

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

Bibliographie

- *Le Langage C++ (The C++ Programming Language)*, de Bjarne Stroustrup, Addison-Wesley – dernière version en 2005 - prochaine version Avril 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++*, 6ème Édition de Claude Delannoy, Eyrolles, 2004
- *Exercices en langage C++*, de Claude Delannoy, Eyrolles, 1999

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/>
- + voir sur <http://www.infotheque.info/> et <http://www.freetechbooks.com/>
- <http://www.stroustrup.com/C++.html>
- ***Aide en ligne* : <http://www.cplusplus.com/>**

Historique du Langage 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://www2.research.att.com/~bs/C++0xFAQ.html>
 - <http://www.softwarequalityconnection.com/2011/06/the-biggest-changes-in-c11-and-why-you-should-care/>
 - 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] :

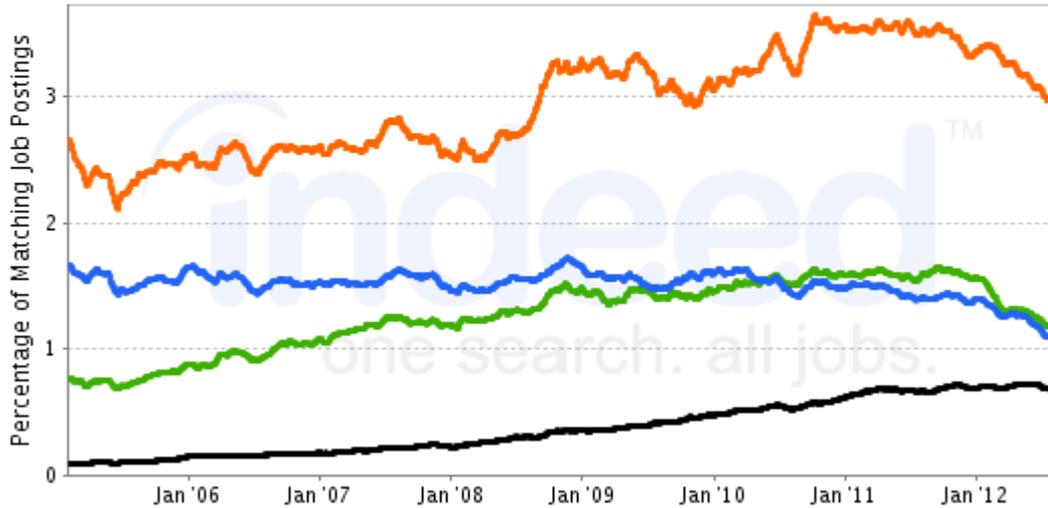
- Java
 - Création des objets par allocation dynamique (*new*)
 - Accès aux objets par références
 - Destruction automatique des objets par le ramasse miettes
- C++
 - Allocation des objets en mémoire statique (variables globales), dans la **pile** (variables automatiques) ou dans le **tas** (allocation 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

Scale: **Absolute** - [Relative](#)

Job Trends from Indeed.com

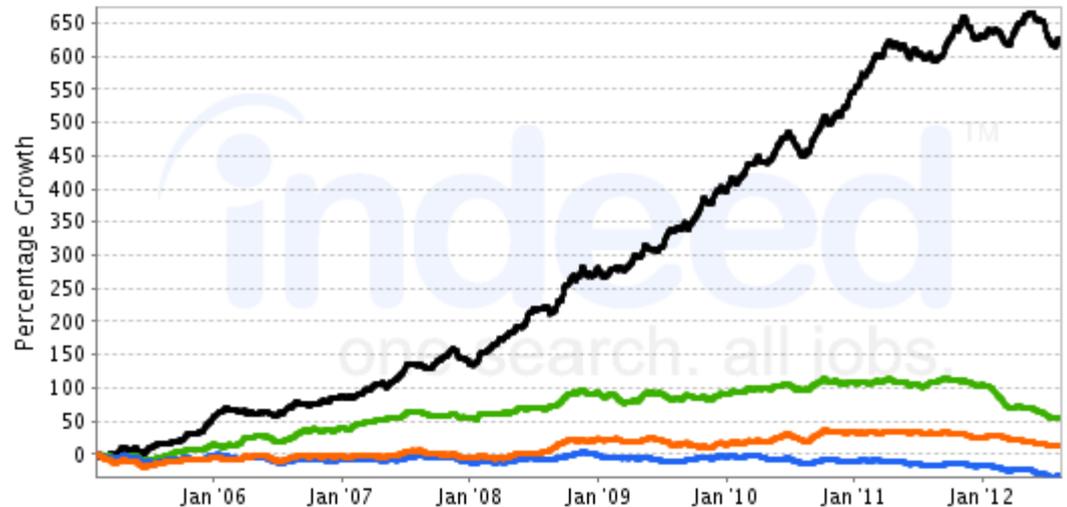
— Java — C++ — C# — Python



Scale: [Absolute](#) - **Relative**

Job Trends from Indeed.com

— Java — C++ — C# — Python



Rappel : Pile et Tas (1/6)

