ; }
   T & operator [] (int i) { return tab[i] ;}
} ;

int main()
{ // Déclaration d'un tableau t à 2 éléments,
  // chaque élément étant un tableau à 3 objets instances de la classe point
  // Attention à bien mettre un espace avant le dernier <
   tableau <2,tableau<3,point> > t;
  // Appel de affiche() pour l'objet correspondant au 2ème point du 1er tableau
   t[1][2].affiche();
}
```

```
constr point 1 1
constr point 1 1
constr point 1 1
construction tableau à 3 éléments
constr point 1 1
constr point 1 1
constr point 1 1
construction tableau à 3 éléments
construction tableau à 2 éléments
Coordonnees : 1 1
```

1er élément de t

2ème élément de t

UNIVERSITE PARIS DAUPHINE

Master Mathématiques, Informatique, Décision, Organisation (MIDO)

1ère année

Le langage C++ (partie IV)

Maude Manouvrier

- Généralité sur la STL (*Standard Template Library*)
- Gestion des exceptions
- Flots

Généralités sur la STL

- **STL (*Standard Template Library)*** : patrons de classes et de fonctions
- Définition de structures de données telles que les conteneurs (vecteur, liste, map), itérateurs, algorithmes généraux, etc.

```
#include <vector>
#include <stack>
#include <list>
int t[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 } ;
vector<int> v1(4, 99) ;  // vecteur de 4 entiers egaux à 99
vector<int> v2(7, 0) ;   // vecteur de 7 entiers
vector<int> v3(t, t+6) ; // vecteur construit a partir de t
// Pile d'entiers utilisant un conteneur vecteur
stack<int, vector<int> > q ;
cout << "taille initiale  : " << q.size() << endl ;
for (i=0 ; i<10 ; i++) q.push(i*i) ;
list<char> lc2 ; // Liste de caractères
list<char>::iterator il1 ; // itérateur sur une liste de char
il2 = lc2.begin() ;
for (il1=lc1.begin() ; il1!=lc1.end() ; il1++) { …}
```

- « *STL - précis et concis* » de Ray Lischner, O'Reilly (Français), février 2004, ISBN: 2841772608 et www.cplusplus.com

Gestion des exceptions (1/9)

- **Exception** :
  - **Interruption de l'exécution d'un programme** suite à un événement particulier
  - Rupture de séquence **déclenchée par une instruction `throw(expression typée)`**
  - Déclenchement d'une exception ⇒ montée dans la pile d'appel des fonctions jusqu'à ce qu'elle soit attrapée sinon sortie de programme
  - Caractérisation de chaque exception par un type et le choix du bon gestionnaire d'exceptions
- Gestion des exceptions ⇒ gestion simplifiée et plus sûre des erreurs

Gestion des exceptions (2/9)

- Exception détectée à l'intérieur d'un bloc d'instructions :
  `try { // instructions}`
- Récupération et traitement de l'exception par : **`catch(type d'exception)`**

```
#include <iostream>
#include <cstdlib>  // Ancien <stdlib.h> : pour exit
using namespace std ;

class vect
{ int nelem ;
  int * adr ;
 public :
  vect (int) { adr = new int [nelem = n] ; };
  ~vect () { delete adr ; };
  int & operator [] (int) ;
} ;

// déclaration et définition d'une classe vect_limite
class vect_limite { // vide pour l'instant} ;

int & vect::operator [] (int i)
{ if (i<0 || i>=nelem) { vect_limite l ; throw (l) ;}
  return adr [i] ;
}
```

Si le paramètre i n'est pas correct, déclenchement d'une exception

Gestion des exceptions (3/9)

```
// Programme exemple d'interception
// de l'exception vect_limite
int main ()
{
  try // Zone de surveillance
    {
      vect v(10) ;
      v[11] = 5 ;     // Ici déclenchement d'une exception
                      // car l'indice est trop grand
    }

    catch (vect_limite l) // Traitement de l'exception
    {
      cout << "exception limite" << endl ;
      exit (-1) ; // Sortie du programme
    }
}
```

```
exception limite
```

Gestion des exceptions (4/9)

```
// déclaration - définition des deux classes pour les exceptions
class vect_limite
{ public :
   int hors ; // valeur indice hors limites (public)
   vect_limite (int i) { hors = i ; }   // constructeur
} ;

class vect_creation
{ public :
   int nb ; // nombre éléments demandes (public)
   vect_creation (int i) { nb = i ; }  // constructeur
} ;

// Redéfinition du constructeur de la classe vect
vect::vect (int n)
{ if (n <= 0)
    { vect_creation c(n) ;      // anomalie
      throw c ;
    }
  adr = new int [nelem = n] ; // construction si n est correct
}
```

Gestion des exceptions (5/9)

```
int & vect::operator [] (int i)
{ if (i<0 || i>nelem)  { vect_limite l(i) ;      // anomalie
                         throw l ; }
  return adr [i] ;  // fonctionnement normal
}

// Programme exemple pour intercepter les exceptions
int main ()
{
  try // Zone de surveillance
  { vect v(-3) ;       // provoque l'exception vect_creation
    v[11] = 5 ;        // provoquerait l'exception vect_limite
  }
  catch (vect_limite l) // Traitement de l'exception vect_limite
  { cout << "exception indice " << l.hors
      << " hors limites " << endl ;
    exit (-1) ;
  }
  catch (vect_creation c) // Traitement de l'exception vect_creation
  { cout << "exception creation vect nb elem = " << c.nb << endl ;
    exit (-1) ;
  }
 }
```

```
exception creation vect nb elem = -3
```

Gestion des exceptions (6/9)

- Fichier en-tête **<stdexcept>** bibliothèque standard fournissant des classes d'exceptions
- Plusieurs exceptions standard susceptibles d'être déclenchées par une fonction ou un opérateur de la bibliothèque standard

  Ex. classe **bad_alloc** en cas d'échec d'allocation mémoire par **new**

```
vect::vect (int n) // Constructeur de la classe vect
{ adr = new int [nelem = n] ;}

int main ()
{ try { vect v(-3) ; }
  catch (bad_alloc) // Si le new s'est mal passé
  { cout << "exception création vect
      avec un mauvaise nombre d'éléments " << endl ;
    exit (-1) ;
  }
}
```

```
exception creation vect avec un mauvaise nombre d'éléments
```

- En cas d'exception non gérée ⇒ appel automatique à **terminate()** qui exécute **abort();**

Gestion des exceptions (7/9)

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

Classes derivées de `exception` :

`bad_alloc, bad_cast, bad_exception, bad_typeid` …

Gestion des exceptions (8/9)

- Classe **`exception`** ayant une méthode virtuelle **`what()`** affichant une chaîne de caractères expliquant l'exception

```
int main ()
{
  try { vect v(-3) ;}
  catch (bad_alloc b)
  { // Appel de la méthode what pour l'exception bad_alloc
    cout << b.what() << endl ; exit (-1) ;
  }
}
```

```
St9bad_alloc
```

- Possibilité de dériver ses propres classes de la classe exception

```
class mon_exception : public exception
{ public :
  mon_exception (char * texte) { ad_texte = texte ; }
  const char * what() const throw() { return ad_texte ; }
  private :
    char * ad_texte ;
} ;
```

Gestion des exceptions (9/9)

