

Langage C++

J. B. OTHMAN

Jbo@univ-paris13.fr

Le langage C Rappels

■

Introduction

- Langage Compilé

Compilateur

Texte
Ascii



Programme Exécutable

- Langage de bas niveau permettant le développement d'os, de prgmes applicatifs scientifiques et de gestion
- Langage Structuré

Introduction

- Langage évolué qui permet d'effectuer des opérations de bas niveau (assembleur, primitives systèmes)
- Portabilité (en respectant la norme ANSI) due à l'emploi de biblios stdards dans lesquelles sont régulées les fonctionalités liées à la machines
- Grande efficacité et puissnce
- Vitesse des prgs maximales

Histoire

- Création du langage en 1970 par Dennie Ritchie au lab. Bell d'AT&T
- C'est l'aboutissement de deux langages :
 - BPCL (1967 par Martin Richards)
 - B (en 1970 par Ken Thompson)
- 1ère publication en 1978 par Brian Kernighan et Dennie Ritchie « The C programming Langage »
- Un travail de normalisation établi par l'ANSI (norme X3J11)
 - Publication de « The C programming language 2ème édition »

Structure d'un Programme C

- Constitué par un ensemble de procédures et fonctions
- La définition des fonctions ou procédures peuvent être définies dans un ordre qcq
- La syntaxe (prototype) d'une fonction est la suivante :

[Type] nom de la fonction ([type et nom des paramètres])

{ /*début de la procédure*/
 corps de la procédure

J. BEN-OTHMAN }/*fin de la procédure*/

Structure d'un Programme C

- Type désigne le type de la valeur renvoyée par la fonction
- Nom est l'identifiant de la fonction
- Corps est le code composé de :
 - L'ensemble des déclarations des variables
 - L'ensemble des instructions

Fonctions particulières

- Tous programmes C contient une fonction particulière :
 - La fonction *main* correspondant à l'adresse en mémoire de laquelle commence l'exécution du programme
- Exemple de programme C :

```
#include <stdio.h> /* bibliothèque d'entrées-sorties standard */
void main()
{
    puts("BONJOUR"); /* utilisation d'une fonction-bibliothèque */
}
```

Ecriture d'un programme C

- Éditer un fichier contenant le programme (on met en général une extension .c),
- Écrire le programme et le sauvegarder
- Compiler le fichier source :
 - Cc –o (comme output) [le nom de l'exécutable] [prg.c]
- Exécuter le programme (output)
- Le langage C distingue les minuscules, des majuscules. Les mots réservés du langage C doivent être écrits **en minuscules**.

Commentaires

- Un prg en C peut contenir des commentaires entre /* et */
- #include est une directive du prépocesseur (ou précompilateur)
- Include fait appel à une bibliothèque contenant des fonctions
- Stdio.h est la bibliothèque des entrées/ sorties
- Le point virgule sépare deux instructions

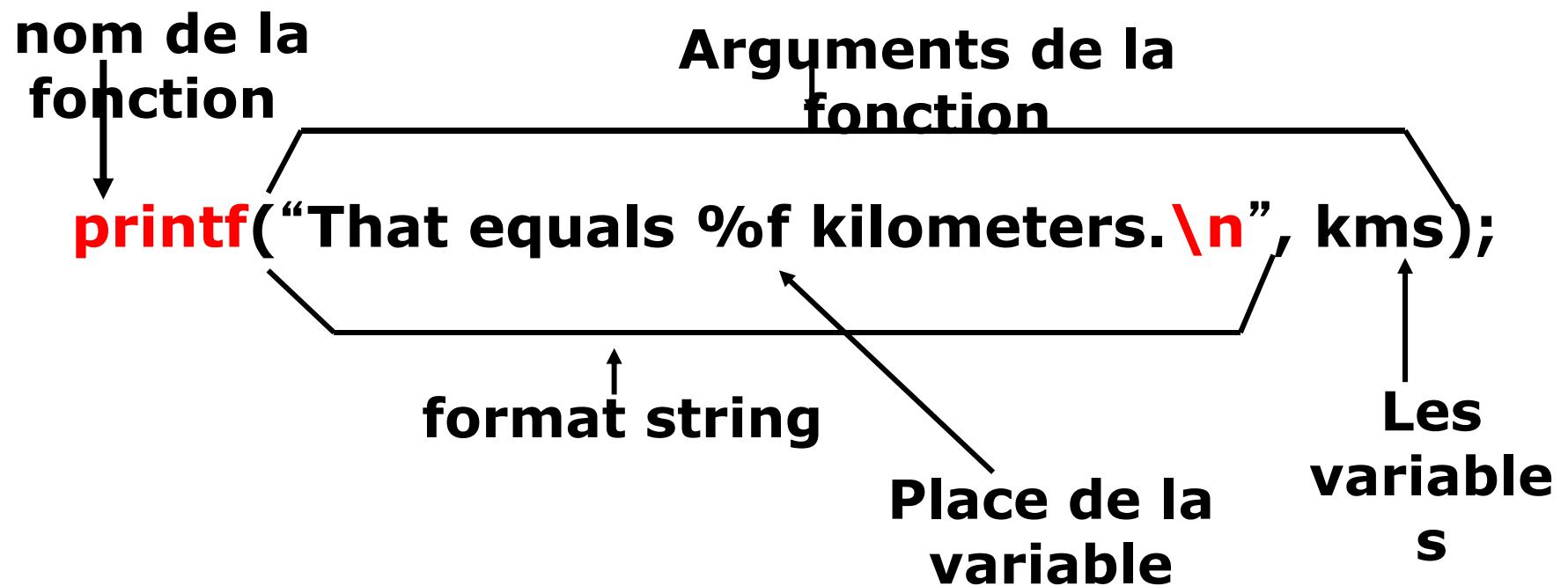
Printf et scanf

- Les opérations Input/Output sont faites par les Fonctions
 - **printf**
 - **scanf**

Input/Output

- **Input operation** – Transfer des données de l’extérieur à l’ordinateur
- **Output operation** – Affichage
- **Input/output functions** -
 - `printf` = output function
 - `scanf` = input function

La fonction printf



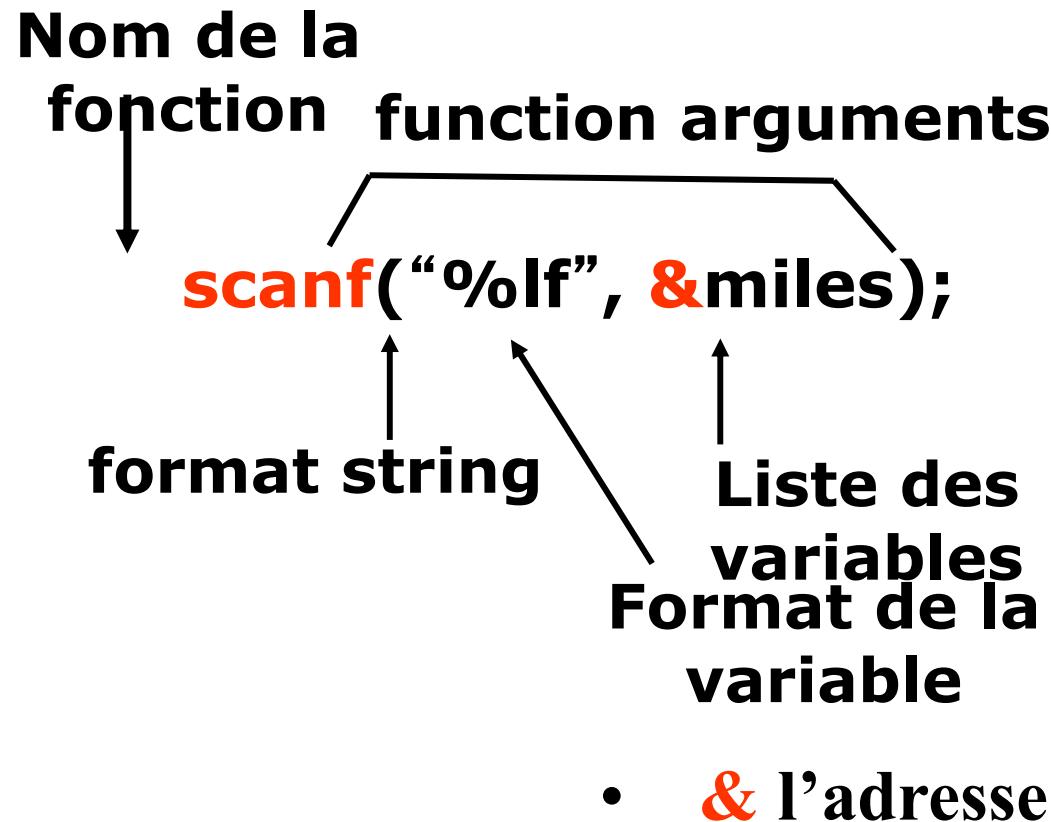
Placeholders

- Place des variables commence toujours par le symbol %

Format d'affichage	Type de variable	Function Utilisant
%c	char	printf / scanf
%d	int	printf / scanf
%f	double	printf
%lf	double	scanf

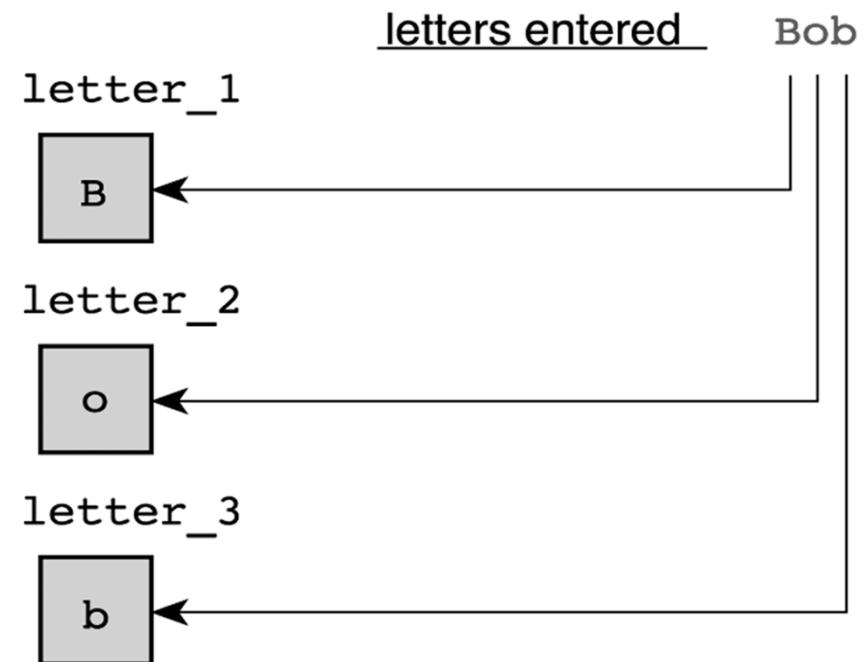
- Retour à la ligne est représenté par – ‘\n’

La Fonction scanf



Exemple de scanf

```
char letter_1, letter_2, letter_3;  
...  
scanf ("%c%c%c", &letter_1, &letter_2, &letter_3);
```



LES DIFFERENTS TYPES DE VARIABLES

Le langage C distingue plusieurs types d'entiers:

TYPE	DESCRIPTION	TAILLE MEMOIRE
int	entier standard signé	4 octets: $-2^{31} \leq n \leq 2^{31}-1$
unsigned int	entier positif	4 octets: $0 \leq n < 2^{32}$
short	entier court signé	2 octets: $-2^{15} \leq n \leq 2^{15}-1$
unsigned short	entier court non signé	2 octets: $0 \leq n < 2^{16}$
char	caractère signé	1 octet : $-2^7 \leq n \leq 2^7-1$
unsigned char	caractère non signé	1 octet : $0 \leq n < 2^8$

Déclaration d'une variable :

<type> identificateur (éventuellement une affectation);

Expressions logiques

- Le type booléen n'existe pas. Le résultat vaut 1 si elle est vrai 0 sinon
- Réciproquement toutes valeurs non nulles est considérée comme fausse
- Les opérateurs logiques sont :
 - Inférieur ‘<’
 - Inférieur ou égal ‘<=’
 - Supérieur ‘>’
 - Supérieur ou égal ‘>=’
 - Opération de négation ‘!’
 - Comparaison identique (‘==’) différents (‘!=’)
 - Opérateurs de conjonction et logique (‘&&’) ou logique (‘||’)

Résultat des expressions

- `!expr1`
 - est vrai si `expr1` est faux (0)
 - Est faux si `expr1` est vrai (1)
- `expr1 && expr2` est vrai si les deux expressions sont vraies sinon faux
- `expr1 || expr2` est vraie si l'une des deux expressions au moins est vrai, faux sinon

Opération sur les bits/variables

- Sur les bits :
 - `~` est le nom logique
 - `&` est le et logique
 - `|` est le ou logique
 - `^` est le ou exclusif
 - `>>` opérateur de décalage à droite
 - `<<` opérateur de décalage à gauche
- Sur les variables :
 - `Var --` évalue var et décrémente var
 - `--Var` décrémente var et évalue var
 - `Var ++` évalue var et incrémente var
 - `++Var` incrémente var et évalue var

Sémantiques des instructions

- Si une instruction
 Instruction (condition)
 instruction
- Si plusieurs instructions :
 Instruction (condition)
 {
 instruction1;
 instruction2;
 .
 .
 instruction n;
 }

if

- if (condition)
 - action 1; /*si condition vrai*/
 - else
 - action 2; /*si condition fausse*/

Rq 1 écriture possible (comme un p.... !)

Condition ? Expression 1 : expression 2

Rq 2 l'action 2 peut être aussi un autre if

Rq 3 si on a trop de if imbriqué on peut utiliser une autre fonction switch

switch

switch (condition)

{

case étiquette 1 : actions; break;

case étiquette 2 : actions; break;

.

.

.

default : actions;

}

Les Boucles

- Il existe deux types de boucles :
 - Boucle itérative
 - Si on connaît le nombre de passage
 - On spécifie le nombre de passage dans la boucle
 - Boucle conditionnelle
 - Si on ne connaît pas le nombre de passage dans la boucle
 - La condition arrête la boucle
- Rq en c on peut utiliser les deux boucles qqs le type de traitement !

Boucle itérative

- `for (condition de départ; condition d'arrêt; incrément)`
 - Condition de départ est évaluée 1 seule fois
 - La condition 2 est évalué avant chaque passage dans la boucle :
 - Si la condition est vrai il n'y a pas de passage dans la boucle
 - Sinon on fait un passage dans la boucle
 - La condition 3 est évaluée à chaque passage

Boucle conditionnelle

- Avec évaluation au départ :
 - `while (condition)`
 {
 instructions;
 }
 le corps de la boucle est exécutée si la condition est vraie
- Avec évaluation à la fin :
 - `do`
 {
 instructions;
 }
 `while (condition)`
 le corps de la boucle est exécutée si la condition est vraie

Le préprocesseur

- Il effectue un pré traitement du prg source avant qu'il soit compilé
- Le préprocesseur exécute des instructions particulières appelées directives
- Ces directives sont identifiées par le caractère # en première colonne
- `#define` permet la définition de pseudo constante
 - # define identificateur chaîne de substitution : le préprocesseur remplace tous les identificateurs par la chaîne de caractère
- `#include` permet d'insérer le contenu d'un fichier dans un autre (ils sont suffixés par .h)
 - `#define <nom de fichier>` si le fichier est /usr/include
 - `#define « nom de fichier »` dans un répertoire particulier

Les tableaux

- Un tableau est une structure permettant de stocker plusieurs valeurs référencées par une seule étiquette
- Type identificateur [taille] pour une dimension
- Type identificateur [taille] [taille] pour deux dimensions

Rappels

```
const int MAXI_AGE = 134;

int age;
char sexe;

/* saisir et valider le sexe */
do {
    printf("Entrez un caractere parmi f, F, m ou M :");
    fflush(stdin); /* vider la memoire temporaire */
    scanf("%c", &sexe);

    if (sexe != 'f' && sexe != 'F' &&
        sexe != 'm' && sexe != 'M')
        printf("Erreur! Retapez S.V.P. \n");
} while (sexe != 'f' && sexe != 'F' &&
         sexe != 'm' && sexe != 'M');

/* saisir et valider l'age */
do {
    printf("Entrez l'age entre 1 et %d : ", MAXI_AGE);
    scanf("%d", &age);

    if (age < 1 || age > MAXI_AGE)
        printf("Erreur! Retapez S.V.P. \n");
} while (age < 1 || age > MAXI_AGE);
```

```
#include <stdio.h>
```

```
int main()
{
    int n;

    /* Saisie et validation */
    do {
        printf("Entrez un entier positif : ");
        scanf("%d", &n);

        if (n < 0)
            printf("n = %d est negatif\n", n);
    } while (n < 0);

    printf("L'entier lu : %d\n", n);

    printf("A l'envers : ");

    do {
        printf("%d", n % 10);
        n = n / 10;
    } while (n > 0);

    return 0;
}
```

$$S = 1 + 2 + 3 + \dots + (n - 1) + n = \sum_{i=1}^n i = \frac{n(n + 1)}{2}.$$

• □ □ □

Start here exorevision.c

```
1 #include <stdio.h>
2
3
4 int main()
5 {
6     int i,j,somme;
7
8     do
9     {
10         printf("introduire la valeur de n \n\n");
11         scanf("%d",&i);
12     }
13     while (i<0);
14     somme=0;
15     for (j=1; j<=i;j=j+1)
16         somme=somme+j;
17
18     printf ("la valeur théorique est : %d la valeur calculée est :%d\n", (i*(i+1))/2, somme);
19 }
20
21
```

```
int k = 1, j = 10;
```

```
do {
    if (k >= 3)
        j++;
    else
        j--;
    k++;
    printf("%3d%3d\n", j, k);
} while (j < 10);

printf("%5d\n", j * k);
printf("FIN");
```

Solution :
(On utilise le symbole ^ pour représenter une espace)

j	k	Affichage à l'écran
	10	1
9	2	^9^2
8	3	^8^3
9	4	^9^4
10	5	10^5
		^50
		FIN

Les pointeurs

Modes d'adressage de variables. Définition d'un pointeur.
Opérateurs de base. Opérations élémentaires. Pointeurs et tableaux.
Pointeurs et chaînes de caractères. Pointeurs et enregistrements.
Tableaux de pointeurs. Allocation dynamique de la mémoire.
Libération de l'espace mémoire.

L'importance des pointeurs

- On peut accéder aux données en mémoire à l'aide de pointeurs i.e. des variables pouvant contenir des adresses d'autres variables.
- Comme nous le verrons dans le chapitre suivant, en C, les pointeurs jouent un rôle primordial dans la définition de fonctions :

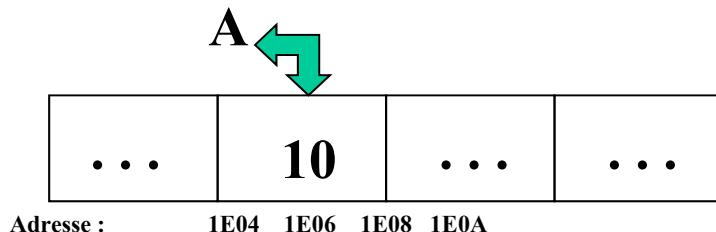
Les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions.
- Le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.
- Les pointeurs nous permettent de définir de nouveaux types de données : les piles, les files, les listes,
- Les pointeurs nous permettent d'écrire des programmes plus compacts et plus efficaces.
- Mais si l'on n'y prend pas garde, les pointeurs sont une excellente technique permettant de formuler des programmes incompréhensibles.

Mode d'adressage direct des variables

Adressage direct :

- Jusqu'à maintenant, nous avons surtout utilisé des variables pour stocker des informations.
- La valeur d'une variable se trouve à un endroit spécifique dans la mémoire de l'ordinateur.

`short A;
A = 10;`



- Le nom de la variable nous permet alors d'accéder directement à cette valeur.

Dans l'adressage direct, l'accès au contenu d'une variable se fait via le nom de la variable.

