Introduction au C++

Vincent Torri

1 Généralités

Le C++ est une évolution du C, écrit au début des années 80 par Bjorne Soustrup. Comme le C, un compilateur comprenant le C++ est nécessaire. Il est compatible avec le C (tout code écrit correctement en C est compilable avec un compilateur C++).

Il ajoute un nombre important de fonctionalités. En particulier, c'est un langage orienté objet (comme le Java). C'est aussi un langage plus abstrait et plus strict que le C, plus difficile à apréhender que le C, au niveau syntaxe et niveau d'abstraction.

De plus, la manière de programmer en C++ est différente de celle du C. Plus que la syntaxe, c'est la conception (design) des programmes qui est différente et très importante. Par exemple, pour l'ecriture de grands programmes, il est essentiels de bien structurer le code, de l'écrire de façon modulaire, avec des interfaces parfaitement définies.

1.1 Quelques fonctionalités du C++

— compatibilité ascendente avec le C,
— langage très typé,
— portabilité,
— gestion des erreurs,
— surcharge de fonctions,
— patrons (templates),
— STL (Standard Template Library).

1.2 Différences techniques avec le C

Un fichier source écrit en C++ a pour extension *.cpp* et non *.c* (par défaut, un compilateur se base sur l'extension des fichiers sources pour savoir s'il va compiler pour du langage C ou pour du langage C++). C'est le cas du compilateur de Visual Studio.

Les fichiers d'en-tête ont en général pour extension *.h* et parfois *.hpp*.

2 Ajouts de fonctionalités

2.1 Commentaires

Les commentaires C sont toujours disponibles. Un nouveau commentaire apparaît : //

Il commente tout ce qui est après, jusqu'à la fin de la ligne.

2.2 Types et variables

Rappel : le *transtypage* (*cast* en anglais) est l'opération qui permet changer le type d'une variable en un autre type. Syntaxe en C : *(type)var* et en C++ : *type(var)*.

2.2.1 Nouveaux types

— Le type *bool* pour la gestion des booléens (vrai ou faux). Les valeurs numériques sont **true** et **false**.
Exemple :

```
bool b = false;
int i;
// i est initialisé
if (i == 6)
b = true;
if (b) // pareil que if (b == true)
std::cout << "b est vrai" << std::endl;
```

Rappel : la convention en C est toujours valide : valeur 0 pour faux et toute valeur non nulle pour vrai.

— La *référence* : elle est synomyme d'identificateur et permet de manipuler un identificateur sous un autre nom.
Syntaxe :

```
[const] type & ref;
```

où *type* est un type connu (autre que void).
Exemple :

```
int i = 0;
int & ref = i;
```

```
ref = ref + i;
```

Toute référence est associée à un identificateur, donc elle **doit** être définie. De plus, si on donne pour valeur à une référence une valeur numérique, la référence doit être constante :

```
const int & ref1 = 3;
int & ref2 = 2;          // erreur
```

Lien avec les pointeurs : la référence et son identificateur associé partage la même adresse. Ainsi, manipuler une référence revient à manipuler un pointeur pointant vers l'identificateur associé à la référence : Exemple avec les pointeurs :

```
int i = 0;
int *p = &i;
*pi = *pi + 1;
```

Exemple avec les références :

```
int i = 0;
int &r = i;
r = r + 1;
```

Donc, les références s'utilisent comme une variable, mais manipulent en fait l'adresse de cette variable comme le ferait un pointeur, sans la lourdeur syntaxique des pointeurs. La grosse différence est qu'une référence est toujours initialisée et que le déréférencement est inutile. Pour le passage en paramètre des références, voir plus loin.

2.2.2 Déclaration et définition

On peut déclarer les variables n'importe où dans un bloc (et plus seulement au début d'un bloc). Il est de plus conseillé de les définir au moment de la déclaration (si possible) et de les déclarer le plus tard possible (au moment où on les utilise).

2.3 Structures de contrôle

2.3.1 Boucle for

On peut en C++ déclarer une variable dans la partie "initialisation" d'une boucle **for**. Cette variable ne pourra être utilisée que pour la boucle et non à l'extérieur de la boucle. L'intérêt est un code plus sûr (utilisation involontaire et erronée de cette variable à l'extérieur de la boucle, par exemple).

Exemple :

```
for (int i = 0; i < 10; i++)
{
  // code de la boucle
}
```

2.4 Fonctions

2.4.1 Déclaration

On peut assigner une valeur par défaut à un des arguments lors de la déclaration de la fonction (et non lors de la définition, si celle-ci ne sert pas aussi de déclaration). Si un argument a une valeur par défaut, tous les arguments suivants doivent avoir une valeur par défaut. Sinon, une erreur de compilation interviendra.

Syntaxe :

```
type_ret nom_fct(t1 v1, t2 v2 = val1, t3 v3 = val2);
```

Exemples :

```
int f1(int i, int j, int k = 0);
int f2(int i, int j = 1, int k = 0);
int f3(int i, int j = 3, int k); // erreur compilation
```

Utilisation :

```
int i1 = f1(1, 2, 3);
int i2 = f1(1, 2); // k dans f1 vaut 0
int i3 = f2(1, 2); // k dans f2 vaut 0
int i4 = f2(1); // j et k dans f2 valent resp. 1 et 0
```

2.4.2 Surcharge de fonction

En C, il est interdit de déclarer plusieurs fonctions ayant le même nom. En C++, c'est possible, ceci se nomme la *surcharge de fonction*. Il faut néanmoins respecter les règles suivantes pour que le compilateur puisse différencier les fonctions :

— Soit le nombre d'arguments des fonctions doit être différent.
— Soit il est le même et dans ce cas le type des arguments doit être différent.

