

## Chapitre 4

### Spécificités du C++

## 1. Commentaires

### comme en C

Pour couvrir toute une zone (une ou plusieurs lignes à la fois)

```
/* quelque chose à mettre en commentaires  
   la suite de ce commentaire  
   enfin la dernière ligne! */
```

sinon

```
#ifndef un_commentaire  
    quelque chose à mettre en commentaires  
    sur plusieurs lignes aussi!  
#endif
```

### à la C++

Pour couvrir une seule ligne

```
// quelque chose à mettre en commentaires que sur une ligne!
```

Faire attention aux commentaires imbriqués,

```
if (b>a) {  
    // temp = a; //swap de a & b ← inutile mais OK!  
    // etc.  
}
```

Par contre dans l'exemple suivant, c'est une belle pagaille! Des commentaires imbriqués qui vont générer une erreur à la compilation.

```
if (b>a) {  
    /* temp = a; /* swap de a & b */ ← C'est l'équivalent de /* temp = a; swap de a & b */  
    etc. */ ← L'erreur est à ce niveau. Les caractères */ vont se retrouver seuls, d'où l'erreur de compilation!  
}
```

On peut imbriquer les deux styles de commentaires :

- Entre /\* et \*/ le style // n'a pas une signification spéciale.
- Dans // le style de commentaires /\* et \*/ n'a pas de signification spéciale.

Trop de commentaires tue le commentaire! Pas besoin d'en rajouter !

```
// Boucle for ! ← Commentaire inutile! On s'est bien que c'est une boucle for !  
for (int i=0 ; i<10 ; i++) {  
    }  
}
```

Dans le précédent exemple, il était plus judicieux de nous dire par exemple à quoi peut-elle servir cette boucle for.

## 2. Déclarations locales

En C, déclarations toujours avant les instructions ou bien dans {}

En C++, possibilité d'avoir les déclarations près de l'utilisation,

```
#include <iostream>  
  
using namespace std ;  
  
int main(){  
    int N=10; // déclaration avant l'instruction  
  
    /* La variable i est déclarée au moment de son utilisation.  
       Cette variable est visible uniquement dans le bloc {}  
    */  
  
    for (int i=0;i<N;i++) {  
  
        int j=10; // La variable j est déclarée dans le bloc {}  
        i+=j;  
        cout << "i vaut: " << i << endl;  
  
    } // Fin de portée de i  
    return 0;  
}
```

Affiche en sortie:

```
i vaut : 10
```

Avantages => déclarations près de l'utilisation (voir l'exemple précédent),

Initialisation avec expression inconnue à l'entrée du bloc,

```
int taille;
cin >> taille;
int tabl[taille]; // assez dangereux, car (taille ≤ 0) => des tests!
```

La portée => fin du bloc.

### 3. Déclarations dans les instructions de contrôle

```
if (int x=fonction()) {traitement(x);} ← OK
```

```
while(int x=1) {x=fonction();} ← OK
```

(La déclaration est valable aussi pour un `switch`, à faire par vous).

```
do {x=fonction();} while(int x=1) ← FAUX, la variable x a été déclarée après son utilisation.
```

### 4. Généralisation de la résolution de portée

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
int i=500;
```

```
cout << "i du main vaut avant: " << i << endl;
```

```
for (int i=0; i< 1; i++) cout << "i boucle vaut: " << i << endl;
```

```
cout << "i du main vaut après: " << i << endl;
```

```
return 0;
```

```
}
```

En sortie:

```
i du main vaut avant: 500
i boucle vaut: 0
i du main vaut après: 500
```

Il est impossible d'afficher la valeur de la variable `i` de la méthode `main` à partir de la boucle `for`.