*Mémoire Allouée
Dynamiquement*

Tas

*Mémoire allouée de
manière statique*

Variables globales

Arguments

Valeurs de retour

Pile

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)

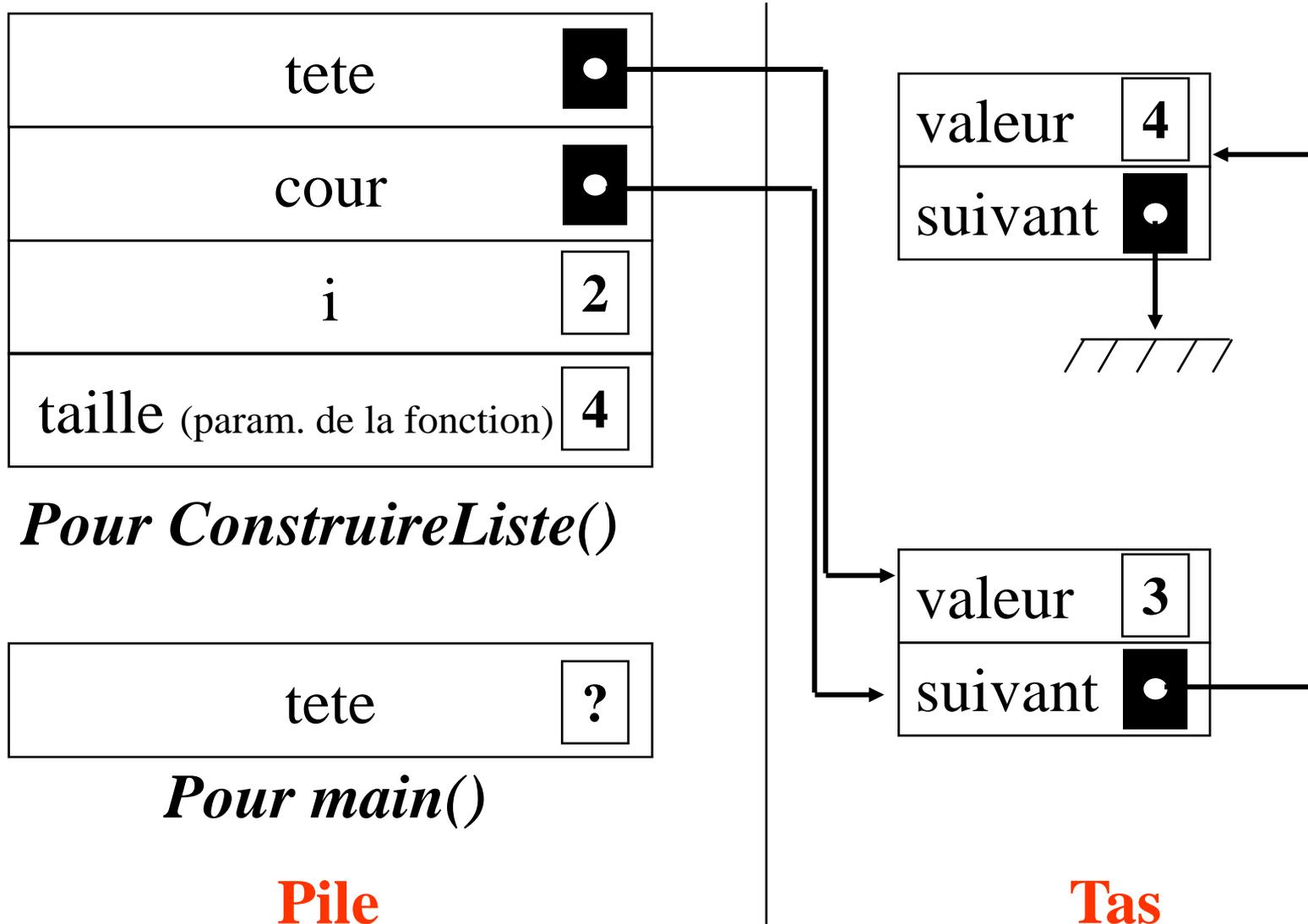
Exemple de programme en C :

```
#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;
}
```

