Le C++20

F. Lassabe

Table des matières

1 Les templates

1.1 Présentation

Les *templates* sont la solution retenue en C++ pour permettre la généricité. En effet, il est courant dans un programme de construire des listes, des tableaux, etc. de données de différents types. Sans généricité, il est fastidieux de construire ces ensembles. En effet, si on veut une liste de doubles, une liste d'entiers, et une liste de chaînes de caractères, il sera nécessaire d'implémenter le fonctionnement de 3 listes différentes, et de manipuler autant de types de données (liste_entiers, liste_doubles, etc.). La généricité, et donc les *templates*, répond à ce problème en permettant de créer des classes dont le type de certains membres sont paramétrables.

Les *templates* peuvent s'appliquer directement à une classe, ou à une fonction. Schématiquement, le code est dupliqué avec les différents types utilisés avant d'être compilé. Pour être utilisés, les templates reposent sur une syntaxe pour les déclarer, et une syntaxe pour les utiliser.

— Déclaration d'un template

```
template <typename T> // de préférence
// ou
template <class T>
// puis, élément sur lequel le template s'applique :
class A {
```

```
T _member;
// ...
};
// ou
int do_something(const T &obj) { // ...
```

— Utilisation du template

```
// Sur classe :
A<int> objet_a;
// Sur fonction :
int res = do_something<double>(42.0);
```

En raison de la façon de passer du code générique au programme réel, il n'est pas possible de procéder à de la compilation séparée quand on utilise les templates. On écrira donc le code générique dans un seul fichier, par convention suffixé .hpp, qui sera ensuite inclus par ses utilisateurs et compilé en même temps qu'eux.

1.2 Application

Cet exercice consistera à implémenter un conteneur dynamique similaire au `std::vector` simplifié. Nous le nommerons `vecteur`. Notre conteneur est paramétrable (il peut contenir n'importe quel type de donnée) grâce à un template. Il gère ses données avec 3 attributs :

— `memory_allocated` de type `size_t`, qui définit combien de mémoire est actuellement disponible pour stocker des éléments,
— `elements_count` de type `size_t`, qui définit combien d'éléments sont déjà dans le tableau. `elements_count` a donc une valeur qui indique l'indice qui suit la position du dernier élément inséré.
— `content` de type pointeur sur le type contenu (T*), est un tableau dynamique alloué sur le tas (i.e. avec new)

Ces attributs sont gérés de la manière suivante : à la construction du vecteur, la taille allouée est de 4. Une fois les 4 éléments affectés, en cas de nouvelle insertion, on double la taille du vecteur. Pour cela, il est nécessaire d'allouer un nouveau tableau, d'y recopier les éléments de l'ancien, puis d'échanger les valeurs des pointeurs (`content` va donc pointer sur le nouveau tableau), et enfin de supprimer l'ancien tableau. Évidemment, `memory_allocated` est mise à jour pour refléter la nouvelle taille du tableau.

Les méthodes de la classe vecteur sont les suivantes :

— Constructeur par défaut
— Destructeur
— `push_back` : ajoute un élément à la fin
— `pop_back` : enlève le dernier élément
— `back` : retourne une référence vers le dernier élément
— `front` : retourne une référence vers le premier élément
— `insert` : insère un élément à la position passée en paramètre
— `size` : retourne le nombre d'éléments assignés
— `empty` : retourne vrai si le vecteur est vide, faux sinon

2 Les exceptions

2.1 Les exceptions en C++

La gestion des exceptions permet au programme de réagir à une condition exceptionnelle qui l'empêcherait de fonctionner. L'avantage des exceptions est de rendre le programme plus lisible, notamment car le bloc **try** ne contiendra que le chemin d'exécution idéal (celui où tout se passe bien), ainsi que de bien couvrir les erreurs possibles. Il y a peu d'inconvénients à utiliser les exceptions, notamment lorsqu'on les compare à la même couverture d'erreurs que des tests de retours de fonctions.

La syntaxe des exceptions en C++ est proche de celle du Java, avec les mot-clés : **try/catch**, **throw**. Certaines classes de la STL lancent des exceptions. Par exemple, certains conteneurs ont des méthodes qui lancent une exception si on essaie d'accéder à des données hors de la plage existante. Vos propres classes peuvent être implémentées de façon à lancer des exceptions, qui peuvent être personnalisées en définissant des classes héritant de `std::exception`.