```
int main()
{ try
  { cout << "bloc try 1" << endl;
    throw mon_exception ("premier type") ;
  }
  catch (exception & e)
  { cout << "exception : " << e.what() << endl; }
  try
  { cout << "bloc try 2" << endl;
    throw mon_exception ("deuxieme type") ;
  }
  catch (exception & e)
  { cout << "exception : " << e.what() << endl;
  }
}
```

```
bloc try 1
exception : premier type
bloc try 2
exception : deuxieme type
```

Les flots (1/15)

- **Flot** : « Canal »
  - **Recevant de l'information – flot de sortie**
  - **Fournissant de l'information – flot d'entrée**
- **`cout`** connecté à la « sortie standard »
- **`cin`** connecté à l'« entrée standard »
- 2 opérateurs **`<<`** et **`>>`** pour assurer le transfert de l'information et éventuellement son formatage
- 2 classes définies sous la forme de patrons
  - **`ostream`**
  - **`istream`**

Les flots (2/15)

Classe ostream

- **`ostream & operator << (expression)`**

  Réception de 2 opérandes :
  - La classe l'ayant appelé (implicitement **`this`**)
  - Une expression de type de base quelconque

  Possibilité de redéfinir l'opérateur << pour les types utilisateurs

- **`cerr`** : flot de sortie connecté à la sortie standard d'erreur sans tampon intermédiaire (pour l'écriture des messages d'erreur)

  ```
  cerr << "Une erreur est survenue!"
  ```

- **`clog`** : flot de sortie connecté à la sortie standard d'erreur avec tampon intermédiaire (pour les messages d'information)

Les flots (3/15)

Classe ostream : formatage

```
par defaut      : 12000
en hexadecimal : 2ee0
en decimal      : 12000
en octal        : 27340
par defaut        : 1
avec noboolalpha : 1
avec boolalpha    : true
```

```
#include <iostream>
using namespace std ;
int main()
{   int n = 12000 ;
    cout << "par defaut      : "          << n << endl ;
    // utilisation du manipulateur hex (base 16)
    cout << "en hexadecimal : " << hex << n << endl ;
    // utilisation du manipulateur dec (base 10)
    cout << "en decimal      : " << dec << n << endl ;
    // utilisation du manipulateur oct (base 8)
    cout << "en octal        : " << oct << n << endl ;

    bool ok = 1 ;    // ou ok = true
    cout << "par defaut        : "                  << ok << endl ;
    // utilisation du manipulateur noboolalpha
    // => (affichage sous forme numérique : 0 ou 1)
    cout << "avec noboolalpha : " << noboolalpha << ok << endl ;
    // utilisation du manipulateur boolalpha
    // => (affichage sous forme alphabétique : true ou false)
    cout << "avec boolalpha    : " << boolalpha   << ok << endl ;

}
```

Les flots (4/15)

Classe istream

- **`istream & operator >> (type_de_base &)`**

  Réception de 2 opérandes :
  - La classe l'ayant appelé (implicitement **`this`**)
  - Une « lvalue » de type de base quelconque

  Possibilité de redéfinir l'opérateur >> pour les types utilisateurs

- Pas de prise en compte des espaces, tabulations (`\t` ou `\v`), des fins de ligne (`\n`) etc.

- Pour prendre en compte ces caractères :
  `istream& get(char&)`

```
char c;
…
while(cin.get(c)) cout.put(c);
  // ⇔ while(cin.get(c)!=EOF) cout.put(c);
```

Les flots (5/15)

Redéfinition des opérateurs << et >> pour les types utilisateurs :

- **Opérateur** prenant un flot en premier argument donc **devant être redéfinie en fonction amie**

```
ostream & operator << (ostream &, expression_de_type_classe &)
istream & operator >> (istream &, expression_de_type_classe &)
```

- Valeur de retour obligatoirement égale à la référence du premier argument

```
class point
{ int x, y ;
  public :
   point (int abs=0, int ord=0){ x = abs ; y = ord ; }
   friend ostream & operator << (ostream &, point &) ;
   friend istream & operator >> (istream &, point &) ;
} ;
```

Les flots (6/15)

```
// Redéfinition de l'opérateur << en fonction amie
// de la classe Point
ostream & operator << (ostream & sortie, point & p)
 { sortie << "<" << p.x << "," << p.y << ">" ; return sortie ; }
// Redéfinition de l'opérateur << en fonction amie
// de la classe Point
istream & operator >> (istream & entree, point & p)
{  char c = '\0' ;
   int x, y ; bool ok = true ;
   entree >> c ; // saisie d'un caractère au clavier
   if (c != '<') ok = false ;
      else { entree >> x >> c ; // saisie de x (entier)
                                // et d'un autre caractère au clavier
             if (c != ',') ok = false ;
              else  { entree >> y >> c ; // même chose pour y
                      if (c != '>') ok = false ; }
           }
  // Si la saisie a été correcte, on affecte à p
   if (ok=true) { p.x = x ; p.y = y ; }
    // Statut d'erreur du flot géré par un ensemble de bits d'un entier
    // clear (activation de bits d'erreur)
    // badbit (bit activé quand flot dans un état irrécupérable)
    // rdstate () pur activer le bit badbit sans activer les autres
     else entree.clear (ios::badbit | entree.rdstate () ) ;
  // on retourne le flot d'entrée
   return entree ;
}
```

Les flots (7/15)

```
int main()
{
  point b ;
  cout << "donnez un point : " ;
  if (cin >> b)
    cout << "Affichage du point : " << b << endl ;
   else cout << "** information incorrecte" << endl ;
}
```

```
donnez un point : fdfdsf
** information incorrecte
```

```
donnez un point : <3,67>
Affichage du point : <3,67>
```

Les flots (8/15)

Connexion d'un flot à un fichier :

- Librairie à inclure : `#include <fstream>`
- Classe `ifstream` : Interface pour les fichier en lecture (*input*)
- Classe `ofstream` : Interface pour les fichier en écriture (*output*)
- Classe `fstream` : Interface pour manipuler les fichiers (Lecture/Écriture)
- Ouverture d'un fichier : par le constructeur de la classe ou par la méthode `open` des classes `ifstream`, `ofstream` et `fstream`

```
void open (const char * filename, openmode mode = in | out);
```

En cas d'erreur : `bool bad () const;` retourne `true`

Mode d'ouverture : possibilité de cumuler les modes avec |

- `ios_base::app` : Ouverture en ajout (à la fin) - mode *append*
- `ios_base::ate` : Ouverture et position du curseur à la fin du fichier
- `ios_base::binary` : Ouverture d'un fichier en binaire (plutôt qu'en texte)
- `ios_base::in` : Ouverture en écriture : Ouverture en lecture
- `ios_base::trunc` : Ouverture du fichier et écrasement du contenu à l'écriture

- Fermeture d'un fichier : méthode des classes `ifstream`, `ofstream` et `fstream`