```
#include <iostream>

using namespace std;

int i = 1000;

int main() {
    cout << "i du main vaut avant: " << i << endl;
    cout << "i en dehors du main vaut avant: " << ::i << endl;
    for (int i=0; i < 1; i++)    cout << "i boucle vaut: " << i << endl;
    cout << "i du main vaut après: " << i << endl;
    cout << "i en dehors du main vaut après: " << ::i << endl;

    return 0;
}
```

En sortie:

```
i du main vaut avant: 500
i en dehors du main vaut avant: 1000
i boucle vaut: 0
i du main vaut apres: 500
i en dehors du main vaut apres: 1000
```

**::** est l'opérateur unaire de résolution de portée ou bien l'opérateur de visibilité (scope qualifier operator) dans cet exemple, **::** fait référence à l'espace de nom anonyme.

5. Espace de nom (namespace)

Les namespaces =>

- Permettent de diviser l'espace global,
- Aident le programmeur à développer des nouveaux composants logiciels sans générer de conflits de noms avec les composants existants.

Déclaration

```
namespace nom_espace{ // bloc d'instructions }
```

Utilisation

```
using namespace nom_espace;
```

#include <iostream>

```
namespace { int i = 1000; } // espace de nom anonyme
namespace test { int i=2500; } // espace de nom test
using namespace std; // on charge tout le contenu de l'espace standard
```

```
int main() {
    cout << "i de l'espace anonyme vaut: " << ::i << endl; // 1000
    cout << "i de test vaut: " << test::i << endl; //2500

    return 0;
}
```

### 5.1. Accès à tous les membres d'un namespace

```
#include <iostream>

namespace test {int i=2500;int j=200;}

using namespace test; // utilisation de tout le contenu de test
using namespace std; // utilisation de tout le contenu de std

int main() {
    cout << "i de test vaut: " << test::i << endl; //2500
    cout << "que vaut j? " << ::j << endl; //200
    return 0;
}
```

Si la ligne: **using namespace test;** était absente du programme =>

Le compilateur va afficher: undeclared variable "j" (first use here)

### 5.2. Emploi d'un membre de namespace individuel

```
#include <iostream>

namespace test {int i=2500;}

using test::i; // définition de i pour le reste du programme
using namespace std;

int main() {
    cout << ::i; // 2500
    return 0;
}
```

### MAIS ...

```
#include <iostream>

int i = 1000; // première définition de i

namespace test {int i=2500;}

using test::i; // définition de i ..... or la variable i a été déjà définie.
using namespace std ;

int main() {
    cout << ::i;
    return 0;
}
```

## 6. Type bool

En C, il n'y a pas de type booléen, on utilise `int` ou bien un type prédéfini:

```
#include <stdio.h>
typedef enum { false=0, true=1} Boolean;

int main() {
    Boolean valide = false;
    printf("%d\n", valide);
    valide = true;
    printf("%d\n", valide);
    return 0;
}
```

Par contre en C++, le type `bool` a été défini. Une variable du type `bool` peut prendre la valeur `0` ou `1`, où:

0 → false  
1 → true

```
#include <iostream>

using namespace std;
int main() {
    bool valide=false;
    cout << valide << endl; // affiche: 0
    valide = true;
    cout << valide << endl; // affiche: 1
    return 0;
}
```

## 7. Surdéfinition de fonctions

Un nom (de fonction, d'opérateur, etc.) est surdéfini s'il désigne plus d'une chose à la fois.

5/2 division entière (2, et le reste est perdu)

5.0/2.0 division réelle (2.5)

opérateur / a un double rôle: la division des nombres entiers et des nombres réels.

En C++, on peut définir différentes fonctions ayant le même nom mais le nombre et le type de paramètres sont différents.

```
#include <iostream>

using namespace std;

void affiche(int x) {cout << "un entier:" << x << endl;}
void affiche(double w) {cout << "un double: " << w << endl;}

int main() {
    int n=100;
    double z=259.6;

    affiche(n); // appel de affiche(int x)
    affiche(z); // appel de affiche(double w)
    return 0;
}
```

S'il y a plusieurs arguments, on essaye chacun séparément. S'il y a une fonction qui est parmi les meilleures, le compilateur la prend, sinon erreur.

