Les lambdas en C++

Christophe PICHAUD
.NET Rangers by Sogeti
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com
www.windowscpp.net

Les expressions lambdas font leur entrée dans la programmation C++. C'est une surprise parce qu'il n'y pas de nouvelle syntaxe dans le langage. Chaque chose que fait une lambda peut être quelque chose de fait avec un peu plus de code. Mais les lambdas sont faciles à utiliser pour créer des objets fonctions, et leur impact sur la programmation jour après jour est énorme.

Sans les lambdas, les algorithmes « if » comme std ::find, std ::remove, std ::count_if sont verbeux car il faut définir une callback ; avec les lambdas, ces algorithmes ont un nouveau charme. Il en est de même pour les algorithmes de comparaisons comme std ::sort, std ::nth_element, std ::lower_bound. En dehors de la STL, les lambdas permettent de créer rapidement des deleters custom pour std ::unique et std ::shared_ptr. Au-delà de la STL, les lambdas facilitent la spécification de fonctions callback on-the-fly, des fonctions de contexte spécifique pour des appels directs. Les lambdas rendent le C++ encore plus plaisant à programmer. Le vocabulaire associé aux lambdas peut être confus. Voici un léger rappel : une expression lambda est juste une expression. C'est une partie du code source.

```
auto it = std::find_if(v.begin(), v.end(),
    [](int val) { return val>=5; });
```

Une closure est l'objet runtime créé par la lambda. Tout dépend du mode de capture, les closures gardent une copie ou une référence des données capturées. Dans l'appel à std ::find_if ci-dessus, la closure est l'objet qui est passé au runtime comme 3ème argument de std ::find_if. Une classe closure est une classe depuis laquelle une closure est instanciée. Chaque lambda oblige le compilateur à générer une classe de closure unique. Les opérations à l'intérieur d'une lambda deviennent des instructions exécutables dans une fonction membre de sa classe closure. Une lambda est souvent utilisée pour créer une closure qui n'est employée que comme un argument à une fonction. C'est le cas dans l'appel à std::find_if ci-dessus.

Eviter les modes de capture par défaut

Il y a deux modes de capture par défaut en C++11 : par référence et par valeur. Le mode par défaut par référence peut amener des problèmes. Une capture par référence produit une closure qui contient une référence à une variable locale ou un paramètre qui est disponible dans le scope là où la lambda est définie. Si la durée de vie de la closure créée par la lambda dépasse la durée de vie de la variable locale ou du paramètre, la référence dans la closure est foireuse !
Par exemple, supposons que nous avons un container avec un filter :

```
using FilterContainer = std::vector<std::function<bool(int)>>;
FilterContainer filters;
filters.emplace_back(
    [](int value) { return value % 5 == 0; }
);
```

Cela ressemble à ce qu'on veut mais on ne peut pas hard-coder 5 dans la lambda. Nous devons ajouter un filtre :

```
void addDivisorFilter()
{
  using FilterContainer = std::vector<std::function<bool(int)>>;
  FilterContainer filters;
  auto calc1 = 10;
  auto calc2 = 20;
  auto divisor = 30;
  filters.emplace_back(
      [&](int value) { return value % divisor == 0; }
  );
}
```

Ce code est un code à problème. La lambda réfère à la variable locale divisor mais cette variable cesse d'exister quand addDivisorFilter retourne. C'est immédiatement après que filters.emplace_back retourne donc c'est la mort à l'arrivée. Cela définit un comportement non défini.
Le problème est le même si divisor est capture par référence.

```
filters.emplace_back(
    [&divisor](int value) { return value % divisor == 0; }
```

Avec une capture explicite, c'est plus facile de voir que la viabilité de la lambda est dépendante de la durée de vie de divisor. Ecrire divisor nous rappelle que divisor vit aussi longtemps que la closure de la lambda.
Si vous savez que la closure est utilisée immédiatement (comme passer à un algorithme de la STL) et ne sera pas copiée, il n'y a aucun problème de capture de la variable locale et du paramètre dans l'environnement dans lequel est créée la lambda.
Prenons un autre exemple qui fonctionne avec un appel à std ::all_of :

```
template<typename C>
void workWithContainer(const C& container)
{
  auto calc1 = 10;
  auto calc2 = 20;
  auto divisor = 30;
  using ContElemt = typename C::value_type;
  if (std::all_of(begin(container), end(container),
      [&](const ConstElemt& value)
  {
      return value % divisor == 0;
  }))
  {  }
  else
  {  }
}
```

C'est vrai ça marche, mais c'est précaire.
Sur le long-terme, c'est mieux de lister explicitement les variables locales

et les paramètres dont une lambda dépend.
Il est possible via C++14 d'utiliser auto dans les lambdas.

```
if( std::all_of(begin(container), end(container),
    [&](const auto& value)
{
    return value % divisor == 0;
}))
```

