GERMINOLEGRAND

Zeste de savoir

C++17 : les décompositions entrent en scène !

15 janvier 2019

Table des matières

1. Introduction

Une des nouveautés importantes du C++17 est la décomposition, ou *« structured bindings »* selon le terme utilisé dans le standard. Je nous propose d'en faire le tour car cela va fortement changer les habitudes. En effet cela balaye entre autres une limitation de longue date : il est maintenant possible d'avoir de multiples valeurs en retour de fonction sans que cela soit trop verbeux.

2. Premières décompositions

Prenons comme exemple un dictionnaire qui indique les villes en fonction de l'année où une organisation française très connue[1] y a fait son congrès.

```
1 using Year = int;
2 using City = std::string;
3 std::map<Year, City> dictio = {
4     {1895, "Limoges"},
5     {1896, "Tours"},
```

1. Vous gagnez un point Pelloutier si vous devinez laquelle !

```
    {1897, "Toulouse"},
    {1898, "Rennes"},
    {1900, "Paris"},
    {1902, "Montpellier"},
    {1904, "Bourges"},
    {1906, "Amiens"}
};
```

2.1. Dans une boucle `for`

Tout d'abord on voudrait pouvoir afficher la liste des congrès. Sans décomposition, il faudrait faire ainsi :

```
for(auto& entry : dictio){
    std::cout << "In " << entry.first <<
        ", a congress took place in " << entry.second << '\n';
}
```

Il n'est pas très pratique de récupérer `entry`, car ce n'est pas cela qui nous intéresse mais ses deux éléments `[year, city]` qu'on accède par `entry.first` et `entry.second` puisque `entry` est une `std::pair<Year, City>`.

Avec une décomposition, on peut directement récupérer les valeurs qui nous intéressent :

```
for(auto& [year, city] : dictio){
    std::cout << "In " << year << ", a congress took place in " <<
        city << '\n';
}
```

Le code est ainsi plus simple, mais également plus lisible : plutôt que des noms génériques comme `first` et `second`, on identifie directement ce que contiennent ces variables en leur donnant un nom.

2.2. Pour initialiser des variables

On souhaite maintenant pouvoir ajouter ou modifier une entrée. Pour cela on va coder une petite fonction :

```
void addCongress(std::map<Year, City>& dictio, Year year, City
    place) {
    auto insertRet = dictio.insert({year, place});
```

```
3
4       std::cout << "For the year " << insertRet.first->first <<
            ", a congress ";
5
6       if(insertRet.second) {
7           std::cout << "is now registered in " <<
                insertRet.first->second << "." << std::endl;
8       } else {
9           std::cout << "was already registered in " <<
                insertRet.first->second << " and was not updated to "
                << place << "." << std::endl;
10      }
11  }
```

Comme on le voit, l'utilisation du type retourné par `map::insert` nécessite un bon nombre de `first` et de `second`, et il est pratiquement impossible de tomber juste du premier coup en l'écrivant.

Dans ce code, `insertRet` est de type `pair<map<Year, City>::iterator, bool>` et `map<Year, City>::iterator` est équivalent à `pair<Year, City>*`, soit deux paires imbriquées l'une dans l'autre avec des pointeurs au milieu pour rendre le tout plus distrayant — et imbuvable.

On peut donc ici aussi faire usage des décompositions pour nommer les éléments qu'on manipule par la suite. Une première fois sur la paire `[iterator, bool]`, puis sur la paire `[Year, City]`.

```
1   void addCongress(std::map<Year, City>& dictio, Year year, City
        place) {
2       auto [it, inserted] = dictio.insert({year, place});
3       auto const& [date, city] = *it;
4
5       std::cout << "For the year " << date << ", a congress ";
6
7       if(inserted) {
8           std::cout << "is now registered in " << city << "." <<
                std::endl;
9       } else {
10          std::cout << "was already registered in " << city <<
                " and was not updated to " << place << "." <<
                std::endl;
11      }
12  }
```

Plus d'obscurs `first` ou `second`, on sait ce qu'on manipule, le code est bien plus lisible.