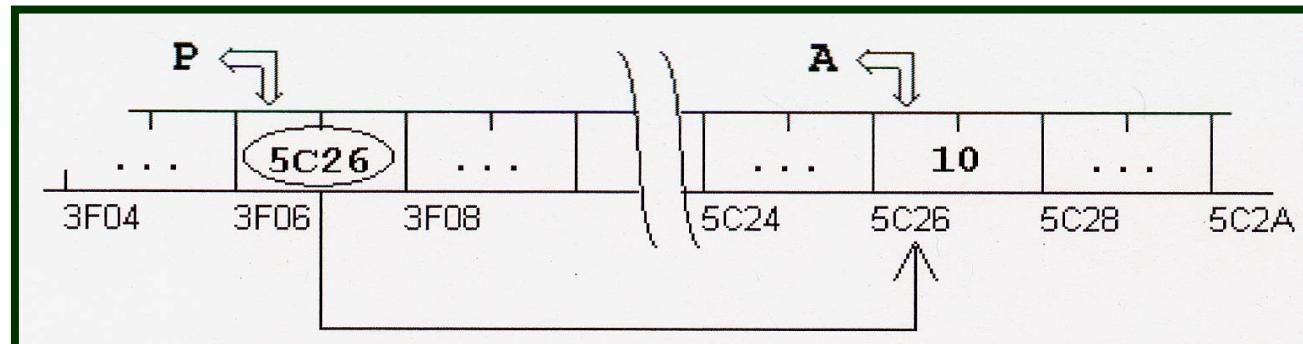
Mode d'adressage indirect des variables

Adressage indirect :

- Si nous ne voulons ou ne pouvons pas utiliser le nom d'une variable A, nous pouvons copier l'adresse de cette variable dans une variable spéciale, disons P, appelée pointeur.
- Nous pouvons alors retrouver l'information de la variable A en passant par le pointeur P.

Dans l'adressage indirect, l'accès au contenu d'une variable se fait via un pointeur qui renferme l'adresse de la variable.

Exemple : Soit A une variable renfermant la valeur 10, et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit :



Définition d'un pointeur

Un pointeur est une variable spéciale pouvant contenir l'adresse d'une autre variable.

- En C, chaque pointeur est limité à un type de données. Il ne peut contenir que l'adresse d'une variable de ce type. Cela élimine plusieurs sources d'erreurs.

Syntaxe permettant de déclarer un pointeur :

type de donnée * identificateur de variable pointeur;

Ex. : int * pNombre;

pNombre désigne une variable pointeur pouvant contenir uniquement l'adresse d'une variable de type int.

Si pNombre contient l'adresse d'une variable entière A,
on dira alors que pNombre pointe vers A.

- Les pointeurs et les noms de variables ont le même rôle : ils donnent accès à un emplacement en mémoire.

Par contre, un pointeur peut contenir différentes adresses mais le nom d'une variable (pointeur ou non) reste toujours lié à la même adresse.

- Bonne pratique de programmation : choisir des noms de variable appropriés (Ex. : pNombre, NombrePtr).

Comment obtenir l'adresse d'une variable ?

- Pour obtenir l'adresse d'une variable, on utilise l'opérateur & précédant le nom de la variable.

Syntaxe permettant d'obtenir l'adresse d'une variable :

& nom de la variable

Ex. : int A;
int * pNombre = &A;

ou encore,
int A;
int * pNombre;
pNombre = &A;

pNombre désigne une variable pointeur initialisée
à l'adresse de la variable A de type int.

Ex. : int N;
printf("Entrez un nombre entier : ");
scanf("%d", &N);



scanf a besoin de l'adresse de chaque paramètre
pour pouvoir lui attribuer une nouvelle valeur.

Note :

L'opérateur & ne peut pas être appliqué à des constantes ou des expressions.

Comment accéder au contenu d'une adresse ?

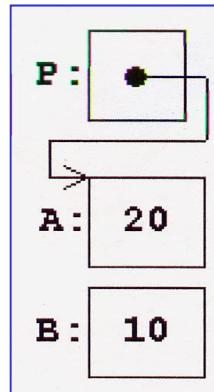
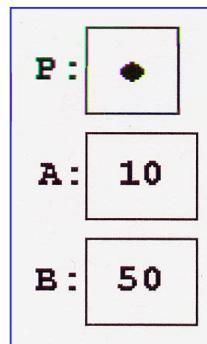
- Pour avoir accès au contenu d'une adresse, on utilise l'opérateur ***** précédant le nom du pointeur.

Syntaxe permettant d'avoir accès au contenu d'une adresse :

*** nom du pointeur**

Ex. : **int A = 10, B = 50;**
 int * P;

P = &A;
B = *P;
***P = 20;**



***P et A désignent le même emplacement mémoire et *P peut être utilisé partout où on peut écrire A (ex. : cin >> *P;).**

Priorité des opérateurs * et &

- Ces 2 opérateurs ont la même priorité que les autres opérateurs unaires (!, ++, --).
- Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

Après l'instruction

`P = &X;`

les expressions suivantes, sont équivalentes:

`Y = *P+1` \Leftrightarrow `Y = X+1`
`*P = *P+10` \Leftrightarrow `X = X+10`

`*P += 2` \Leftrightarrow `X += 2`
`++*P` \Leftrightarrow `++X`
`(*P)++` \Leftrightarrow `X++`

Parenthèses

obligatoires sans quoi, cela donne lieu à un accès non autorisé.

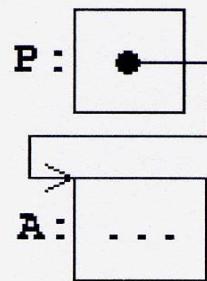
Le pointeur **NULL**

- Pour indiquer qu'un pointeur pointe nulle part, on utilise l'identificateur **NULL**
(On doit inclure stdio.h ou iostream.h).
- On peut aussi utiliser la valeur numérique **0** (zéro).

```
int * P = 0;
```

```
if (P == NULL) printf("P pointe nulle part");
```

En résumé ...



Après les instructions:

```
int A;  
int *P;  
P = &A;
```

A désigne le contenu de A

&A désigne l'adresse de A

P désigne l'adresse de A

***P** désigne le contenu de A

En outre:

&P désigne l'adresse du pointeur P

***A** est illégal (puisque A n'est pas un pointeur)

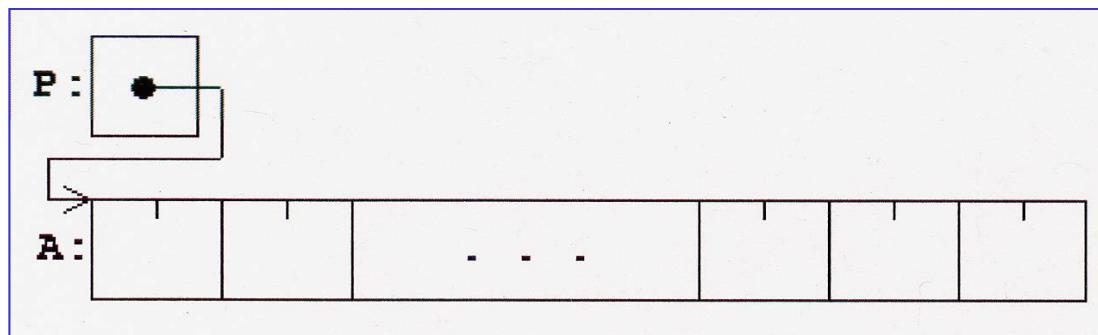
$$A == *P \Leftrightarrow P == \&A$$

$$A == *\&A \text{ et } P == \&*P$$

Pointeurs et tableaux

- Chaque opération avec des indices de tableaux peut aussi être exprimée à l'aide de pointeurs.
- Comme nous l'avons déjà constaté, le nom d'un tableau représente l'adresse de la première composante.
&tableau[0] et **tableau** sont une seule et même adresse.
- Le nom d'un tableau est un pointeur constant sur le premier élément du tableau.

```
int A[10];
int * P;
P = A;  est équivalente à      P = &A[0];
```



Adressage des composantes d'un tableau

- Si P pointe sur une composante quelconque d'un tableau, alors $P + 1$ pointe sur la composante suivante.

$P + i$ pointe sur la $i^{\text{ième}}$ composante à droite de $*P$.

$P - i$ pointe sur la $i^{\text{ième}}$ composante à gauche de $*P$.

Ainsi, après l'instruction $P = A;$

$* (P+1)$ désigne le contenu de $A[1]$

$* (P+2)$ désigne le contenu de $A[2]$

... ...

$* (P+i)$ désigne le contenu de $A[i]$

- Incrémantation et décrémentation d'un pointeur

Si P pointe sur l'élément $A[i]$ d'un tableau, alors après l'instruction

$P++;$ P pointe sur $A[i+1]$

$P+=n;$ P pointe sur $A[i+n]$

$P--;$ P pointe sur $A[i-1]$

$P-=n;$ P pointe sur $A[i-n]$

Ces opérateurs (+, -, ++, --, +=, -=) sont définis seulement à l'intérieur d'un tableau car on en peut pas présumer que 2 variables de même type sont stockées de façon contiguë en mémoire.

Calcul d'adresse des composantes d'un tableau

Note : Il peut paraître surprenant que $P + i$ n'adresse pas le i^{ème} octet après P, mais la i^{ème} composante après P.

Pourquoi ? Pour tenter d'éviter des erreurs dans le calcul d'adresses.

Comment ? Le calcul automatique de l'adresse $P + i$ est possible car, chaque pointeur est limité à un seul type de données, et le compilateur connaît le # d'octets des différents types.

Soit A un tableau contenant des éléments du type **float** et P un pointeur sur **float**:

```
float A[20], x;  
float *P;
```

Après les instructions,

```
P = A;  
x = *(P+9);
```

X contient la valeur du dixième élément de A, i.e. celle de A[9].

Une donnée de type float ayant besoin de 4 octets, le compilateur obtient l'adresse $P + 9$ en ajoutant $9 * 4 = 36$ octets à l'adresse dans P.

Soustraction et comparaison de 2 pointeurs

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*:

$P1 - P2$ fournit le nombre de composantes comprises entre P1 et P2.

Le résultat de la soustraction $P1 - P2$ est

- négatif, si P1 précède P2
- zéro, si $P1 = P2$
- positif, si P2 précède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

- Comparaison de deux pointeurs

On peut comparer deux pointeurs par $<$, $>$, $<=$, $>=$, $==$, $!=$.

Mêmes tableaux :

Comparaison des indices correspondants.

Tableaux différents :

Comparaison des positions relatives en mémoire.

Différence entre un pointeur et le nom d'un tableau

Comme A représente l'adresse de A[0],

* (A+1) désigne le contenu de A[1]

* (A+2) désigne le contenu de A[2]

...

* (A+i) désigne le contenu de A[i]

- Un *pointeur* est une variable,
donc des opérations comme P = A ou P++ sont permises.

- Le *nom d'un tableau* est une constante,
donc des opérations comme A = P ou A++ sont impossibles.

Résumons ...

Soit un tableau A de type quelconque et i un indice d'une composante de A,

A

désigne l'adresse de

A[0]

A+i

désigne l'adresse de

A[i]

* (A+i)

désigne le contenu de A[i]

Si P = A, alors

P

pointe sur l'élément

A[0]

P+i

pointe sur l'élément

A[i]

* (P+i)

désigne le contenu de A[i]

Copie des éléments positifs d'un tableau S dans un tableau T

```
#include <iostream.h>

void main()
{
    int S[10] = { -3, 4, 0, -7, 3 , 8, 0, -1, 4, -9};
    int T[10];
    int i, j;

    for (i = 0, j = 0; i < 10; i++)
        if (*(S + i) > 0)
    {
        *(T + j) = *(S + i);
        j++;
    }

    for (i = 0; i < j; i++) cout << *(T + i) << " ";
    cout << endl;
}
```

Rangement des éléments d'un tableau dans l'ordre inverse

```
#include <iostream.h>
void main()
{
    int N;
    int tab[50];
    int somme = 0;
    cout << "Entrez la dimension N du tableau : ";
    cin >> N;
    for (int i = 0; i < N; i++)
    {
        cout << "Entrez la " << i << " ieme composante : ";
        cin >> *(tab+i);
        somme += *(tab+i);
    }
    for (int k = 0; k < N / 2; k++)
    {
        int echange = *(tab+k);
        *(tab+k) = *(tab + N - k - 1);
        *(tab + N - k - 1) = echange;
    }
}
```

Rangement des éléments d'un tableau dans l'ordre inverse

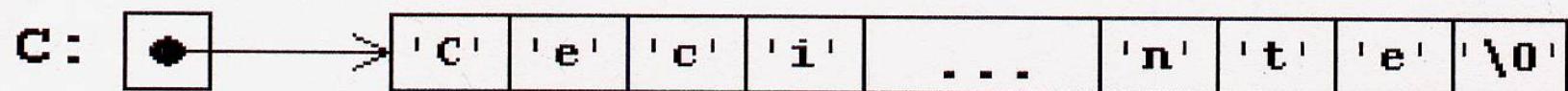
```
cout << endl << endl << "Affichage du tableau inverse." << endl;
for (int j = 0; j < N; j++)
{
    if((j % 3) == 0) cout << endl;
    cout << "tab[ " << j << " ] = " << *(tab+j) << "\t";
}
}
```

Pointeurs et chaînes de caractères

- Tout ce qui a été mentionné concernant les pointeurs et les tableaux reste vrai pour les pointeurs et les chaînes de caractères.
- En plus, un pointeur vers une variable de type char peut aussi contenir l'adresse d'une chaîne de caractères constante et peut même être initialisé avec une telle adresse.

Exemple

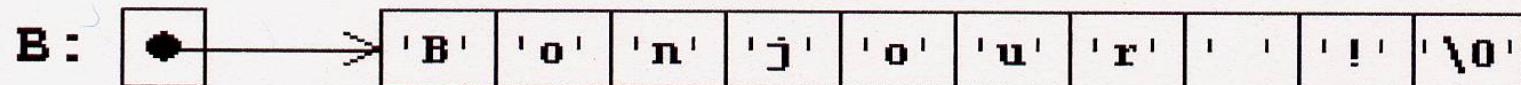
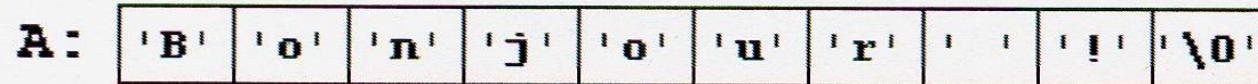
```
char *C;  
C = "Ceci est une chaîne de caractères constante";
```



```
char *B = "Bonjour !";
```

Distinction entre un tableau et un pointeur vers une chaîne constante

```
char A[] = "Bonjour !"; /* un tableau */
char *B = "Bonjour !"; /* un pointeur */
```



A a exactement la grandeur pour contenir la chaîne de caractères et \0. Les caractères peuvent être changés mais A va toujours pointer sur la même adresse en mémoire (pointeur constant).

Exemple

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B; /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```

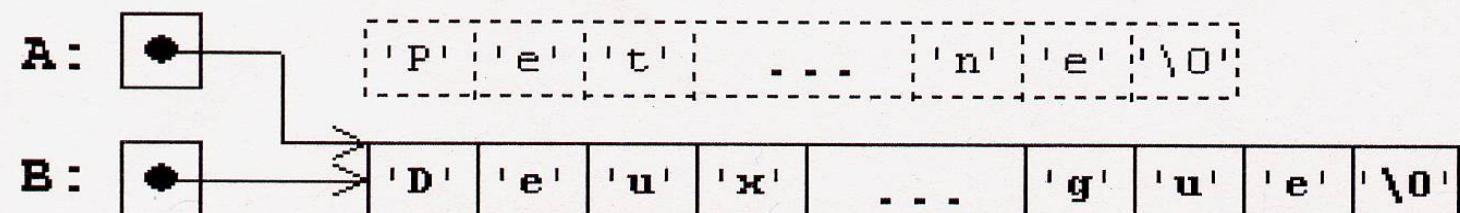
Distinction entre un tableau et un pointeur vers une chaîne constante

B pointe sur une chaîne de caractères constante. Le pointeur peut être modifié et pointer sur autre chose (la chaîne de caractères constante originale sera perdue). La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée (**B[1] = 'o'**; est illégal).

Exemple

```
char *A = "Petite chaîne";
char *B = "Deuxième chaîne un peu plus longue";
A = B;
```

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:



Un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur.

Avantage des pointeurs sur char

- Un pointeur vers char fait en sorte que nous n'avons pas besoin de connaître la longueur des chaînes de caractères grâce au symbole \0.
- Pour illustrer ceci, considérons une portion de code qui copie la chaîne CH2 vers CH1.

```
char * CH1;  
char * CH2;
```

...

1^{ière} version :

```
int I;  
I=0;  
while ((CH1[I]=CH2[I]) != '\0')  
    I++;
```

2^{ième} version :

Un simple changement de notation nous donne ceci :

```
int I;  
I=0;  
while ((* (CH1+I)=* (CH2+I)) != '\0')  
    I++;
```

Avantage des pointeurs sur char

Exploitons davantage le concept de pointeur.

```
while ((*CH1==*CH2) != '\0')
{
    CH1++;
    CH2++;
}
```

Un professionnel en C obtiendrait finalement :

```
while (*CH1++ == *CH2++)
;
```

Pointeurs et tableaux à deux dimensions

Soit $\text{int } M[4][10] = \{ \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\},$
 $\quad \quad \quad \{10,11,12,13,14,15,16,17,18,19\},$
 $\quad \quad \quad \{20,21,22,23,24,25,26,27,28,29\},$
 $\quad \quad \quad \{30,31,32,33,34,35,36,37,38,39\} \};$

M représente l'adresse du 1^e élément du tableau et pointe vers le tableau M[0] dont la valeur est : { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. De même, M + i est l'adresse du i^{ème} élément du tableau et pointe vers M[i] dont la valeur est la i^{ème} ligne de la matrice.

```
cout << (*M+2)[3]; // 23
```

Explication :

Un tableau 2D est un tableau unidimensionnel dont chaque composante est un tableau unidimensionnel. Ainsi, M + i désigne l'adresse du tableau M[i].

Question :

Comment accéder à l'aide de pointeurs uniquement à une composante M[i][j] ?

Il s'agit de convertir la valeur de M qui est un pointeur sur un tableau de type int en un pointeur de type int.

Pointeurs et tableaux à deux dimensions

Solution :

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}};
int *P;
P = (int *)M; /* conversion forcée */
```

Puisque le tableau 2D est mémorisé ligne par ligne et que cette dernière affectation entraîne une conversion de l'adresse `&M[0]` à `&M[0][0]` *, il nous est maintenant possible de traiter `M` à l'aide du pointeur `P` comme un tableau unidimensionnel de dimension 40.

- * `P` et `M` renferme la même adresse mais elle est interprétée de deux façons différentes.

Pointeurs et tableaux à deux dimensions

Exemple : Calcul de la somme de tous les éléments du tableau 2D M.