2.2 Mise en application

2.2.1 Exercice 1

À l'aide d'un bloc **try/catch**, sécuriser le code suivant :

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> v{1, 2, 3, 4};
  cout << v.at(0) << " " << v.at(4) << endl;
  return 0;
}
```

Il vous sera nécessaire de lire la documentation de la méthode **at** de la classe **vector** pour résoudre ce problème.

2.2.2 Exercice 2

Dans cet exercice, vous modifierez la classe vecteur de la première partie du TP pour lui ajouter des méthodes d'accès aléatoire : une méthode at(int idx) qui retourne l'élément à la position idx, et une surcharge de l'opérateur `[]` qui fait de même. Les deux devront retourner une exception `std::out_of_range` si l'index demandé n'existe pas.

Vous définirez ensuite une exception nommée `empty_container_exception` qui sera lancée par les méthodes `pop_back`, `back` et `front` dans le cas où le vecteur est vide. ,

3 Les pointeurs intelligents

3.1 RAII

RAII est un idiome C++ qui vise à lier le cycle de vie d'une ressource à celle de la classe qui en a la responsabilité. Plus qu'une question d'initialisation (contrairement à ce que le nom laisse penser), RAII nécessite de libérer les ressources à la destruction d'un objet. Pour celà, RAII s'appuie sur le destructeur de la classe pour libérer l'ensemble des objets gérés hors de la pile (variables sur le tas, flux ouverts, sémaphores, mutex, etc.). Cet idiome et son application ont pour objectif et permettent d'éviter les fuites mémoire, les blocages de ressource, etc. Pour implémenter des applications conformes à l'idiome RAII, on s'appuie notamment sur la STL (Standard Template Library) et des classes *wrapper*. Notamment, la mémoire dynamique gagne à être gérée par des pointeurs intelligents.

3.2 Les pointeurs intelligents

Ils sont définis dans les bibliothèques standard du C++, et utilisables avec l'inclusion du fichier d'entête `memory`. Il existe deux principales classes de pointeurs intelligents : le pointeur unique (classe `std::unique_ptr`) et pointeur partagé (classe le `std::shared_ptr`). Une troisième classe, pour éviter des cycles de dépendance entre pointeurs partagés, existe, le pointeur faible (classe `std::weak_ptr`), mais nous ne la traiterons pas dans cette ressource.

3.2.1 Le pointeur unique

Le pointeur unique est une classe qui gère un pointeur ou un tableau et en détient la propriété sans partage. En conséquence :

- Le pointeur unique n'est pas copiable (il ne serait plus unique)
- On peut en obtenir le pointeur brut sous-jacent (pour utilisation uniquement)
- Il est détruit et dé-alloué quand on sort de son *scope*

```
void f() {
  std::unique_ptr<double> uptrd = std::make_unique<double>(1234.0);
  std::cout << *uptrd << std::endl;
} // Sortie du scope : mémoire libérée
```

Plusieurs méthodes notables de la classe `std::unique_ptr` existent, dont les suivantes :

- `get()` : retourne le pointeur sous-jacent
- `reset(pointeur=nullptr)` : réaffecte le pointeur (dé-alloue l'ancienne cible)
- Opérateur booléen : teste la validité
- Opérateur `*` : déréférence (accède à la variable gérée)
- Opérateur `->` : accède aux champs de la variable pointée
- Opérateur `[]` : pour les tableaux, accès à un élément

Exemple d'utilisation de get() :

```
#include <memory>
#include <iostream>

void mod_double(double *d, double v) { // modifie un double
  *d = v;
}

int main() {
  std::unique_ptr<double> uptrd = std::make_unique<double>(1234.0);
  std::cout << *uptrd << " ";
  mod_double(uptrd.get(), 5555.0);
  std::cout << *uptrd << std::endl;
  return 0;
}
```

Lors de l'utilisation, le programme affichera quelque chose de similaire à ça :

```
$ ./unique
1234 5555
```

La méthode reset est utilisée dans l'exemple suivant deux fois : une première fois pour supprimer l'objet géré (et le dé-allouer de la mémoire), une seconde fois pour confier la responsabilité d'un nouvel objet au pointeur unique (ce n'est pas la façon la plus sûre de le faire, il serait préférable d'utiliser la fonction `std::make_unique`).