```
void close ();
```

Les flots (9/15)

Connexion d'un flot à un fichier :

- **Méthode héritée de la classe `istream` par `ifstream` et `fstream` :**
  - `istream& read (char* s, streamsize n );` `// Lire`
  - `streampos tellg ( );` `// Retourner la position du curseur`
  - `istream& seekg (streampos pos)` `// Déplacer le curseur`
  - `// Déplacement du curseur de off octets à partir dir`
    `istream& seekg (streamoff off, ios_base::seekdir dir);`
    - `ios_base::beg` : Depuis le début
    - `ios_base::cur` : Depuis la position courante
    - `ios_base::end` : depuis la fin
  - Lecture d'une chaîne jusqu'à un délimiteur (`'/n'` par défaut) – insertion de `'/0'`
    - `istream& getline/get (char* s, streamsize n );`
    - `istream& getline/get (char* s, streamsize n, char delim );`
- **Méthode héritée de la classe `ostream` par `ofstream` et `fstream` :**
  - `ostream& write (const char* str , streamsize n);` `// Écrire`
  - `streampos tellp ( );`
  - `ostream& seekp (streampos pos);` `// Déplacer le curseur`
  - `ostream& seekp (streamoff off, ios_base::seekdir dir);`

Les flots (10/15)

Exemple :

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{ int length;
   char * buffer;

  // Création et ouverture de deux fichiers via les constructeurs
  ifstream infile ("test.txt",ifstream::binary);
  ofstream outfile ("new.txt",ofstream::binary);

  // Calcul de la taille du fichier
  infile.seekg (0, ios::end); // Position à la fin
  // Récupération de la position de la fin de fichier
  length = infile.tellg();
  infile.seekg (0, ios::beg); // Position du curseur au début

   // Allocation mémoire pour stocker le contenu du fichier
   buffer = new char [length];

   // Lecture du contenu d'un fichier en un seul bloc
  infile.read (buffer,length);

  outfile.write (buffer,size); // Écriture dans le fichier

  delete[] buffer;

  infile.close(); outfile.close(); // Fermeture des fichiers

  return 0;
}
```

Les flots (11/15)

Surcharge des opérateurs << et >> :

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

int main()
{
  ifstream fichierin;
  ofstream fichierout;
  int n1, n2, n3;

  fichierin.open("test1.txt", ios::in);
  fichierout.open("test2.txt", ios::out | ios::trunc);

  // Si l'ouverture n'a pas réussi - bad() retourne true
  if (fichierin.bad()) return (1); // Erreur à l'ouverture, on quitte...

  // Lecture du contenu du fichier
  // et affectation des 3 entiers lus à n1, n2 et n3
  fichierin >> n1 >> n2 >> n3; // Utilisation de l'opérateur >>

  // Ecriture du contenu du fichier par l'opérateur <<
  fichierout <<  n1 << endl << n2 << endl << n3;

  // Fermeture des fichiers
  fichierin.close();
  fichierout.close();

  return (0);
}
```

Les flots (12/15)

Exemple d'écriture d'entiers dans un fichier :

```
#include <iostream>
#include <fstream> // Librairie contenant la classe ofstream
#include <iomanip> // Librairie pour utiliser setw
using namespace std ;const int LGMAX = 20 ;

const int LGMAX = 20 ;

int main()
{   char nomfich [LGMAX+1] ; int n ;
    cout << "nom du fichier a creer : " ;
    cin >> setw (LGMAX) >> nomfich ;
    // Déclaration d'un fichier associé au flot de sortie
    // de nom nomfich et ouvert en écriture
    ofstream sortie (nomfich, ios::out) ;
    if (!sortie)
     { cout << "creation impossible " << endl ; exit (1) ; }
    do { cout << "donnez un entier : " ;
         cin >> n ;
         // Écriture dans le fichier
         if (n) sortie.write ((char *)&n, sizeof(int) ) ;
       }
    while (n && (sortie)) ;
    // fermeture du fichier
    sortie.close () ;
}
```

Les flots (13/15)

Exemple de lecture d'entiers dans un fichier :

```
#include <iostream>
#include <fstream> // Librairie contenant la classe ofstream
#include <iomanip> // Librairie pour utiliser setw
using namespace std ;const int LGMAX = 20 ;

const int LGMAX = 20 ;

int main()
{
   char nomfich [LGMAX+1] ;
   int n ;
   cout << "nom du fichier a lister : " ;
   cin >> setw (LGMAX) >> nomfich ;

   // Déclaration d'un fichier associé au flot d'entrée
   // de nom nomfich et ouvert en lecture
   ifstream entree (nomfich, ios::in) ;
   if (!entree) { cout << "ouverture impossible " << endl;
                  exit (-1) ;}
   // Tant qu'on peut lire dans le fichier
   while ( entree.read ( (char*)&n, sizeof(int) ) )
        cout << n << endl ;

   // Fermeture du fichier
   entree.close () ;
}
```

Les flots (14/15)

Exemple de déplacement de curseur :

```
const int LGMAX_NOM_FICH = 20 ;
int main()
{ char nomfich [LGMAX_NOM_FICH + 1] ;
  int n, num ;
  cout << "nom du fichier a consulter : " ;
  cin >> setw (LGMAX_NOM_FICH) >> nomfich ;
  ifstream entree (nomfich, ios::in|ios::binary) ;   // ou ios::in
  if (!entree) {cout << "Ouverture impossible" ; exit (-1); }
  do
    { cout << "Numero de l'entier recherche : " ; cin >> num ;
      if (num)
         { // Placement du curseur à la position de l'entier
           // (num-1) - la position commençant à zéro
           // et par rapport au début du fichier (ios::beg)
           entree.seekg (sizeof(int) * (num-1) , ios::beg ) ;
           entree.read ( (char *) &n, sizeof(int) ) ;
           if (entree) cout << "-- Valeur : " << n << endl ;
             else {  cout << "-- Erreur" << endl ;
                     entree.clear () ;}
         }
    }
  while (num) ;
  entree.close () ;
}
```

Les flots (15/15)

Transformer un entier en string en utilisant la classe `stringstream` :

```
#include<iostream>
#include<string>
#include<sstream>

using namespace std;

// Fonction de conversion d'un int en string
string itos(int i)
{
   stringstream s; // string sous forme de flot d'E/S
   s << i;
   return s.str(); // retourne la string associée au flot
}

int main()
{
   int i = 127;
   string ss = itos(i);
   cout << ss << " " << endl;
}
```

UNIVERSITE PARIS DAUPHINE

Master Mathématiques, Informatique, Décision, Organisation (MIDO)

1ère année

Le langage C++ (partie V)

Maude Manouvrier

Compléments d'informations :

- Conversion et opérateur de conversion
  - Conversion d'un objet en type de base
  - Conversion d'un objet en objet d'une autre classe
- Exemple d'optimisation du compilateur

Convertir un objet en type de base (1/4)

```
// Adapté de [Delannoy, 2004]
class A
{ int x ;
  public:
   A(int i =0) {x=i;} // constructeur
};

void fct (double v)
{ cout << "$$ Appel de la fonction avec comme argument :" << v << endl;}

int main()
{ A obj(1);
  int entier1; double reel1, reel2;
  entier1= obj + 1.75; // instruction 1
  cout << "entier1= " << entier1 << endl;
  reel1= obj + 1.75; // instruction 2
  cout << "reel1= " << reel1 << endl;
  reel2= obj; // instruction 3
  cout << "reel2= " << reel2 << endl ;
  fct(obj); // instruction 4
}
```

```
error: no match for 'operator+'
in 'obj + 1.75e+0'
```

```
error: no match for 'operator+'
in 'obj + 1.75e+0'
```

```
error: cannot convert `A' to
`double' in assignment
```

```
error: cannot convert `A' to `double' for argument `1' to
`void fct(double)'
```

Convertir un objet en type de base (2/4)