```
int M[4][10] = {{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 },
                 {10,11,12,13,14,15,16,17,18,19},
                 {20,21,22,23,24,25,26,27,28,29},
                 {30,31,32,33,34,35,36,37,38,39}};
int *P;
int I, SOM;
P = (int*)M;
SOM = 0;
for (I=0; I<40; I++)
    SOM += * (P+I);
```

Note : Dans cet exemple, toutes les lignes et toutes les colonnes du tableau sont utilisées. Autrement, on doit prendre en compte

- le nombre de colonnes réservé en mémoire,
- le nombre de colonnes effectivement utilisé dans une ligne,
- le nombre de lignes effectivement utilisé.

Tableaux de pointeurs

Syntaxe :

type * identificateur du tableau[nombre de composantes];

Exemple :

```
int * A[10]; // un tableau de 10 pointeurs  
// vers des valeurs de type int.
```

Tableaux de pointeurs vers des chaînes de caractères de différentes longueurs

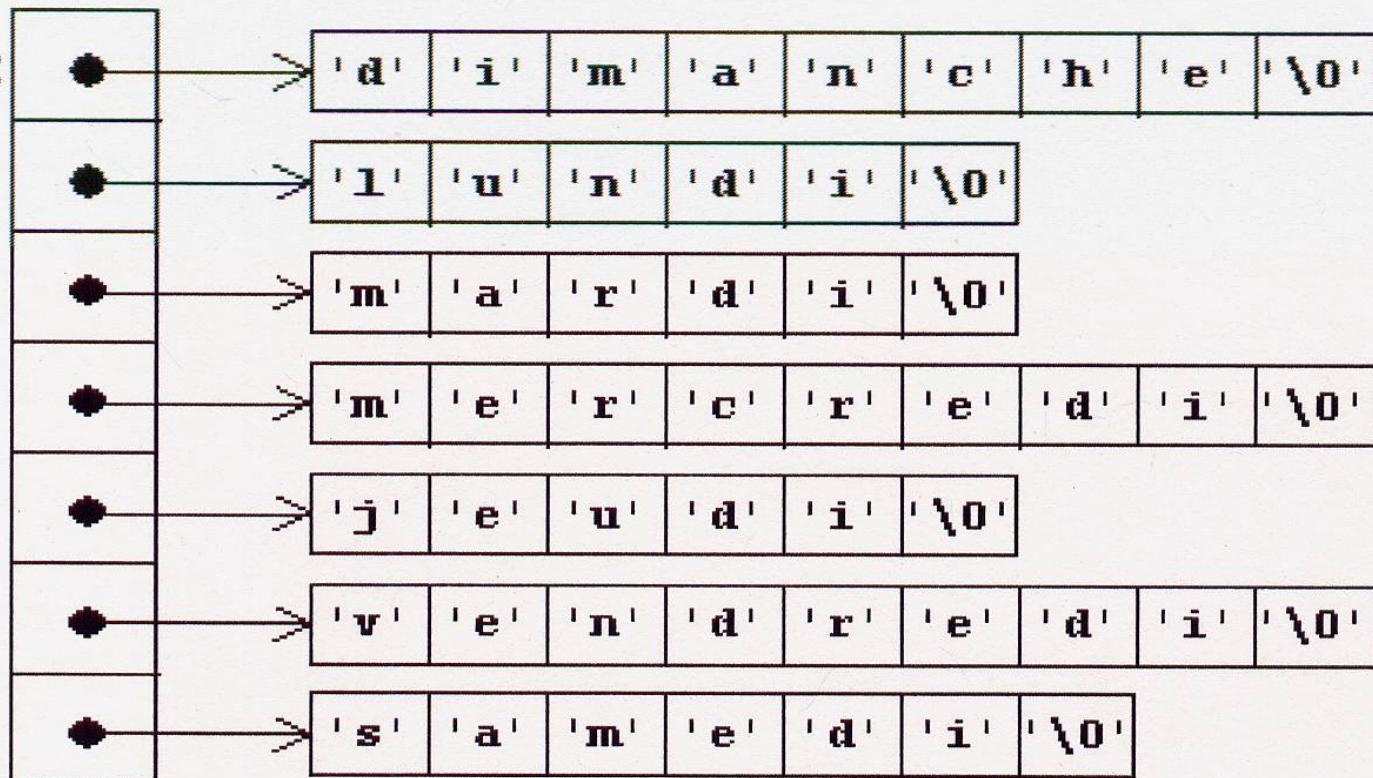
Exemple

```
char *JOUR[] = {"dimanche", "lundi", "mardi",  
"mercredi", "jeudi", "vendredi",  
"samedi"};
```

Nous avons déclaré un tableau JOUR[] de 7 pointeurs de type char, chacun étant initialisé avec l'adresse de l'une des 7 chaînes de caractères.

Tableaux de pointeurs

JOUR :



Affichage :

```
int I;
for (I=0; I<7; I++) printf("%s\n", JOUR[I]);
```

Pour afficher la 1^e lettre de chaque jour de la semaine, on a :

```
int I;
for (I=0; I<7; I++) printf("%c\n", *JOUR[I]);
```

Tableaux de pointeurs

Si $D[i]$ pointe dans un tableau,

$D[i]$	désigne l'adresse de la première composante
$D[i]+j$	désigne l'adresse de la j -ième composante
$* (D[i]+j)$	désigne le contenu de la j -ième composante

Les tableaux de pointeurs vers des chaînes de caractères de différentes longueurs sont d'un grand intérêt mais ce n'est pas le seul (à suivre).

Allocation statique de la mémoire

- Jusqu'à maintenant, la déclaration d'une variable entraîne automatiquement la réservation de l'espace mémoire nécessaire.
- Le nombre d'octets nécessaires était connu au temps de compilation; le compilateur calcule cette valeur à partir du type de données de la variable.

Exemples d'allocation statique de la mémoire

```
float A, B, C;          /* réservation de 12 octets */
short D[10][20];         /* réservation de 400 octets */
char E[] = {"Bonjour !"};
                        /* réservation de 10 octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                /* réservation de 40 octets */
```

Il en est de même des pointeurs ($p = 4$).

```
double *G;              /* réservation de p octets */
char *H;                 /* réservation de p octets */
float *I[10];             /* réservation de 10*p octets */
```

Allocation statique de la mémoire

Il en est de même des chaînes de caractères constantes ($p = 4$).

Exemples

```
char *J = "Bonjour !";
        /* réservation de p+10 octets */
float *K[] = {"un", "deux", "trois", "quatre"};
        /* réservation de 4*p+3+5+6+7 octets */
```

Allocation dynamique de la mémoire

Problématique :

Souvent, nous devons travailler avec des données dont nous ne pouvons prévoir le nombre et la grandeur lors de l'écriture du programme.

La taille des données est connue au temps d'exécution seulement.

Il faut éviter le gaspillage qui consiste à réservé l'espace maximal prévisible.

But :

Nous cherchons un moyen de réservé ou de libérer de l'espace mémoire au fur et à mesure que nous en avons besoin pendant l'exécution du programme.

Exemples :

La mémoire sera allouée au temps d'exécution.

```
char * P; // P pointera vers une chaîne de caractères  
          // dont la longueur sera connue au temps d'exécution.
```

```
int * M[10]; // M permet de représenter une matrice de 10 lignes  
              // où le nombre de colonnes varie pour chaque ligne.
```

La fonction malloc et l'opérateur sizeof

- La fonction malloc de la bibliothèque stdlib nous aide à localiser et à réserver de la mémoire au cours de l'exécution d'un programme.
- La fonction malloc fournit l'adresse d'un bloc en mémoire disponible de N octets.

`char * T = malloc(4000);`

Cela fournit l'adresse d'un bloc de 4000 octets disponibles et l'affecte à T.
S'il n'y a plus assez de mémoire, T obtient la valeur zéro.

- Si nous voulons réservé de l'espace pour des données d'un type dont la grandeur varie d'une machine à l'autre, on peut se servir de `sizeof` pour connaître la grandeur effective afin de préserver la portabilité du programme.

```
sizeof <var>
    fournit la grandeur de la variable <var>
sizeof <const>
    fournit la grandeur de la constante <const>
sizeof (<type>)
    fournit la grandeur pour un objet du type <type>
```

La fonction malloc et l'opérateur sizeof

Exemple :

```
#include <stdio.h>
void main()
{
    short A[10];
    char B[5][10];
    printf("%d%d%d%d%d", sizeof A, sizeof B,
           sizeof 4.25,
           sizeof "Bonjour !",
           sizeof(float),
           sizeof(double));
```

Exemple :

```
}
```

205081048

Réserver de la mémoire pour X valeurs de type int où X est lire au clavier.

```
int X;
int *PNum;
printf("Introduire le nombre de valeurs :");
scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

La fonction malloc et l'opérateur sizeof

Note :

S'il n'y a pas assez de mémoire pour satisfaire une requête, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** de **stdlib** et de renvoyer une valeur non nulle comme code d'erreur.

Exemple : Lire 10 phrases au clavier et ranger le texte.

La longueur maximale d'une phrase est fixée à 500 caractères.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
{
    /* Déclarations */
    char INTRO[500];
    char *TEXTE[10];
    int I;
    /* Traitement */
    for (I=0; I<10; I++)
    {
```

La fonction malloc et l'opérateur sizeof

```
gets(INTRO);
/* Réservation de la mémoire */
TEXTE[I] = malloc(strlen(INTRO)+1);
/* S'il y a assez de mémoire, ... */
if (TEXTE[I])
    /* copier la phrase à l'adresse */
    /* fournie par malloc, ... */
    strcpy(TEXTE[I], INTRO);
else
{
    /* sinon quitter le programme */
    /* après un message d'erreur. */
    printf("ERREUR: Pas assez de mémoire \n");
    exit(-1);
}
return 0;
}
```

Ex. : Matrice triangulaire inférieure (partie I)

Exemple :

12			
-2	4		
9	-17	50	
-98	19	25	75

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int * M[9];
    int i, j;

    //      Allocation dynamique de la mémoire.

    for (i = 0; i < 9; i++) M[i] = malloc((i+1) * sizeof(int));
```

Ex. : Matrice triangulaire inférieure (partie I)

// Initialisation de la matrice.

```
for (i = 0; i < 9; i++)
    for (j = 0; j <= i; j++)
        *(M[i] + j) = (i + 1) * 10 + j + 1;
```

// Affichage des éléments de la matrice.

```
for (i = 0; i < 9; i++)
{
    for (j = 0; j <= i; j++)
        printf("\t%d", *(M[i] + j));
    for (j = i+1; j < 9; j++)
        printf("\t0");
    printf("\n");
}
```

Reprendons le même exemple où, cette fois, le # de lignes de la matrice est lu.

Ex. : Matrice triangulaire inférieure (partie II)

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    int * * M;    ||| 
    int N;
    int i, j;

    printf("Entrez le nombre de lignes de la matrice : ");
    scanf("%d", &N);

    //      Allocation dynamique de la mémoire.
    ||| 
    M = malloc(N * sizeof(int *));
}

for (i = 0; i < N; i++) M[i] = malloc((i+1) * sizeof(int));
```

Ex. : Matrice triangulaire inférieure (partie II)

// Initialisation de la matrice.

```
for (i = 0; i < N; i++)
    for (j = 0; j <= i; j++)
        *((*(M+i)) + j) = (i + 1) * 10 + j + 1;
```

// Affichage des éléments de la matrice.

idem

```
for (i = 0; i < N; i++)
{
    for (j = 0; j <= i; j++)
        printf("\t%d", M[i][j]);
    for (j = i+1; j < N; j++)
        printf("\t0");
    printf("\n");
}
```

Libération de l'espace mémoire

- Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, nous pouvons le libérer à l'aide de la fonction **free** de la librairie **stdlib**.

free(pointeur);



Pointe vers le bloc à libérer.

À éviter :

Tenter de libérer de la mémoire avec **free** laquelle n'a pas été allouée par **malloc**.

Attention :

La fonction **free** ne change pas le contenu du pointeur.

Il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était rattaché.

Note :

Si la mémoire n'est pas libérée explicitement à l'aide de **free**, alors elle l'est automatiquement à la fin de l'exécution du programme.

Allocation dynamique, libération de l'espace mémoire en C++

- Le mot clé **new** permet de réservé de l'espace selon le type de donnée fourni.

Syntaxe : **new type de donnée**

L'opérateur renvoie l'adresse du bloc de mémoire allouée.
Si l'espace n'est pas disponible, l'opérateur renvoie 0.

Exemple :

```
unsigned short int * pPointeur;  
pPointeur = new unsigned short int;
```

ou encore,

```
unsigned short int * pPointeur = new unsigned short int;
```

- On peut également affecter une valeur à cette zone. Ex. : ***pPointeur = 25;**
- Pour libérer l'espace alloué avec **new**, on utilise l'opérateur **delete** une seule fois.
Ex.: **delete pPointeur;**

Autrement, il se produira une erreur à l'exécution. Pour éviter ceci, mettez le pointeur à 0 après avoir utilisé l'opérateur **delete**.

Allocation dynamique, libération de l'espace mémoire en C++

- Libérer le contenu d'un pointeur nul est sans incidence.

```
int * p = new int;  
delete p;           // libérer la mémoire.  
p = 0;  
...  
delete p;           // sans incidence sur le programme.
```

- Réaffecter une valeur à un pointeur alors que celui-ci n'est pas nul génère une perte de mémoire.

```
unsigned short int * pPointeur = new unsigned short int;  
*pPointeur = 72;  
pPointeur = new unsigned short int;  
*pPointeur = 36;
```

À chaque instruction **new** devrait correspondre une instruction **delete**.

- Tenter de libérer le contenu d'un pointeur vers une constante ou une variable allouée de manière statique est une erreur.

Ex. : **const int N = 5;**
int * p = &N;
delete p;

Allocation dynamique, libération de l'espace mémoire en C++

- Comment libérer l'espace mémoire d'un tableau ?

```
float * p = new float[10];
```

...

```
delete [] p;           // libérer la mémoire.  
p = 0;
```

car, nous sommes en présence d'un tableau.



Matrice de réels - exemple

```
#include <iostream.h>

void main()
{
    // Saisie de la dimension de la matrice.

    int M, N;
    cout << "Nombre de lignes : ";
    cin >> M;
    cout << "Nombre de colonnes : ";
    cin >> N;

    // Construction et initialisation d'une matrice réelle M x N.

    typedef float * pReel;
    pReel * P;
    P = new pReel[M];
```

Matrice de réels - exemple

```
for (int i = 0; i < M; i++)
{
    P[i] = new float[N];
    for (int j = 0; j < N; j++) P[i][j] = (float) 10*i + j;
}

// Affichage d'une matrice M x N.

for (i = 0; i < M; i++)
{
    cout << endl;
    for (int j = 0; j < N; j++) cout << P[i][j] << " ";
}
cout << endl;

// Libération de l'espace.

for (i = 0; i < M; i++) delete [] P[i];
delete [] P;
```

Pointeur générique

Une variable de type **void *** est un pointeur générique capable de représenter n'importe quel type de pointeur.

```
#include <stdio.h>
void main()
{
    int A = 5;           void * P = &A;
    if ((*int * P) == 5) // *P est invalide car on ne connaît
                          printf("%d", (*int * P)); // pas le type pointé par P.
}
```

On peut affecter un pointeur à un autre si les 2 sont de même type.
S'ils ne le sont pas, il faut effectuer une conversion explicite.

La seule exception est le type **void ***. On ne peut toutefois pas affecter un pointeur **void *** directement à un pointeur d'un autre type.

```
#include <iostream.h>
void main()
{
    void * P;           float * R;           int Q = 5;
    P = &Q;             R = (float *)P;
    cout << *R;         // Donne des résultats erronés. }
```

Usage de const avec les pointeurs

- Un pointeur constant est différent d'un pointeur à une constante.

```
int n = 44;  
int* p = &n;  
++(*p);  
++p;  
int* const cp = &n; → un pointeur constant  
++(*cp);  
++cp; → illégal  
const int k = 88;  
const int * pc = &k; → un pointeur à une constante  
++(*pc); → illégal  
++pc;  
const int* const cpc = &k; → un pointeur constant  
    à une constante  
++(*cpc); → illégal  
++cpc; → illégal
```

Avant-goût des structures de données

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    // Déclaration des types de données.
    struct Fiche_etudiant
    {
        char nom[25];
        char prenom[25];
        int age;
        bool sexe;
        int matricule;
        struct Fiche_etudiant * suivant;
    };
    struct Fiche_etudiant * pEnsemble_des_fiches = NULL;
    struct Fiche_etudiant * pointeur = NULL;

    int Nombre_de_fiches = 0;
    char test;
```

// Saisie des fiches des étudiants.

```
for (Nombre_de_fiches = 0; Nombre_de_fiches < 50; Nombre_de_fiches++)  
{  
    printf("\nVoulez-vous entrer les donnees d'une %s fiche (O ou N) ? ",  
          Nombre_de_fiches ? "autre " : "");  
    scanf(" %c", &test);  
    if(test == 'N' || test == 'n') break;  
  
    pointeur=(struct Fiche_etudiant *) malloc(sizeof(struct Fiche_etudiant));  
    (*pointeur).suivant = pEnsemble_des_fiches;  
    pEnsemble_des_fiches = pointeur;  
  
    scanf("%s%s%i%i%i",  
          (*pEnsemble_des_fiches).nom,  
          (*pEnsemble_des_fiches).prenom,  
          &(*pEnsemble_des_fiches).age,  
          &(*pEnsemble_des_fiches).sexe,  
          &(*pEnsemble_des_fiches).matricule);  
}
```

```
// Affichage de l'ensemble des fiches d'étudiants.  
  
pointeur = pEnsemble_des_fiches;  
while (pointeur != NULL)  
{  
    printf("\n%s %s %i %i %i ", (*pointeur).nom,  
          (*pointeur).prenom,  
          (*pointeur).age,  
          (*pointeur).sexe,  
          (*pointeur).matricule);  
    pointeur = (*pointeur).suivant;  
}
```

La structure de données utilisée est une pile.

Les pointeurs en C

- Définition : un pointeur est une variable ou une constante dont la valeur est une adresse
- L'adresse d'un objet est indissociable de son type
 - Exple : pointeur de caractère, d'entier
- L'opération fondamentale effectuée sur les pointeurs est l'indirection, c.a.d l'évaluation de l'objet pointé.
 - Rq : le résultat de cette indirection dépend du type pointé

Propriétés des pointeurs

- Les deux opérateurs de base sont & et * permettant d'obtenir respectivement l'adresse d'un objet et la valeur d'un objet pointé
- Il est possible de définir des pointeurs sur des objets complexes
- Les tableaux n'ont pas d'adresse, on ne peut donc pas extraire l'adresse d'un tableau avec &
- La taille d'un pointeur est le nombre d'octets utilisés pour le codage d'une adresse de l'espace d'adressage virtuel (en général 4 octets)
- Rq : un tableau est référencé par l'adresse de son 1ere élément, les opérations sur les tableaux sont identique à celles des pointeurs

Opération sur les pointeurs

- Affectation
 - Pour modifier le contenu d'un pointeur il faut une valeur de même type
 - Exple
 - Int n,*p,*q; déclaration d'une variable et de deux pointeurs
 - P=&n;l'adresse de n est affectée à p
 - *p=0; le contenu de p est mis à 0
 - Q=p; q pointe sur le même case que q
 - Pour affecter un pointeur sur type 1 de la valeur d'un pointeur de type 2 différent est possible en faisant un cast :
 - Type 1 *ptr1
 - Type 2 *ptr2
 - Ptr1=(type 1 *)ptr2

Réservation de la place mémoire

- Après la déclaration d'un pointeur il faut résERVER de la mémoire

```
#include <stdio.h>
```

```
Void * malloc (size_t taille)
```

La valeur renvoyée est un pointeur sur la 1ere case réservée ou NULL s'il n'y a plus de place

Exple

```
Q=(int *) malloc (sizeof(int))
```

Libérer la mémoire

```
#include <stdio.h>
Void free (void * ptr)
```

Visibilité des variables

- On appelle visibilité ou portée des variables les règles qui régissent l'utilisation des variables. Les mêmes règles régissent les types définis par l'utilisateur.
- Règle 1 : variables globales
Les variables déclarées avant la 1ere fonction peuvent être utilisées dans toutes les fonctions. Ces variables sont dites globales.

```
#include "stdio.h"
int i;

void f1 () {
    i = i+1;
}

void main(){
    i=0;
    f1();
    printf("%d\n", i) -> 1
}
```

Visibilité des variables

- Règle 2 : variables locales

Les variables déclarées dans une fonction ne peuvent être utilisées que dans cette fonction. Ces variables sont dites locales.

```
void f1 () {
    int i;
    i = i+1;
}
void main() {
i=0;    -> ERREUR : i n'existe pas pour main
...
}
```

Visibilité des variables

- Règle 3: arguments = variables locales

Les arguments d'une fonction sont aussi des variables locales de la fonction.

```
void f1 (int i) {  
    i = i+1; /* i est une variable locale de la fonction */
```

```
}
```

```
void main() {
```

```
    int j=1;
```

```
    f1(j);
```

```
    printf ("%d\n", j)
```

```
}
```

- Règle 4: Au sein d'une fonction, toutes les variables doivent avoir des noms distincts

```
void f1 () {
```

```
    int i;
```

```
    char i; -> ERREUR : i existe déjà
```

```
    i = i+1;
```

```
}
```

Visibilité des variables

- Règle 5 : Des variables déclarées dans des fonctions différentes peuvent porter le même nom sans ambiguïté.

```
void f1 () {  
    int i;      ← sous-entendu i_f1  
    ...  
}  
void f2 () {  
    char i;    ← sous-entendu i_f2  
    ...  
}  
void main() {  
    int i;      ← sous-entendu i_main  
    ...  
}
```

Ces 3 variables n'ont rien de commun

Visibilité des variables

- Règle 6 : Si une variable globale et une variable locale ont le même nom, on accède à la variable locale dans la fonction où elle est déclarée.. Si il n'y a pas de déclaration locale, on accède à la variable globale.

```
int i;
void f1 () {
    int i;
    i=2; /* i de f1 */
}
void main(){
    i=0; /* i global */
    f1();
    printf ("%d\n", i); -> 0
}
```

Conseils

- Evitez autant que possible l'usage des variables globales => limitation des effets de bord indésirables

```
int i;  
void f1 () {  
    ...  
    i=i+1;  
}  
void main() { i=0 ; f1() ;     printf ("%d\n", i) ; -> 1  
}
```

- Dans f1, on travaille sur i global :
 - Est-ce bien ce que l'on désirait (oubli de déclaration d'une nouvelle variable locale ?)
 - Débogage difficile : il faut inspecter le code en détail pour voir où sont modifiées les variables.