Un moyen de résoudre le problème avec divisor serait une capture par valeur par défaut :

```
filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
);
```

Ceci est un exemple et la capture par valeur par défaut n'est pas le mode qui peut vous exonérer de tous les dangers. Le problème c'est de capter un pointeur par valeur ce qui copie le pointeur dans la closure et que l'on ne peut pas garantir que le code va faire un delete du pointeur et mettre votre application en carafe. On se rassure en se disant que l'on utilise tous des smart pointeurs et qu'il n'existe plus que des fous pour utiliser des pointeurs et jouer avec new et delete ! Prenons un autre exemple :

```
class Widget
{
public:
  Widget() {}
  void addFilter();

  using FilterContainer = std::vector<std::function<bool(int)>>;
  FilterContainer filters;

private:
  int divisor = 10;
};

void Widget::addFilter()
{
  filters.emplace_back(
      [=](int value) { return value % divisor == 0; }
  );
}
```

A première vue, le code est safe… La lambda dépend de divisor mais le mode de capture par valeur par défaut protège bien le code.
Faux ! Les captures ne s'appliquent que pour les variables locales non static (inclure les paramètres) visibles dans le scope là où la lambda est créée. Dans le corps de Widget::addFilter, divisor n'est pas une variable locale, c'est une donnée membre de la classe Widget. Elle ne peut pas être capturée. Le code ne compilera pas !

```
void Widget::addFilter()
{
  filters.emplace_back(
      [](int value) { return value % divisor == 0; }
  );
}
```

Si vous tentez une capture explicite par valeur ou par référence, le code ne compilera pas car divisor n'est pas une variable locale ou un paramètre.

```
void Widget::addFilter()
{
  filters.emplace_back(
      [divisor](int value) { return value % divisor == 0; }
  );
}
```

L'explication tient dans l'utilisation du raw pointeur "this". Chaque fonction membre non static possède un pointeur this et on utilise celui-ci à chaque fois que l'on accède à une variable membre de la classe.
Dans la fonction membre de Widget, le compilateur remplace divisor par this->divisor. Dans la version de Widget::addFilter avec une capture par défaut par valeur,

```
void Widget::addFilter()
{
  filters.emplace_back(
      [=](int value) { return value % divisor == 0; }
  );
}
```

Ce qui est capturé c'est le pointeur this de Widget et non divisor. Le compilateur traite le code comme si on avait écrit cela :

```
void Widget::addFilter()
{
  auto currentObjectPtr = this;
  filters.emplace_back(
      [currentObjectPtr](int value) { return value % currentObjectPtr->divisor == 0; }
  );
}
```

Considérons un autre exemple :

```
using FilterContainer = std::vector<std::function<bool(int)>>;
FilterContainer filters;

void doSomeWork()
{
  auto pw = std::make_unique<Widget>();
  pw->addFilter();
}
```

Quand un appel est fait à doSomeWork, un filtre est créé grâce à un objet Widget créé par std ::make_unique. Le code est foireux…
Ce problème peut être résolu en faisant une copie locale d'une donnée membre que l'on veut capturer. Et ensuite capturer la copie.

```
void Widget::addFilter()
{
  auto divisorCopy = divisor;
  filters.emplace_back(
      [divisorCopy](int value) { return value % divisorCopy == 0; }
  );
}
```

Si vous prenez cette approche, la capture par valeur par défaut marche aussi.

```
void Widget::addFilter()
{
  auto divisorCopy = divisor;
```

```
  filters.emplace_back(
      [=](int value) { return value % divisorCopy == 0; }
  );
}
```

La capture par valeur par défaut a rendue possible la capture du this alors que l'on croyait capturer divisor. En C++14, un meilleur moyen de capturer les données membres est d'utiliser la capture de lambda généralisée.

```
void Widget::addFilter()
{
  filters.emplace_back(
      [divisor = divisor](int value) { return value % divisor == 0; }
  );
}
```

Considérons une nouvelle version de la fonction addDivisorFilter vue précédemment :

```
void addDivisorFilter2()
{
  using FilterContainer = std::vector<std::function<bool(int)>>;
  FilterContainer filters;
  static auto calc1 = 10;
  static auto calc2 = 20;
  static auto divisor = 30;
  filters.emplace_back(
      [=](int value) { return value % divisor == 0; }
  );
  ++divisor;
}
```

N'importe quel lecteur de ce code voit "[=]" et peut penser « OK, la lambda fait une copie de tous les objets ».
La lambda n'utilise aucune des variables non static. En réalité, rien n'est capturé ! En réalité, le code de la lambda utilise la variable static divisor. Si vous restez en dehors du mode de capture par valeur par défaut, vous éliminez le risque que votre code soit foireux !

Conclusion : à retenir

La capture par référence par défaut peut amener des références foireuses. La capture par valeur par défaut peut amener des pointeurs foireux. ●