3. Décortiquons la décomposition

Après ce premier exemple qui donne une idée de l'utilité des décompositions, il est temps de voir comment elle se caractérise précisément.

Une décomposition peut déclarer exactement autant de variables que d'éléments comportés par la structure décomposée ; ni plus, ni moins.

```
1  // Les trois sont identiques
2  auto [x, y, z] = getPosition3D();
3  auto [x, y, z](getPosition3D());
4  auto [x, y, z]{getPosition3D()};
5
6  auto position = getPosition3D();
7  auto& [a, b, c] = position;
8  auto const& [f, g, h] = position;
```

Il est possible de décomposer par valeur ou par référence, constante ou non. Attention toutefois, le `const` a des comportements inattendus[2]. Le plus simple est d'utiliser `auto& [x, y, z]` si l'on souhaite décomposer par référence, et `auto [x, y, z]` si l'on souhaite des valeurs.

Les noms des variables déclarées ne sont pas liées au nom interne de la variable décomposée, attention donc aux erreurs d'inattention : si vous déclarez `[y, x, z]` au lieu de `[x, y, z]`, le compilateur ne pourra pas voir la différence.

L'objet à décomposer peut être de trois sortes : une structure, un tableau, un tuple.

3.1. Décomposer une structure

Le premier type est simple : il s'agit d'une structure dont tous les membres sont publics.

```
1  struct Position3D
2  {
3      int x;
4      int y;
5      int z;
6  };
7
8  Position3D getPosition3D()
9  {
10     return {10, 20, 30};
```

2. Le `const` ne s'applique pas directement à `x`, `y`, `z` mais à la façon dont est récupérée l'expression. Ainsi attendez-vous à des surprises si vous manipulez des références.

```
}

auto pos = getPosition3D();
auto& [x, y, z] = pos;
y = 40;
assert(pos.y == 40);
```

Cela peut tout aussi bien être une classe dont toutes les variables membres sont en public. Cela ne fonctionne pas si la classe contient des membres privés ou protégés.[3]

Si certaines variables membres sont des références, alors quelque soit la méthode de décomposition, la référence restera une référence et pointera vers le même objet.

```
struct MyStruct
{
    int a;
    int& b;
};

int i = 4;
MyStruct mstr{42, i};

auto [val, refer] = mstr;
val = 10;
refer = 20;

assert(mstr.a != val);
assert(mstr.a == 42);
assert(mstr.b == refer);
assert(mstr.b == 20);
assert(i == 20);
assert(&refer == &i);
```

3.2. Décomposer un tableau natif

Il est recommandé d'utiliser `std::array` plutôt que des tableaux natifs, cela évite des tracas et permet de les manipuler avec la même facilité que les autres conteneurs.[4]

Ici aussi, il est nécessaire de déclarer autant de variables qu'en contient la totalité du tableau décomposé.

3. Une rectification rétroactive a par la suite été adoptée pour exiger seulement que tous les membres soient accessibles dans la portée de l'utilisation de la décomposition.

4. FAQ C++ : Pourquoi dois-je bannir les tableaux C-style et utiliser std : :array à la place ? ; C++ Core Guidelines : Prefer using STL array or vector instead of a C array

```cpp
int pos[3] = {10, 20, 30};
auto& [x, y, z] = pos;
y = 40;
assert(pos[1] == 40);
```

3.3. Décomposer un tuple

Il est possible de décomposer un `std::tuple` et tout type disposant des mêmes éléments qu'un tuple (c'est le cas de `std::pair` et de `std::array` par exemple).

```cpp
std::tuple<int, int, int> getPosition3D()
{
    return {10, 20, 30};
}

auto pos = getPosition3D();
auto& [x, y, z] = pos;
y = 40;
assert(std::get<1>(pos) == 40);
```

