

Chapitre 7

Les fonctions

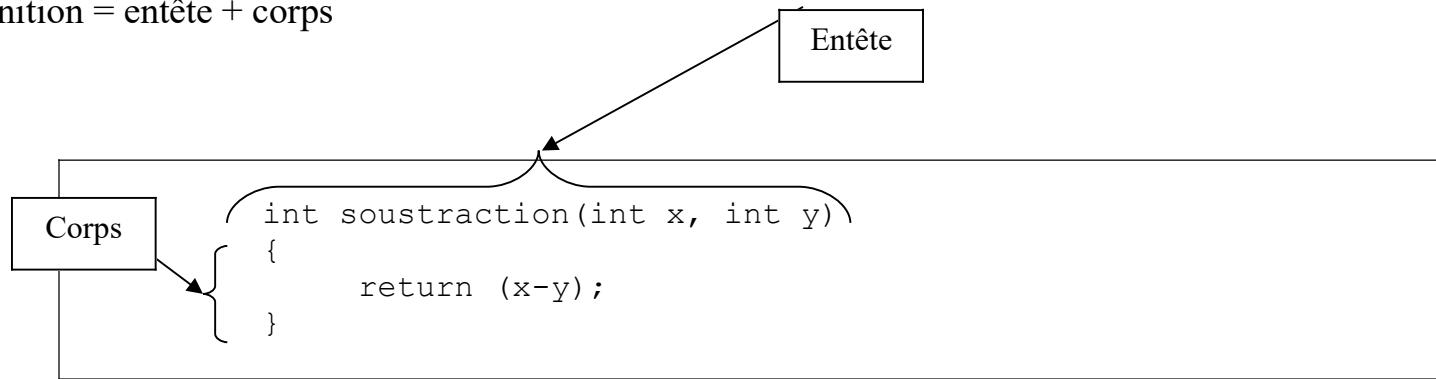
1. Définition d'une fonction

La définition se fait comme suit :

```
type identificateur (paramètres) {  
    ... /* Instructions de la fonction. */  
}
```

- « type » est le type de la valeur renvoyée.
- « identificateur » est le nom de la fonction.
- « paramètres » est une liste de paramètres.

Une définition = entête + corps



- Une fonction en C++ peut être définie globalement dans une autre fonction ou dans une classe. Dans ce dernier cas, la fonction porte le nom de « méthode ».
- Il n'y a pas de fonctions en Java seulement des méthodes.

2. Déclarations de fonctions (prototypes)

- Prototype => Description de l'entête d'une fonction avant son 1^{er} appel,

```
type identificateur (paramètres) ;
```

- Le prototype est facultatif en C mais OBLIGATOIRE en C++,
- Une fonction doit être déclarée avant d'être appelée.

```
// déclaration de la fonction
int soustraction(int,int);

int main(){
    int x=11, y=10, z=0;
    // appel de la fonction
    z = soustraction(x,y);
    return 0;
}
// définition de la fonction
int soustraction(int a, int b) {
    return (a-b);
}
```

Écrire

```
int soustraction(int,int);
```

Ou bien

```
int soustraction(int a,int b);
```

- Les deux précédentes écritures reviennent à dire la même chose.
- Le compilateur dans tous les cas ignore le nom des variables dans la déclaration d'une fonction.
- L'appel d'une fonction se fait en donnant son nom puis les valeurs de ses paramètres entre parenthèses.

3. Prototype et définition en même temps

- Écrire la définition avant le premier appel de la fonction (son utilisation).

```
// Prototype et définition de la fonction
int soustraction(int a, int b) {
    return (a-b);
}

int main() {
    int x=11, y=10, z=0;
    // appel de la fonction
    z = soustraction(x,y);
    return 0;
}
```

4. Cas de VOID

4.1. Fonction sans paramètres (arguments)

- Une fonction sans arguments restera sans type

```
int fonction_test(void) ← C
```

```
int fonction_test() ← C++
```

4.2. Fonction sans valeur de retour

```
fonction_exemple(int a,double b);
```

- En C par défaut retourne un « int ».
- En C++ déclaration du type de la valeur renvoyée est **OBLIGATOIRE**,

```
void fonction_exemple(int a,double b);
```

5. 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, le résultat est 2, et le reste est perdu.
 - $5.0/2.0$ division réelle, le résultat est 2.5.
- L'opérateur « / » a un double rôle : la division des nombres entiers et des nombres réels.
- En C++ (Java aussi), on peut définir différentes fonctions ayant le même nom mais le nombre et le type de paramètres sont différents.