```
// Ajout de l'opérateur de conversion int() dans la classe A
operator A::int() // opérateur de cast A --> int
  { cout << "**Appel de int() pour l'objet d'attribut "<< x << endl;
    return x;
  }

int main()
{ A obj(1);
  int entier1; double reel1, reel2;
  entier1= obj + 1.75; // instruction 1
  cout << "entier1= " << entier1 << endl;
  reel1= obj + 1.75; // instruction 2
  cout << "reel1= " << reel1 << endl;
  reel2= obj; // instruction 3
  cout << "reel2= " << reel2 << endl ;
  fct(obj); // instruction 4
}
```

```
**Appel de int() pour l'objet d'attribut 1
```

```
**Appel de int() pour l'objet d'attribut 1
```

```
**Appel de int() pour l'objet d'attribut 1
```

```
**Appel de int() pour l'objet d'attribut 1
$$ Appel de la fonction avec comme argument :1
```

```
warning: converting to `int' from `double' : pour l'instruction 1
```

Convertir un objet en type de base (3/4)

```
// Ajout de l'opérateur de conversion double() dans la classe A
operator A::double() // opérateur de cast A --> double
  { cout << "**Appel de double() pour l'objet d'attribut"<< x << endl;
    return x;
  }

int main()
{ A obj(1);
  int entier1; double reel1, reel2;

  entier1= obj + 1.75; // instruction 1
  cout << "entier1= " << entier1 << endl;

  reel1= obj + 1.75; // instruction 2
  cout << "reel1= " << reel1 << endl;

  reel2= obj; // instruction 3
  cout << "reel2= " << reel2 << endl ;

  fct(obj); // instruction 4
}
```

Le compilateur ne sait pas s'il doit convertir `obj` en `int` ou en `double`.
Il a le choix => bug à la compilation!!

```
error: ambiguous overload for 'operator+' in 'obj + 1.75e+0'
note: candidates are: operator+(double, double) <built-in>
operator+(int, double) <built-in>
```

Convertir un objet en type de base (4/4)

```
int main()
{ A obj(1);
  int entier1; double reel1, reel2;
  // cast explicite de obj en entier => appel de int()
  entier1= (int)obj + 1.75; // instruction 1
  cout << "entier1= " << entier1 << endl;
  // cast explicite de obj en double => appel de double()
  reel1= (double)obj + 1.75; // instruction 2
  cout << "reel1= " << reel1 << endl;
  reel2= obj; // instruction 3
  cout << "reel2= " << reel2 << endl ;
  fct(obj); // instruction 4
};
```

```
**Appel de int() pour l'objet d'attribut 1
entier1= 2
**Appel de double() pour l'objet d'attribut 1
reel1= 2.75
**Appel de double() pour l'objet d'attribut 1
reel2= 1
**Appel de double() pour l'objet d'attribut 1
$$ Appel de la fonction avec comme argument :1
```

Convertir un objet en objet d'une autre classe (1/7)

```
class B
{ int b1;
  public :
   // constructeur sans argument
   B() {cout << " Passage dans B::B(); " << endl;}
   // constructeur à un argument
   B(int b) {b1=b; cout << "Passage dans B::B(int); " << endl;}
   int GetB() {return b1;}
};

class C
{ int c1;
  public :
   // Constructeur sans argument
   C() {cout << "Passage dans C::C(); " << endl; }
   // Constructeur à un argument de type entier
   C(int c) {c1=c; cout << " Passage dans C::C(int); " << endl;}
   C& operator = (const C& obj)
    { c1=obj.c1;
      cout << "Passage dans C& operator = (const C& obj)" << endl;
      return *this;
    }
 // Surdéfinition de l'opérateur + par une fonction amie
 friend C& operator+(const C&,const C&);
}
```


Convertir un objet en objet d'une autre classe (2/7)

```
int main()
{ B obj2B1, obj2B2, obj2B3;
  C obj2C1, obj2C2, obj2C3;

  obj2C1=obj2B1;
```

```
error: no match for 'operator=' in 'obj2C1 = obj2B1'
note: candidates are: C& C::operator=(const C&)
```

```
  obj2B1=obj2C1;
```

```
error: no match for 'operator=' in 'obj2B1 = obj2C1'
note: candidates are: B& B::operator=(const B&)
```

```
  obj2C1=obj2C2+obj2C3;  // => Appel de l'opérateur + et
                         // de l'opérateur = de la classe C

  obj2C1=obj2B2+obj2B3;
```

```
error: no match for 'operator+' in 'obj2B2 + obj2B3'
note: candidates are: C& operator+(const C&, const C&)
```

```
  obj2B2+obj2B3;
```

```
error: no match for 'operator+' in 'obj2B2 + obj2B3'
note: candidates are: C& operator+(const C&, const C&)
```

```
  obj2B1=obj2B2+ obj2B3;
```

```
error: no match for 'operator+' in 'obj2B2 + obj2B3'
note : candidates are: C& operator+(const C&, const C&)
```

```
}
```

Convertir un objet en objet d'une autre classe (3/7)

```
// Surdéfinition de l'opérateur + par une fonction amie
C& operator+(const C& obj1,const C& obj2)
{
  // Déclaration et initialisation d'un objet local
  // => Appel au constructeur C::C(int)
  C* obj3 = new C(obj1.c1+ obj2.c1);
  cout << "Passage dans operator+(C obj1,C obj2)" << endl;
  return *obj3;
}

// Ajout dans la classe C
// Constructeur à un argument de type instance de la classe B
C::C(B obj)
  {
    c1=obj.GetB();
    cout << "Passage dans C::C(B obj)" << endl;
  }
```

Convertir un objet en objet d'une autre classe (4/7)

```
int main()
{ B obj2B1, obj2B2, obj2B3;
  C obj2C1, obj2C2, obj2C3;

  obj2C1=obj2B1;
```

```
Passage dans C::C(B obj)
Passage dans C& operator = (const C& obj)
```

```
  obj2B1=obj2C1;
```

```
error: no match for 'operator=' in 'obj2B1 = obj2C1'
note: candidates are: B& B::operator=(const B&)
```

```
  obj2C1=obj2C2+obj2C3;  // => Appel de l'opérateur + et
                         // de l'opérateur = de la classe C
```

```
  obj2C1=obj2B2+obj2B3;
```

```
Passage dans C::C(B obj)
Passage dans C::C(B obj)
Passage dans C::C(int); // pour l'objet local obj3
Passage dans operator+(C obj1,C obj2)
Passage dans C& operator = (const C& obj)
```

```
  obj2B2+obj2B3;
```

```
Passage dans C::C(B obj)
Passage dans C::C(B obj)
Passage dans C::C(int); // pour l'objet local obj3
Passage dans operator+(C obj1,C obj2)
```

```
  obj2B1=obj2B2+ obj2B3;
}
```

```
error: no match for 'operator=' in 'obj2B1 = operator+(((const
C&)(&C(obj2B2))), ((const C&)(&C(obj2B3))))'
 note: candidates are: B& B::operator=(const B&)
```

Convertir un objet en objet d'une autre classe(5/7)

```
// Remplacement du constructeur C::C(B)
// par un opérateur de conversion B --> C
// dans la classe B
 operator C ()
 { // Déclaration et initialisation d'une variable locale
   // => Appel à C::C(int)
   C* obj = new C(b1);
   cout << "Passage dans B::operator C(); " << endl;
   return *obj;
 }