Qu'il s'agisse d'une vraie structure ou d'un tuple, la décomposition s'utilise de façon identique, on peut passer de l'un à l'autre de façon transparente.

```cpp
using namespace std::literals;

auto bookmark = std::make_tuple("https://zestedesavoir.com"s,
    "Zeste de Savoir"s, 30);
// Le type déduit de `bookmark` est donc `std::tuple<std::string,
    std::string, int>`

auto& [url, title, visits] = bookmark;
++visits;
assert(std::get<2>(bookmark) == 31);
```

4. Une décomposition personnalisée

De la même façon qu'une décomposition peut s'appliquer sur un `std::tuple` ou un `std::pair`, il est possible de créer ses propres classes à décomposer.

Il n'est pas nécessaire de savoir créer des décompositions sur des classes pour les utiliser. Néanmoins, il peut être intéressant de comprendre comment cela fonctionne.

Prenons un exemple de `Position2D` qui inclue un nom optionnel. Si la position n'a pas de nom défini, elle est nommée automatiquement en fonction de sa position.

```
struct Position2D
{
    Position2D() = default;
    Position2D(int x, int y): x(x), y(y) {}
    Position2D(int x, int y, std::string name): x(x), y(y),
        m_name(std::move(name)) {}

    int x = 0;
    int y = 0;

    std::string getName() const
    {
        if(m_name.empty()){
            return std::to_string(x) + ":" + std::to_string(y);
        }
        return m_name;
    }

private:
    std::string m_name;
};
```

On cherche à décomposer par valeur (autrement cela créerait des références fantômes) :

```
int main()
{
    {
        Position2D positionA {10, 20, "Player"};

        auto [x, y, name] = positionA;
        assert(name == "Player");
    }
    {
        Position2D positionB {10, 20};

        auto [x, y, name] = positionB;
        assert(name == "10:20");
    }
}
```

Il nous faut maintenant définir trois choses pour que la décomposition soit possible :

— la classe `tuple_size` doit être spécialisée pour indiquer la taille de la décomposition ;
— la classe `tuple_element` doit être spécialisée pour indiquer le type de chacun des éléments de la décomposition ;
— la fonction `get` doit être spécialisée pour récupérer chaque élément.

4.1. `tuple_size`

La première étape est d'indiquer la taille de la décomposition. Pour cela, on va spécialiser `std::tuple_size`. C'est un des rares cas où il est autorisé d'ajouter quelque chose dans le `namespace std`.

```
1 namespace std {
2     template<> class tuple_size<::Position2D>: public
          integral_constant<size_t, 3> {};
3 }
```

L'expression `std::tuple_size<Position2D>::value` vaut maintenant `3`.

i

À noter l'usage de `::Position2D` pour préciser qu'il s'agit de la classe `Position2D` qui se trouve dans l'espace de nom global, et non pas de `std::Position2D`.
Si votre classe est dans un `namespace malib`, faites bien attention à refermer votre `namespace` avant de déclarer la spécialisation, autrement vous allez créer une classe `malib::std::tuple_size` et vous arracher les cheveux pour comprendre pourquoi votre code ne compile pas.

4.2. `tuple_element`

C'est le même principe pour `std::tuple_element` que nous allons spécialiser pour renseigner le type de chaque élément de la décomposition.

```
1 namespace std {
2     template<> class tuple_element<0, ::Position2D>{ public: using
          type = int; };
3     template<> class tuple_element<1, ::Position2D>{ public: using
          type = int; };
4     template<> class tuple_element<2, ::Position2D>{ public: using
          type = std::string; };
5 }
```

`std::tuple_element<0, Position2D>::type` vaut maintenant `int` ;

`std::tuple_element<1, Position2D>::type` vaut maintenant `int`;

`std::tuple_element<2, Position2D>::type` vaut maintenant `std::string`.

Même remarque que précédemment.

