

Chapitre 2

C++ versus C

* C++ est une extension de ANSI-C

* Tout programme C est un programme C++, mais quelques incompatibilités existent dues à des règles de typage plus strictes en C++.

* Il est toujours possible de se servir de la librairie C dans C++.

1. Définition d'une fonction

Une définition = entête + corps

En langage C (non ansi)

en langage C++

```
Int soustraction(x,y)
int x;
int y;
return (x-y);
}
```

```
int soustraction(int x, int y)
{
    return (x-y);
}
```

2. Déclarations de fonctions (prototypes)

- prototype \Rightarrow Description de l'entête d'une fonction avant son 1^{er} appel,

type identificateur (paramètres)

- identificateur est le nom de la fonction,
- paramètres une liste de paramètres passés à la fonction,
- type est le type de la valeur renvoyée (retournée) par la fonction.

- pas obligatoire en C, 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

ou bien

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

```
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.

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 retournée est **OBLIGATOIRE**.

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

5. const et define

5.1. #define

La commande permet:

- La substitution de texte

```
#define dimension 10
```

C'est une directive au préprocesseur qui demande le remplacement lexical de **dimension** par **10**, aucun espace mémoire n'est alloué pour stocker **10**.

```
const int dimension=10;
```

C'est une directive au compilateur, espace mémoire de taille **int** est réservé à la variable **dimension** qui doit être initialisée. Cette variable a été initialisée avec la valeur **10**. Elle ne pourra pas être modifiée.

Privilégier **const** à **define**, c'est plus sécuritaire car le compilateur connaît l'existence de la variable et peut repérer les erreurs lors de son utilisation, par ailleurs la variable devient disponible dans la table des symboles. De ce fait, le débogueur connaît aussi son existence.

- La définition de macros (sera étudiée un peu plus loin dans le cours, voir l'utilisation de la fonction **inline**).

5.2. Les constantes

- La valeur d'une constante ne peut pas être modifiée,
- L'adresse d'une constante ne peut pas être affectée à une variable,

1^{er} cas de figure

```
const int j=100;
j=50; // Erreur la variable j est constante, on ne peut pas modifier sa valeur

int* ptr;
ptr=&j; // Erreur, car on risque de modifier une variable déclarée comme étant constante
```

A vrai dire, le compilateur g++ ne signale qu'un warning! Et modifie le type de j de const int vers int uniquement. Autant écrire les choses proprement: soit garder j comme une constante et enlever ptr=&j ou bien déclarer j comme étant un int et garder ptr=&j.

2^e cas de figure

```
const int* ptr; // ptr pointe sur une variable constante, mais ptr n'est pas une constante
ptr = &j; // OK
*ptr=200; // Erreur car on tente de modifier la valeur pointée par ptr qui est une constante!

int k=30;
ptr = &k; // OK
```

3^e cas de figure

```
int a = 10;
int* const ptr = &a; // ptr est une constante mais pas la variable pointée
int b = 20;

ptr = &b; // Erreur, l'adresse est constante

*ptr = 400; // OK, et par la même occasion a=400
```

4^e cas de figure

```
int i=20;
const int j=10;
const int* const ptr = &j; // ptr est constant et pointe sur une valeur constante

*ptr=30; // Erreur, on tente de modifier la valeur de j qui est constante

ptr = &i; // Erreur, on modifie l'adresse pointée par ptr qui est constante
```

6. Conversion de pointeurs

Type `void*` dénote un pointeur quelconque.

Ansi-C permet de convertir implicitement:

`type*` \rightarrow `void*`

et

`void*` \rightarrow `type*`

En C++,

la conversion `void*` \rightarrow `type*` est INTERDITE.

`int*` `p`;

`void*` `v`;

`v = p`; OK conversion vers `void*` acceptable.

`p = v`; ERREUR, conversion non autorisée en C++, mais OK en C.

Le compilateur n'a aucune information pour interpréter le contenu de la variable afin de la convertir, le compilateur ne connaît pas le nombre précis d'octets auquel le pointeur réfère: dans le cas d'un pointeur vers `void`, il ne peut pas déterminer le nombre d'octets à partir du type. Il est nécessaire de faire un cast explicite, i.e.:

```
p = (int*) v; // écriture à la C (valable en C++).
```

```
p = static_cast<int*>(v); // nouveau style (non valable en C).
```

7. Conversion de type

En plus des règles de forçage de conversion de type du langage C, utilisées aussi en C++ ; 4 nouveaux opérateurs ont été introduits en C++, pour forcer la conversion de type. Ces opérateurs sont:

7.1. `static_cast`

Cet opérateur est utilisé pour effectuer les opérations de conversion standard, du `void*` vers un `int*` par exemple, où un `int` en `float`, un `float` en `char` etc.

```
double x = 5.89;
```

```
int y = static_cast<int*>(x); // conversion double vers int.
```

```
double z = static_cast<double>(y); // conversion int vers double.
```

Cet opérateur ne permet le forçage de types `const` vers non `const` ainsi que le forçage de types sans relation.

```
const int x = 8;
int *p = static_cast<int*>(&x); // La variable x est constante, on tente une conversion constante
// vers non constante => Erreur.
int j =250;
int *p = static_cast<int*>(j); //Le pointeur p est du type int* et la variable j est int, il n'y a pas
// de relation entre les deux types => Erreur.
```

7.2. const_cast

Cet opérateur permet uniquement la conversion de types `const` vers non `const`.

```
const int x = 8;
int *p = const_cast<int*>(&x); // Ok! Conversion const vers non const.

int y = 8;
int *r = const_cast<int*>(&y); // Inutile car y n'est pas const.

double yz = 8.7;
int *rr = const_cast<int*>(&yz); // Erreur+Inutile! Erreur car conversion de type double vers int or
le rôle de const_cast se limite à la conversion const vers non const.
```

7.3. reinterpret_cast

Cet opérateur est utilisé essentiellement pour la conversion de types de relation différente (non standard), `int` et `int*`, `void*` et `int` etc. Il ne permet pas la conversion `const` vers non `const`.

```
int j =250;
int *p = reinterpret_cast<int*>(j); // Ok! Conversion de types: int vers int* mais ...
```

Faire très attention lors de l'utilisation de cet opérateur.

```
int j =250;
int *p = reinterpret_cast<int*>(j);
cout << *p; // peut provoquer un comportement bizarre (parfois même une erreur due à un accès interdit à une
zone mémoire) lors de l'exécution du programme.
```

7.4. dynamic_cast

Cet opérateur est utilisé en programmation polymorphique, où la conversion est différée au moment de l'exécution du programme. Nous allons voir cela avec plus de détails dans le chapitre 13: "fonctions virtuelles et classes abstraites".

8. Les nouveaux mots-clés

Une trentaine de nouveaux identificateurs, exemple: `static_cast`, `new`, `private`, `protected`, `class`, `virtual`, `friend`, etc. Un programme C avec ces identificateurs compilé en C++ ne marchera pas.