  // Attention pour cet opérateur la classe C doit être
  définie avant la classe B
```

Convertir un objet en objet d'une autre classe (6/7)

```
int main()
{ B obj2B1, obj2B2, obj2B3;
  C obj2C1, obj2C2, obj2C3;

  obj2C1=obj2B1;

  obj2B1=obj2C1;

  obj2C1=obj2C2+ obj2C3; // => Appel de l'opérateur + et
                         // de l'opérateur = de la classe C

  obj2C1=obj2B2+obj2B3;

   obj2B2+obj2B3;

  obj2B1=obj2B2+ obj2B3;
}
```

```
Passage dans C::C(int); // pour la variable locale
Passage dans B::operator C();
Passage dans C& operator = (const C& obj)
```

```
error: no match for 'operator=' in 'obj2B1 = obj2C1'
note: candidates are: B& B::operator=(const B&)
```

```
Passage dans C::C(int); Passage dans B::operator C();
Passage dans C::C(int); Passage dans B::operator C();
Passage dans C::C(int);
Passage dans operator+(C obj1,C obj2)
Passage dans C& operator = (const C& obj)
```

```
Passage dans C::C(int); Passage dans B::operator C();
Passage dans C::C(int); Passage dans B::operator C();
Passage dans C::C(int); Passage dans operator+(C obj1,C obj2)
```

```
error: no match for 'operator=' in 'obj2B1 = operator+(((const
C&)((const C*)(&(&obj2B2)->B::operator C()))), ((const C&)((const
C*)(&(&obj2B3)->B::operator C()))))'
note: candidates are: B& B::operator=(const B&)
```

Convertir un objet en objet d'une autre classe (7/7)

Si définition de l'opérateur +
par une méthode dans la classe C :

```
C& C::operator+(const C& obj1)
{ c1=obj1.c1;
  cout << "Passage dans C::operator+(C obj1)"
     << endl;
  return *this;
}
```

⇒**Bug pour l'instruction :** `obj2B2+ obj2B3;`

```
error: no match for 'operator+' in 'obj2B2 + obj2B3'
```

Car recherche de l'opérateur + pour la classe B qui n'existe pas!!

Mais OK pour `obj2C2+obj2B3;` ⇔ `obj2C2.operator+(obj2B3);`

Exemple d'optimisation du compilateur (1/4)

```
class Chaine
{
   …
    // Fonction amie surchargeant l'opérateur +
    friend Chaine operator+(const Chaine&, const Chaine&);
   …
}

Chaine operator+(const Chaine& c1,const Chaine& c2)
{  // Déclaration d'une variable locale
     Chaine locale (c1.taille+c2.taille);

    … // Corps de la fonction (A faire en TD!!)

    // Retour d'une variable locale
    //déclarée dans le corps de la fonction
    return locale;
}
```

Exemple d'optimisation du compilateur (2/4)

```
int main()
{
   …
   Chaine a3; a3.AfficherChaine();
    a3=a1+a2; a3.AfficherChaine();
   Chaine a4=a1+a2; a4.AfficherChaine();
   …
}
```

a3 a pour adresse mémoire 0x22ee70

```
**Constructeur sans paramètre pour l'objet 0x22ee70 avec une taille de 255 et une valeur de ""
```

```
La valeur de l'objet 0x22ee70 est "" et sa taille est : 0
```

Exemple d'optimisation du compilateur (3/4)

```
a3=a1+a2; a3.AfficherChaine();
```

```
**Entrée de l'opérateur + avec un paramétre c1 de valeur="ReBonjour" et
de taille 9 et un paramétre c2 de valeur=" Toto" et de taille 5
**Constructeur avec un paramètre entier pour l'objet 0x22ee60 avec une
taille max de 14 et une valeur de ""
**Sortie de l'opérateur + avec une variable locale d'adresse 0x22ee60 de
valeur "ReBonjour Toto" et de taille 14
```

Déclaration de la variable locale

```
**Entrée dans l'Operateur = avec un paramatre c de valeur="ReBonjour
Toto" et taille 14
**Sortie dans l'Operateur = avec pour l'objet courant valeur="ReBonjour
Toto" et taille=14
```

```
**Destructeur pour l'objet 0x22ee60 de valeur "ReBonjour Toto" et de
taille=14 **
```

Destruction de la variable locale après avoir copié sa valeur dans a3

```
La valeur de l'objet 0x22ee70 est "ReBonjour Toto" et sa taille est 14
```

La variable locale de la méthode surchargeant l'opérateur + a pour adresse 0x22ee60

Exemple d'optimisation du compilateur (4/4)

```
Chaine a4=a1+a2; a4.AfficherChaine();
```

```
**Entrée de l'opérateur + avec un paramétre c1 de valeur="ReBonjour" et
de taille 9 et un paramétre c2 de valeur=" Toto" et de taille 5
**Constructeur avec un paramètre entier pour l'objet 0x22ee60 avec une
taille max de 14 et une valeur de ""
**Sortie de l'opérateur + avec une variable locale d'adresse 0x22ee60 de
valeur "ReBonjour Toto" et de taille 14
```

← Déclaration de la variable locale

Pas de destruction de la variable locale après avoir copié sa valeur dans a4
Optimisation du compilateur : a4 prend l'adresse de la variable locale (0x22ee60)
donc pas d'appel au copy constructeur!

```
La valeur de l'objet 0x22ee60 est "ReBonjour Toto" et sa taille est 14

#Destruction des objets du dernier crée au premier crée.##
*Destructeur pour l'objet 0x22ee60 de valeur "ReBonjour Toto" et de
taille=14 **
**Destructeur pour l'objet 0x22ee70 de valeur "ReBonjour Toto" et de
taille=14 **
**Destructeur pour l'objet 0x22ee80 de valeur " Toto" et de taille=5 **
**Destructeur pour l'objet 0x22ee90 de valeur "ReBonjour" et de taille=9
```

← a4

← a3

UNIVERSITE PARIS DAUPHINE

Master Mathématiques, Informatique, Décision, Organisation (MIDO)

1ère année

Le langage C++ (partie VI)

Maude Manouvrier

Création de Bibliothèques dynamiques / DLL :

- **Définitions**
- **Exemple de DLL C++**
- **Exemple de programme C++ utilisant une DLL de manière dynamique**
- **Exemple de programme C++ utilisant une DLL de manière statique**
- **Exemple de DLL C++ contenant une classe**

Qu'est qu'une DLL

- ***Librairy* ou bibliothèque** : fichier contenant plusieurs fonctions d'un programme
- **DLL** (*Dynamic Link Library*) ou bibliothèque liée dynamiquement (Windows) :
  - **Compilée**, donc prête à être utilisée, **chargée dynamiquement en mémoire** lors du démarrage d'un programme (i.e. n'étant pas incluse dans le programme exécutable)
  - Indépendante du(des) programmes qui l'utilise(nt) et pouvant être utilisée par plusieurs programmes en même temps
  - Stockée une seule fois sur le disque dur
  - Permettant aux développeurs (i) de distribuer des fonctions réutilisables par tout programme, sans en dévoiler les sources, (ii) de réduire la taille des exécutables et (iii) de mettre à jour les librairie indépendamment des programmes les utilisant

Développement d'une DLL C++

- Structure d'une DLL : Code exécutable en vue d'être appelé par un programme externe
- Table d'exportation de symboles (*Symbols Export Table*) : liste de toutes les fonctions exportées (donc disponibles par un programme externe) ainsi que de leur point d'entrée (adresse du début du code de la fonction)
- Pour développer une DLL sous Visual C++ :
  - Créer un nouveau projet de type *Win32 Dynamic Link Library*
  - Choisir un projet de type « *Empty project* »
  - Créer 3 fichiers :
    - `nom_fichier.h` : contenant les inclusions de bibliothèques (au minimum la bibliothèque standard et `windows.h`) et la définition des fonctions de la bibliothèque
    - `Nom_fichier.cpp` : fichiers contenant le corps des fonctions (et incluant le fichier `nom_fichier.h`)
- DLL existantes :

  `http://msdn.microsoft.com/en-us/library/ms723876(vs.85).aspx`

Exemple de DLL C++ (1/4)

- Fichier **`MaDll.h`:**