```
#include <iostream>
using namespace std;
double exemple(double x, int w){
    cout << "configuration -A-: " << x << " " << w << "\n";
    return x;
}

int exemple(int x, double w){
    cout << "configuration -B-: " << x << " " << w << "\n";
    return x;
}

int main() {
    double a=150.8,w;
    int b=200,v;

    w = exemple(a,b);           // configuration -A- double exemple(double, int)
    v = exemple(b,a);           // configuration -B- int exemple(int, double)

    v = exemple(a,a);           // conversion de int vers double du 1er argument -> configuration A (ou) conversion
                                // de double vers int du 2er argument -> configuration B d'où erreur (ambiguïté),
                                // impossible de choisir entre les deux configurations A & B. Le type de retour n'est
                                // pas considéré lors du processus de conversion de types.

    return 0;
}
```

Pour un seul argument, le compilateur essaie dans l'ordre:

- correspondance exacte de types.
- promotion numérique

origine	conversion vers
char, short	int
int	long
float	double

```
short zz=12;
affiche(zz); // appel de affiche(int) si affiche(short) n'existe pas.

int ww=15;
affiche(ww); // appel de affiche(double) si affiche(int) ou affiche(long) n'existent pas.
```

- conversion dégradante

origine	conversion vers
int	short
double	float

## 8. Référence

Référence est un alias à un espace mémoire.

Syntaxe: Type& var = valeur;

```
#include <iostream>

int main() {
    using namespace std ;
    int i=0, j=20;
    int& valeur =i;
    cout << "i: " << i << " valeur: " <<valeur << endl;
    i = 50;
    cout << "i: " << i << " valeur: " <<valeur << endl;
    valeur = j;
    cout << "i: " << i << " valeur: " <<valeur << endl;
    return 0;
}
```

en sortie:

```
i: 0 valeur: 0
i: 50 valeur: 50
i: 20 valeur: 20
```

© Mohamed N. Lokbani

v3.00

POO avec C++

### Chapitre 4 : Spécificités du C++

- Les variables de référence doivent être initialisées au moment de leurs déclarations.

```
#include <iostream>

int main() {
    int i=0;
    int& valeur;
    // Erreur valeur a été déclarée comme référence mais n'a
    // pas été initialisée. La manière correcte est: int& valeur = i;
    std::cout << valeur << " " << i << endl;
    return 0;
}
```

- Ne peuvent être réaffectées (réassignées) comme alias à d'autres variables.

```
#include <iostream>

int main() {
    int i=0, j=20;
    int& valeur = i; // valeur a été déclarée comme référence alias à la variable i
    valeur = j; // valeur n'a pas été réaffectée à j, elle est toujours alias de i.
    // Cette ligne sert uniquement à affecter 20 (donc le contenu de j) à valeur (donc i).
    std::cout << valeur << " " << i << " " << j << std::endl;
    return 0;
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++



- Une fois que la référence est déclarée alias d'une autre variable, toutes les opérations qu'on croit effectuer sur l'alias (c'est à dire la référence) sont en fait exécutées sur la variable d'origine.

```
#include <iostream>

int main() {
    int i=0;
    int& valeur = i; // valeur = i =0
    valeur++; // valeur = 1
    std::cout << i << std::endl; // affiche: 1
    return 0;
}
```

### Danger ...

```
// bugref.cpp
#include <iostream>

int main() {
    double i=0;
    int& valeur = i; // valeur = i =0
    valeur = 250; // valeur = 250
    std::cout << i << std::endl; // affiche: 0
    return 0;
}
```

Le compilateur compile le programme sans erreur et génère les warnings suivants:

```
bugref.cpp: In function 'int main()':
bugref.cpp:5: warning: converting to 'int' from 'double'
bugref.cpp:5: warning: argument to 'int' from 'double'
bugref.cpp:5: warning: initialization of non-const reference type 'int&' from
rvalue of type 'double'
```

Le compilateur a créé une variable temporaire (signalée dans le second warning), comme suit:

```
double i=0;
int temp = i; // conversion double vers int et initialisation de temp avec la valeur de i
int& valeur = temp; // valeur est une référence à temp (mais pas i)
```

Modifier **valeur** revient à modifier **temp**, mais pas **i** car il n'y a aucun lien entre **i** et **temp**. De ce fait, faire attention de déclarer le type de la référence identique à la variable référencée, pour éviter des surprises lors de l'exécution de votre programme.