```
#include <iostream>
#include <memory>
#include <string>

int main() {
  std::unique_ptr<std::string> uptrs = std::make_unique<std::string>("Hello");
  if (uptrs) {
    std::cout << *uptrs << std::endl;
  }
  uptrs.reset(); // Plus d'objet géré
  if (!uptrs) {
    std::cout << "Aucun objet géré" << std::endl;
    uptrs.reset(new std::string{"abcd"}); // pas très safe
    std::cout << *uptrs << std::endl;
  }
  return 0;
}
```

On obtiendra une sortie de ce genre :

```
$ ./unique2
Hello
Aucun objet géré
abcd
```

L'exemple ci dessous illustre l'utilisation du pointeur unique pour gérer une structure, ainsi que l'accès aux membres de la structure avec l'opérateur `->` (ligne 12) de déréférencement à partir d'un pointeur, ou l'opérateur `*` (ligne 13) de déréférencement de pointeur, suivi de la notation pointée d'accès aux membres d'une classe :

```cpp
#include <iostream>
#include <memory>
#include <string>

struct S {
  int a = 42;
  std::string s = "hi!";
};

int main() {
  auto uptrstruct = std::make_unique<S>();
  std::cout << uptrstruct->a << " "
            << (*uptrstruct).s << std::endl;
  return 0;
}
```

On observera le résultat suivant :

```
$ ./unique3
42 hi!
```

Ce dernier exemple démontre l'utilisation du pointeur unique pour déclarer un tableau dynamique et l'utiliser :

```cpp
#include <iostream>
#include <memory>

int main() {
  auto uptra = std::make_unique<int[]>(10);
  for (size_t i=0; i<10; ++i)
    uptra[i] = i + 1;
  for (size_t i=0; i<10; ++i)
    std::cout << uptra[i] << " ";
  std::cout << std::endl;
  return 0;
}
```

La sortie sera :

```
$ ./unique4
1 2 3 4 5 6 7 8 9 10
```

3.2.2 Le pointeur partagé

Comme son nom l'indique, le pointeur partagé `std::shared_ptr` partage la responsabilité de la mémoire gérée avec d'autres pointeurs partagés, construits par copie. La classe connaît le nombre de références à la mémoire, ce qui lui permet de dé-allouer la mémoire cible lors de la suppression de la dernière référence (i.e. la dernière instance de `std::shared_ptr`) qui lui est faite.

L'exemple ci-dessous utilise une fonction pour instancier un double et renvoie le double sous forme d'un pointeur partagé, qui sera ensuite copié dans une variable du main. La référence sptrd dans f est détruite, mais comme il en existe encore une (sptrd dans main), l'instance du double n'est pas dé-allouée. À la sortie du main, sptrd est détruit, amenant le compteur de références à zéro, ce qui provoque la dé-allocation de la mémoire gérée.

```cpp
auto f() {
  std::shared_ptr<double> sptrd = make_shared<double>(4321.0);
  return sptrd;
} // pas détruit car copié dans l'appelant

int main() {
  auto sptrd = f();
  std::cout << *sptrd << std::endl;
  return 0;
} // sptrd détruit ici
```

Les méthodes présentées ci-dessus pour manipuler les pointeurs uniques sont également offertes par les pointeurs partagés. D'autres méthodes y sont ajoutées, en particulier `use_count()`, dont voici un exemple :

```
#include <iostream>
#include <memory>

void fun(std::shared_ptr<int> sp) {
    std::cout << "in fun(): sp.use_count() == " << sp.use_count()
              << " (object @ " << sp << ")\n";
}

int main() {
    auto sp1 = std::make_shared<int>(5);
    std::cout << "in main(): sp1.use_count() == " << sp1.use_count()
              << " (object @ " << sp1 << ")\n";

    fun(sp1);
}
```

Le résultat sera celui-ci :

