SHAREMAN

Zeste de savoir

Les conversions de types en C++

12 août 2019

Table des matières

Le langage C++ possède ses propres mots-clefs qui nous permettent de *caster* des variables ou des objets. Concrètement, un cast est une conversion de types. Dans ce tutoriel, nous allons voir comment mettre en œuvre les différents types de cast en C++.

Je n'entre volontairement pas dans certains détails trop techniques. Je présente la chose de manière globale. Si vous souhaitez aller encore plus loin, on trouve les informations supplémentaires à la pelle sur internet.

1. Introduction

Dans ce cours, vous allez apprendre la bonne méthode pour convertir des types d'objets en C++. Quand je dis "objet", je sous-entends variable, pointeur, référence, etc. Il arrive fréquemment qu'on ait à faire ce genre de choses et heureusement pour nous, c'est simple ; il suffit d'être rigoureux et d'utiliser le bon cast en fonction de notre besoin.

On peut distinguer quatre types de conversion possibles et réalisables en C++ :

— La conversion **statique** de types ;
— La totale **ré-interprétation** des données d'un type vers un autre ;
— La "conversion" d'un pointeur (ou référence) **constant(e)** vers un pointeur (ou référence) non-constant(e) ;
— La conversion de types **dynamique**.

Notez que l'on parle de "dynamisme" en C++ pour qualifier une action réalisée pendant l'exécution de votre programme. Une conversion de types dynamique est donc une conversion qui va s'effectuer pendant l'exécution de votre programme et non par le compilateur comme c'est le cas pour les trois autres cast existants.

La conversion de types dynamique s'effectuant donc pendant l'exécution du code, on peut tomber sur une erreur d'implémentation et si cette erreur se produit, une exception est lancée : `std::bad_cast`. On va donc facilement pouvoir gérer une erreur de ce type.

Il est également important de savoir qu'un pointeur (ou une référence) constant(e) n'est pas un pointeur ou une référence dont on ne peut modifier la valeur mais dont l'élément pointé ne peut être modifié en passant par ce pointeur (ou cette référence). Cela peut porter à confusion, mais c'est très simple au fond, retenez juste que le mot-clé `const` n'a pas la même signification utilisé pour un objet qu'utilisé pour un pointeur ou une référence.

Vous noterez que dans la liste que je vous ai dressée au-dessus, j'ai volontairement marqué des mots en rouge pour qu'ils ressortent bien. En effet, je voulais porter votre attention sur ces mots-là, car ils ont un lien direct avec les mots-clefs associés. C'est très simple, on a respectivement les mots-clefs suivants :

- **static_cast**
- **reinterpret_cast**
- **const_cast**
- **dynamic_cast**

Nous allons sans plus tarder voir de quoi il s'agit réellement et comment mettre en œuvre ces cast de manière simple et sûre dans un programme. J'illustrerai mes propos avec des exemples concrets pour vous donner une idée.

2. static_cast et reinterpret_cast

Ici, nous allons nous intéresser au type de cast le plus simple et le plus rencontré, `static_cast`, et à un autre type de cast tout aussi intéressant, `reinterpret_cast`.

2.0.1. Le cast "statique"

Le mot-clef associé à ce type de cast est `static_cast`, évidemment.

Il s'agit ici souvent d'expliciter une conversion de types implicite qui peut avoir lieu quand on essaye de copier le contenu d'une variable d'un type de la même famille que le type de la variable de destination. C'est donc très simple : `static_cast` permet de caster des types de même famille. Après, il faut savoir ce que l'on entend par "famille".

C'est encore très simple : les pointeurs forment une famille, les références une autre, etc. Par contre, il existe bien sûr certaines conversions entre types de même famille (comme de double *vers float* par exemple) qui ne sont pas réalisables avec un simple `static_cast`, mais nous allons voir cela dans un second temps. Des cast comme long vers int ou void *vers double* sont des exemples qui nécessitent l'utilisation d'un `static_cast` (la plupart du temps, c'est juste une question de rigueur).

Pour l'exemple, voici un code qui **ne compile pas** :

```
1 void* vp;
2 long* lp = vp;
```

La solution est bien sûr d'appliquer un `static_cast`. Cela se fait de la manière suivante :

```
1 long* lp = static_cast<long*>(vp);
```

On indique le type de destination entre "<>" suivi de la variable à caster **entre parenthèses**.

On ne devrait jamais avoir à utiliser void* en C++ mais c'est juste pour l'exemple.

Voici un autre exemple où une conversion implicite suffirait, mais où un compilateur bien réglé cracherait au moins un warning sans `static_cast` :

```
1 double d;
2 float f = static_cast<float>(d);
```

C'est donc enfantin. Pensez à expliciter vos conversions implicites en utilisant `static_cast`, c'est une très bonne pratique, et un programmeur qui lira votre code aura toutes les chances de comprendre plus vite ce que vous essayez de faire et surtout il sait que vous savez ce que vous faites.