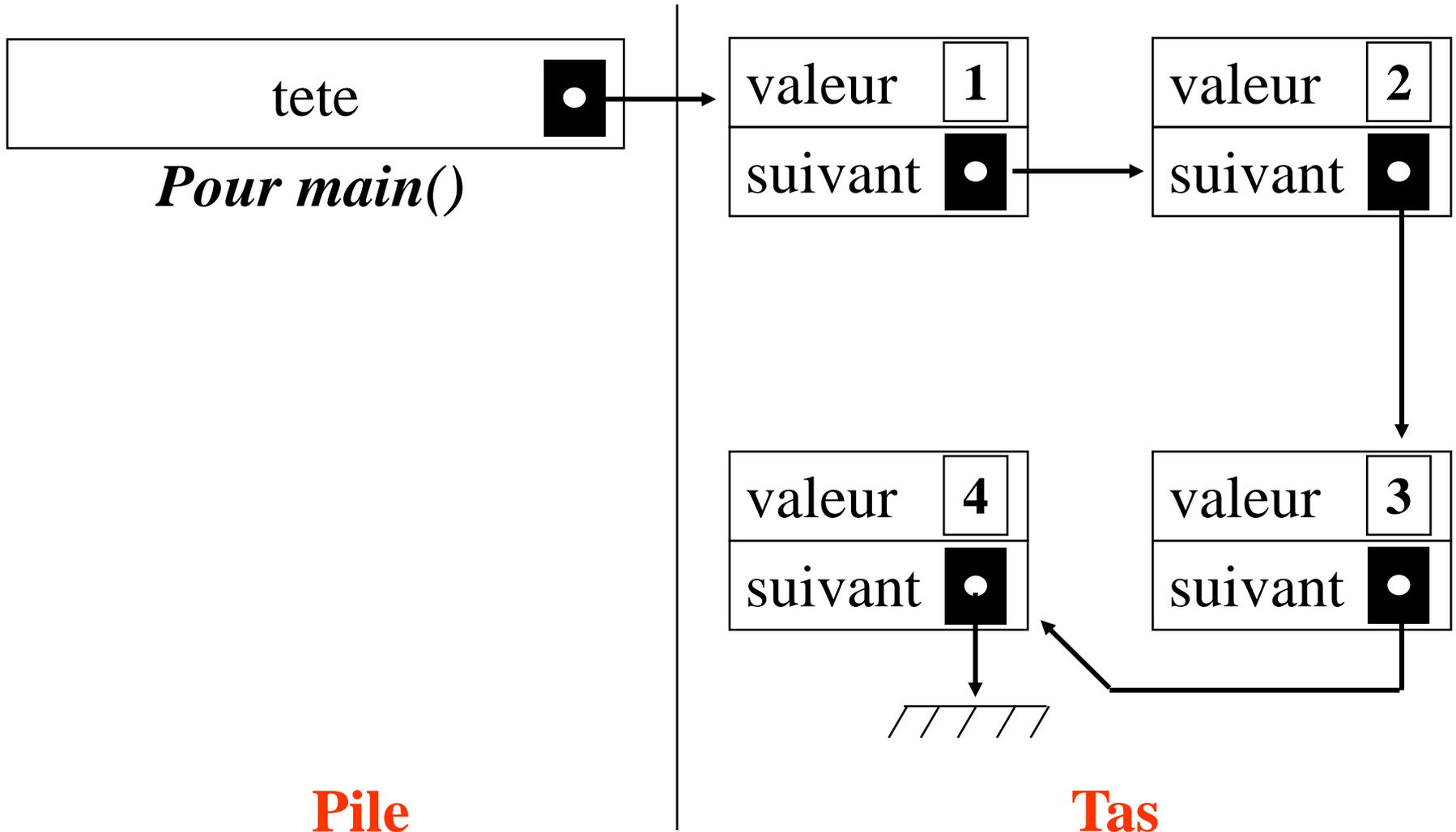
Rappel : Pile et Tas (5/6)

