C++ templates

S. Genaud

July 29, 2022

Outline

Table of Contents

Fonctions ou Classes

- la notion de `template` est une arme supplémentaire pour rendre votre code générique
- Le template peut être utilisé dans des fonctions, ou dans des classes
- Vous avez déjà intuitivement utilisé des classes munies de templates :

```
std::vector<int> tabint = { 1, 2, 3, 4 };
std::vector<std::string> tabstr = { "Hello", "World" };
```

Motivation

- On a déjà vu la surcharge de fonctions : des fonctions ayant le même nom mais pas les mêmes paramètres.
- Exemple:

```
int max(int,int);
double max(double,double);
```

- Intéressant mais le programmeur doit définir la fonction pour chaque type.
- Le template permet de demander au compilateur de gérer ça.

Fonctions Template

- Une fonction template utilise un type générique annoncant un type générique : `template<typename T>`

```
template <typename T>
T max(const T& a, const T& b) {
 return a > b ? a : b;
}
```

- Le compilateur va générer le code en inférant le type adéquat.
- Dans l'appel, on instancie le type avec < > (appelé aussi spécialisation du type)

```
...
int a = max<int>(3,5);
...
```

Déduction de type

- Pour les fonctions templates le compilateur est capable de déduire les types à utiliser sans nécessairement les spécialiser.
- Dans l'exemple précédent, on peut écrire l'appel de deux façons équivalentes:

```
...
int a = max<int>(3,5);
int a = max(3,5); // equivalent
...
```

car le compilateur sait que 3 et 5 sont des entiers, et donc que `T` est `int`.

Fonctions Template

- Une fonction template peut avoir plusieurs types génériques ou ordinaires.

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

template <typename U, typename V>
U foo(V vect, U factor, std::string sep)
{
   U res(0); // donc U peut etre un int, float, double, ...
   for (auto v : vect ) { // auto deduit type U
      std::cout << v << sep;
      res += v * factor;
   }
   std::cout << std::endl;
   return res;
}


int main() {
    std::vector<int> v = {1,2,3,4};
    int sum = foo(v, 4., ";");
    std::cout << "sum=" << sum << std::endl;
    return 0;
}
```

- A noter : le prototype et la définition d'une fonction template doivent obligatoirement se trouver dans un fichier d'en-tête.

Exemple affichage générique d'un itérable

- Dans cet exemple : une fonction template fait l'hypothèse qu'un objet à afficher est de type `iterable`. Le code utilise le `for` applicable à tout itérable.

```
template <typename I>
void afficher(I const & iterable) {
   for (auto const & e : iterable)
      cout << e << "␣";
   cout << endl;
}
```

```
// Un std::vector est iterable, donc aucun probleme.
 vector<int> tab_entiers { 1, 3, 5, 7, 9 };
 afficher(tab_entiers);

 // std::array.
 array<double, 4> const tab_reels { 1.2, 3.1415, 12.5, 2.7 };
 afficher(tab_reels);

 // std::list.
 list<string> const liste_chaines { "abc", "def", "ghi"};
 afficher(liste_chaines);

 // Une chaine de caracteres est aussi iterable.
 string const chaine { "ceci␣est␣une␣chaines␣de␣caracteres" };
 afficher(chaine);
```

Classe Template

- De la même manière, une classe peut utiliser un type générique. Exemple du rectangle :

```
 template <typename T>
 class Rectangle {
  private:
    T hauteur;
    T largeur;
  public:
    Rectangle(T hauteur, T largeur)
    : hauteur(hauteur), largeur(largeur)
    { }

    T surface () const {
       return hauteur*largeur;
    }
};
```

- Vous remarquez que la déclaration du type `T` devant la classe étend sa portée aux membres de la classe (inutile de redéclarer `T`).

Class Template

- Il est possible d'écrire l'implémentation des méthodes à l'extérieur de la classe ....

```
template <typename T>
class Rectangle{
  public:

    // ...
    T surface() const;
    // ...
  };
```

- ... mais
  - elle doit apparaitre dans le même fichier `.h`.
  - il faut redéfinir le type `T` ainsi que le type `T` pour la classe:

```
template<typename T>
T Rectangle<T>::surface() const {
   return hauteur * largeur;
}
```

Déduction de Type pour Class Template

- On a vu que la déduction de type pour les fonctions template permettait de ne pas instancier le type lors de l'appel lorsqu'il pouvait être déduit
- Pour les class template, cette possibilité n'existe qu'à partir du C++17. Appelée Class template argument deduction (CTAD) .
- Par exemple, jusqu'à C++14 inclus, créeer une paire d'entier nécessite de spécialiser le type même si les arguments permettent de le déduire:

```
std::pair<int, int> p(3, 5);
```

- Pour contourner le problème une fonction template est fournie :