— Le type de retour des fonctions n'intervient pas dans la différenciation des fonctions.

Exemples :

```
int f(int i, int j);
int f(double i, double j);
double f(double i, double j); // erreur compilation
int f(int i, int j); // erreur compilation
int f(int i, int j, int k = 0); // voir ci-dessous
```

Utilisation :

```
int i1 = f(1, 3, 0); // derniere fonction
int i2 = f(1.0, 2); // 2eme fonction
int i3 = f(1, 2); // erreur : peut etre la 1ere
                  // ou la derniere fonction
```

2.4.3 Fonction d'affichage

Bien qu'elle soit disponible (pour des raisons de compatibilité ascendante), la fonction **printf()** ne s'utilise plus, au profit d'un autre moyen : **std : :cout** et **std : :endl**. Il faut inclure le fichier *iostream* (pas d'extension ".h") et ils s'utilisent ainsi :

```
#include <iostream>
...
std::cout << "bonjour" << std::endl;
```

affiche le texte "bonjour" dans la console. **std : :endl** sert (plus ou moins) à aller à la ligne (l'équivalent du \n du C). On peut remplacer la chaîne de caractères par tout object comprenant la syntaxe ci-dessus, comme les type entiers ou flottant, les pointeurs, etc... :

```
#include <iostream>
...
int i = 2;
int j = i + 4;
std::cout << i << "+" << 4 << "=" << j << std::endl;
```

Affiche "2+4=6".

La syntaxe "std : :" est un espace de nom décrit dans la section suivante.

3 Espaces de nom

3.1 Syntaxe

Les espaces de nom sont une solution élégante pour regrouper, en rajoutant un préfixe commun, des fonctions, des classes, des variables, etc...

Exemple déjà vu :

```
std::cout
```

Ici, **cout** est une variable (plus précisément un instance de classe, voir 5), et on lui a rajouté le préfixe **std : :**.

Il est à noter que un espace de nom ne fait que modifier le nom d'un objet, il ne change pas les caractéristiques de cet objet.

Intérêts :

- Meilleure structuration du programme : certaines parties ont un préfixe, d'autres en ont un autre, donc on sait à quelles parties du programme une fonction (par exemple) appartient, juste en lisant son nom.
- Ceci permet d'utiliser le même nom de fonctions ou variables, avec juste le préfixe qui est différent.

Pour ajouter un espace de nom, on utilise la syntaxe suivante :

```
namespace nom_du_namespace {
// on met du code ici :
// variables globales,
// classe ou structure,
// declaration ou definition de fonction, ...
} // namespace nom_du_namespace
```

Ici, *nom_du_namespace* est un identificateur et est le nom du namespace. Tout ce qui est dans le namespace sera préfixé par *nom_du_namespace : :*.
On peut imbriquer les namespaces :

```
namespace ns1 { namespace ns2 {
...
} } // namespace ns1::ns2
```

3.2 Utilisation à l'intérieur d'un namespace

On peut ne pas utiliser le préfixe :

```
namespace garcon {
```

```
struct Fred
{
  int id;
};

int g() { return 1; }

void f()
{
  Fred f;
  f.id = g();
}
} // namespace garcon
```

3.3 Utilisation à l'extérieur d'un namespace

On doit utiliser le préfixe :

```
namespace garcon {
struct Fred
{
  int id;
};

int g() { return 1; }
} // namespace garcon

void f()
{
  garcon::Fred f;
  f.id = garcon::g();
}
```

3.4 Déclaration *using*

On peut s'affranchir d'ajouter le préfixe du namespace pour un objet en utilisant une déclaration **using** :

```
namespace garcon {
struct Fred
{
```

```
  int id;
};

int g() { return 1; }
} // namespace garcon

void f()
{
  using garcon::Fred; // on peut la mettre hors de f()
  Fred f;
  f.id = garcon::g(); // necessaire pour g
}
```

3.5 Directive *using*

On peut s'affranchir d'ajouter le préfixe du namespace pour tous les objets dans ce namespace en utilisant une directive **using** :

```
namespace garcon {
struct Fred
{
  int id;
};

int g() { return 1; }
} // namespace garcon

using namespace garcon;

void f()
{
  Fred f;
  f.id = g();
}
```

Attention : utiliser la directive **using** avec le namespace **std** est une mauvaise idée.
Attention : utiliser trop de directives **using** peut conduire à des erreurs :

```
namespace A {
  int x = 1;
  int y = 2;
```

```
} // namespace A

namespace B {
  int x = 3;
  int z= 2;
} // namespace B

using namespace A;
using namespace B;

void f()
{
 std::cout << x << std::endl; // erreur: A::x ou B::x ?
}
```

3.6 Utilisation d'espaces de nom imbriqués

Supposons que l'on ait ces espaces de nom :

```
namespace garcon { namespace F {
  int Fred = 1;
  int Francois = 2;
} } // namespace garcon::F
```

On peut les utiliser comme ceci :

```
garcon::F::Fred = 8;
using garcon::F::Fred; // declaration
using garcon::F; // directive
```

ou bien une directive juste sur *garcon* :

```
using garcon;
F::Fred = 5;
```

4 Gestion des erreurs

5 Classes

5.1 Déclaration, définition et usage

5.2 Construteur, constructeur de copie et destructeur

Table des matières