Utilité: En général, pour éviter d'avoir affaire aux pointeurs. En particulier, lors des passages de paramètres (des classes, des structures), afin d'éliminer la surcharge causée par la copie de grandes quantités de données.

```
#include <iostream>

typedef struct test {
    char echantillon[200];
    int taille;
} TEST;

TEST Maj (TEST x) { // Mise à Jour
    x.taille = 200;
    return x;
}

int main() {
    TEST entree, sortie;

    sortie = Maj(entree); // il y aura une copie complète de la structure entreee dans la variable x.
                        // Si la structure était plus complexe, imaginez le temps nécessaire pour
                        // réaliser cette copie.

    std::cout << sortie.taille << std::endl; // affiche: 200
    return 0;
}
```

Réécrive ce code en utilisant le passage par référence tel que décrit dans les paragraphes précédents. La variable sortie a été utilisée pour montrer que la fonction Maj s'est exécutée correctement. Dans ce contexte, la variable sortie est-elle utile lors d'un passage par référence?

## 9. Passage de paramètres

2 types de paramètres:

- paramètres réels: dans l'appel de fonction
- paramètres formels: dans l'entête de la fonction

```
#include <iostream>

int multiplication(int x,int y){
    return (x*y);
}

int main(){
    std::cout << multiplication(3,4)<< std::endl;
    return 0;
}
```

formels

réels

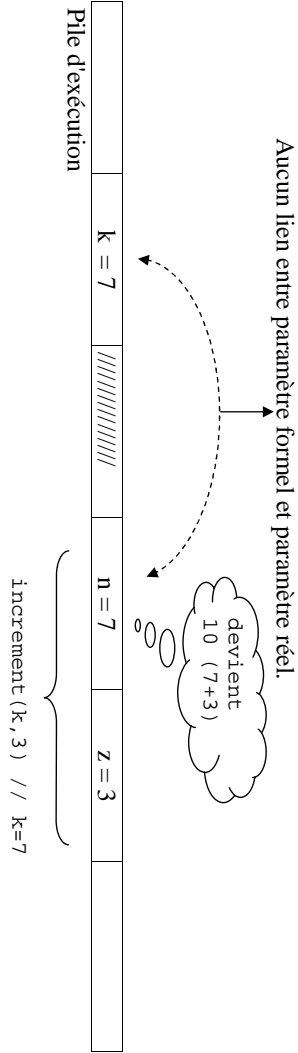
En C++, 3 types de passages de paramètres:

- par valeur (existe aussi dans le langage C)
- par adresse (existe aussi dans le langage C)
- par référence (uniquement en C++)

9.1. Passage de paramètres par valeur

```
#include <iostream>
void increment(int n, int z) {
    n=n+z;
}

int main() {
    int k=7;
    increment(k,3);
    std::cout << k << std::endl; // affiche 7, mais pas 10 (i.e. 7+3)
    return 0;
}
```



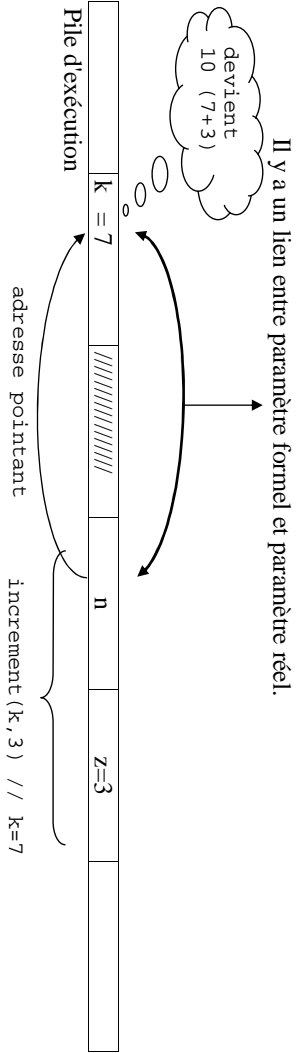
Paramètre formel **n** est initialisé avec la valeur du paramètre réel **k=7**.