```
std::pair<int, int> p = std::make_pair(3, 5); // retourne std::pair<int, int>
auto p = std::make_pair(3, 5); // idem mais c++11 permet auto
```

- A partir de C++17, possibilité d'écrire directement

```
auto p = std::pair(3, 5);
```

Class Template : type par défaut

- Il est possible de préciser un défaut pour le type :

```
template <typename T = double>
class Rectangle {
  private:
    T hauteur;
    T largeur;
  public:
    Rectangle(T hauteur, T largeur)
    : hauteur(hauteur), largeur(largeur)
    { }

    T surface () const {creturn hauteur*largeur;c}
};

int main() {
 Rectangle<> r1; // possible de ne rien preciser, ce sera double
 Rectangle<int> r2; // toujours possible de preciser, ici int
 // ...
}
```

Class Template : non-type template parameters

- Il est possible aussi de donner des paramètres par défaut au template. Ces non-type template parameters sont limités et doivent pouvoir s'évaluer à la compilation. On ne peut utiliser que des entiers par exemple pour préciser une valeur, souvent utilisées pour paramètrer la taille d'un objet.

```
template<typename T = int, size_t taille=1>
class Tableau {
  private:
    T * data;

  public:
  // constructeur, alloue dynamiq. l'espace
  Tableau() : data(new T [taille] )
   {}

  // destructeur
   ~Tableau() {
     delete [] data;
   }
};

int main() {
  Tableau<> T1; // tableau de 1 entier
  Tableau<std::string, 10> T2; // tableau de 10 chaines
}
```

Table of Contents

Objectifs

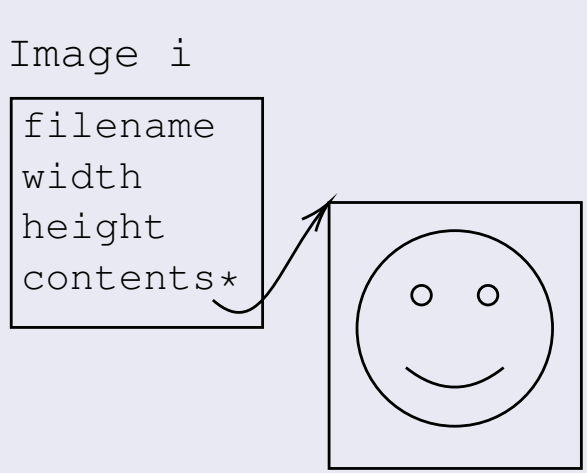
- L'allocation et la libération de mémoire sont explicites avec `new` et `delete`
- Les mécanismes de *class* et *template* permettent d'encapsuler cette gestion. Objectifs: le programmeur
  - n'a plus à gérer explicitement la libération
  - est protégé de certains pièges comme l'*aliasing*
- Deux classes essentielles sont disponibles :
  - `unique_ptr`
  - `shared_ptr`

Exemple motivation pointeur partagé

- Classe `Image` dont le contenu `contents` est alloué sur le tas
- La classe doit libérer les ressources qu'elle a allouées

Image.hpp

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

class Image {
   private:
      std::string filename;
      size_t width;
      size_t height;
      size_t * contents;

   public:
      Image(std::string filename, size_t width, size_t height)
      : filename(filename),width(width),height(height),
        contents(new size_t [width * height])
        {}

      ~Image() {
         delete [] contents;
      }
      std::string get_name() const { return filename;}
};
```



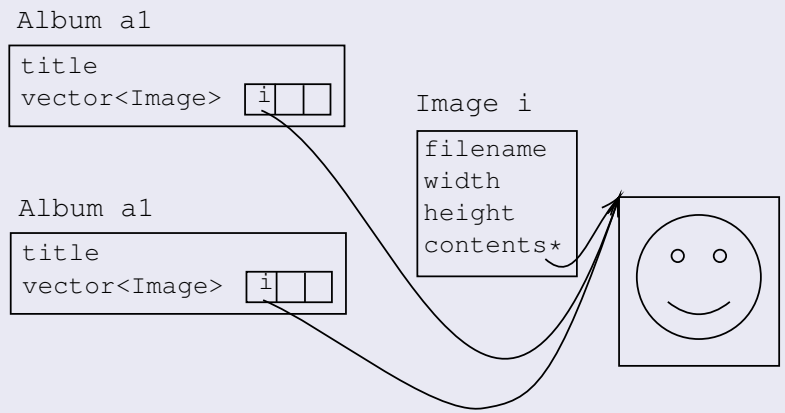
Exemple motivation pointeur partagé

- Deux albums utilisant la même image

Album.hpp

```
#include <iostream>
#include <vector>
using namespace std;

#include "image.hpp"

class Album {

  private:
    string title;
    vector<Image> images;
  public:
    Album(string title) : title(title) {}

    void add(Image image) {
       images.push_back(image);
    }

    ~Album() {}
 };
```

On veut:



Exemple motivation pointeur partagé

- Utilisation

main_album.cpp

```
#include "image.hpp"
#include "album.hpp"

 void add_image_to_albums(Image i, Album * albums[2]) {
   for (size_t cnt=0;cnt<2;cnt++)
       // add Image i to the vector
       albums[cnt]->add(i);
 }

int main() {
    Image i("smiley.jpg",256,256);
    Album a1("album1");
    Album a2("album2");

    // Create a list of 2 albums
    Album * va[2] = {&a1, &a2};

    // Add the image to the 2 albums
    add_image_to_albums(i, va);

    return 0;
 };
```

Exemple motivation pointeur partagé

- Quel problème se pose, par exemple quand on supprime l'un des deux albums ?

