C++ Lambda

Origin story

introduction

- Apparaît dans le standard de 2011
- Est un sucre syntaxique sur ce qui est déjà possible de faire en cpp
- Permet de créer les outils qui n'ont pas à être référencé
- https://en.cppreference.com/w/cpp/language/lambda

Une fonction en Cpp

```cpp
int plus_one(int x)
{
    return x + 1;
}

int i = plus_one(1);
```

- Un nom unique
- Une liste de parametres
- Un type de retour
- Un body qui contient les instructions a executé

Surcharge de fonction

```
int plus_one(int x)
{
    return x + 1;
}

double plus_one(double x)
{
    return x + 1;
}

int i = plus_one(1);
double i = plus_one(1.1);
```

- Un seul noms mais 2 fonction
- Le paramètre passé permet au compilateur de déduire la fonction a choisir

Name mangling

```
int plus_one(int x)
{
    return x + 1;
}

double plus_one(double x)
{
    return x + 1;
}
```

```
//asm
_Z8plus_onei:
//body


_Z8plus_oned:
//asm
```

- Le compilateur génère 2 nouveau nom
- Name mangling / mangling : plus_one/i/d
- https://gcc.godbolt.org/z/TfK1Ps

Fonction template

```
template <typename T>
T plus_one(T x)
{
    return x + 1;
}

int main() {
    plus_one(1);
    plus_one(1.1);
}
```

```
_Z8plus_onei:
//body


_Z8plus_oned:
//body
```

- Template offre au compilateur de créer les fonctions pour vous
- Globalement équivalent à créer les fonctions soit même.
- https://gcc.godbolt.org/z/o769Wx

Fonction membre d'une classe

```
class Plus {
    const int value;
    public:
    Plus( int v ) : value(v) {}

    int plus_me(int x) {
        return x + value;          _ZNK4Plus7plus_meEi:
    }
};

int main()
{
    auto p = Plus(1);
    p.plus_me(42);
}
```

- Une fonction membre d'un objet reviens toujours à créer une fonction seule
- Quand vous appelez une fonction membre, le compilateur sait que p est de type Plus, que vous tentez d'appeler une méthode plus_me.
- https://gcc.godbolt.org/z/rqWPa9

operator ()

```
class Plus {
    const int value;
    public:
    Plus( int v ) : value(v) {}

    int operator()(int x)          _ZNK4PlusclEi:
    {
        return x + value;
    }
};

int main()
{
    auto p = Plus(1);
    auto i = p(42); // 43
}
```

- Prend utilise la même logique que précédemment.
- Mais ici Plus instancié, se comporte, fonctionnellement comme une fonction simple.
- Functors/Closure pattern
- Utilisé dans QConnect/qt Algo, std algorithm, etc...
- https://gcc.godbolt.org/z/xav8n3

Lambda : faire des functors facilement

```
class Plus {
    const int value;
    public:
    Plus( int v ) : value(v) {}

    int operator()(int x)
    {
        return x + value;
    }
};

int main()
{
    auto p = Plus(1);
    auto i = p(42); // 43
}
```

```
int main()
{
    auto p =  [value=1](int x) { return x + value; };
    // nom généré: _ZZ4mainENKUliE_clEi
    auto i = p(42); // 43
}
```

- Exacte même implementation & comportement.
- Il n'y plus de nom de classe, le compilateur en génère un pour vous.
- Le type de retour de la fonction est déduit automatiquement.
- Les [] qui sont les variable membre, sont appelé capture dans le standard

Exemple

```
bool has_object_with_value(const std::vector<Object>& vec, int val)
{
    auto has_value = [value = val](const Object& obj) {
        return obj.value() == value;
    };

    return std::find_if(vec.begin(), vec.end(), has_value) != vec.end();
}
```

- https://gcc.godbolt.org/z/3sej6h
- Qt connect : https://doc.qt.io/qt-5/signalsandslots.html#advanced-signals-and-slots-usage