2.0.2. La ré-interprétation des données

Le cast de type `static_cast` est déjà fort pratique mais n'est pas assez puissant pour régler toutes les situations. Par exemple, avec un `static_cast`, il est impossible de convertir un double *vers un float*. La solution serait de ré-interpréter les données stockées par les pointeurs. En gros, on devrait pouvoir récupérer la valeur hexadécimale stockée par la variable double* et la considérer comme l'adresse d'une variable float.

Ce genre de pratique est tout à fait réalisable en C++ mais cette fois, on utilisera le mot-clef `reinterpret_cast`. Pour l'exemple de conversion double *vers float*, on procédera donc de la manière suivante :

```
1 double* dp;
2 float* fp = reinterpret_cast<float*>(dp);
```

Ce code compile sans problème et fait exactement ce à quoi on s'attend. Mais `reinterpret_cast` ne se limite pas aux types de même famille. Il est possible de réaliser des cast entre certains types de familles différentes et cela peut parfois s'avérer très pratique.

Imaginons que vous souhaitez réaliser un programme qui demande à l'utilisateur une adresse mémoire et lui affiche le contenu de la case demandée. Comment allez-vous procéder ? Si l'on ne connaît pas `reinterpret_cast`, c'est très difficilement réalisable. Pour ceux qui y auraient pensé, non, `std::cin` ne fonctionne pas sur les pointeurs ; par contre, `std::cin` fonctionne sur les int ! La solution, vous l'aurez deviné, est de demander une valeur hexadécimale à l'utilisateur, de la stocker dans une variable de type int puis de ré-interpréter la donnée comme étant une adresse mémoire.

Le code peut donc ressembler à ceci :

```
int i;
std::cout << "Adresse : ";
std::cin >> std::hex >> i >> std::dec;
std::cout << "Contenu : " << *(reinterpret_cast<int*>(i));
```

Et ce code compile sans problème ! On va même le tester en prenant un exemple de variable que l'on aura auparavant initialisée afin que l'on puisse bien se rendre compte que le code fonctionne :

```
int var = 67;
std::cout << "Exemple d'adresse : " << &var << std::endl;
int i;
std::cout << "Adresse : ";
std::cin >> std::hex >> i >> std::dec;
std::cout << "Contenu : " << *(reinterpret_cast<int*>(i));
```

On crée une variable var et on lui donne la valeur 67. On affiche son adresse. Maintenant, vous vous attendez à retrouver la valeur 67 en saisissant l'adresse affichée à l'écran, et effectivement :

```
Exemple d'adresse : 0x22ff44
Adresse : 0x22ff44
Contenu : 67
```

Ça fonctionne comme on le souhaite. Comme vous le voyez, `reinterpret_cast` est très simple et pratique.

Nous allons sans plus tarder nous attaquer à un cast un peu plus particulier, `const_cast` !

3. Les cast de pointeurs sur constante

En ce qui concerne le titre de cette sous-partie, j'inclus les références dans le terme de pointeur.

Pour vous expliquer ce qu'est réellement `const_cast` et à quoi cela va nous servir, je vais partir d'un exemple concret : vous étiez tous déjà dans la situation dans laquelle vous avez un pointeur (ou une référence) sur constante et vous ne pouvez pas modifier l'élément pointé (car celui-ci est protégé par le const). Par exemple, le code suivant ne compile pas :

```
int i = 65;
const int& r_i = i;
r_i = 75;
```

En effet, la référence *r_i* est déclarée "sur constante", donc il n'y a pas moyen de modifier *i* en passant par *r_i*. Il est donc également impossible d'assigner le contenu de cette référence à une référence du même type mais non-"sur constante" :

```
1 int i = 65;
2 const int& r_i = i;
3 int& r_i2 = r_i;
```

Cela constitue naturellement une sécurité et c'est très pratique dans certains cas de pouvoir ainsi empêcher le programmeur de toucher à certaines choses (c'est fait pour ça, const).

Mais... il existe bien une solution pour cracker cette sécurité, c'est le fameux `const_cast` !

`const_cast` est à utiliser avec modération et seulement s'il n'y a pas d'autres solutions alternatives.

Avec le précédent code, on applique le `const_cast` au moment de l'affectation du contenu de *r_i* à *r_i2* et ce de la manière suivante :

```
1 int i = 65;
2 const int& r_i = i;
3 int& r_i2 = const_cast<int&>(r_i);
```

En résumé, ce type de cast permet de supprimer les attributs *const* ou *volatile* (mais pour ce dernier point, cela peut se faire implicitement).

`const_cast` ne fonctionne que sur des pointeurs ou des références et n'est pas fait pour modifier la valeur d'une variable constante d'un type d'une autre famille ; de toute manière, cela ne compilerait pas. Mais comme dit, on ne devrait jamais avoir à utiliser `const_cast` dans un programme à moins de savoir exactement ce que l'on fait. En effet, cela peut devenir dangereux.

Je vais prendre pour exemple la méthode `std::string::c_str()` qui retourne un pointeur sur constante de type `const char*` sur la chaîne de caractères stockée par l'objet `std::string` en mémoire vive. Avec un `const_cast`, il est donc possible de modifier cette chaîne sans passer par l'instance de `std::string` et donc toutes les autres informations relatives à cette chaîne stockées dans l'objet perdront leur signification et on peut aboutir à un bug dans votre programme. Prudence donc.