4.3. get

La dernière étape, spécialiser la fonction `get` pour récupérer chaque élément. Contrairement aux précédents, elle doit être dans le même espace de nom que notre `Position2D`.

```
template<size_t I>
constexpr auto get(Position2D const& pos)
{
    if constexpr(I == 0){
        return pos.x;
    } else if constexpr(I == 1){
        return pos.y;
    } else if constexpr(I == 2){
        return pos.getName();
    }
}
```

Le paramètre template `I` indique l'élément que l'on souhaite récupérer. Avec une série de `if constexpr`, on retourne la valeur appropriée.

On peut maintenant faire `int y = get<1>(maPosition);` pour récupérer la composante Y de `maPosition`.

Cela fonctionne aussi si la fonction `get()` est membre de la classe `Position2D`, on pourra faire `maPosition.get<1>()` pour obtenir la position en Y.

4.4. Utilisation de notre décomposition et code complet

Nous voici au bout de notre décomposition !

On peut par exemple l'utiliser dans une boucle `for` :

```
std::vector<Position2D> positions = {
    {10, 20},
    {30, 40, "Door"},
```

```
    {50, 60},
    {70, 80, "Window"}
};

for(auto [x, y, name] : positions){
    std::cout << "{" << x << ", " << y << "} is named " <<
        std::quoted(name) << '\n';
}
```

`std::quoted(name)` permet de mettre facilement le nom entre guillemets (et de gérer les cas particuliers où il y a des guillemets dans le nom par exemple).

Enfin voici le code complet :

Contenu masqué n°1

5. Conclusion

Nous avons fait le tour des décompositions. On ne crée pas une décomposition personnalisée tous les matins, la plupart du temps on peut se contenter d'utiliser celles qui existent déjà.

C'est très bien comme cela puisque l'objectif est de faciliter l'usage et la lisibilité du C++ .

Contenu masqué

Contenu masqué n°1

```
#include <iostream>
#include <iomanip>
#include <cassert>
#include <tuple>
#include <vector>
#include <string>

struct Position2D
{
    Position2D() = default;
    Position2D(int x, int y): x(x), y(y) {}
    Position2D(int x, int y, std::string name): x(x), y(y),
        m_name(std::move(name)) {}
```

Contenu masqué

```
13
14      int x = 0;
15      int y = 0;
16
17      std::string getName() const
18      {
19          if(m_name.empty()){
20              return std::to_string(x) + ":" + std::to_string(y);
21          }
22          return m_name;
23      }
24
25  private:
26      std::string m_name;
27  };
28
29  namespace std {
30      template<> class tuple_size<::Position2D>: public
            integral_constant<size_t, 3> {};
31      template<> class tuple_element<0, ::Position2D>{ public: using
            type = int; };
32      template<> class tuple_element<1, ::Position2D>{ public: using
            type = int; };
33      template<> class tuple_element<2, ::Position2D>{ public: using
            type = std::string; };
34  }
35
36  template<size_t I> constexpr auto get(Position2D const& pos)
37  {
38      if constexpr(I == 0){
39          return pos.x;
40      } else if constexpr(I == 1){
41          return pos.y;
42      } else if constexpr(I == 2){
43          return pos.getName();
44      }
45  }
46
47  int main(){
48      {
49          Position2D positionA{10, 20, "Player"};
50
51          auto [x, y, name] = positionA;
52          assert(name == "Player");
53      }
54      {
55          Position2D positionB{10, 20};
56
57          auto [x, y, name] = positionB;
58          assert(name == "10:20");
```

Contenu masqué

```
    }

    std::vector<Position2D> positions = {
        {10, 20},
        {30, 40, "Door"},
        {50, 60},
        {70, 80, "Window"}
    };

    for(auto [x, y, name] : positions){
        std::cout << "{" << x << ", " << y << "} is named " <<
            std::quoted(name) << '\n';
    }
}
```

Listing 1 – Exécuter dans Wandbox

Retourner au texte.