Capture par copie par défaut

```
bool has_object_with_name(const std::vector<Object>& vec, std::string name)
{
    auto has_name = [value = name](const Object& obj) {
        return obj.name() == value;
    };

    return std::find_if(vec.begin(), vec.end(), has_name) != vec.end();
}
```

- `name` est copié par défaut dans la variable membre `value`
- Le type des variables, est déduit de la même façon que si vous aviez utilisé `auto`, et sont marqué const par défaut.
- https://gcc.godbolt.org/z/Yrxe4E

Capture par référence

```
bool has_object_with_name(const std::vector<Object>& vec, std::string name)
{
    auto has_name = [&value = name](const Object& obj) {
        return obj.name() == value;
    };

    return std::find_if(vec.begin(), vec.end(), has_name) != vec.end();
}
```

- name est capturé par référence, aucune copie
- Attention aux références qui deviennent invalides
- https://gcc.godbolt.org/z/5b85aW

Capture par move

```
bool has_object_with_name(const std::vector<Object>& vec, std::string name)
{
    auto has_name = [value = std::move(name)](const Object& obj) {
        return obj.name() == value;
    };

    return std::find_if(vec.begin(), vec.end(), has_name) != vec.end();
}
```

- value va voler le contenu de name.
- https://gcc.godbolt.org/z/5b85aW

Raccourci pour la capture

```
bool has_object_with_name(const std::vector<Object>& vec, std::string name)
{
    auto has_name1 = [name](const Object& obj) { return obj.name() == name; };
    // copie name en gardant le même nom

    auto has_name2 = [&name](const Object& obj) { return obj.name() == name; };
    // prend la référence en gardant le même nom

    //auto invalid = [name, &name](const Object& obj) { return obj.name() == name; };
    // vous ne pouvez pas capturer plusieurs fois avec le même nom

    auto has_name3 = [=](const Object& obj) { return obj.name() == name; };
    // copie ce dont la lambda a besoin

    auto has_name4 = [&](const Object& obj) { return obj.name() == name; };
    // prend les références dont la lambda a besoin
    // '&' et '=' ne font jamais de capture des variables globales, vous y avez déjà accès
```

Autre feature

- Convertissable vers un pointer de fonction, si il y a pas de variable capturé :

```
int (*fp)(int) =  [](int x) { return x + 1; };
```

- Constexpr par défaut (depuis c++17):

```
auto lambda = [](int x){ return x + 1 };
static_assert(lambda(42) == 43);
```

Lambda mutable

- Les variable capturé par copie sont marqué const par défaut.
- Si vous voulez les modifier vous devez marquer votre lambda avec le mot clé mutable.

```
int main()
{
    int i = 0;
    //auto lb1 = [i]() { i++; return i; };
    auto lb1 = [i]() mutable { i++; return i; };
    auto lb2 = [&i]() { i++; return i; };

    return lb2();
}
```

- https://gcc.godbolt.org/z/jKnGfq

Forcer une reference constante

```
int main()
{
    int i = 0;
    auto lb2 = [&i = std::as_const(i)]() {
      //i++ ne compile pas
      std::cout << i << '\n';
      };

    return lb2();
}
```

This capture

```
struct Toto
{
    int a, b, c;

    void do_something()
    {
        auto lb = [this]()
        {
            a++;
            b++;
            c++;
        };

        lb();
    }
};
```

- Quand une lambda est créée dans une méthode, vous pouvez capturer this pour pouvoir accéder implicitement à tous les attributs & méthodes dans le body.

Lambda et Qt

```
connect(action, &QAction::triggered, engine,
        [=]() { engine->processAction(action->text()); });
```

- Le 3ème paramètre 'engine' sert pour savoir quel est le QThread qui va être appelé pour exécuter la lambda

Et voila :)

Vous êtes maintenant des pro de la lambda.

Des question ?