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



Pile

Tas

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** << **expr₁** << ... << **expr_n**
 - Instruction affichant *expr₁* puis *expr₂*, 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** >> **var₁** >> ... >> **var_n**
 - Instruction affectant aux variables *var₁*, *var₂*, 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 :

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

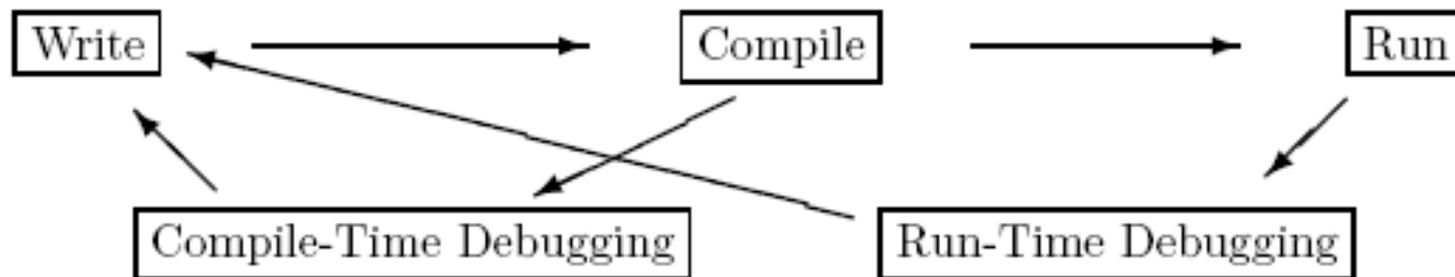
■ Compilateurs propriétaires :

- **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 bibliothèques 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` *:: opérateur de résolution de portée*
`std::endl`
- 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 booléenne
int i = 0; // entier
long j = 123456789; // entier long
float f = 3.1; // réel
// réel à double précision
double pi = 3.141592653589793238462643;
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/1 octets

- **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)

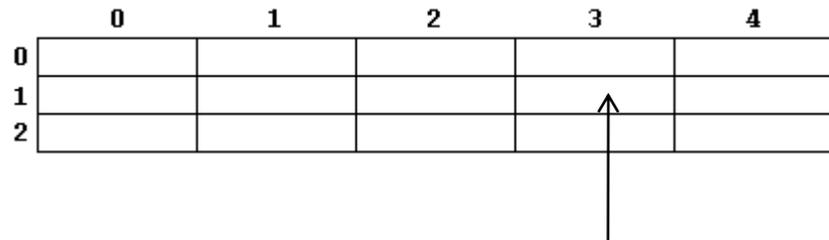
`\b` (*backspace*)

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/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

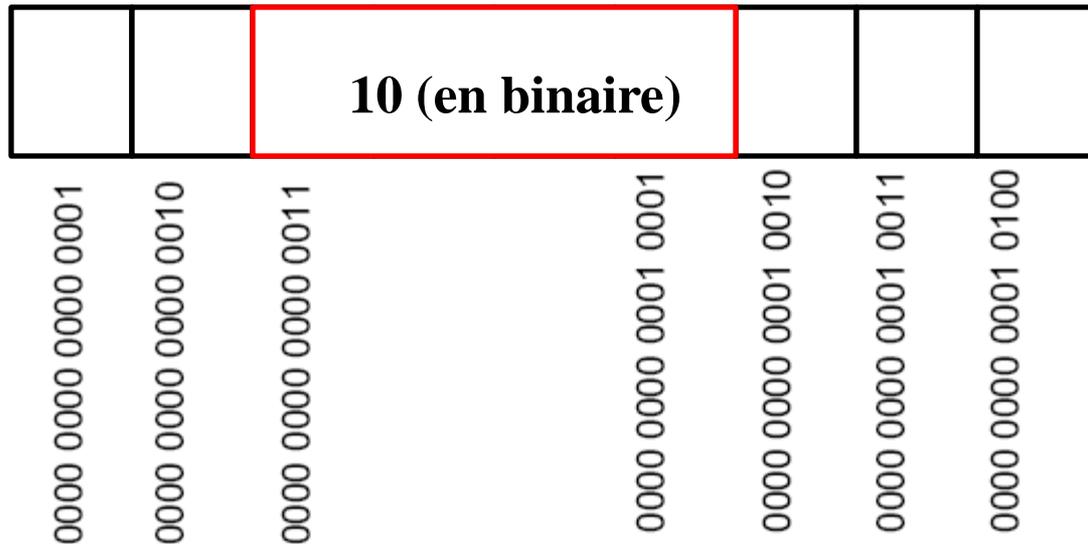
```
int i = 5;  
int & j = i; // j reçoit i  
int k = j; // k reçoit 5  
j += 2; // i reçoit i+2 (=7)  
j = k; // i reçoit k (=5)
```