Conseils

- Si l'on ne peut éviter les variables globales, respecter un code pour différencier les variables globales des variables locales.
- Par exemple :
 - si l'initiale de la variable est une majuscule -> globale : Vglob
 - minuscule -> locale : vloc
- ou bien
- le nom de chaque variable globale commence par G_ : G_variable
- etc...
- Pas de confusion entre variables locales et globales.
- Mêmes règles pour les déclarations de type que pour les variables

Compléments : static

- Static : une telle variable maintient sa valeur à travers les appels de la fonction

```
void inc( )  
{  
    int i=0;  
    i++;  
    printf ("%d", i);  
}
```

1, 1, 1, 1, ...

```
void inc( )  
{  
    static int i=0;  
    i++;  
    printf ("%d", i);  
}
```

1, 2, 3, 4, ...

Compléments : register

- Une déclaration "register" indique au compilateur qu'une variable sera utilisée fréquemment.
- Si c'est possible, le compilateur utilisera un registre pour implanter la variable plutot qu'un emplacement mémoire (vitesse d'exécution)
- `register int i;`

Structures

- Une structure permet de rassembler sous un même nom des données de types différents
- Une structure peut contenir des données entières, flottantes, tableaux , caractères, pointeurs, etc... Ces données sont appelés les membres de la structure.
- Exemple : fiche d'identification d'un personne
 - nom, prénom, âge, liste des diplômes, etc...

Les structures en C

- **Différence entre une structure et un tableau**
 - Un tableau
 - permet de regrouper des éléments de même type
 - Les structures permettent de remédier à cette lacune des tableaux, en regroupant des objets (des variables) au sein d'une entité repérée par un seul nom de variable.
 - Les objets contenus dans la structure sont appelés **champs de la structure**.

Définition d'une structure

- Déclaration d'une structure : syntaxe

```
struct nomdelastructure {  
    typemembre1 nommembre1 ;  
    typemembre2 nommembre2 ;  
    ...  
    typemembren nommembren ;  
}
```

- Exemple : compte bancaire

```
struct compte {  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde ;  
};
```

```
struct compte a,b,c; /*déclaration de 3 variables de ce type*/
```

Déclarations de variables

- Autres façons de déclarer des variables structure

```
struct compte {  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde;  
} a, b; /*déclaration de 2 variables de ce type*/  
struct compte c; /*déclaration de 1 variable de ce type*/  
  
struct { /* le nom de la structure est facultatif */  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde;  
} a,b,c; /*déclaration de variables de ce type ici */  
/* mais plus de possibilité de déclarer d'autres variables de  
ce type*/  
        Déconseillé
```

Déclarations de variables

- Autres façons de déclarer des variables structure

```
typedef struct compte {  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde;  
} cpt ;  
/* cpt est alors un type équivalent à struct compte */
```

```
cpt a,b,c; /*déclaration de variables de ce type*/  
Recommandé
```

- Dans ce cas puisque on ne se sert plus de "struct compte" par la suite

```
typedef struct {  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde;  
} cpt ;
```

Structures imbriquées

- Une structure peut être membre d'une autre structure

```
struct date {  
    int jour;  
    int mois;  
    int annee;  
};
```

```
struct compte {  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde;  
    struct date dernier_versement;  
};
```

- Remarque : ordre de déclaration des structures

Structures

- Tableaux de structures

```
struct compte client[100];
```

- La portée du nom d'un membre est limité à la structure dans laquelle il est défini. On peut avoir des membres homonymes dans des structures distinctes.

```
struct s1 {  
    float x;  
    int y ;  
};
```

Pas de confusion

```
struct s2{  
    char x;  
    float y;  
};
```

Manipulation des structures

- Initialisation à la compilation

```
struct compte {  
    int no_compte ;  
    char etat ;  
    char nom[80] ;  
    float solde;  
    struct date dernier_versement;  
};
```

```
struct compte c1 = {12345,'p',"Dupond",2000.45,01,10,2019};
```

- Accès aux membres : opérateur . Syntaxe : variable.membre

```
1/ c1.solde = 3834.56;
```

```
2/ struct compte c[100];  
y=c[33].solde;
```

```
3/ c1.dernier_versement.jour = 15;  
c[12].dernier_versement.mois = 10;
```

Manipulation des structures

- Sur les structures elles-mêmes

- Affectation :

c [4] = c1

- Pas de comparaison , il faut comparer tous les membres

Structures et pointeurs

- L'adresse de début d'une structure s'obtient à l'aide de l'opérateur &

```
typedef struct {  
    int no_compte ;  
    char etat ;  
    char nom[80];  
    float solde;  
    struct date dernier_versement;  
} cpt ;
```

- `cpt c1, * pc;`
`c1` est de type `cpt`, `pc` est un pointeur sur une variable de type `cpt`
`pc = &c1;`
- Accès au membres à partir du pointeur
`*pc.no-compte = ...`  **Incorrect . est plus prioritaire que ***
`(*pc).no-compte = ...`
- Opérateur ->
`pc->no-compte = ...`

Structures et fonctions

- Les membres d'une structure peuvent être passés comme paramètres à des fonctions avec ou sans modification
- Ex1 (sans modification)

```
float ajoute_au_compte(float soldel, float somme1) {
    soldel = soldel+somme1;
    return (soldel);
}
void main ()
{
    .....
cpt c1;
c1.solde = 0.;
ajoute_au_compte(c1.solde,1000.0);
printf("%f\n",c1.solde); -> 0.000000
c1.solde=ajoute_au_compte(c1.solde,1000.0);
printf("%f\n",c1.solde); -> 1000.000000
```

Structures et fonctions

- Les membres d'une structure peuvent être passés comme paramètres à des fonctions avec ou sans modification
- Ex1 (sans modification)

```
float ajoute_au_compte(float soldel, float somme1) {  
    soldel = soldel+sommel;  
    return (soldel);  
}  
void main () {  
    ....  
    cpt c1;  
    c1.solde = 0.;  
    ajoute_au_compte(c1.solde,1000.0);  
    printf("%f\n",c1.solde);  -> 0.000000  
    c1.solde=ajoute_au_compte(c1.solde,1000.0);  
    printf("%f\n",c1.solde);  -> 1000.000000
```

Structures et fonctions

- Ex2 (avec modification)

```
void ajoute_au_compte(float * soldel, float sommel) {
    *soldel = *soldel+sommel;
}

void main () {
    .....
cpt c1;
c1.solde = 0.;
ajoute_au_compte(&(c1.solde),1000.0); /* ou &c1.solde */
printf("%f\n",c1.solde); -> 1000.000000
```

Structures et fonctions

- Un argument de fonction peut-être de type structure

```
float ajoute_au_compte(cpt c, float somme1) {  
    return(c.solde+somme1);  
}
```

```
void main () {  
cpt c1;  
c1.solde = ajoute_au_compte(c1,1000.0);  
printf ("%f\n",c1.solde); -> 1000.000000
```

- Ou pointeur sur structure

```
void ajoute_au_compte (cpt * c, float somme1) {  
    c->solde = c->solde + somme1;  
}
```

```
void main () {  
cpt c1;  
ajoute_au_compte(&c1 ,1000.0);  
printf ("%f\n",c1.solde); -> 1000.000000
```

Structures et fonctions

- La valeur de retour d'une fonction peut être une structure

```
cpt ajoute_au_compte(cpt c, float somme1) {
    cpt c2;
    c2=c;
    c2.solde=c.solde+somme1;
    return(c2);
}

void main () {
    .....
    cpt c1;
    c1.solde = 0.;
    c1=ajoute_au_compte(c1,1000.0);
    printf("%f\n",c1.solde); -> 1000.000000
```

Récursion

- *Définitions :*
- Une notion est dite récursive quand elle fait référence à elle-même soit directement soit indirectement.
- Récursion directe : $A \rightarrow A \rightarrow A$
- Récursion indirecte : $A \rightarrow B \rightarrow \dots \rightarrow A$
- *Exemples :*
- arbre : racine et des branches vers des sous-arbres
- $n! = n * (n-1)!$
- Un problème peut être représenté par un algorithme récursif quand il peut être décomposé en un ou plusieurs sous-problèmes de même type mais de taille inférieure.

Récursion

- *Méthode générale :*
 - 1) Le paramétrage consiste à mettre en évidence les éléments dont dépend la solution, en particulier la taille du problème.
 - 2) La recherche et la résolution d'au moins un cas trivial : consiste à résoudre le problème directement, c-à-d sans appel récursif, dans un cas particulier.
 - 3) La décomposition du cas général consiste à passer d'un problème de taille N à un ou des problèmes de taille < N.
- *Exemple 1 : calcul de factorielle.*
 - 1) paramétrage : n
 - 2) cas triviaux : $1! = 0! = 1$
 - 3) décomposition : $n! = n * (n-1)!$

Récursion

```
int facto (int n) {  
    int p;  
    if (n==0) return (1);  
    else {  
        p = n * facto(n-1); /* appel récursif à la fonction  
facto */  
        return (p);  
    }  
}
```

- Lors du calcul de $\text{facto}(n)$ il y a n appels à la fonction facto . La pile contient n "assiettes" correspondant à cette fonction.

Récursion

Exemple 2 : Suite récurrente : Suite de Fibonacci (récursion double);

$$U_n = U_{n-1} + U_{n-2}$$

$$U_1 = U_0 = 1$$

- 1) paramétrage : n
- 2) cas triviaux : $U_1 = U_0 = 1$;
- 3) décomposition : $U_n = U_{n-1} + U_{n-2}$

- Implantation en langage C

```
int fibo (int n) {  
    if ((n==0) || (n==1)) return (1);  
    else return ( fibo(n-1) + fibo (n-2) ) ; /* appel  
récursif à la fonction fibo */  
}
```

- Lors du calcul de $fibo(n)$ il y a 2^n appels à la fonction $fibo$. La pile contient n "assiettes" correspondant à cette fonction.

Récursion

Exemple 3 : Combinasions C_n^p (méthode du triangle de Pascal);

- 1) paramétrage : n et p
- 2) cas triviaux : $C_n^0 = 1$ $C_n^n = 1$
- 3) décomposition : $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$

Implantation en langage C

```
int combinaisons (int n, int p) {  
    if ((n==p) || (p==0)) return (1);  
    else return (combinasions (n-1, p-1)+combinasions (n-1, p));  
/* appel récursif à la fonction combinasions */  
}
```

Récursion

Exemple 4 : Tri par partition d'un tableau T sur l'intervalle [a , b]

- 1) paramétrage : a et b
- 2) cas triviaux : $a \geq b-1$, rien à faire
- 3) décomposition :

Trier T sur $[a,b]$: - Faire la partition de T sur $[a, b]$ et soit adpivot l'adresse du pivot après partition

- Trier T sur $[a , adpivot - 1]$
- Trier T sur $[adpivot + 1, b]$

```
void tri (tab T , int a , int b) {  
    int adpivot;  
    if (b > a+1) { /* sinon ne rien faire */  
        partition (T,a,b,&adpivot);  
        tri(T,a,adpivot-1);  
        tri(T,adpivot+1,b);  
    }  
}
```

Récursion

Exemple 5 : Recherche du zéro d'une fonction continue sur [a , b]

Pb : Trouver un zéro x_0 d'une fonction continue sur $[a , b]$ avec une précision $\varepsilon > 0$ donnée si il en existe, sinon détecter qu'il n'y a pas de zéros.

On cherche x et y / $x < y \leq x + \varepsilon$ et tels que $f(x).f(y) < 0$.

Principe de dichotomie :

- si a et b sont tels que $f(a). f(b) < 0$ alors il existe un zéro dans $[a,b]$.
- sinon, on divise l'intervalle en 2 moitiés et on recherche dans le premier intervalle. Si l'on n'a pas trouvé de zéro on cherche dans le second.

1) paramétrage : a , b et ε

2) cas triviaux :

$a < b \leq a + \varepsilon$ et $f(a). f(b) < 0$ alors il existe un zéro dans $[a,b]$ et $x_0 = a$

$a < b \leq a + \varepsilon$ et $f(a). f(b) > 0$ alors il n'existe pas de zéro dans $[a,b]$

3) décomposition : principe de dichotomie

RécurSION

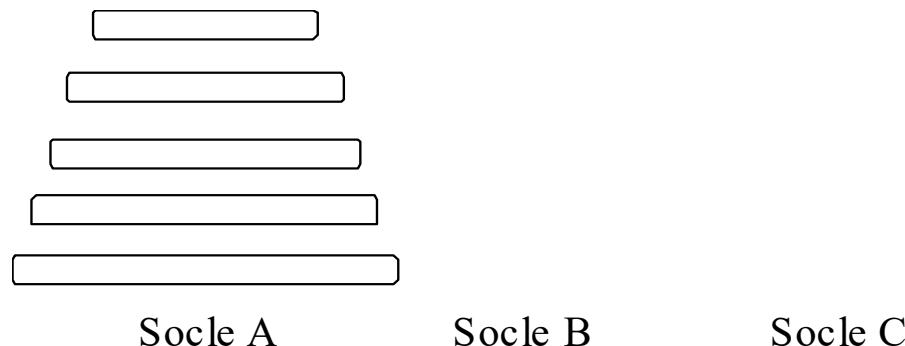
```
int zero  (float a, float b, float epsilon, float * x)
{
/* retourne 0 si il n'existe pas de zero dans
l'intervalle a , b; 1 sinon */
int trouve;
if  ( b-a <= epsilon)
    if (f(a) * f(b) < 0 ) { *x = a; return (1); }
    else return (0);
else      {
    trouve = zero (a, (a+b)/2,epsilon,x);
    if (trouve) return(1);
    else return(zero ((a+b)/2,b,epsilon,x));
}
}
```

Récursion

Exemple 6 : Les tours de Hanoi :

Soient 3 socles A, B et C. Sur le socle A sont posées n disques de taille décroissante. Le problème consiste à transférer tous les disques du socle A au socle B en respectant les contraintes suivantes :

- on ne déplace qu'un disque à la fois
- on ne peut déplacer que les disques se trouvant en haut de chaque socle
- on ne peut déplacer un disque que si on le pose sur un disque plus grand ou sur un socle vide. Exemple avec 5 disques :



Récursion

- 1) paramétrage : N : nombre de disques, socle de départ, socle relais, socle final
- 2) cas triviaux : N = 0 ne rien faire ; N= 1, et socle final vide : déplacer de départ à final
- 3) décomposition :

Transférer N disques de A vers B en passant par C :

- Transférer N-1 disques de A vers C en passant par B
- Déplacer 1 disque de A vers B
- Transférer N-1 disques de C vers B en passant par A

```
void hanoi (int n, char depart, char final, char relais) {  
if (N>0) {  
    hanoi(n-1, depart, relais, final);  
    printf("deplacement de %c à %c\n", depart, final);  
    hanoi(n-1, relais, final, depart) ; } ;  
}
```

Récursion

remarques :

- la seule véritable action est faite par le printf
- pour n disques il y a 2^n appels à la fonction hanoi

appels successifs

hanoi(3, 'A', 'B', 'C')

|hanoi(2, 'A', 'C', 'B')

 |hanoi(1, 'A', 'B', 'C')

 |hanoi(0, 'A', 'C', 'B')

 A vers B

 |hanoi(0, 'C', 'B', 'A')

 A vers C

 |hanoi(1, 'B', 'C', 'A')

 |hanoi(0, 'B', 'A', 'C')

 B vers C

 |hanoi(0, 'A', 'C', 'B')

 A vers B

Actions

rien

A vers B (1)

rien

A vers C (2)

rien

B vers C (3)

rien

A vers B (4)

hanoi(2, 'C', 'B', 'A')

|hanoi(1, 'C', 'A', 'B')

 |hanoi(0, 'C', 'B', 'A')

 C vers A

 |hanoi(0, 'B', 'A', 'C')

 C vers B

rien

C vers A (5)

rien

C vers B (6)

 |hanoi(1, 'A', 'B', 'C')

 |hanoi(0, 'A', 'C', 'B')

 A vers B

rien

A vers B (7)



Socle A

Socle B

Socle C

1



Socle A

Socle B

Socle C

2



Socle A



Socle B



Socle C

3



Socle A

Socle B



Socle C

4

Socle A

Socle B

Socle C

Socle A

Socle B

Socle C

5

6



Socle A



Socle B

Socle C

Socle A

Socle B

Socle C

7



Déclaration d'une structures

- Lors de la déclaration de la structure, on indique les champs de la structure, c'est-à-dire le type et le nom des variables qui la composent:

```
struct Nom_Structure {  
    type_champ1 Nom_Champ1;  
    type_champ2 Nom_Champ2;  
    type_champ3 Nom_Champ3;  
    type_champ4 Nom_Champ4;  
    type_champ5 Nom_Champ5; ... };
```

Exemple

```
struct MaStructure {  
    int Age;  
    char Sexe; char Nom[12];  
    float MoyenneScolaire;  
    struct AutreStructure StructBis;  
    /* en considerant que la structure AutreStructure est definie */};
```

Définition d'une variable structurée

- La définition d'une variable structurée est une opération qui consiste à créer une variable ayant comme type celui d'une structure que l'on a précédemment déclaré
- La définition d'une variable structurée se fait comme suit:
 - `struct Nom_Structure Nom_Variable_Structuree;`
Nom_Structure représente le nom d'une structure que l'on aura préalablement déclarée.
Nom_Variable_Structuree est le nom que l'on donne à la variable

Accès aux membres de la structure

- Chaque variable de type structure possède des champs repérés avec des noms uniques.
- Pour accéder aux champs d'une structure on utilise l'opérateur de champ (un simple point) placé entre le nom de la variable structurée que l'on a définie et le nom du champ :
 - `Nom_Variable.Nom_Champ`; Ainsi, pour affecter des valeurs à la variable *Pierre* (variable de type *struct Personne* définie précédemment), on pourra écrire:
 - `Pierre.Age = 18; Pierre.Sexe = 'M';`

Exemple

```
#include <stdio.h>
struct adresse
{
    char Nom [50];
    char Prenom [50];
    char Adresse [50];
    char telephone [50];
}
int main ()
{
    struct adresse une_adresse;
    printf (« donner votre nom »);
    gets(une_adresse.Nom);
    printf (« donner votre prénom »);
    gets(une_adresse.Prenom);
    printf (« donner votre Adresse »);
    gets(une_adresse.Adresse);
    printf (« donner votre téléphone »);
    gets(une_adresse.telephone);
    printf (« votre saisie : \n Nom : %s \n : \n Prénom : %s \n : \n Adresse : %s \n : \n
téléphone: %s \n », une_adresse.Nom, une_adresse.Prenom, une_adresse.Adresse,
une_adresse.telephone );
}
```

Les fonctions en c

- Définition : Les fonctions sont des parties de code source qui permettent de réaliser le même type de traitement plusieurs fois ou sur des objets différents
- Nous utilisons des fonctions prédéfinies dans les bibliothèques (stdio.h)

Caractéristiques des fonctions

- Une fonction en langage C peut :
 - modifier des données globales. Ces données sont dans une zone de mémoire qui peut être modifiée par le reste du programme. Une fonction peut dans ces conditions réaliser plusieurs fois le même traitement sur un ensemble de variables défini statiquement à la compilation.
 - communiquer avec le reste du programme par une interface. Cette interface est spécifiée à la compilation. L'appel de la fonction correspond à un échange de données à travers cette interface, au traitement de ces données (dans le corps de fonction), et à un retour de résultat via cette interface. Ainsi, une fonction permet de réaliser le même traitement sur des ensembles de variables différents.