Exemple motivation pointeur partagé

- Quel problème se pose, par exemple quand on supprime l'un des deux albums ?
- Un album contient un `vector` de copies d'`Image`
- Quand l'album est supprimé $\rightarrow$ le vector `images` est supprimé
- Une `Image` est supprimée $\rightarrow$ son destructeur est invoqué, `contents` est libéré
- Les copies de cette image dans d'autre albums pointent vers la zone libérée ☹
- Toute suppression d'un autre album (ou copie temporaire d'album) contenant cette image conduira à un *double free*.

shared_ptr

- shared_ptr encapsule un pointeur vers une ressource
- tient le compte du nombre de références sur la ressource
- libère si besoin si la ressource si 0 références à la ressource

shared_ptr

```
#include <iostream>
#include <memory>
using namespace std;

int main() {
  int * tab = new int [10]; // ressource
  // on declare un pointeur partage pointant sur tab
  shared_ptr<int *> p1 = make_shared<int *>(tab);

  cout << "#references␣a␣la␣ressource␣:" << p1.use_count() << endl; // affiche 1

  // on declare un autre pointeur sur la ressource
  shared_ptr<int *> p2(p1); // ou p2 = p1;

  cout << "#references␣a␣la␣ressource␣:" << p1.use_count() << endl; // affiche 2
  // aussi 2 pour p2.use_count()

  // Si on dereference le shared_ptr, on obtient l'adresse de la ressource pointee.
  cout << "valeur␣pointee␣par␣p1␣et␣p2:␣*p1==" << *p1 << "==*p2==" << *p2 << endl ;
  // aussi *p1.get() et *p2.get()

  // acces ecriture/lecture a la ressource:
  (*p1)[0] = 42; // ecriture dans la premiere case du tableau pointe
  cout << (*p2)[0] << endl; // affiche 42
 return 0;
}
```

shared_ptr::use_count()

```
#include <iostream>
#include <memory>
using namespace std;

int main() {

  int * tab = new int [10]; // ressource
  shared_ptr<int *> p1 = make_shared<int *>(tab);
  cout << "#references␣a␣la␣ressource␣:" << p1.use_count() << endl; // affiche 1

  {
     // on declare un autre pointeur sur la ressource
     shared_ptr<int *> p2(p1); // ou p2 = p1;

     cout << "#references␣a␣la␣ressource␣:" << p1.use_count() << endl; // affiche 2

  } // --> a la fin de ce bloc, p2 n'existe plus

  cout << "sortie␣bloc" << endl;
  cout << "#references␣a␣la␣ressource␣:" << p1.use_count() << endl; // affiche 1

  return 0;
}
```

Piege

un nouveau `shared_ptr` ne partage que si `shared_ptr<T> p2 = p1;`

Exemple avec shared

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

class Image {
   private:
      std::string filename;
      size_t width;
      size_t height;
      size_t * contents = nullptr;

   public:
      Image(std::string filename, size_t width, size_t height)
      : filename(filename),width(width),height(height),contents(new size_t [width * height])
      { }
      ~Image() {
         // inutile : delete [] contents;
      }
      std::string get_name() const { return filename;}

};
```

Exemple avec shared

```
#include <memory>
#include <iostream>
#include <vector>
using namespace std;

#include "image_shared.hpp"

class Album {
  private:
    string title;
    vector<std::shared_ptr<Image>> images;

  public:
    Album(string title) : title(title) {
      cout << "Album ’" << title << "’ created." << endl;
    }

    void add(Image image) {
        images.push_back(std::make_shared<Image>(image));
    }

    ~Album() {
      cout <<"Album ’" << title << "’ deleted." << endl;
    }
 };
```

Exemple avec shared

```
#include "image_shared.hpp"
#include "album_shared.hpp"

 void add_image_to_albums(Image i, Album * albums[2]) {
   for (size_t cnt=0;cnt<2;cnt++)
       // add Image i to the vector
       albums[cnt]->add(i);
 }

int main() {
    Image i("smiley.jpg",256,256);
    Album a1("album1");
    Album a2("album2");

    // Create a list of 2 albums
    Album * va[2] = {&a1, &a2};

    // Add the image to the 2 albums
    add_image_to_albums(i, va);

    return 0;
 };
```