Même nom, mais les arguments sont différents

```
#include <iostream>

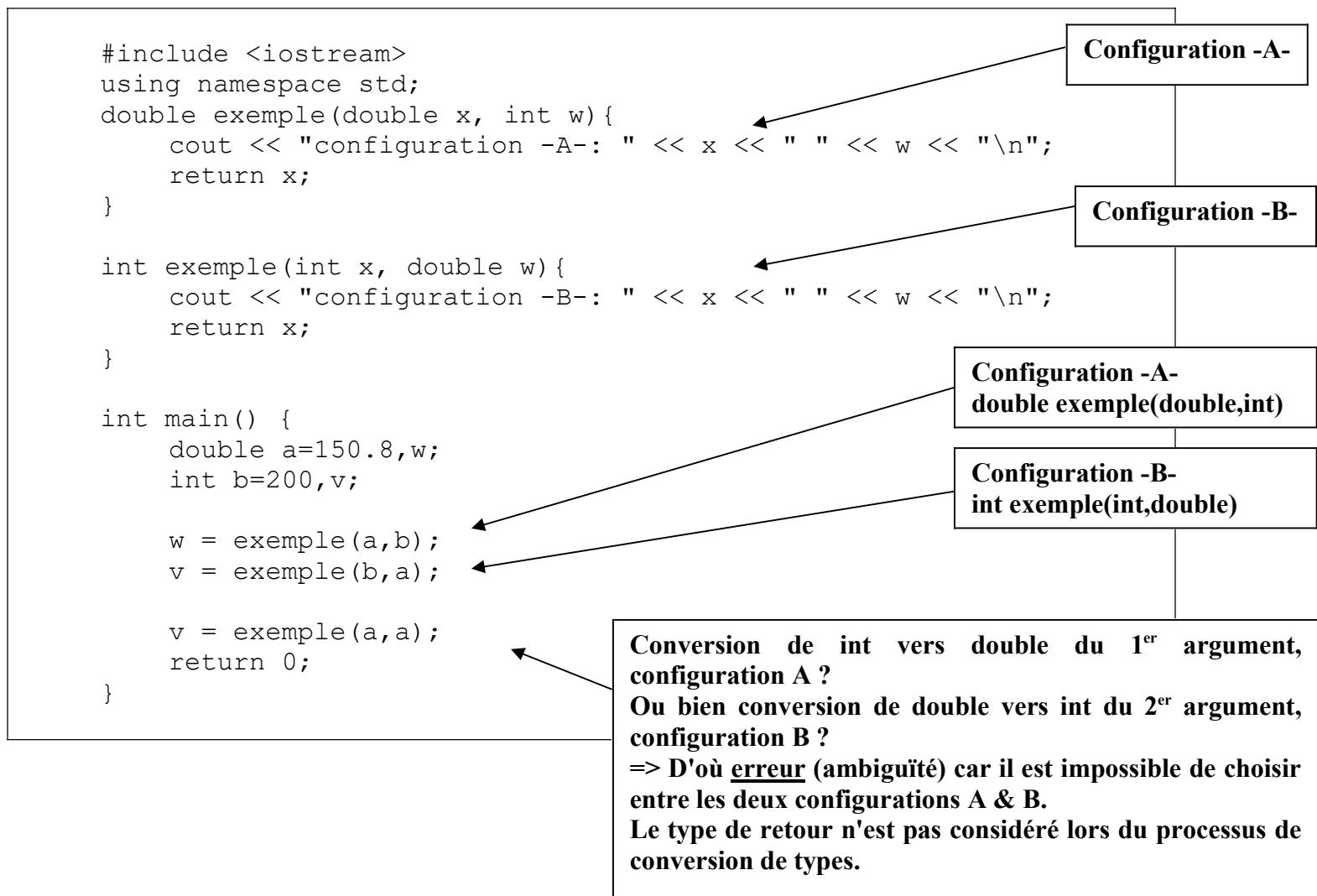
using namespace std;

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;
}
```