```
in main(): sp1.use_count() == 1 (object @ 0x20eec30)
in fun(): sp.use_count() == 2 (object @ 0x20eec30)
```

3.3 Application

Reprenez les exercices du TP1 avec pointeurs, et remplacez les pointeurs par des pointeurs intelligents.

4 La STL

4.1 Présentation succincte

La STL, pour *Standard Template Library*, répond à des besoins récurrents dans l'écriture de programmes informatiques, notamment la gestion de séquences de données. Une séquence ou une collection de données peut se représenter sous plusieurs formes en mémoire :

— Contigües : tableaux, tableaux dynamiques (`std::vector`)
— Associatifs : maps
— Triées par nature : set
— Non contigües : list, deque

Comme les données stockées dans des collections ne seront pas les mêmes d'un programme à l'autre (il sera nécessaire de pouvoir stocker parfois des doubles, des chaînes de caractères, etc.), la STL propose une implémentation paramétrable des collections, avec une interface cohérente pour leur utilisation. Les structures de données proposées sont notamment :

array : tableau de taille fixe

vector : tableau de taille modifiable

list : liste doublement chaînée

set : séquence de valeurs ordonnées, accessible en temps logarithmique

map : association de clés (ordonnées) et de valeurs, accessible en temps logarithmique

unordered_set : comme set, mais en table de hachage (accès en temps constant)

unordered_map : version table de hachage de la map

Comme la structure sous-jacente des conteneurs varie, leurs méthodes de manipulation ne sont pas toutes les mêmes. Ils disposent néanmoins d'un socle commun, dont les éléments les plus utiles sont :

— opérateur d'affectation
— begin/end : range-for
— empty, size (sauf `forward_list`)
— swap
— insert, clear (sauf `stack` et `queue`)

Les conteneurs courants sont décrits dans les sous-sections suivantes. Pour en savoir plus sur leur fonctionnement, il sera nécessaire de lire la documentation de chacun des conteneurs sur le site cppreference.com.

4.1.1 La classe array

La classe array définit un tableau de taille fixe connue à la compilation. Ses deux paramètres de template sont le type contenu et le nombre d'éléments du tableau. La classe array n'est pas dynamique. Voici un exemple d'utilisation de cette classe :

```
int main(int argc, char *argv[]) {
  std::array<double, 10> a;
  for (const double d: a)
    std::cout << d << " ";
  std::cout << std::endl;
  return 0;
}
```

4.1.2 La classe vector

La classe vector définit un conteneur séquentiel contigu (comme un tableau) dont la taille peut évoluer au fil du temps. La méthode `capacity` permet de connaître le nombre d'éléments alloués actuellement à un vecteur (ce nombre est supérieur ou égal au nombre d'éléments déjà affectés dans le vecteur). L'exemple ci-dessous montre un usage simple de la classe vector :

```
int main(int argc, char *argv[]) {
  std::vector<int> v{1, 2, 3};
  for (int i=1; i<argc; ++i)
    v.push_back(std::strtoi(argv[i], NULL, 10));
  for (const int value: v)
    std::cout << value << " ";
  std::cout << std::endl;
  return 0;
}
```

4.1.3 La classe list

Ce conteneur est une liste doublement chaînée. En conséquence, les ajouts et suppression d'éléments sont peu coûteux, mais le parcours et l'accès à une position de la liste sont coûteux (il faut partir d'un bout de la liste et parcourir élément par élément jusqu'à atteindre l'élément souhaité). L'exemple qui suit est un usage simple de la liste :

```
int main(int argc, char *argv[]) {
  std::list<int> liste{1, 2, 3, 4};
  auto iter = liste.find(3);
  liste.insert(iter, 42);
  for (const int entier: liste)
    std::cout << entier << " ";
  std::cout << std::endl;
  return 0;
}
```

On remarque que l'usage n'est pas vraiment différent de celui du vecteur, grâce à la cohérence des API de la STL. En revanche, les structures de données internes étant très différentes, les performances varieront grandement en fonction du cas d'utilisation. Il est donc nécessaire de choisir soigneusement la structure utilisée en fonction de l'application. Il faut cependant noter que, comme dans le cas courant, trouver un élément à supprimer dans une séquence nécessite de parcourir la séquence avec une liste, si l'on a un doute sur l'avantage d'utiliser une list plutôt qu'un vector, il sera généralement mieux de choisir le vector.

4.1.4 La classe set

La classe set stocke une séquence ordonnée d'éléments. Les objets contenus par un set doivent être des instances d'une classe qui a une méthode de comparaison "inférieur à" (ou `operator<`). L'exemple suivant, une fois compilé et exécuté, affiche les mots du set dans leur ordre alphabétique.