Variables locales

- Les variables déclarées dans un bloc d'instructions sont uniquement visibles à l'intérieur de ce bloc. On dit que ce sont des variables locales à ce bloc. Exemple : La variable NOM est définie localement dans le bloc extérieur de la fonction HELLO.

```
void HELLO(void);
{
    char NOM[20];
    printf("Introduisez votre nom :");
    gets(NOM);
    printf("Bonjour %s!\n",NOM);
}
```

Variables globales

- Définition
 - Ce sont les variables déclarées au début du fichier, à l'extérieur de toutes les fonctions
 - Elles sont disponibles à toutes les fonctions du programme.
 - En général déclarées immédiatement derrière les instructions #include au début du programme.
 - Attention!!! Les variables déclarées au début de la fonction principale main ne sont pas des variables globales, mais elles sont locales à main !
- Exemple : La variable STATUS est déclarée globalement pour pouvoir être utilisée dans les fonctions A et B
-

```
#include <stdio.h>
int STATUS;

void A(...)
{
...
if(STATUS>0)
    STATUS--;
else
    ...
}

void B(...)
{
...
STATUS++;
}

}
```

Déclaration définition de fonctions

- En général, le nom d'une fonction apparaît à trois endroit différents dans un programme :
 1. lors de la déclaration
 2. lors de la définition
 3. lors de l'appel

```
#include<stdio.h>
main()
{
    /*Fonctions appelées*/
    int ENTREE(void);
    Int MAX(int N1, int N2);

    /*Déclaration des variables*/
    int A,B;

    /*Traitement avec appel des fonctions*/
    A=ENTREE();
    B=ENTREE();
    printf("Le maxi est %d\n",MAX(a,B));
}

/*Définition de la fonction ENTREE*/
int ENTREE(void)
{
    int NOMBRE;
    puts("Entrez un nombre entier :");
    scanf("%d",&NOMBRE);
    return NOMBRE;
}

/*Définition de la fonction MAX*/
int MAX(int N1, int N2)
{
    if(N1>N2)
        return N1;
    else
        return N2;
}
```

Définition d'une fonction

- Lors de leur définition ou de leur utilisation les fonctions sont distinguées des variables par la présence des parenthèses ouvrantes et fermantes. Une définition de fonction, contient : une interface et un corps de fonction.
- Exemple : Structure d'une fonction
-

```
int add(int a, int b)
{
    int c;
    c=a+b;
    return c;
}
```

INTERFACE
CORPS
BLOC

Code retour

- Une fonction en c renvoie une valeur avec la fonction return
- Une procédure a comme code de retour void

```
void LIGNE(int L)
{
    /*Déclarations des variables locales*/
    int I;
    /*Traitements*/
    for (I=0;I<L;I++)
        printf("*");
    printf("\n");
}
```

```
double CARRE(double X)
{
    return X*X
}
```

Résumé

- Langage c est un langage de bas niveau
 - Avantages pas de limites pour la programmation
 - Inconvénients les bugs
- Le manque de structure évolué fait qu'on ne peut pas définir des fonctions dans les structures

Programmation Objet

Object-Oriented Programming

An *algorithm* is a step-by-step process.

A *computer program* is a step-by-step set of instructions for a computer.

Every computer program is an algorithm.

Algorithms have a long history in science, technology, engineering and math.

Object-Oriented Programming

Early computers were far less complex than computers are today.

Their memories were smaller and their programs were much simpler.



Object-Oriented Programming

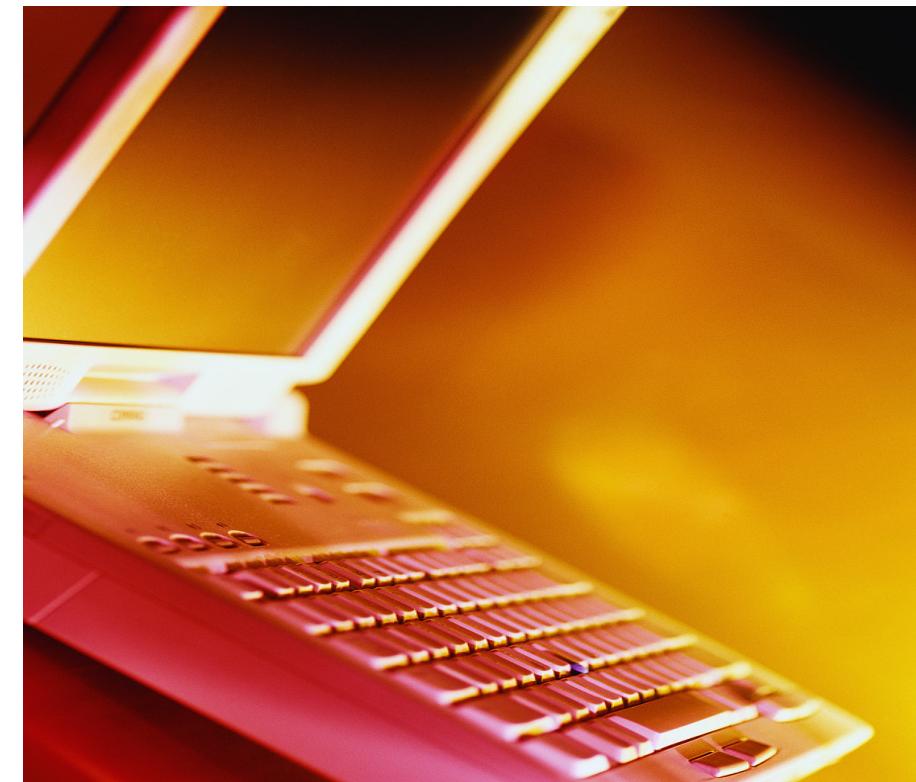
They usually executed only one program at a time.



Object-Oriented Programming

Modern computers are smaller, but far more complex than early computers.

They can execute many programs at the same time.



Programming Languages

- Programming languages allow programmers to code software.
- The three major families of languages are:
 - Machine languages
 - Assembly languages
 - High-Level languages

Machine Languages

- Comprised of 1s and 0s
- The “native” language of a computer
- Difficult to program – one misplaced 1 or 0 will cause the program to fail.
- Example of code:

1110100010101

10111010110100

111010101110

10100011110111

Assembly Languages

- Assembly languages are a step towards easier programming.
- Assembly languages are comprised of a set of elemental commands which are tied to a specific processor.
- Assembly language code needs to be translated to machine language before the computer processes it.
- Example:
ADD 1001010, 1011010

High-Level Languages

- High-level languages represent a giant leap towards easier programming.
- The syntax of HL languages is similar to English.
- Historically, we divide HL languages into two groups:
 - Procedural languages
 - Object-Oriented languages (OOP)

Procedural Languages

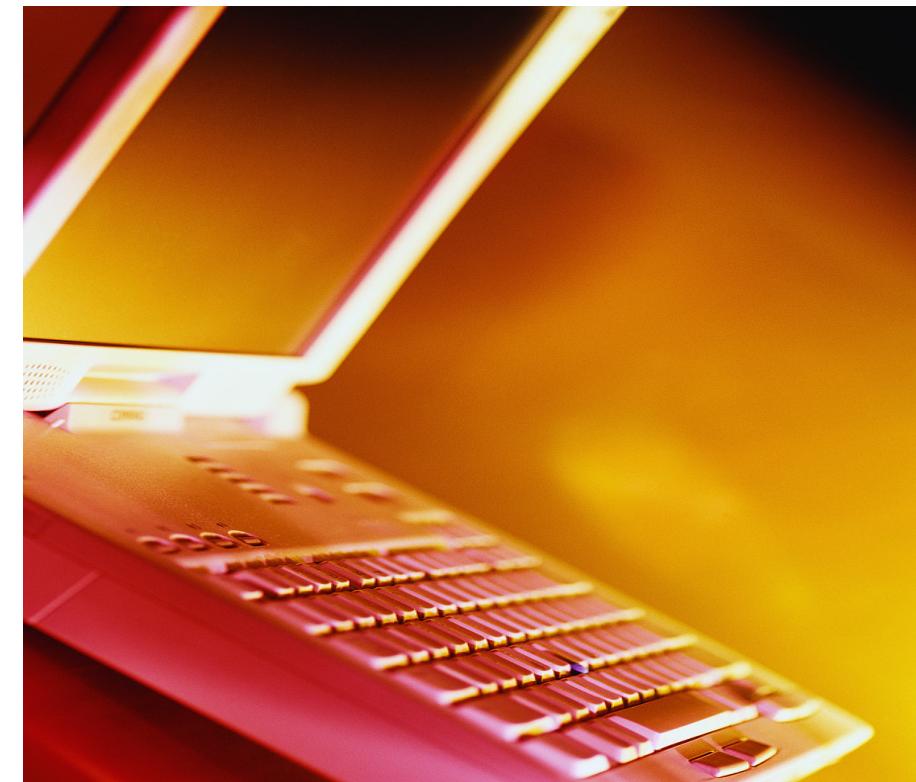
- Early high-level languages are typically called procedural languages.
- Procedural languages are characterized by sequential sets of linear commands. The focus of such languages is on *structure*.
- Examples include C, COBOL, Fortran, LISP, Perl, HTML, VBScript

Object-Oriented Languages

- Most object-oriented languages are high-level languages.
- The focus of OOP languages is not on structure, but on *modeling data*.
- Programmers code using “blueprints” of data models called *classes*.
- Examples of OOP languages include C++, Visual Basic.NET and Java.

Object-Oriented Programming

Computer scientists have introduced the notion of *objects* and *object-oriented programming* to help manage the growing complexity of modern computers.



Object-Oriented Programming

An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Object-Oriented Programming

An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Numbers



Object-Oriented Programming

An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Text



Object-Oriented Programming

An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Pictures



Object-Oriented Programming

An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Sound



Object-Oriented Programming

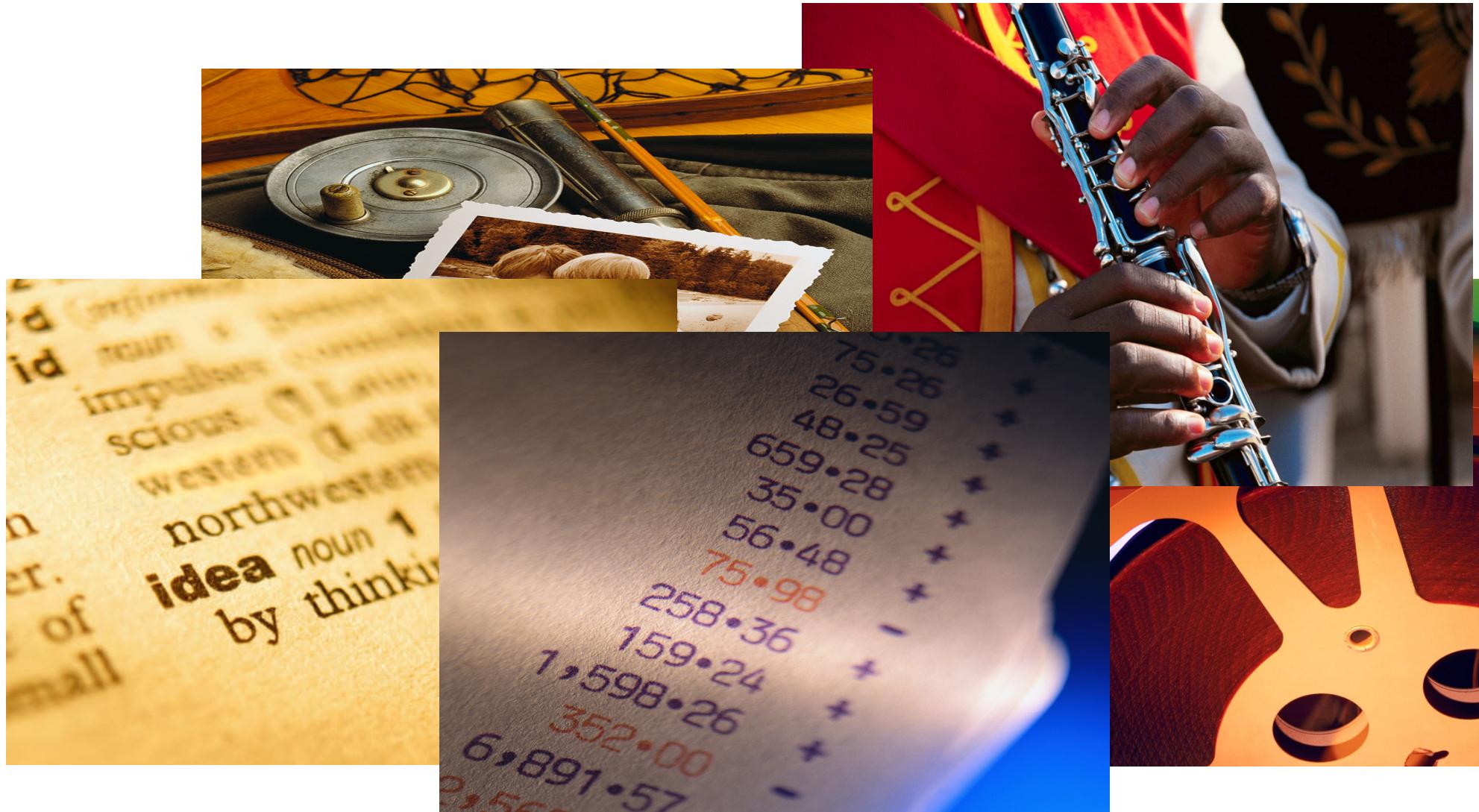
An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Video



Object-Oriented Programming

An object is anything that can be represented by data.



Object-Oriented Programming

An object can be something in the physical world or even just an abstract idea.

An airplane, for example, is a physical object that can be manipulated by a computer.



Object-Oriented Programming

An object can be something in the physical world or even just an abstract idea.

A bank transaction is an example of an object that is not physical.



Object-Oriented Programming

To a computer, an object is simply something that can be represented by data in the computer's memory and manipulated by computer programs.



Object-Oriented Programming

The data that represent the object are organized into a set of *properties*.

The values stored in an object's properties at any one time form the *state* of an object.

Name: PA 3794

Owner: US Airlines

Location: 39 52' 06" N 75
13' 52" W

Heading: 271°

Altitude: 19 m

AirSpeed: 0

Make: Boeing

Model: 737

Weight: 32,820 kg

Object Oriented Programming

- **Object** – Unique programming entity that has *methods*, has *attributes* and can react to *events*.
- **Method** – Things which an object can do; the “verbs” of objects. In code, usually can be identified by an “action” word -- *Hide*, *Show*

Object Oriented Programming

- **Attribute** – Things which describe an object; the “adjectives” of objects. In code, usually can be identified by a “descriptive” word – *Enabled, BackColor*
- **Events** – Forces external to an object to which that object can react. In code, usually attached to an event procedure

Object Oriented Programming

- **Class** – Provides a way to create new objects based on a “meta-definition” of an object (Example: The automobile **class**)
- **Constructors** – Special methods used to create new instances of a class (Example: A Honda Civic is an **instance** of the automobile **class**.)

Object-Oriented Programming

Computer programs implement algorithms that manipulate the data.

In object-oriented programming, the programs that manipulate the properties of an object are the object's **methods**.

```
class Bicycle {  
  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
  
    void applyBrakes(int decrement) {  
        speed = speed - decrement;  
    }  
}
```

Object-Oriented Programming

We can think of an object as a collection of properties and the methods that are used to manipulate those properties.

Properties

Methods

```
class Bicycle {  
    int cadence = 0;  
    int speed = 0;  
    int gear = 1;  
  
    void changeCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    void changeGear(int newValue) {  
        gear = newValue;  
    }  
  
    void speedUp(int increment) {  
        speed = speed + increment;  
    }  
  
    void applyBrakes(int decrement) {  
        speed = speed - decrement;  
    }  
}
```

Object-Oriented Programming

A *class* is a group of objects with the same properties and the same methods.

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}
```

Object-Oriented Programming

Each copy of an object from a particular class is called an *instance* of the object.



Object-Oriented Programming

The act of creating a new instance of an object is called **instantiation**.



Object-Oriented Programming

A class can be thought of as a blueprint for instances of an object.



Object-Oriented Programming

Two different instances of the same class will have the same properties, but different values stored in those properties.



OOP - Encapsulation

- Incorporation into a class of data & operations in one package
- Data can only be accessed through that package
- “Information Hiding”

OOP - Inheritance

- Allows programmers to create new classes based on an existing class
- Methods and attributes from the parent class are inherited by the newly-created class
- New methods and attributes can be created in the new class, but don't affect the parent class's definition

OOP - Polymorphism

- Creating methods which describe the way to do some general function
(Example: The “drive” method in the automobile class)
- Polymorphic methods can adapt to specific types of objects.

Classes and Objects

- A **class** is a data type that allows programmers to create objects. A class provides a definition for an object, describing an object's attributes (data) and methods (operations).
- An object is an *instance* of a class. With one class, you can have as many objects as required.
- This is analogous to a variable and a data type, the class is the data type and the object is the variable.

Sample Class Definition