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



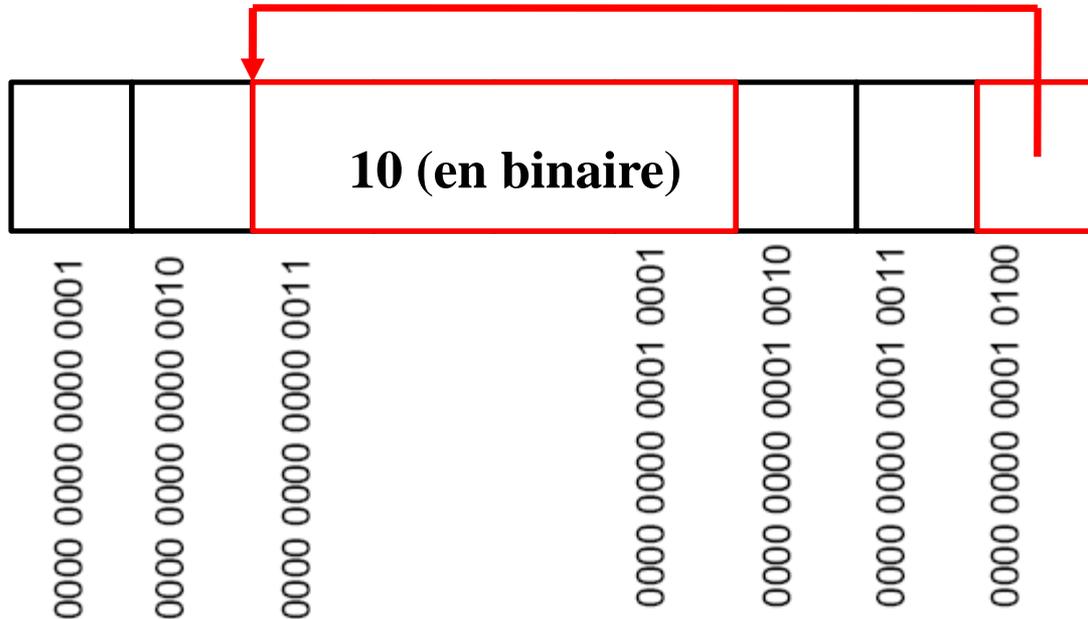
```
int i=10 ;
```

1. Réservation d'une zone mémoire de 4 octets (la 1^{ère} 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 souhaitée (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 repositionné au début de la zone mémoire
// allouée par new
```

■ 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;

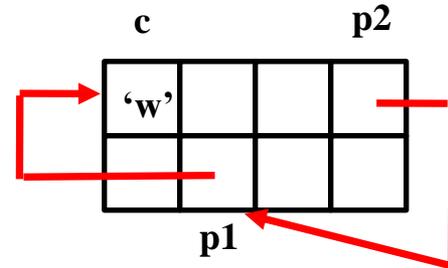
(*tab)++; // => tab[1]=4

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

```
#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



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

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 pas 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 \"<string>\"<< MaChaine`
`<< "\" a pour taille \" << MaChaine.size() << \".\"`
`<< endl;`
`string AutreChaine("!!");`
`cout << "Concaténation des deux chaines : \"<string>\"`
`<< MaChaine + AutreChaine<<\"<string>\".<< << 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
```

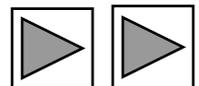
```
    void SetHoraire(int, int, int);
```

```
    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 \Rightarrow 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 ;
```



Attention au nom des constantes

```
// 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();
```

```
}
```



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 copie  
  return 0;  
}
```

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

```
** 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 copie
      Exemple(const Exemple &);
      ~Exemple();
    };
};

// Définition du constructeur par copie
Exemple::Exemple (const Exemple & e)
{ cout << "** Appel du constructeur par copie ";
  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 copie :

```
** Appel du constructeur - valeur de l'attribut = 1
** Appel du constructeur par copie - 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 \Rightarrow 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 copie
}
```

- 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 copie
}
```

- 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 \Leftrightarrow **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) {...}
```