4. Les cast dynamiques

Les cast dynamiques sont des conversions de types s'effectuant pendant l'exécution du programme (cela n'étant pas fait à la compilation). Il s'agit toujours de types personnalisés (qu'on va donc devoir définir à l'aide des classes). Le mot-clef associé à ce type de cast est `dynamic_cast`. Le fonctionnement est un petit peu plus délicat que celui des autres cast mais si vous faites bien

attention, vous n'aurez aucune peine à comprendre ! Encore une fois, ce cast ne concerne que les références ou les pointeurs.

Sans vouloir me lancer directement dans les grosses explications, je vais partir d'un exemple pour vous aider à comprendre :

Je dispose d'une classe mère **polygone** et d'une classe dérivée **carre** (carré). **carre** hérite donc de **polygone** car tout carré est un polygone. Supposons que je dispose d'une référence sur un objet **carre** et que j'aimerais considérer ce carré comme un polygone en copiant cette référence vers une référence sur un **polygone**. C'est possible car **un carré est un polygone** et on fait cela précisément avec un `dynamic_cast` !

Voici un code pour illustrer mes propos :

```
#include <iostream>

class polygone
{
    public :
    virtual void f() {}
};

class carre : public polygone {};

int main()
{
    carre monCarre;
    carre& r_carre = monCarre;

    try {
        polygone& r_polygone = dynamic_cast<polygone&>(r_carre);
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what();
    }
    return EXIT_SUCCESS;
}
```

Pour une question de polymorphisme, il faut que la classe mère possède au moins une fonction virtuelle (je vous conseille de lire le tutoriel de Nanoc sur le polymorphisme ici). Si vous compilez le précédent code, vous vous rendrez compte qu'il fonctionne sans embrouille, aucune exception n'est lancée.

Par contre, si vous essayez d'inverser les types (donc si vous remplacez carre par polygone et inversement), vous vous retrouvez avec une belle exception `std::bad_cast` car **un polygone n'est pas forcément un carré** et donc on ne peut pas considérer tout polygone comme étant un carré. En programmation, tout est donc affaire de logique. Vous noterez qu'il est également impossible de caster avec un `dynamic_cast` entre deux classes dérivant d'une même classe (un pentagone n'est pas un carré).

Mais si je reprends le précédent code, j'aurais très bien pu faire :

```cpp
int main()
{
    carre monCarre;
    polygone& r_polygone = monCarre;

    try {
        carre& r_carre = dynamic_cast<carre&>(r_polygone);
    }
    catch (const std::exception& e)
    {
        std::cerr << e.what();
    }
    return EXIT_SUCCESS;
}
```

Vous vous attendez à ce qu'une exception `std::bad_cast` soit lancée mais... non ! En effet, on utilise bien `dynamic_cast` d'une classe mère vers une classe fille mais le secret réside dans l'affectation d'un objet de type carre à la référence de type *polygone&*, c'est la seule condition pour que cela fonctionne et la logique reste la même : dès le début du programme, on sait que le polygone "pointé" par *r_polygone* est un carré donc un cast vers une référence sur un carre est possible.

Dans la même optique, il est également possible de caster d'une classe (A) vers une autre classe (B) si les deux classes ont une fille en commun (C) et si l'on a instancié la référence sur A à caster vers B& avec un objet de type C. Cela parait difficile, mais c'est très simple et ça reste logique. Supposons que je possède une classe **losange** (A) et une classe **rectangle** (B) ainsi qu'une classe **carre** (carré, C) qui hérite de losange et de rectangle (un carré est un losange et un rectangle). On peut très bien créer une référence sur losange à partir d'un objet carre puis, par la suite, considérer cette référence comme une référence sur un rectangle puisqu'on sait que c'est un carré (et donc un rectangle).

Voici un code d'exemple qui illustre assez bien ce concept :

```cpp
#include <iostream>

class losange
{
    public :
    virtual void f() {}
};

class rectangle
{
    public :
    virtual void f() {}
```

```
13 };
14
15 class carre :
16     public losange, public rectangle {};
17
18 int main()
19 {
20     carre monCarre;
21     losange& r_losange = monCarre;
22
23     try {
24         rectangle& r_rectangle =
               dynamic_cast<rectangle&>(r_losange);
25     }
26     catch (const std::exception& e)
27     {
28         std::cerr << e.what();
29     }
30     return EXIT_SUCCESS;
31 }
```

Cette pratique porte un nom, c'est le cross-casting. *Un losange n'est un rectangle que si c'est un carré.*

Les cast en bon C++ sont souvent méconnus des débutants et c'est très dommage car on peut faire beaucoup avec ! J'espère donc avoir été le plus clair possible en vous ayant enseigné un nouveau concept du C++.

Je remercie Xavinou pour sa relecture attentive ainsi que Nanoc et Alp pour leurs remarques toujours pertinentes.

Bon codage !