```
#include <iostream>
#include <cmath>
#include <windows.h>

// Fonctions exportées de la DLL
extern "C" _declspec(dllexport) double carre(double& arg);
extern "C" _declspec(dllexport) double
   addition(double& arg1, double& arg2);
extern "C" _declspec(dllexport) double
    soustraction(double arg1, double arg2);

// Fonction d'entrée de la DLL
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD ul_reason_for_call,
                       LPVOID lpReserved)
{
   return TRUE;
}
```

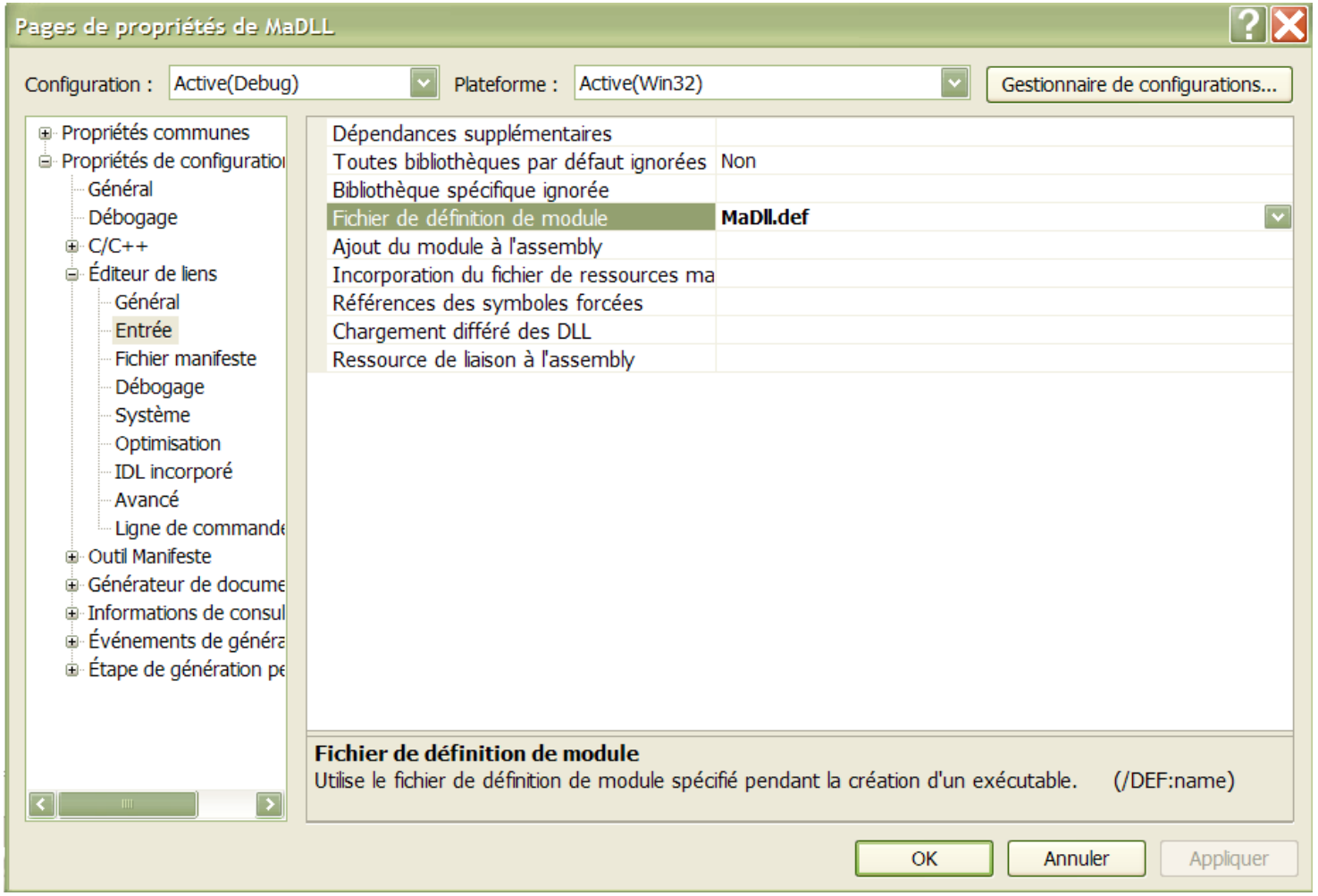
Exemple de DLL C++ (2/4)

- Fichier **`MaDll.cpp`:**

```
#include "MaDll.h"

double _stdcall carre (double& arg)
  {return arg*arg;}

double _stdcall addition (double& arg1, double& arg2)
  {return arg1+arg2;}

double _stdcall soustraction (double arg1, double arg2)
  {return arg1-arg2;}
```

Exemple de DLL C++ (3/4)

Bien spécifier `MaDll.def` dans la propriété *Fichier de Définition de Modules* de l'*Editeur de Liens* (obtenu par un clic droit de la souris sur le nom du projet)



Exemple de DLL C++ (4/4)

- Fichiers générés :
  - `Nom_Dll.dll` : fichier de la bibliothèque dynamique
  - `Nom_Dll.lib` : fichier permettant de faire la liaison avec la bibliothèque (i.e. pour qu'un programme puisse accéder aux fonctions de la bibliothèques)
- Possibilité de lier la bibliothèque au programme C++ l'utilisant :
  - De manière statique : déclaration explicite dans le programme (via `#include`) et résolution de liens effectuée par l'éditeur de lien au moment de la phase de compilation du programme
  - De manière dynamique : demande explicite du chargement d'une bibliothèque durant l'exécution du programme
- Sous linux : bibliothèque dynamique d'extension `.so`

Exemple programme C++ utilisant une DLL de manière dynamique

```
// Définition d'un type pointeur sur fonction
typedef double (_stdcall * importFunction)(double&,double&);

int main(void)

{   // Déclaration d'une variable de type pointeur sur fonction
    importFunction AddNumbers;

    double r=5,d=7,result;

    // Chargement de la DLL
    HINSTANCE hinstLib = LoadLibrary(TEXT("C:\\Chemin_d_acces\\MaDLL.dll"));

    // Si le chargement s'est mal passé!
    if (hinstLib == NULL) { cout << "ERROR: unable to load DLL\n";
       return 1;
       }

    // Récupération de la fonction de la librairie via le pointeur
    AddNumbers = (importFunction)GetProcAddress(hinstLib, "addition");

    // Si la récupération de la fonction s'est mal passée!
    if (AddNumbers == NULL)