```cpp
int main() {
  std::set<std::string> s{"toto", "hello", "world", "all"};
  for (const auto &entry: s)
    std::cout << entry << " ";
  std::cout << std::endl;
  return 0;
}
// affiche: all hello toto world
```

4.1.5 La classe map

Similairement à la classe set, la classe map permet de définir un conteneur associatif, c'est-à-dire qui associe à un objet nommé *clé*, un autre objet nommé *valeur*. Comme pour set, la structure fait que le stockage est ordonné par clé (on pourrait dire qu'un set est une map avec seulement des clés). Pour mieux comprendre la différence, voici un exemple :

```cpp
int main() {
  std::map<std::string, double> m{{"Bob", 12.0}, \
    {"Alice", 25.0}, {"Trudy", 23.0}};
  m.insert({"John", 44.0});
  for (const auto &entry: m)
    std::cout << entry.first << " -> " << entry.second \
              << std::endl;
  return 0;
}
// affiche:
//Alice -> 25
//Bob -> 12
//John -> 44
//Trudy -> 23
```

4.1.6 La classe unordered_map

Cette classe fonctionne comme la map, mais le stockage n'est pas ordonné (ni même dans l'ordre d'insertion). La unordered_map est stockée sous forme de table de hachage. Sur de gros volumes de données, c'est ce type de conteneur qui aura les meilleures performances. L'ordre arbitraire (lié à la fonction de hachage appliquée à la clé) est visible dans l'exemple ci-dessous :

```cpp
int main() {
  std::unordered_map<std::string, double> m{{"Bob", 12.0}, \
    {"Alice", 25.0}, {"Trudy", 23.0}};
  m.insert({"John", 44.0});
  for (const auto &entry: m)
    std::cout << entry.first << " -> " << entry.second \
              << std::endl;
  return 0;
}
// affiche:
//John -> 44
//Trudy -> 23
//Alice -> 25
//Bob -> 12
```

4.1.7 Les multisets et multimaps

Les classes suivantes :

- `multiset`
- `multimap`
- `unordered_multiset`
- `unordered_multimap`

fonctionnent comme leurs homologues non multi, mais permettent d'avoir plusieurs valeurs par clé (plusieurs instances de la même clé dans le cas des *multiset). C'est ce qu'illustre le code ci-dessous :

```
int main() {
  std::unordered_multimap<std::string, double> m{{"Bob", 12.0}, \
    {"Alice", 25.0}, {"Trudy", 23.0}};
  m.insert({"Bob", 28.0});
  for (const auto &entry: m)
    std::cout << entry.first << " -> " << entry.second \
              << std::endl;
  return 0;
}
// affiche:
//Bob -> 12
//Bob -> 28
//Alice -> 25
//Trudy -> 23
```

4.2 Travail à réaliser

Le travail consiste à exploiter des données présentées sous formes de lignes (un vector par ligne) pour en tirer des données agrégées en s'appuyant sur les structures de la STL. Chaque ligne apparaît sous la forme d'un `std::vector<std::string>` dont les champs sont (dans l'ordre d'apparition) :

- Coordonnée x (représentation ASCII d'un double)
- Coordonnée y (représentation ASCII d'un double)
- Coordonnée z (représentation ASCII d'un double)
- Direction (représentation ASCII d'un entier, entre 0 (inconnu), 1 (nord), 2 (est), 3 (sud), 4 (ouest)
- Une succession de champs allant par deux (à déduire de la longueur du array) :
  - Une adresse MAC au format XX :XX :XX :XX :XX :XX
  - Une mesure de signal en dBm (représentation ASCII d'une valeur double)

Pour permettre le parcours des vecteurs de lignes, ces dernières sont stockées dans un `std::vector` (on a donc un vecteur de vecteurs de chaînes de caractères). Votre objectif est de produire deux résultats :

- Une map dont la clé est (x, y, z, direction), et la valeur une map dont la clé est une adresse MAC et la valeur une moyenne des valeurs rencontrées pour cette adresse MAC. Il y a donc autant d'enregistrements dans cette map que dans le vecteur fourni, mais seulement une paire adresse/mesure par adresse différente.
- Un set avec les adresses MAC lues dans l'ensemble des données.

Un fichier cpp à compléter vous est fourni (avec le code nécessaire pour lire un fichier de données et en faire le vecteur de vecteurs), ainsi qu'un fichier des données à agréger au format CSV.