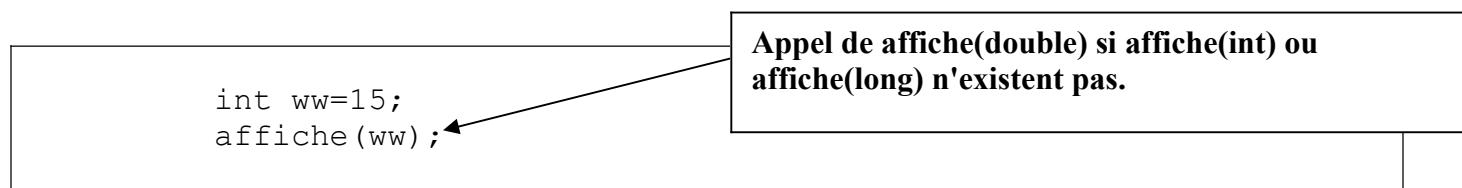
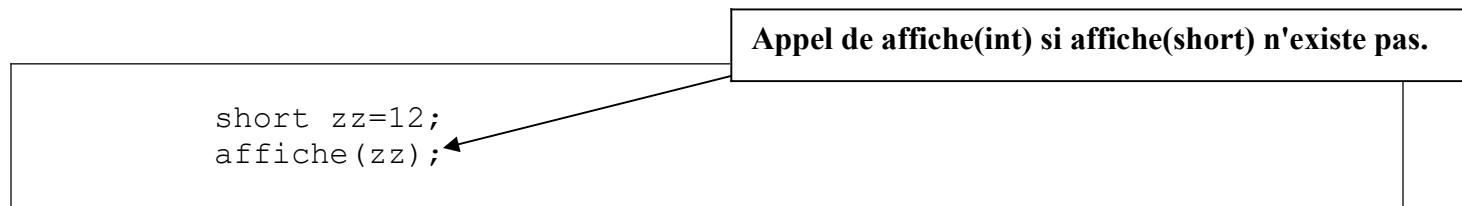
- Le compilateur recherche la « meilleure correspondance » possible.
- S'il y a plusieurs arguments, le compilateur essaye chacune des fonctions séparément.
- Le compilateur signale une erreur de compilation si aucune fonction ne convient ou bien si plusieurs fonctions conviennent. Donc, il y a plusieurs choix possibles (ambiguïté) dans celles-ci.



- 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



- Conversion dégradante

Origine	Conversion vers
int	short
double	float

12. Arguments par défaut

- 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(){
    cout << multiplication(3,4)<< endl;
    return 0;
}
```

The diagram shows a C++ code snippet with annotations. Two arrows point from the parameter names 'x' and 'y' in the function definition to a box labeled 'Formels'. Another arrow points from the argument values '3' and '4' in the main function call to a box labeled 'Réel'.

- On commence à omettre à partir de la fin.
- On peut spécifier des valeurs par défaut

```
#include <iostream>

using namespace std ;

double exemple(double x, char c = 'T', double z=200.5){
    cout << x << " " << c << "\n";
    return z;
}

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

    w = exemple(a,'s',-670.9);
    cout << w << endl;

    w = exemple(a,'s');
    cout << w << endl;

    w = exemple(a);
    cout << w << endl;
    return 0;
}
```

**Nombre de paramètres formels = nombre de paramètres réels.
Aucun traitement à faire.**

**Omission à partir de la fin ...
Omission du dernier argument, la fonction prendra la valeur par défaut du 3^e argument.**

Omission des deux derniers arguments, la fonction prendra la valeur par défaut du 2^e et 3^e argument.

Affichage en sortie

```
150.8 s  
-670.9  
150.8 s  
200.5  
150.8 T  
200.5
```

Erreur de compilation ...

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

13. Fonctions inline (en ligne)

- `#define` est utilisé pour la substitution de texte.
- Une autre utilité 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

using namespace std ;

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

    n = abs(a);
    cout << n << endl; // affiche 4
    n = abs(a)*2;
    cout << n << endl; // affiche 4
    n = abs(b)*2;
    cout << n << endl; // affiche 12

    return 0;
}
```

Définition de la macro `abs`. Elle permet de calculer la valeur absolue d'un nombre.

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

Le résultat est faux. Il faut donc revoir la logique de l'opération.

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

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

Mais qu'en est-il pour le programme suivant, que va-t-il afficher après son exécution ?

```
#include <iostream>

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

using namespace std ;

int main() {
    int a=0,b=1;

    cout << MAX(a,b++) << 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 faibles (surtout pour les petites fonctions).
- Il n'y aura pas d'accès à la table des fonctions. Toutes les fonctions sont considérées `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>

using namespace std;

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

int main() {
    int n, a=-4;
    n = abs(a)*2;
    cout << n << endl; // affiche 8
    return 0;
}
```

Le programme calcule d'abord `abs(a)` puis le résultat est * 2

- 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.