    { cout << "ERROR: unable to find DLL function\n";
      FreeLibrary(hinstLib); // Libération de l'espace de chargement de la DLL
      return 1;
     }
    // Appel de la fonction
    result = AddNumbers(r,d);

    FreeLibrary(hinstLib); // Libération de l'espace de chargement de la DLL

    cout << result << endl;

}
```

Ne pas oublier d'inclure <iostream> et <windows.h> !!

Exemple programme C++ utilisant une DLL de manière statique (1/4)

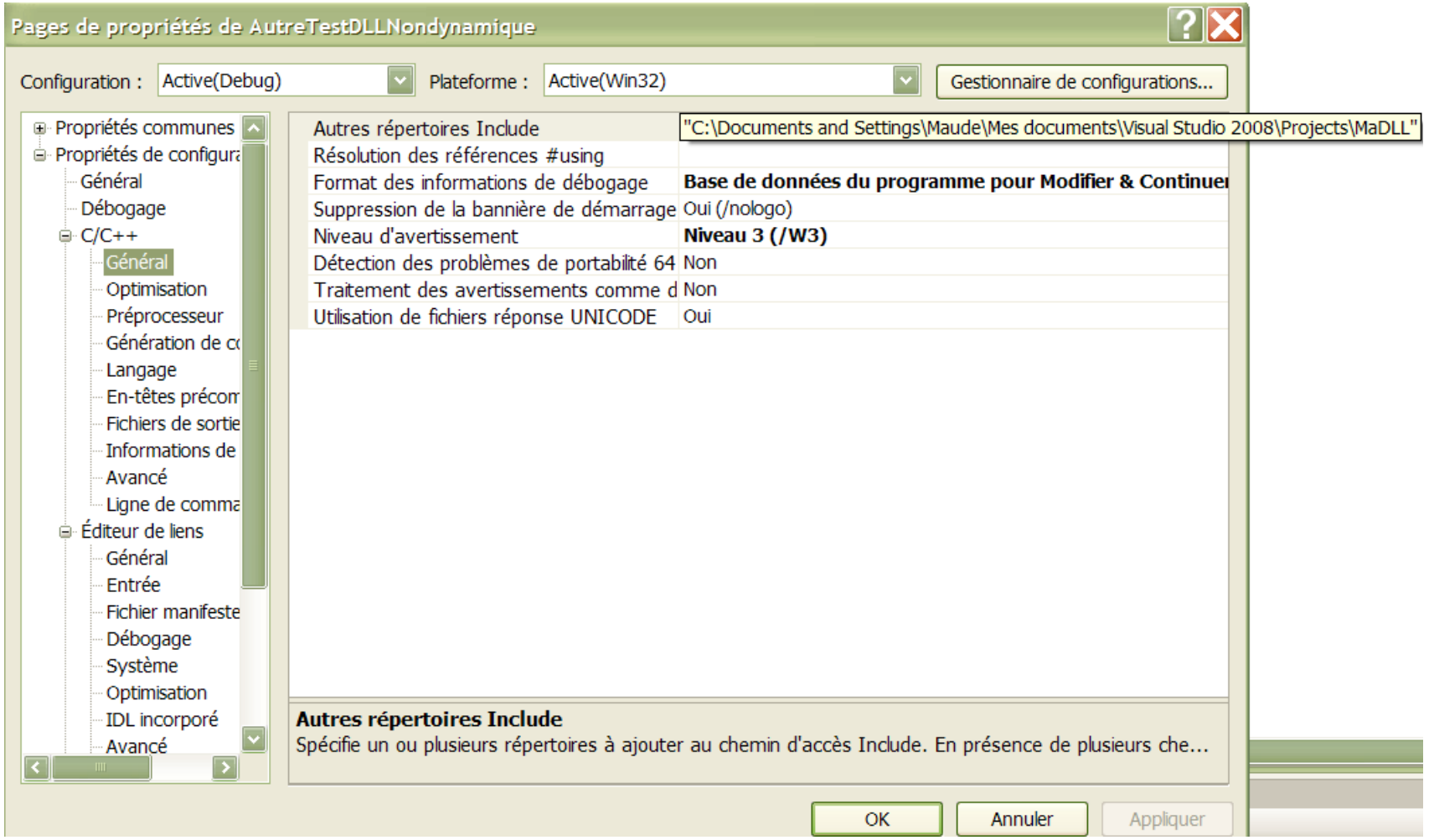
```
#include "MaDll.h"   // Inclusion du .h de la DLL
using namespace std;
int main()
{
     double r,d;
     cin >> r;
     cin >> d;
     cout <<"carre(" << r << ")=" << carre(r) << endl;
     cout <<"addition(" << r << ","<< d << ")=" << addition(r,d) << endl;
}
```

Pour que ça compile et que cela tourne, indiquer dans les propriétés du projet :

- Où trouver le `.h` de la bibliothèque (dans *C/C++/Général/Autres Répertoires Include* – avec **des guillemets)**
- Où trouver le **`.lib` de la bibliothèque (dans *Editeur de Liens/Entrée/Dépendances Supplémentaires*)**
- Où trouver le **`.dll` de la bibliothèque (dans *Débogage/Environnement* taper *`PATH=chemin_acces_au_fichier_dll`*)**

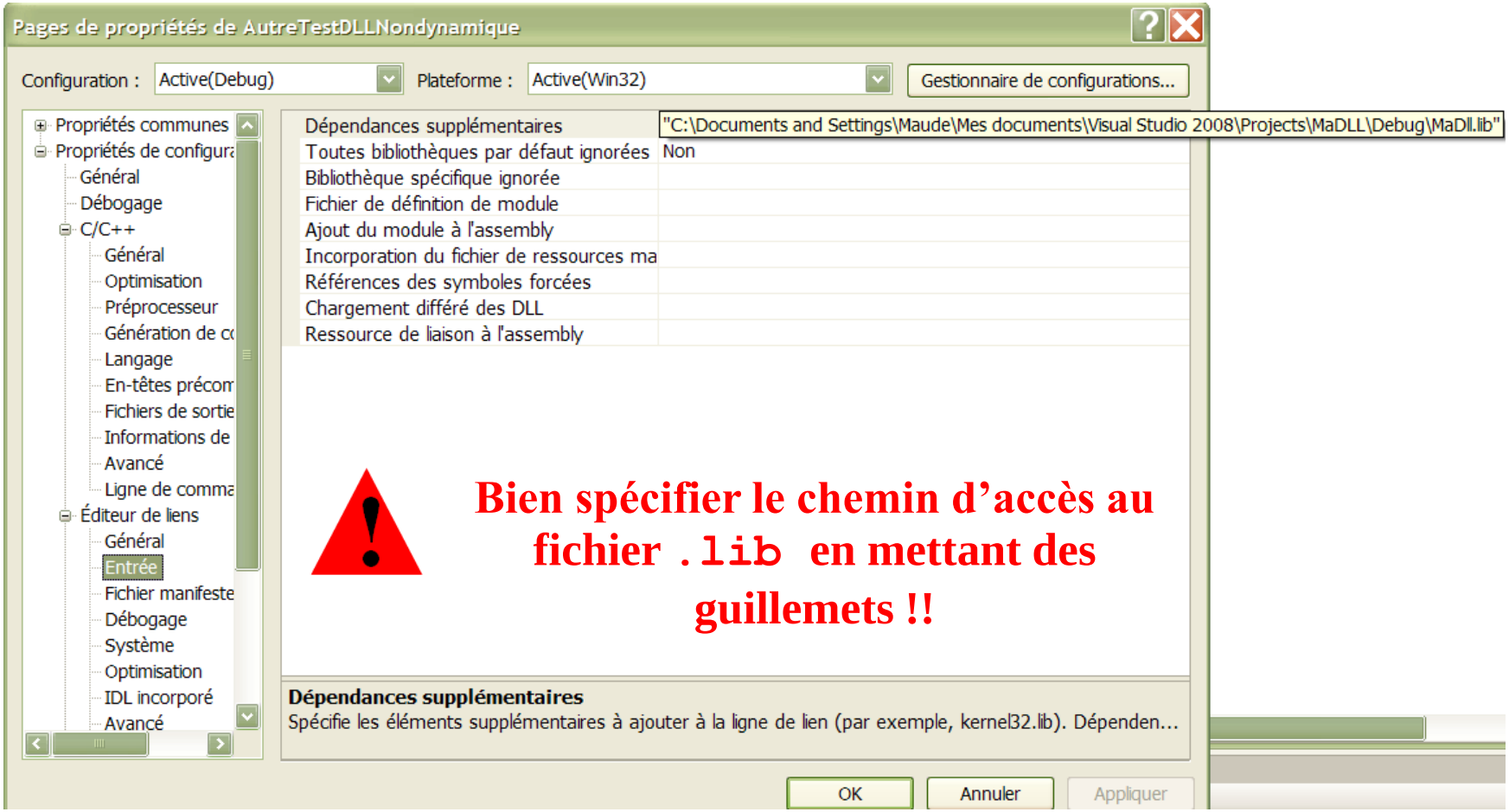
Exemple programme C++ utilisant une DLL de manière statique (2/4)

- Où trouver le `.h` de la bibliothèque :



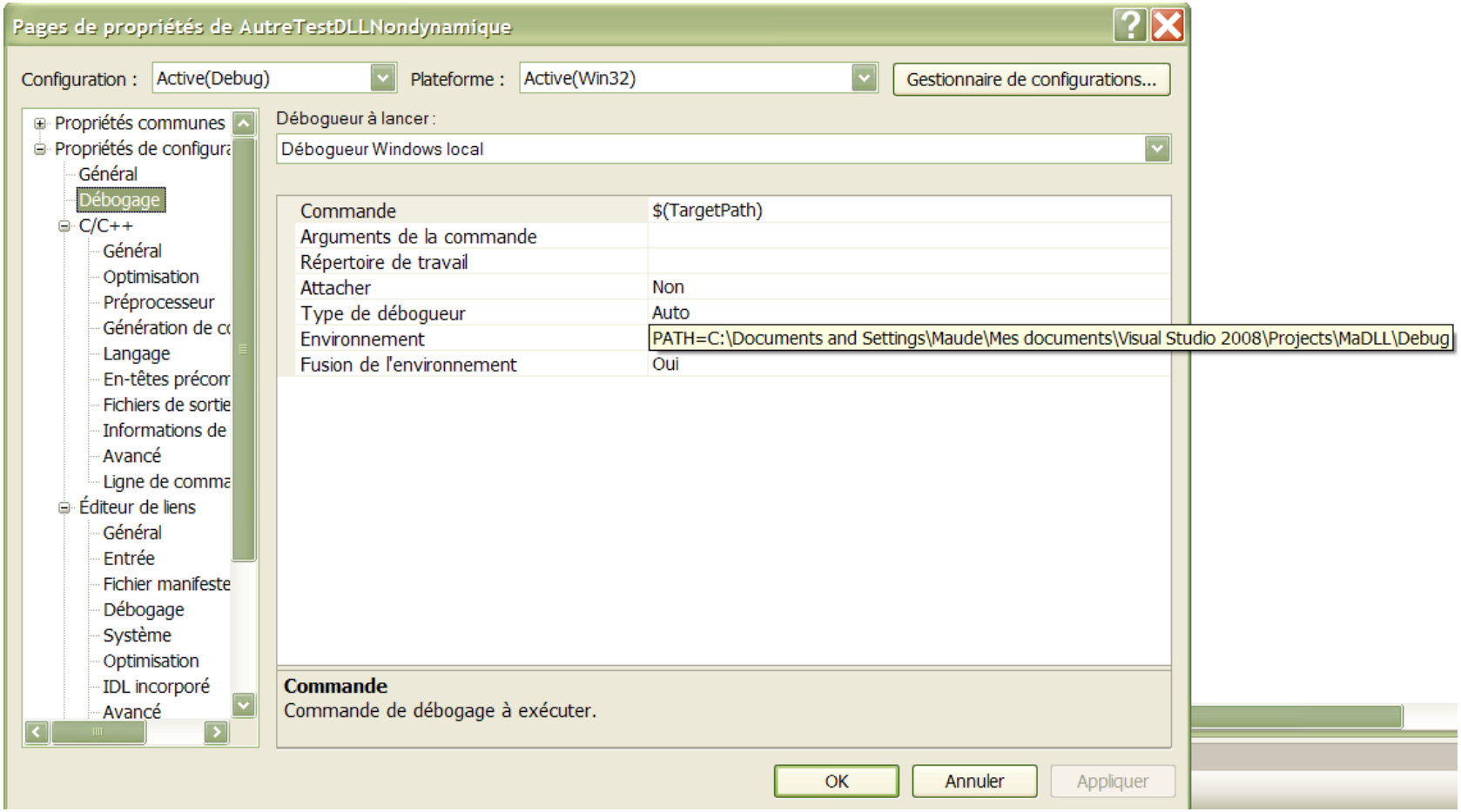
Exemple programme C++ utilisant une DLL de manière statique (3/4)