9.2. Passage de paramètres par adresse

Pour modifier le paramètre réel, on passe son adresse plutôt que sa valeur.

```
#include <iostream>
void increment(int *n, int z) {
    *n= *n + z;
}

int main() {
    int k=7;
    increment(&k,3);
    std::cout << k << std::endl; // affiche 10, car c'est l'adresse qui a été passée.
    return 0;
}
```



Paramètre formel **n** est initialisé avec le contenu de l'adresse pointant sur **k**.

\* Il y a un lien potentiel entre les paramètres formels de la fonction et les paramètres réels.

\* Gestion des adresses est effectuée par le programmeur (lourd et difficile à lire).

### 9.3. Passage de paramètres par référence

\* En C++ le compilateur peut se charger lui même de la gestion des adresses.

\* Le paramètre formel est un alias de l'emplacement mémoire du paramètre réel.

```
#include <iostream>

void increment(int& n, int z) {
    n = n + z;
}

int main() {
    int k=7;
    increment(k,3);
    std::cout << k << std::endl; // affiche 10, car c'est l'adresse qui a été passée.
    return 0;
}
```

#### \* Exemple du swap

Que va afficher le programme suivant, sur la sortie standard? Puis, réécrive le code suivant, dans les 2 cas suivants:

```
-1- void swap(int a, int& b){...}
-2- void swap(int& a, int& b){...}

#include <iostream>

void swap(int a, int b){
    int c;
    c=a;a=b;b=c;
}

int main() {
    int i=4, j=7;
    swap(i, j);
    std::cout << i << " " << j << std::endl;
    return 0;
}
```

## 10. Passage d'un tableau

2 configurations possibles:

### 10.1. par pointeur

```
#include <iostream>

void fonction(int* tableau, int taille) {
    for (int i=0; i<taille;i++) std::cout << tableau[i] << " ";
    std::cout << std::endl;
}

int main() {
    int tableau[2] = {1,2};
    fonction(tableau,2);
    return 0;
}
```


### 10.2. par semi-référence

```
#include <iostream>

void fonction(int tableau[], int taille) {
    for (int i=0; i<taille;i++) std::cout << tableau[i] << " ";
    std::cout << std::endl;
}

int main() {
    int tableau[2] = {1,2};
    fonction(tableau,2);
    return 0;
}
```

au lieu d'un pointeur



## 11. Quand utiliser une référence

(voir aussi P. Prados @ <http://perso.club-internet.fr/pprados/Langage/CPP/ref/ref.html>)

### 11.1. Comme attribut

(voir paragraphe sur les références, page 57).

### 11.2. Comme un paramètre

Une question difficile à trancher

- si le paramètre ne doit être que consulté par la méthode, il faut recevoir une référence (assez souvent constante) ;
- si le paramètre qu'une méthode reçoit peut ne pas être présent, i.e. l'utilisateur peut envoyer la valeur NULL, il faut recevoir un pointeur.

```
void f(const TEST& x){ // x est une référence
    if (&x == NULL) ... // À ne pas faire ... référence ne peut pas être NULL.
}
```

Utilisez dans ce cas la surcharge ...

```
void f(const TEST& x){ // traite le cas d'un passage par référence
    // ...
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

## Chapitre 4 : Spécificités du C++

70

```
void f(const void* x){ // traite le cas d'un pointeur NULL
}
```

Essayez de cogiter sur un exemple complet.

### 11.3. en retour d'une méthode (ou fonction)

```
#include <iostream>
```

```
int& fonction(int &a) {
    a += 10;
    return a;
}
```

```
int main() {
    int val, test = 20;
    val = fonction(test);
    std::cout << "val: " << val << " " << "test: " << test << std::endl;
                                // affiche: val: 30 test: 30
    fonction(test) = 250;
    std::cout << test << std::endl; // affiche: 250
    return 0;
}
```

### Danger ...