```
Class Cube
    Side As Real
    Volume As Real
    Subprogram SetSide(NewSide)
        Set Side = NewSide
        End Subprogram
    Subprogram ComputeVolume()
        Set Volume = Side ^ 3
        End Subprogram
    Function GetVolume() As Real
        Set GetVolume = Volume
        End Function
    Function GetSide() As Real
        Set GetSide = Side
        End Function
End Class
```

The class Cube is similar to the definition of a record, but also has functions added that are part of the objects created. So we can think of it as an object ‘has’ data attributes and function attributes

Sample Instance of Class

Main Program

 Declare Cube1 As Cube

 Write "Enter a positive number:"

 Input Side1

 Call Cube1.SetSide(Side1)

 Call Cube1.ComputeVolume

 Write "The volume of a cube of side ", Cube1.GetSide

 Write "is ", Cube1.GetVolume

End Program

Is JavaScript an OOP language?

- Well, not really ...
- We call JavaScript an "object-inspired" language
- It uses objects by way of supporting inheritance and encapsulation, but it doesn't really provide support for polymorphism.

“Object-Oriented” JavaScript

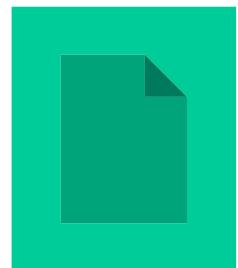
- More like “Object-Inspired” JavaScript
- We can create new, custom, re-usable objects in JavaScript that include their own methods, properties and events.
- Consider the following problem: “I want to record the color, brand, horsepower and price of several cars.”

What is an API?

- API stands for Application Programming Interface
- Allows programmers to extend the current language to include customized components
- Most modern languages incorporate APIs

Solution with OOP Design

- Calls to an API that contains the custom car object
- Much cleaner code
- Re-usable object



Object-Oriented Programming

Object

Property

Method

The same terminology is used in most object-oriented programming languages.

Instantiation

Instance

State

Class

Le langage C++ (partie I)

Crédit de slides : 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

Bibliographie

- *Le Langage C++ (The C++ Programming Language)*, de Bjarne Stroustrup, Addison-Wesley – 4^{ème} édition, mai 2013
- Programmation: Principes et pratique avec C++, , de Bjarne Stroustrup et al., Pearson Education, déc. 2012
- *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++*, 8^{ème} 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/~f1/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%2B%2B/CourseA/>
- ***Thinking in C++* de Bruce Eckel, 2003**
<http://w2.syrnex.com/jmr/eckel/>
- <http://www.stroustrup.com/C++.html>
- <http://channel9.msdn.com/Events/GoingNative/2013/Openings-Keynote-Bjarne-Stroustrup>
- Livres gratuits en ligne : <http://it-ebooks.info/book/1256/>
- Aide en ligne : <http://www.cplusplus.com/doc/tutorial/>

Historique du Langage C++

- **4ème langage le plus utilisé au monde (classements TIOBE de Août 2020 <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>) derrière C, Java, Python**
- **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++14

- C++14 : révision mineure de C++11
 - <http://electronicdesign.com/dev-tools/bjarne-stroustrup-talks-about-c14>
 - <https://parasol.tamu.edu/people/bs/622-GP/C++14TAMU.pdf>
 - http://www1.cs.columbia.edu/~aho/cs4115/lectures/14-01-29_Stroustrup.pdf
- Nouvelle mise à jour annoncée pour 2017
- Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* - O'Reilly Media; 1 edition (December 5, 2014) ISBN-13: 978-1491903995

Nouvelle norme C++ : C++17

- C++11 (C++0x) approuvée par l'ISO en 12/2017 et disponible depuis dec 2017 (norme ISO/CEI 14882:2011)
- Quelques sites explicatifs des nouveautés de la norme 2011/2017 :
 - <http://www.siteduzero.com/tutoriel-3-497647-introduction-a-c-2011-c-0x.html>
 - <https://en.wikipedia.org/wiki/C%2B%2B17>
 - <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/>
 - <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++ ?

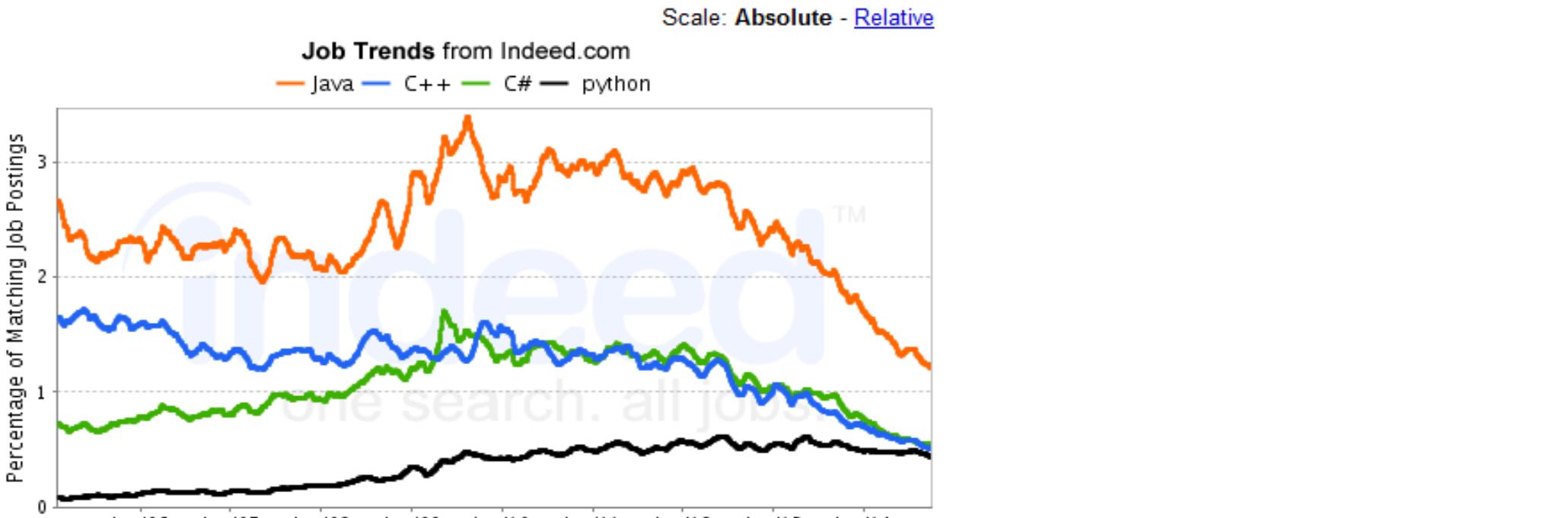
- D'après Bjarne Stroustrup, conception du langage C++ pour :
 - Être meilleur en C,
 - Permettre les abstractions de données
 - Permettre la programmation orientée-objet
- Compatibilité C/C++ [Alard, 2000] :
 - C++ = sur-ensemble de C,
 - C++ ⇒ ajout en particulier de l'orienté-objet (classes, héritage, 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] :

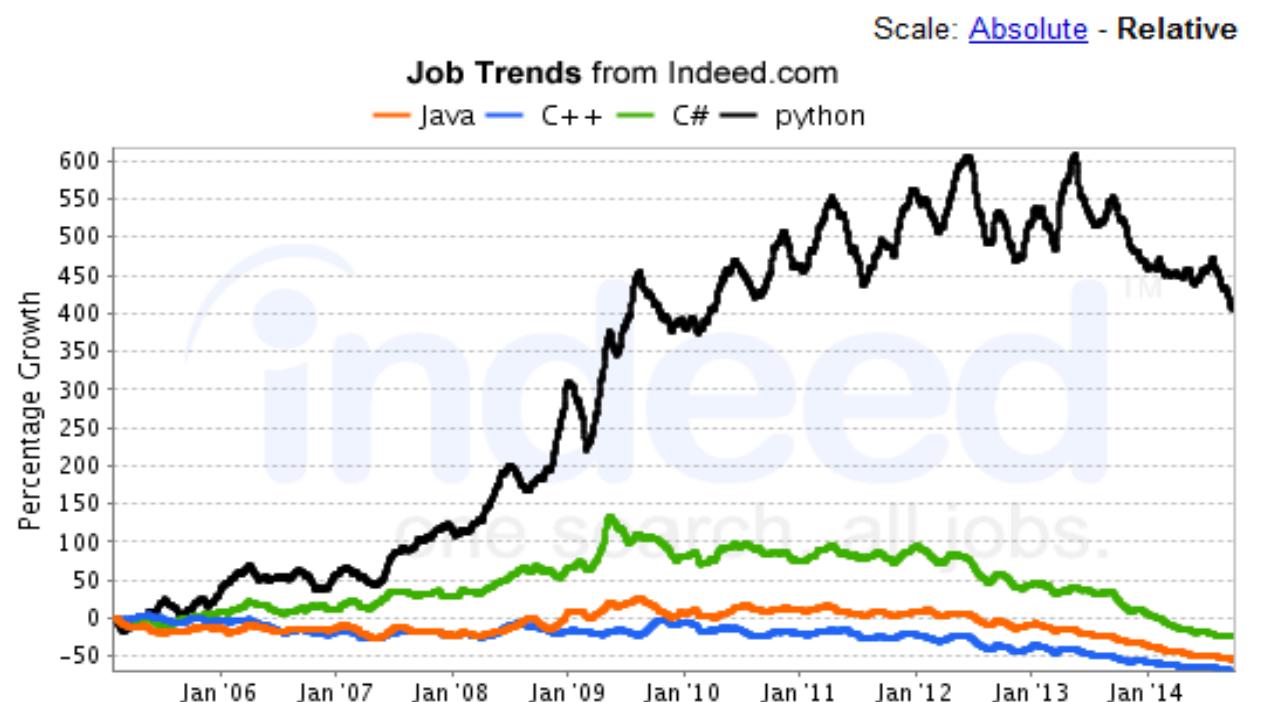
- Java
 - Création des objets par allocation dynamique (*new*)
 - Accès aux objets par références
 - Destruction automatique des objets par le ramasse miettes
- C++
 - Allocation des objets en mémoire statique (variables globales), dans la **pile** (variables automatiques) ou dans le **tas** (allocation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



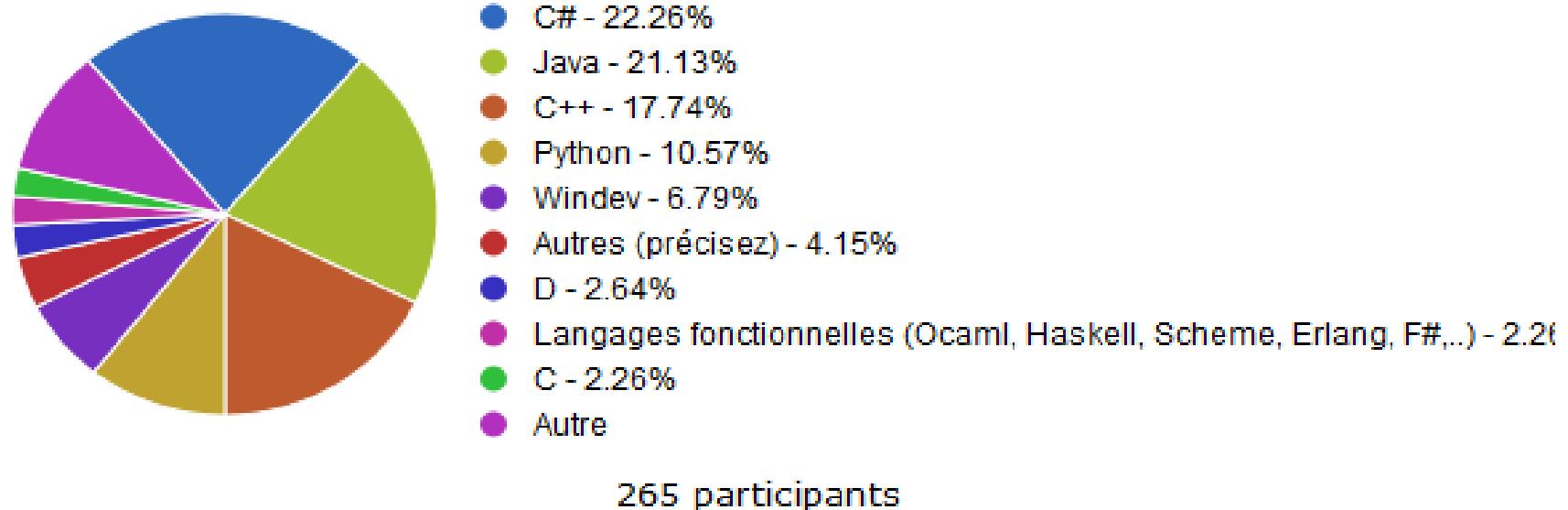
Postes en France en nov. 2014 :

- 8628 en Java
- 3672 en C++
- 3482 en C#
- 1797 en Python



Programmation : le couple C/C++ dominant 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)

Le 18 septembre 2014, par [Lana.Bauer](#), Community Manager

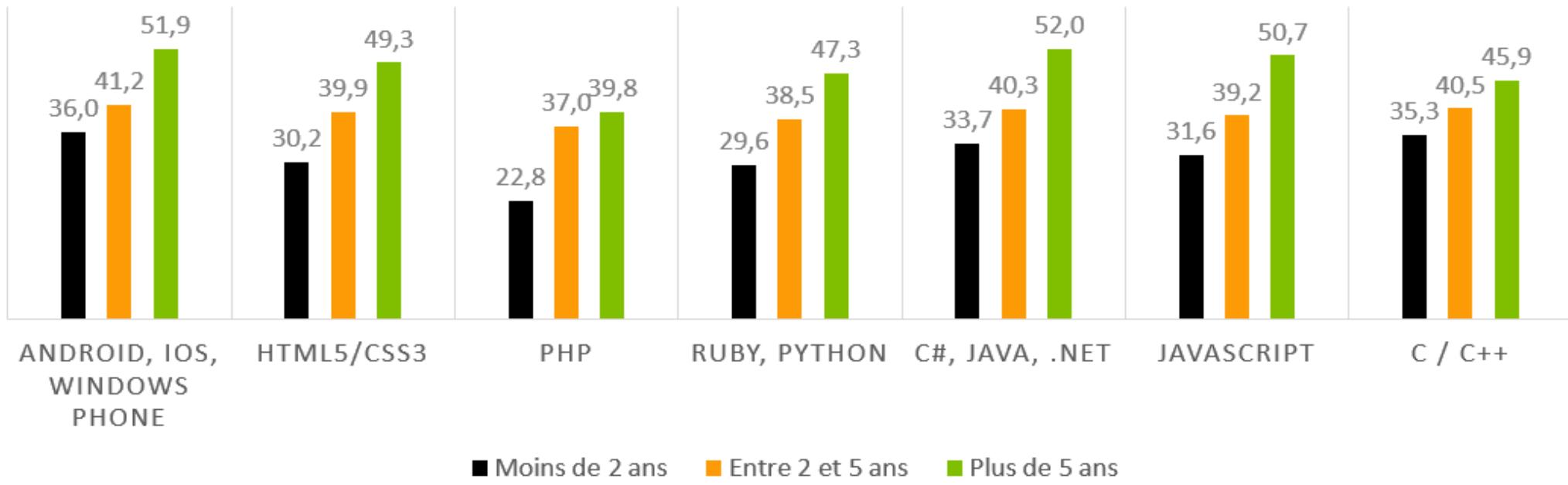


En 2013 :

- C# : 30,28 %
- C++ : 24,39 %
- Java : 21,14 %.

Graphique et statistiques issus de
<http://programmation.developpez.com/actu/75400/Quel-est-votre-langage-de-programmation-prefere-en-2014-Partagez-votre-experience-sur-le-langage-de-votre-choix/>

SALAIRE MOYEN ANNUEL PAR TECHNOLOGIE ET PAR EXPÉRIENCE EN ILE-DE-FRANCE



Issu <http://jobprod.com/salaires-developpeurs-2014>

TIOBE Index for November 2014

Nov 2014	Nov 2013	Change	Programming Language	Ratings	Change
1	1		C	17.469%	-0.69%
2	2		Java	14.391%	-2.13%
3	3		Objective-C	9.063%	-0.34%
4	4		C++	6.098%	-2.27%
5	5		C#	4.985%	-1.04%
6	6		PHP	3.043%	-2.34%
7	8	▲	Python	2.589%	-0.52%

Why Java and C++ developers should sleep well at night -
<http://www.itworld.com/article/2694396/big-data/why-java-and-c---developers-should-sleep-well-at-night.html>

Différences techniques

	C++	Objective C	Ada	Java
Encapsulation	Yes	Yes	Yes	Yes
Inheritance	Yes	Yes	No	Yes
Multiple Inherit.	Yes	Yes	No	No
Polymorphism	Yes	Yes	Yes	Yes
Binding (Early/Late)	Both	Both	Early	Late
Concurrency	Poor	Poor	Difficult	Yes
Garbage Collection	No	Yes	No	Yes
Genericity	Yes	No	Yes	No
Class Libraries	Yes	Yes	Limited	Yes

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 90 */
        tete = cour;
    }
    return tete;
}
```

Rappel : Pile et Tas (4/6)

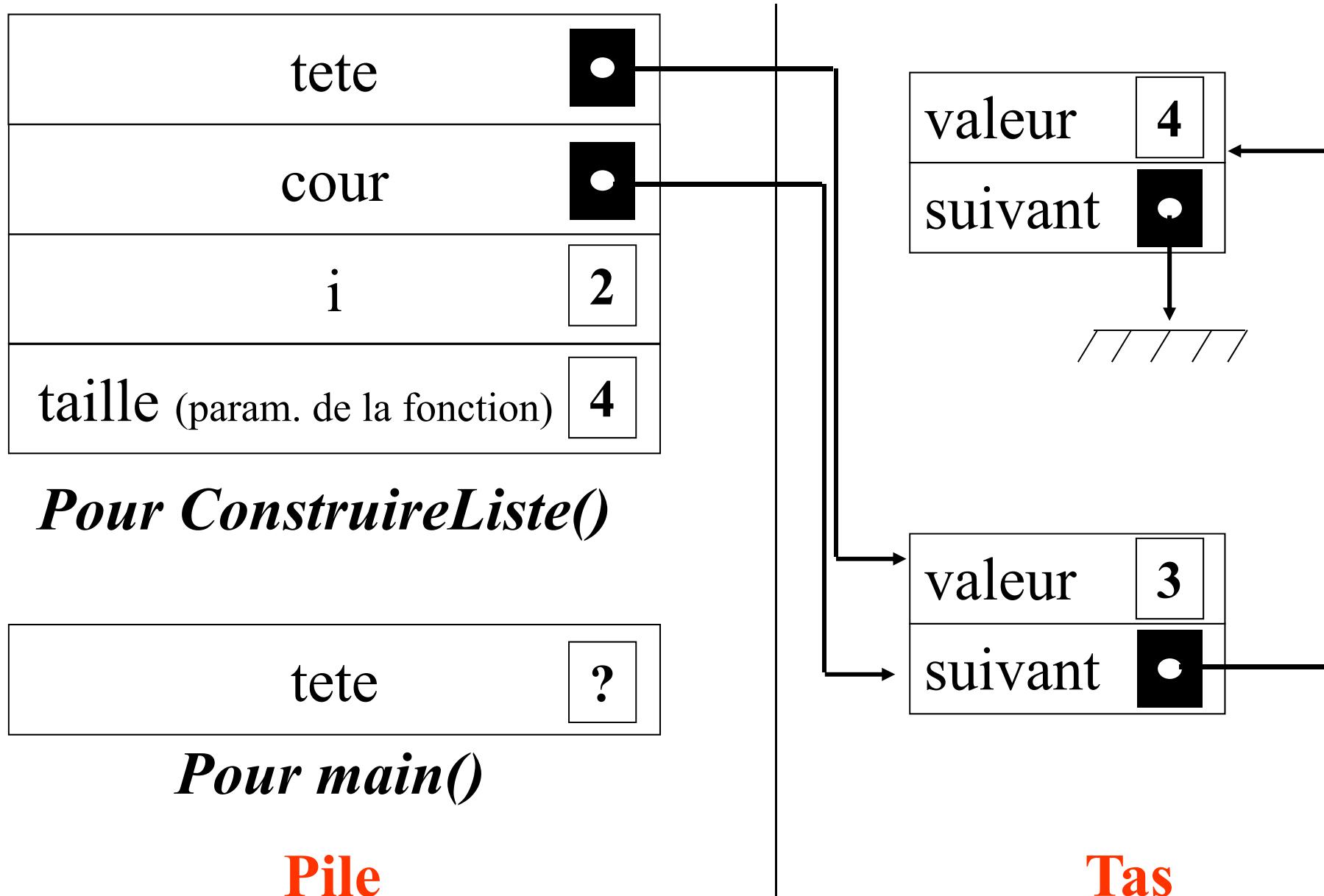
Exemple de programme en C :

```
#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 91 */
    ...
    return 1;
}
```

Rappel : Pile et Tas (5/6)

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



Pour ConstruireListe()