- Où trouver le `.lib` de la bibliothèque :



Exemple programme C++ utilisant une DLL de manière statique (4/4)

- Où trouver le `.dll` de la bibliothèque :



Exemple de DLL C++ contenant une classe (1/2)

- Fichier **MaDll.h:**

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

class ClasseDynamique
{
  public:
    _declspec(dllexport) ClasseDynamique();
    _declspec(dllexport) ~ClasseDynamique();
     void _declspec(dllexport) SetAttribut(int a);
    int _declspec(dllexport) GetAttribut();
    void _declspec(dllexport) Afficher();
  private:
    int attribut;
};
```

Classe C++ pouvant être utilisée dans un programme C++ liée à la DLL de manière statique ou dynamique

extern "C" _declspec(dllexport) pour les méthodes de la classe devant être exportées

Exemple de DLL C++ contenant une classe (2/2)

- Fichier **`MaDll.cpp`:**

```
#include "MaDll.h"
ClasseDynamique::ClasseDynamique() { attribut=0; }

ClasseDynamique::~ClasseDynamique() {}

int ClasseDynamique::GetAttribut() { return attribut; }

void ClasseDynamique::SetAttribut(int a) { attribut=a; }

void ClasseDynamique::Afficher()
  { cout << "attribut=" << attribut << endl; }
```

- Dans le programme utilisant la DLL (de manière statique par exemple) :

La DLL peut être liée de manière statique ou dynamique

```
#include "MaDll.h"
int main()
{   ClasseDynamique o;
    o.SetAttribut(5);
    o.Afficher();
}
```