```
int& exemple() {
    int i=10; // i est une variable locale de la fonction exemple, durée de vie entre {}
    return i; // exemple va retourner un alias sur une variable qui va cesser d'exister!
}
```

© Mohamed N. Lokbani

v3.00

POO avec C++

## 12. Arguments par défaut

En C++, nombre de paramètres réels ... nombre de paramètres formels.  
On commence à omettre à partir de la fin.  
On peut spécifier des valeurs par défaut

```
#include <iostream>

double exemple(double x, char *msg = "rien", double z=200.5) {
    std::cout << x << " " << msg << "\n";
    return z;
}

int main() {
    double a=150.8,w;

    w = exemple(a,"salut",-670.9);
    std::cout << w << std::endl;

    // Omission à partir de la fin ...
    // Omission du dernier argument, la fonction prendra la valeur par défaut du 3e argument.
    w = exemple(a,"salut");
    std::cout << w << std::endl;

    // Omission des deux derniers arguments, la fonction prendra la valeur par défaut du 2e et 3e argument.
    w = exemple(a);
    std::cout << w << std::endl;
    return 0;
}
```

### affichage en sortie

```
150.8 salut
-670.9
150.8 salut
200.5
150.8 rien
200.5
```

### Erreur de compilation ...

Un appel: `w = exemple()` ; provoquera une erreur de compilation, car aucune valeur par défaut n'a été définie dans l'entête de la fonction `exemple` pour le premier argument.

### 13. Opérateurs new et delete

Allocation dynamique de l'espace mémoire

Langage	allouer	supprimer
C	malloc	free
C++	new	delete

```
new T retourne un pointeur de type T* non nul;  
delete P libère l'espace mémoire pointé par P.
```

	allouer	supprimer
pour un élément	new T	delete P
pour un tableau de taille DIM	new T[DIM]	delete [ ] P

new T(valeur) allocation d'un pointeur de type T\* non nul, et initialisation du pointeur avec valeur (valable uniquement pour un élément).

```
#include <iostream>  
  
int main() {  
  
    double* ptr; // pointeur du type double  
  
    ptr = new double; // allocation de l'espace pour stocker un double  
  
    *ptr = 126789.89; // affectation d'une valeur double  
  
    int* tab = new int[20]; // sur une ligne, allocation d'un tableau de 20 int  
  
    tab[5] = 78; // affectation d'une valeur à tab[5]  
  
    int* autre = new int(250); // allocation d'un espace mémoire pour stocker un int  
                                // et initialisation de cet espace.  
  
    delete ptr; // libération de l'espace alloué pour ptr  
    delete autre; // libération de l'espace alloué pour autre  
    delete [ ] tab; // libération de l'espace alloué pour le tableau tab  
  
    return 0;  
}
```



14. Fonctions inline (en ligne)

À la page 30, nous avons montré l'utilisation du #define pour la substitution de texte, une autre utilisé du #define est la définition de macros.

Une macro est une pseudo fonction, substituée dans tout le programme pendant le prétraitement (préprocesseur) avant la compilation. Elle évite le coût d'un appel de fonction, en contrepartie le code exécutable devient plus volumineux.

```
#include <iostream>

#define abs(x) (x<0)?-x:x

int main() {
    int n, a=-4, b=6;

    n = abs(a);

    std::cout << n << std::endl; // affiche 4

    n = abs(a)*2;
    std::cout << n << std::endl; // affiche 4

    n = abs(b)*2;
    std::cout << n << std::endl; // affiche 12

    return 0;
}
```

```
n = abs(a)*2; est transformée en:

n = (a<0) ? -a : a*2;

n = (-4<0) ? 4 : 8; // puisque -4 est inférieure à 0, on retourne 4

Une solution à ce problème serait d'écrire la macro comme suit:

#define abs(x) ((x<0)?-x:x)
```

Mais quand est-il pour le programme suivant, que va-t-il afficher après son exécution?