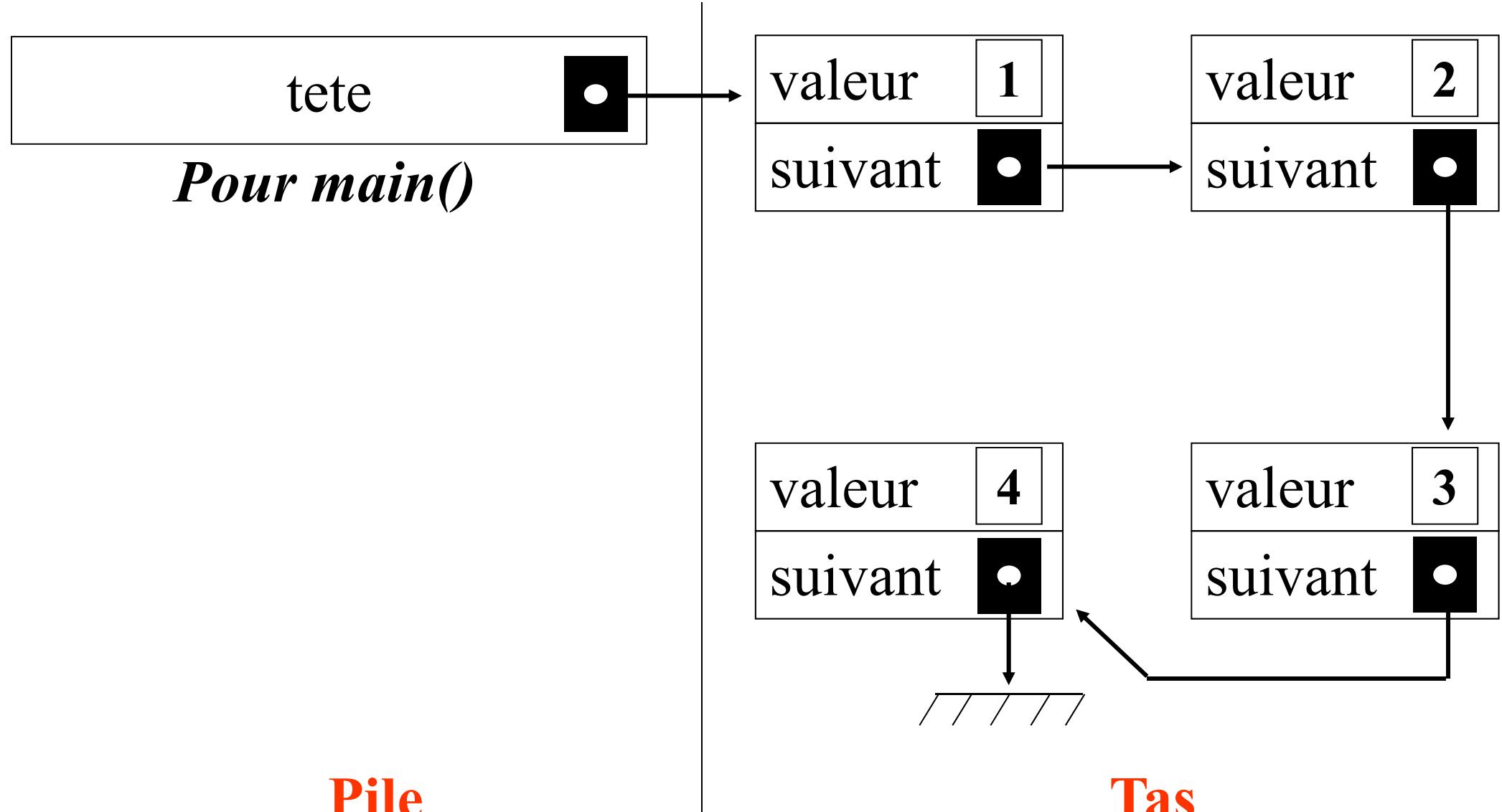
Pour main()

Pile

Tas

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 << 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) \Rightarrow 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 :

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 <i>n</i> caractères
setprecision(int n)	affichage de la valeur avec <i>n</i> 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 à 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 " ;
}
```

MonProgPrincipal.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 à des fonctions dont les bibliothèques sont manquantes

- Erreur d'exécution

Segmentation fault, overflow, division par zéro

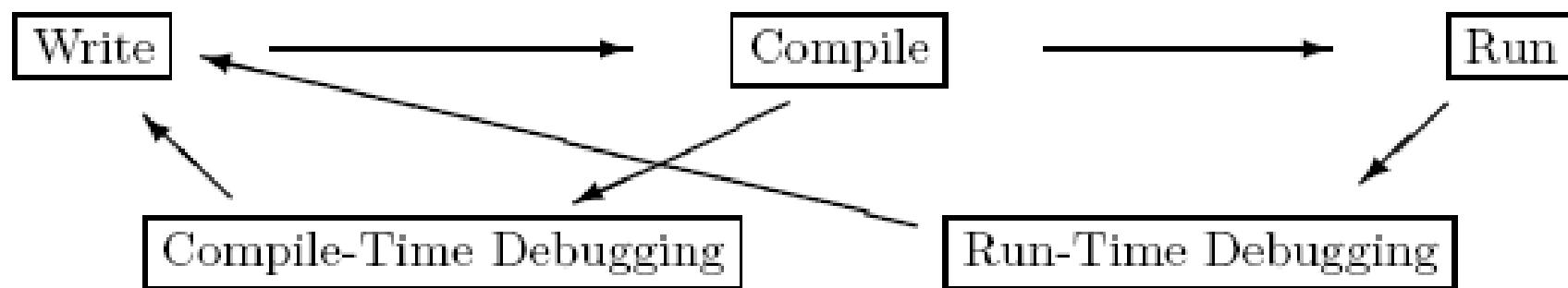
- Erreur logique

Compilateur C++

- Compilateurs gratuits (*open-source*) :
 - **Plugin C++ pour Eclipse**
<http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/lunasr1>
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 CROI INTER-UFR- version gratuite disponible Visual Express mais nécessite de s'inscrire sur le site de Windows :
<http://msdn.microsoft.com/fr-fr/express/>)
 - **Borland C++**

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 booléenne
cout << boolalpha << this_is_true; // pour que cela affiche
                                         // true ou false

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 (mantisso)
- 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/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];
```

	0	1	2	3	4
0					
1					
2					

tab_a_2dim[1][3]

```
char chaîne[] = "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 car i vaut 1

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 << " "; } Instructions exécutées tant que n est  
    { cout << n << " "; } supérieur à zéro  
    --n;  
}  
    return 0;  
}  
  
Résultat affiché :  
8 7 6 5 4 3 2 1
```



```
int main ()  
{  
    int n=0;  
    do { cout << n << " "; } Instructions exécutées une fois puis une  
    { cout << n << " "; } tant que n est supérieur à zéro  
    --n;  
}  
    while (n>0);  
    return 0;  
}  
  
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)
```

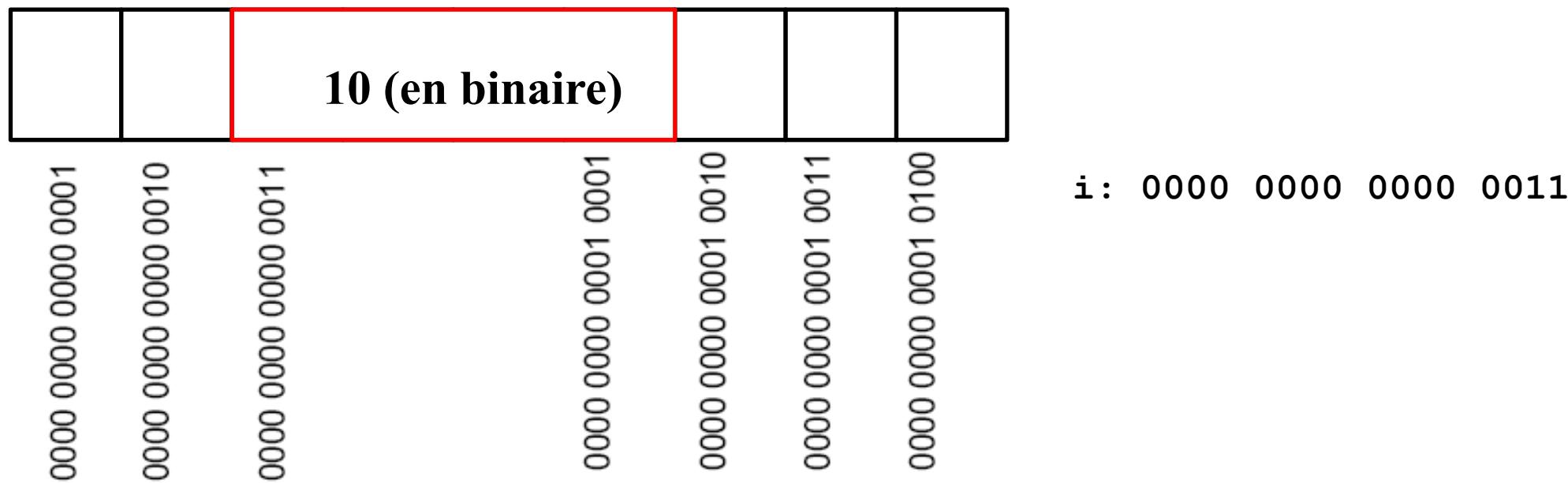
```
j = k; // i reçoit k (=5)
```



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



```
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()                                /* Pour que new retourne un
{   long taille ;                         pointeur nul en cas d'échec */
    int * adr ;
    int nbloc ;
    cout << "Taille souhaitée ? " ;
    cin >> taille ;
    for (nbloc=1 ; ; nbloc++)
    {   adr = new (nothrow) int [taille] ;
        if (adr==0) { cout << "**** manque de mémoire ****\n" ;
                        exit (-1) ;
                    }
        cout << "Allocation bloc numero : " << nbloc << "\n" ;
    }
    return 0;
}
```



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 numéro : " << nbloc << "\n" ;
    }
    return 0;
}

void deborde ()          // fonction appelée en cas de manque de mémoire
{ cout << "Mémoire insuffisante\n" ;
  cout << "Abandon de l'exécution\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[0]=4 car on a fait tab++ avant
           // car on a décalé tab à l'instruction précédente
*(tab+1)=5; // équivalent à tab[1]=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)

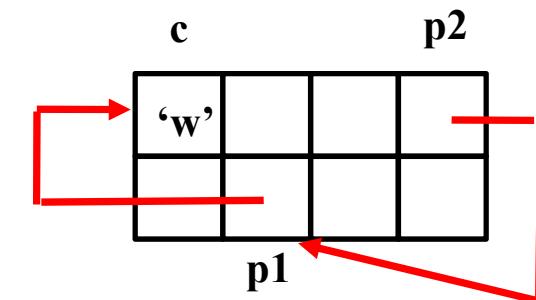
```
int *p1 = new int;  
  
int *p2 = p1 ; // p2 pointe sur la même zone mémoire que p1  
*p2=11; //=> *p1=11; car p1 et p2 pointe sur la même zone mémoire  
  
int *tab = new int [20];  
  
*tab++=3; // équivalent à *(tab++)=tab[0]=3  
          // ⇔ *tab=3; tab++;  
          // *++tab=3 ⇔ tab++; *tab=3;  
  
(*tab)++; // Ici on ne décale pas le pointeur!  
  
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
début echange: 10 20

fin echange: 20 10
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;
```



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!!

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

debut echange: 10 20

fin echange: 20 10

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<<"\".\n" << 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

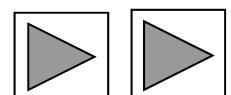
- **Classe :**
 - Regroupement de données (attributs ou champs) et de méthodes (fonctions membres)
 - Extension des structures (*struct*) avec différents niveaux de visibilité (*protected*, *private* et *public*)
- En programmation orientée-objet pure : encapsulation des données et accès unique des données 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 Crédit 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]

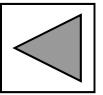
void Horaire::SetHoraire(int h, int m, int s)
{
    heure = (h >= 0 && h < 24) ? h : 0 ;
    minute = (m >= 0 && m < 59) ? m : 0 ;
    seconde = (s >= 0 && s < 59) ? s : 0 ;
}

void Horaire::AfficherMode24h()
{
    cout << (heure < 10 ? "0" : "") << heure << ":"
        << (minute < 10 ? "0" : "") << minute;
}

void Horaire::AfficherMode12h()
{
    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;}
```



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: a
also declared as `std::max' (de gcc 3.2.2)

!

Compilé 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 après 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();  
MaClasse(int);
```

```
Afficher();  
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};  
};
```

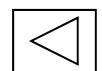
*Définition de la méthode
dans la déclaration même
de la classe*

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();
}

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

Fonction

main

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

Afficher();

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 (**pouvoir être appelée sans devoir instancier 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);
}

main
{{
    Horaire h1, h2;
    ...
    if (h1==h2) ...
}
```

Fonction
main

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));
}

main
{   Horaire h1, h2;
    ...
    if (h1==h2) ...
}
```



Un opérateur binaire peut être défini comme :

- une méthode à un argument
- une fonction 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)
{
    if (this == &h) return * this ; // auto-assignation
    heure=h.heure;
    minute = h.minute;
    seconde= h.seconde;
    return *this;
}

main
{
    Horaire h1(23,16,56),h2;
    ...
    h2=h1; ...
}
```

Fonction

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!!
```

Le langage C++ (partie II)

- 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 déposer(float);
    void retirer(float);
    float getSolde();
};

class ComptePrélèvementAuto : public CompteBanque
{
    float prelev;
public:
    void prélever();
    ComptePrélèvementAuto(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()
{
    pointcol a (2,3,4) ;
    fct (a) ; // appel de fct avec a transmis par valeur
}
```

Résultat :

```
++ point      2 3 } pointcol a (2,3,4) ;
++ pointcol  4   }
CR point      2 3 } fct (a) ;
CR pointcol  4   }

*** entree dans fct ***
```

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 Dérivée : 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();
    ...
};
```

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

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

```
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 Dérivée : 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

Résumé

- **membres publics** : ils restent publics dans une classe dérivée de manière publique, mais deviennent privés dans une classe dérivée de manière privée.
- **membres protégés** : ils restent protégés dans une classe dérivée de manière publique, mais deviennent privés dans une classe dérivée de manière privée.
- **membres privés** : ils ne sont jamais accessibles dans une classe dérivée.

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)
    b.g(c); // conversion de x en int
    ...// 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 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 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; // légale: 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 `ClasseDérivée*'
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 << "opérateur = 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(1,2,3), d;
    d=c;
}

opérateur = 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 << "opérateur = de pointcol" << endl;
        return * this ;
    }
};

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

int main()
{ pointcol c(1,2,3), 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 << "opérateur = de pointcol" << endl;
    return * this ;
}

int main()
{ pointcol c(1,2,3), d;
  d=c;
}
```

opérateur = de point
opérateur = 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 << "opérateur = de pointcol" << endl;
    return * this ;
}

int main()
{
    pointcol c(1,2,3), d;
    d=c;
}
```

```
opérateur = de point
opérateur = de pointcol
```

Le langage C++ (partie III)

- 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                                class Couleur
{
    int x;
    int y;
public:
    Point(...){...}
    ~Point(){...}
    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
    // ...
};

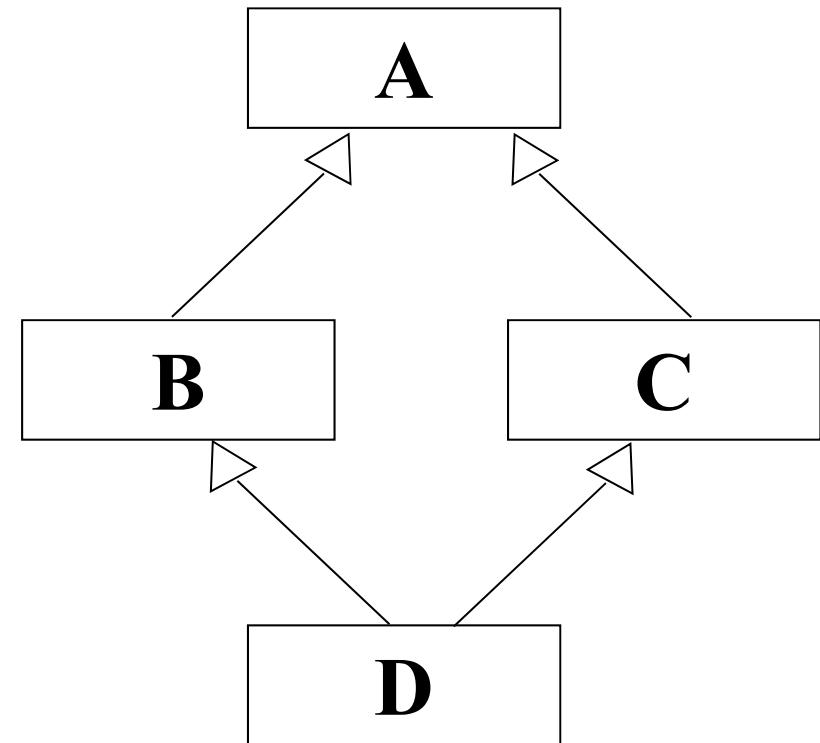
int main()
{
    C objet_c;
    // appel des constructeurs B(), A() et C()
    // ...
}
```

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 virtual 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 {....};

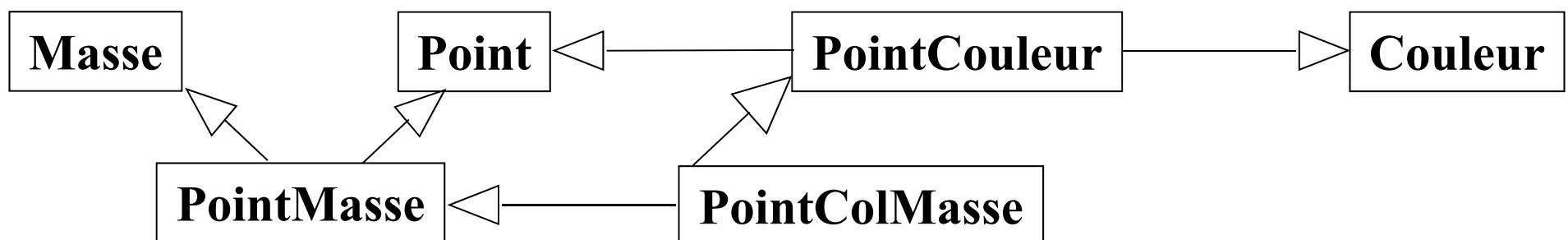
// Ici le mot-clé virtual ne doit pas apparaître!!
class D : public B, public C {....};
```

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
```

The diagram illustrates the inheritance hierarchy for Point objects. It shows four classes: Pointcoul, Pointmasse, Pointcolmasse, and Point. Pointcoul is the base class for Pointmasse and Pointcolmasse. Point is the base class for Pointcoul. The inheritance chain is shown as follows:

- Pointcoul (highlighted in red) is derived from Point (highlighted in blue).
- Pointmasse (highlighted in green) is derived from Pointcoul.
- Pointcolmasse (highlighted in orange) is derived from Pointmasse.
- Point (highlighted in purple) is derived from Pointcoul.

