Lambda functions (depuis C++11)

Core language C++

Construit une fermeture : un objet fonction anonyme capable de capturer des variables dans la portée.

Syntaxe

[*capture* **]** **(** *params* **)** *mutable exception attribute* **->** *ret* **{** *body* **}**	(1)
[*capture* **]** **(** *params* **)** **->** *ret* **{** *body* **}**	(2)
[*capture* **]** **(** *params* **)** **{** *body* **}**	(3)
[*capture* **]** **{** *body* **}**	(4)

1) Déclaration complète 2) Déclaration d'un lambda const: les objets capturés par copie ne peuvent pas être modifiés . 3) Type de retour omis: le type de retour de la fermeture `operator()` est déduite d'après les règles suivantes:

- si le *body* est composé de la mention `return` unique, le type de retour est le type de l'expression retournée (après conversion implicite rvalue-à-lvalue, tableau à pointeur, ou fonction-à-pointeur)
- autrement, le type de retour est `void`

4) Liste des paramètres omis: la fonction ne prend aucun argument, comme si la liste des paramètres est **()**

Explication

mutable - Permet à *body* de modifier les paramètres capturés par copie, et d'appeler leurs fonctions de membre non-const

exception - fournit la spécification d'exception ou la noexcept clause pour l'opérateur () du type de fermeture

attribute - fournit à l'opérateur la spécification d'attribut () du type de fermeture

capture - spécifie les symboles visibles dans le champ où la fonction est déclarée sera visible à l'intérieur du corps de la fonction.

Une liste de symboles peut être adoptée comme suit:

- **[a,&b]** où *a* est capturé par valeur et *b* est capturé par référence.
- **[this]** capte le pointeur **this**
- **[&]** captures all symbols by reference
- **[=]** captures all by value
- **[]** captures nothing

params - La liste des paramètres, comme dans nommée fonctions

ret - Type de retour. S'il n'est pas présent il est implicite dans les énoncés de retour de fonction (ou void si elle ne retourne aucune valeur)

body - Corps de la fonction

L'expression lambda construit un objet non identifié temporaire unique, anonyme non syndiqué non agrégée type, connu sous le nom' **type de fermeture, ce qui comprend les membres suivants:**

ClosureType ::operator()

```
ret operator()(params) const { body }    (le mot-clé mutable n'a pas été utilisé)
ret operator()(params) { body }          (le mot-clé mutable a été utilisé)
```

Executes the body of the lambda-expression, when invoked. When accessing a variable, accesses its captured copy (for the entities captured by copy), or the original object (for the entities captured by reference). Unless the keyword `mutable` was used in the lamda-expression, the objects that were captured by copy are non-modifiable from inside this `operator()`.

Dangling references

If an entity is captured by reference, implicitly or explicitly, and the function call operator of the closure object is invoked after the entity's lifetime has ended, undefined behavior occurs. The C++ closures do not extend the lifetimes of the captured references.

ClosureType ::operator ret(*)(params)

```
typedef ret(*F)(params);
operator F() const;
```

This member function is only defined if the capture list of the lambda-expression is empty.

The value returned by this conversion function is a function pointer that, when invoked, has the same effect as invoking the closure object's function call operator directly.

ClosureType ::ClosureType()

```
ClosureType() = delete;
ClosureType(const ClosureType& ) = default;
ClosureType(ClosureType&& ) = default;
```

Closure types are not DefaultConstructible. The copy constructor and the move constructor are implicitly-declared and may be implicitly-defined according to the usual rules for implicit copier des constructeurs and déplacer des constructeurs.

ClosureType ::operator=()

```
ClosureType& operator=(const ClosureType&) = delete;
```

Closure types are not CopyAssignable.

ClosureType ::~ClosureType()

```
~ClosureType() = default;
```

The destructor is implicitly-declared.

ClosureType ::CapturedParam

```
T1 a;
T2 b;
...
```

If the lambda-expression captures anything by copy (either implicitly with capture clause **[=]** or explicitly with a capture that does not include the character &, e.g. **[a, b, c]**), the closure type includes unnamed non-static data members, declared in unspecified order, that hold copies of all entities that were so captured.

The type of each data member is the type of the corresponding captured entity, except if the entity has reference type (in that case, references to functions are captured as-is, and references to objects are captured as copies of the referenced objects).

For the entities that are captured by reference (with the default capture **[&]** or when using the character &, e.g. **[&a, &b, &c]**), it is unspecified if additional data members are declared in the closure type.

This section is incomplete
Reason: scope rules, capture list rules, nested lambdas, implicit capture vs odr use, decltype

Exemple

Cet exemple montre comment passer un lambda à un algorithme générique et que les objets résultant d'une déclaration lambda, peuvent être stockés dans des objets `std::function` .

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <functional>

int main()
{
    std::vector<int> c { 1,2,3,4,5,6,7 };
    int x = 5;
    c.erase(std::remove_if(c.begin(), c.end(), [x](int n) { return n < x; } ), c.end());

    std::cout << "c: ";
    for (auto i: c) {
        std::cout << i << ' ';
    }
    std::cout << '\n';

    std::function<int (int)> func = [](int i) { return i+4; };
    std::cout << "func: " << func(6) << '\n';
}
```

Résultat :

```
c: 5 6 7
func: 10
```

Voir aussi

auto spécificateur	spécifie un type défini par l'expression (C++11) [edit] (https://fr.cppreference.com/mwiki/index.php?title=Mod%C3%A8le:cpp/language/dsc_auto&action=edit)

function (C++11)	adaptateur générique de foncteur (classe générique) [edit] (https://fr.cppreference.com/mwiki/index.php?title=Mod%C3%A8le:cpp/utility/functional/dsc_function&action=edit)