```
#include <iostream>

#define MAX(a,b) (((a) > (b)) ? (a) : (b) )

int main() {
    int a=0,b=1; // Essayer aussi avec b=0.

    std::cout << MAX(a,b++) << std::endl;

    return 0;
}
```

Le C++ a introduit les fonctions `inline` afin de remédier aux problèmes posés par l'utilisation des macros, mais tout en gardant, comme dans le cas des macros, la gestion des appels de fonctions faible (surtout pour les petites fonctions). Il n'y aura pas d'accès à la table des fonctions car toutes les fonctions sont `inline` dans le programme.

Le qualificatif `inline` recommande au compilateur de faire une copie du code de la fonction en place => code compilé plus long ... **MAIS** le compilateur peut prendre la décision de "désactiver" le qualificatif `inline` d'une fonction, s'il estime qu'il perdra moins de temps à y accéder via la table des fonctions, que de recopier son code.

syntaxe: `inline fonction_donnée`

```
#include <iostream>

int inline abs(int x){
    return (x<0)?-x:x;
}

int main() {
    int n, a=-4;

    n = abs(a)*2; // le programme calcule d'abord abs(a) puis le résultat est * 2
    std::cout << n << std::endl; // affiche 8

    return 0;
}
```

Pourquoi `inline` au lieu d'une fonction tout court? Afin d'éviter l'accès à la fonction à travers la table des fonctions (là où est stocké un index vers la fonction) d'où un gain de temps.

### Solution au problème posé en page 47.

```
switch (char c=mot[0]) {
    case 'a':
        // quelque chose
        break;
    default:
        // quelque chose
        break;
}
```

### Solution au problème posé en page 61.

```
#include <iostream>

typedef struct test {
    char echantillon[200];
    int taille;
} TEST;

void Maj (TEST& x){
    x.taille = 200;
}

int main() {
    TEST entree;
    Maj(entree);
    std::cout << entree.taille << std::endl; // affiche: 200
    return 0;
}
```

**Solution au problème posé en page 66.**

1/ les variables i et j sont passées par valeur, de ce fait, en sortie nous obtenons: 4 7

2/

```
-1- void swap(int a,int& b){...}
```

```
#include <iostream>
```

```
void swap(int a,int& b){
```

```
int c;
```

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

```
}
```

```
int main() {
```

```
int i=4,j=7;
```

```
swap(i,j); // i est passée par valeur, j par référence
```

```
std::cout << i << " " << j << std::endl; // 4 4
```

```
return 0;
```

```
}
```

**Chapitre 4 : Spécificités du C++**

```
-2- void swap(int& a,int& b){...}
```

```
#include <iostream>
```

```
void swap(int& a,int& b){
```

```
int c;
```

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

```
}
```

```
int main() {
```

```
int i=4,j=7;
```

```
swap(i,j); // i et j sont passées par référence, un swap complet.
```

```
std::cout << i << " " << j << std::endl; // 7 4
```

```
return 0;
```

```
}
```

**Solution au problème posé en page 70.**

```
#include <iostream>

using namespace std;

void fonction(const int& x) {
    cout << "valeur: " << x << endl; // cette fonction ne sert qu'à afficher x, qui a été déclarée constante.
}

void fonction(const void* ptr) { // on passe une adresse qui peut être NULL
    if (ptr == NULL) cout << "pointeur NULL\n"; // test si c'est le cas
    else cout << "pointeur: " << ptr << endl; // sinon on affiche l'adresse
}

int main() {
    int i = 10;
    int* ptr = &i; // un pointeur, qui pointe sur l'adresse de la variable i
    void* aptr = NULL;
    fonction(i); // on passe i par référence; affiche: 10
    fonction(ptr); // on passe l'adresse du pointeur; affiche: pointeur: 0xqg_chose
    fonction(aptr); // on passe un pointeur NULL; affiche: pointeur NULL

    return 0;
}
```

**Solution au problème posé en page 76.**

Le programme va afficher 2 au lieu de `MAX(0,1) = 1`.