Each class has its own constructor sequence:

- Pointcoul: ++ Constr. defaut point
++ Constr. coul 2
++++ Constr. pointcoul 3 9 2
- Pointmasse: ++ Constr. defaut point
++ Constr. masse 100
++++ Constr. pointmasse 12 25 100
- Pointcolmasse: pointcolmasse 5 10 20
Coordonnees : 2 5
Couleur : 10
Masse : 20
- Point: ++ Constr. point 2 5
++ Constr. coul 10
++++ Constr. pointcoul 2 5 10
++ Constr. masse 20
++++ Constr. pointmasse 2 5 20
++++ Constr. pointcolmasse ++++ Constr.

```
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 : point *
type de *adp : point
type de adp : point *
type de *adp : pointcol
```

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
⇒ 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)      float minimum(float a, float b)
{ if(a<b) return a;           { if(a<b) return a;
    else return b;             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 :**
 - ⇒ Instructions utilisées par le compilateur pour fabriquer à chaque fois que nécessaire les instructions correspondant à la fonction requise
 - ↔ « 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 << "Coordonées : " << x << " " << y << endl ; }

// Méthode affiche() spécialisée pour les réels
void point<double>::affiche ()
{ cout << "Coordonées : " << 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 () ;
}

    Coordonées : 3 5
    Coordonées : 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

Le langage C++ (partie IV)

- 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 égaux à 99
vector<int> v2(7, 0) ; // vecteur de 7 entiers
vector<int> v3(t, t+6) ; // vecteur construit à 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 ill ; // itérateur sur une liste de char
il2 = lc2.begin() ;
for (ill=lc1.begin() ; ill!=lc1.end() ; ill++) { ... }
```

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

Gestion des exceptions (1/10)

- **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 \Rightarrow 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 \Rightarrow gestion simplifiée et plus sûre des erreurs

Gestion des exceptions (2/10)

- 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 n) { 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 e ; throw (e) ; }
  return adr [i] ;
}
```

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

Gestion des exceptions (3/10)

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

exception limite
```

Gestion des exceptions (4/10)

```
// 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 demandés (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 (5/10)

```
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 (6/10)

- 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 \Rightarrow appel automatique à **terminate()** qui exécute

Gestion des exceptions (7/10)

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

Classes dérivées de `exception` :

`bad_alloc`, `bad_cast`, `bad_exception`, `bad_typeid` ...

Gestion des exceptions (8/10)

- 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 (9/10)

```
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
```

Gestion des exceptions (10/10)

Possibilité d'intercepter une exception dans une méthode (ex. constructeur) :

```
class D
{ int id;
  public:
    D(int i) {if (i<0) throw (-1); else id=i;}
};
```

```
class C
{ D d;
  public:
    C(int i=0)
      try : d(i)
      {}
      catch (int) {
          // traitement exception
      }
};
```



Une exception interrompt l'exécution normale du code, mais avant de passer la main au catch, tous les objets locaux sont détruits!

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

```
#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 ;
}
```

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



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 généré 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;

// Cr ation 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   la fin
// R  cup ration de la position de la fin de fichier
length = infile.tellg();

infile.seekg (0, ios::beg); // Position du curseur au d but

// Allocation m moire 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,length); //  criture dans le fichier

delete[] buffer;

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

return 0;
}
```

Les flots (11/15)

Surcharge des opérateurs << et >> :

```
#include <iostream>
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 ostream
#include <iomanip> // Librairie pour utiliser setw
using namespace std ;

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 ostream
#include <iomanip> // Librairie pour utiliser setw
using namespace std ;

const int LGMAX = 20 ;

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

    // Déclaration d'un fichier associé au flux 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 à 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 << "Numéro 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 float d'E/S
    s << i;
    return s.str(); // retourne la string associée au float
}

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

Le langage C++ (partie V)

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 reell1, reell2;
    entier1= obj + 1.75; // instruction 1
    cout << "entier1= " << entier1 << endl;
    reell1= obj + 1.75; // instruction 2
    cout << "reell1= " << reell1 << endl;
    reell2= obj; // instruction 3
    cout << "reell2= " << reell2 << 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 reell1, reell2;

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

  reell1= obj + 1.75; // instruction 2
  cout << "reell1= " << reell1 << endl; } } } } } }

  reell2= obj; // instruction 3
  cout << "reell2= " << reell2 << 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 reell1, reell2;

  entier1= obj + 1.75; // instruction 1
  cout << "entier1= " << entier1 << endl;
  ←
  reell1= obj + 1.75; // instruction 2
  cout << "reell1= " << reell1 << endl;
  ←
  reell2= obj; // instruction 3
  cout << "reell2= " << reell2 << endl ;
  ←
  fct(obj); // instruction 4
}
```

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

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

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

```
int main()
{ A obj(1);
  int entier1; double reell1, reell2;
  // 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()
  reell1= (double)obj + 1.75; // instruction 2
  cout << "reell1= " << reell1 << endl;
  reell2= obj; // instruction 3
  cout << "reell2= " << reell2 << 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
reell1= 2.75
**Appel de double() pour l'objet d'attribut 1
reell2= 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&)

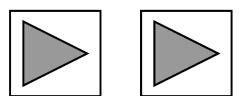
obj2B2+obj2B3;    error: no match for 'operator+' in 'obj2B2 + obj2B3'
                      note: candidates are: C& operator+(const C&, const C&)

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;
}
// Pour empêcher une conversion implicite : explicit
```





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

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)

```
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)

```
obj2B2+obj2B3;
```

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

    obj2B2+obj2B3;          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)

    obj2B1=obj2B2+ obj2B3;  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=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 paramètre c1 de valeur="ReBonjour" et de
taille 9 et un paramètre 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 "" ← Déclaration de la variable locale
**Sortie de l'opérateur + avec une variable locale d'adresse 0x22ee60 de
valeur "ReBonjour Toto" et de taille 14
```

```
**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 paramètre c1 de valeur="ReBonjour" et de
taille 9 et un paramètre 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 "" ← Déclaration de la variable locale
**Sortie de l'opérateur + avec une variable locale d'adresse 0x22ee60 de
valeur "ReBonjour Toto" et de taille 14
```



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 ** ← a4
**Destructeur pour l'objet 0x22ee70 de valeur "ReBonjour Toto" et de
taille=14 ** ← a3
**Destructeur pour l'objet 0x22ee80 de valeur " Toto" et de taille=5 **
**Destructeur pour l'objet 0x22ee90 de valeur "ReBonjour" et de taille=9
```

Le langage C++ (partie VI)

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](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"

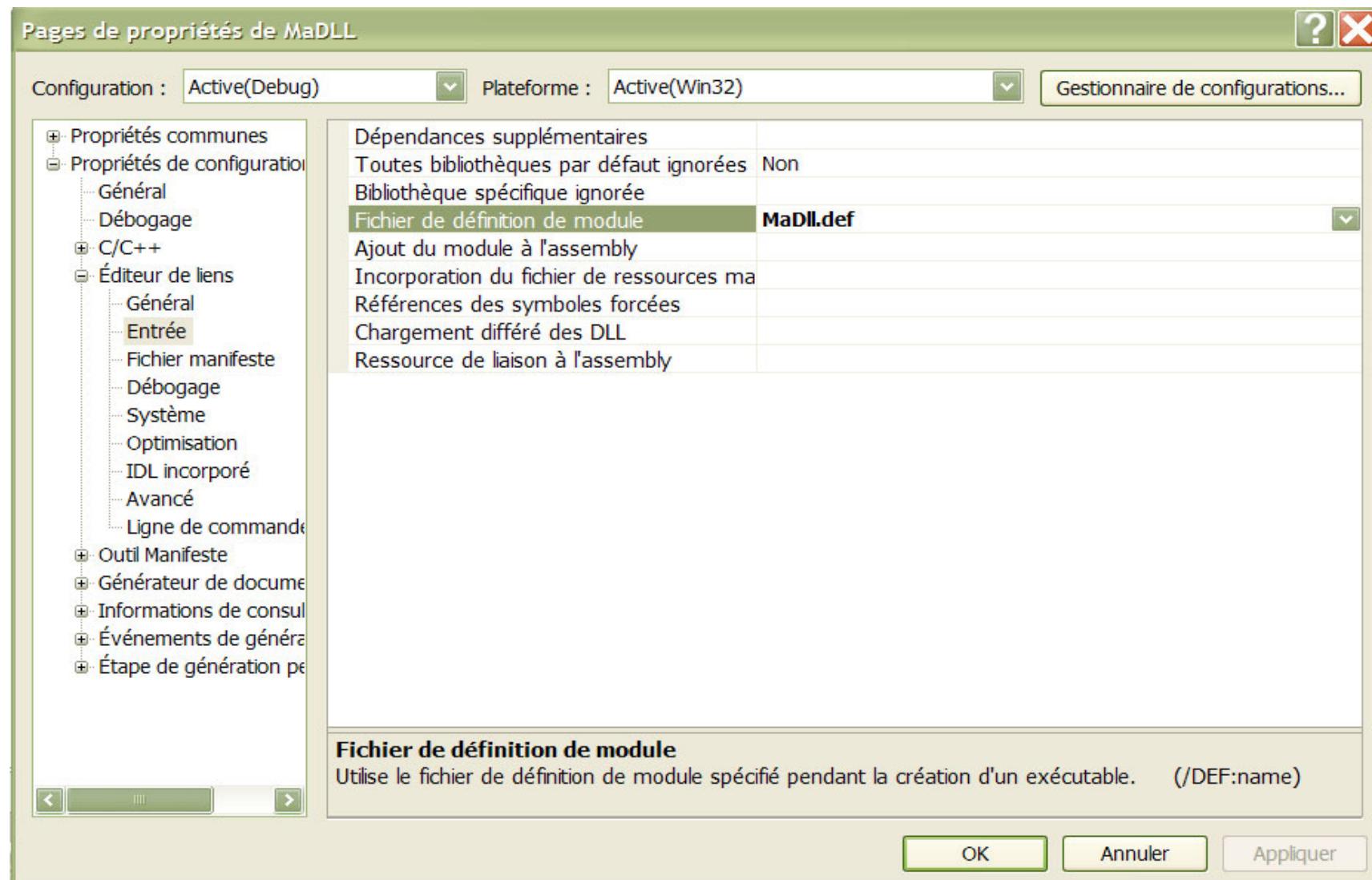
double __stdcall carre (double& arg)
{return arg*arg; }

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 .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
    Ne pas oublier d'inclure <iostream> et
    cout << result << endl;
    <windows.h> !!
```

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

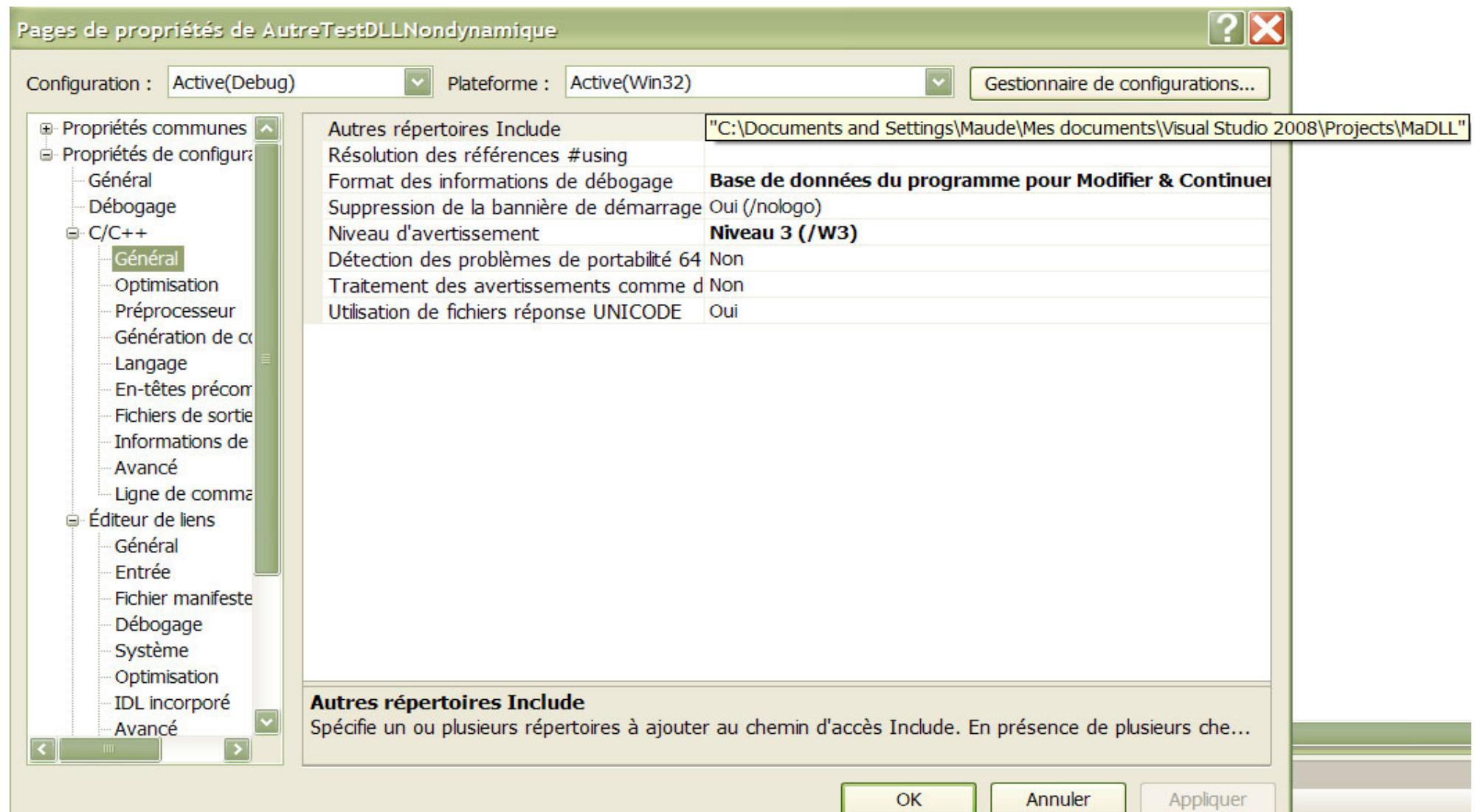
```
#include "MaDll.h"      // Inclusion du .h de la DLL
using namespace std;
int main()
{
    double r,d;
    cin >> r;
    cin >> d;
    cout <<"carre(" << r << ")=" << carre(r) << endl;
    cout <<"addition(" << r << ","<< d << ")=" << addition(r,d) << endl;
}
```

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*)

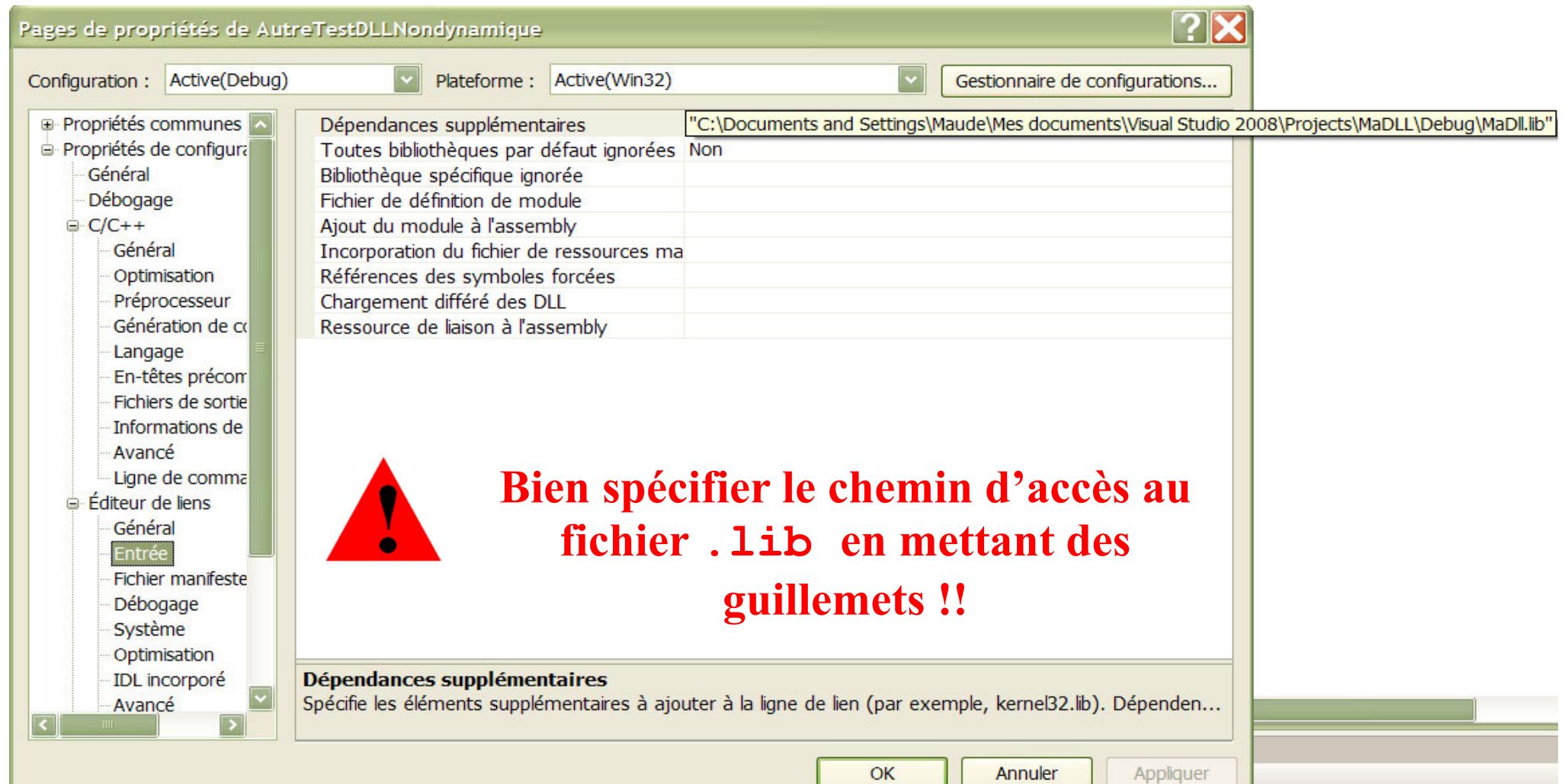
Exemple programme C++ utilisant une DLL de manière statique (2/4)

- Où trouver le .h de la bibliothèque :



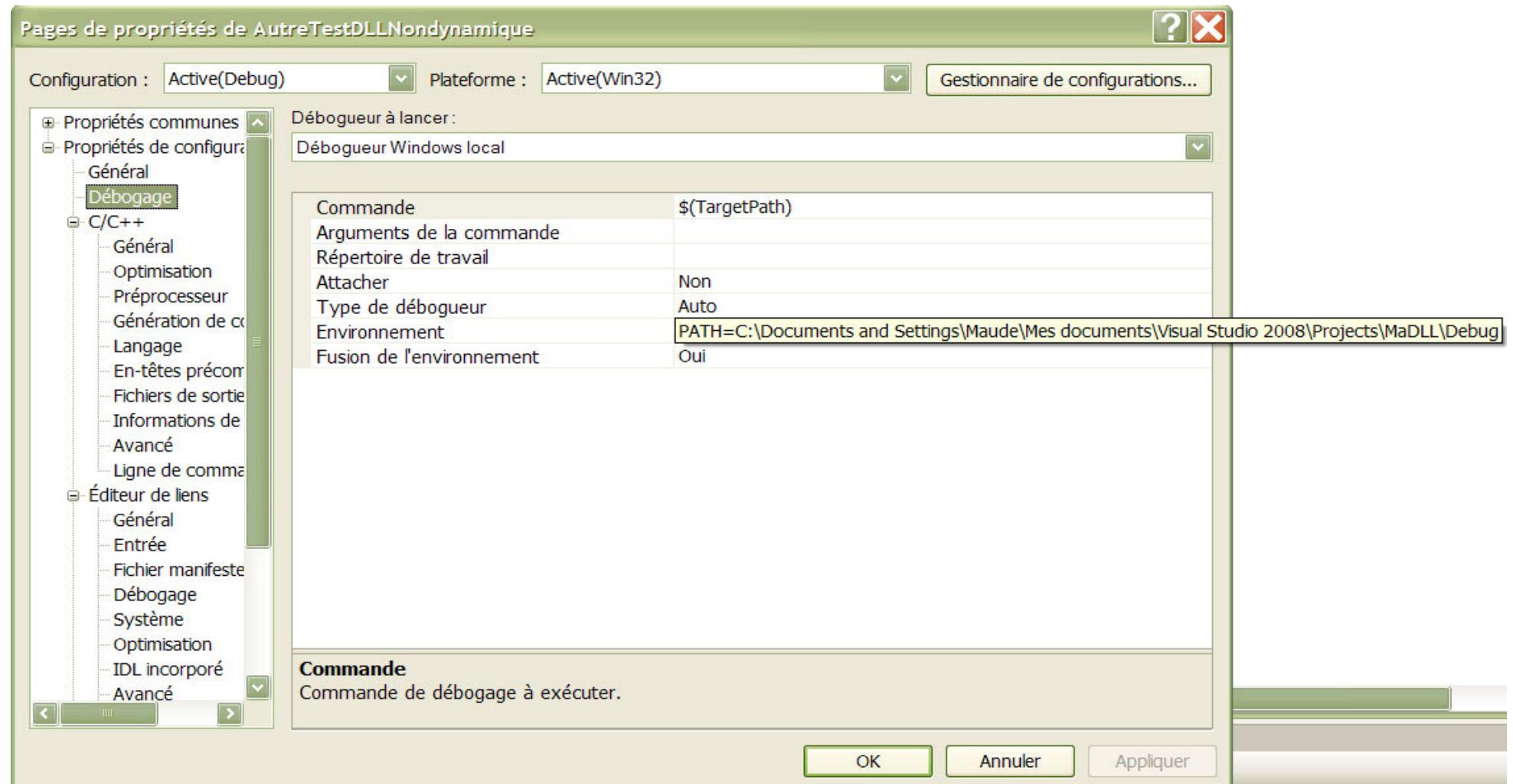
Exemple programme C++ utilisant une DLL de manière statique (3/4)

- 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++ lié 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();

}
```