Тема 8. Лямбда функции

Это понятие позволяет быстро вставить в код некоторые операции, необходимые, например, для обработки наборов данных с помощью алгоритмов. Выше мы говорили, что для этих целей используются функторы. Но функторы представляют собой отдельный класс со специально определенным оператором (), со всеми его атрибутами. А часто бывает, что нам достаточно лишь определить достаточно простую обособленную функцию, например, сравнения для сортировки или предиката для каких-нибудь условий.

Формально, лямбда-функция — это определение “почти обычной” функции, записанное в специальной форме и иногда прямо в том месте, где она используется.

Сначала формальный синтаксис.

```
[ список захвата ] ( параметры ) mutable -> возвр.тип  noexcept { тело функции }
```

В этом выражении многие составные части не всегда нужны и могут отсутствовать. Основное — начальные квадратные скобки, по которым лямбда функции и опознается компилятором.

Предварительно:

список захвата — какие переменные из окружаещего контекста могут использоваться внутри функции;

параметры — это понятно, параметры функции;

mutable — можно ли изменять захваченные переменные (их копии) внутри функции;

тип возвращаемого значения — понятно;

noexcept — функция не может вызывать исключения;

тело функции — тоже понятно.

Итак, несколько простых примеров.

```
vector<int> a = {1, 2, 3, 4, 5, 6, 7, 8, 9};

for_each(a.begin(), a.end(), [](int k) { cout << k << " "; } );
cout << endl;

sort( a.begin(), a.end(),
    [](const int &k1, const int &k2) -> bool { return k1 > k2; }
);

for_each(a.begin(), a.end(), [](int &k) { k+=10; } );

for_each(a.begin(), a.end(), [](int k) { cout << k << " : "; } );
cout << endl;
```

тип возвращяемого значения определяется по контексту или явно указывается.

Захват переменной из текущей области видимости — возможность ее использовать внутри функции.

Есть много особенностей, вот несколько возможных вариантов и примеров

```
vector<int> a = {1, 2, 3, 4, 5, 6, 7, 8, 9};
int x = 1, y = 5, z = 10;

// []   без захвата переменных
// [=]  все переменные захватываются по значению
for_each(a.begin(), a.end(), [=](int &k) { k += x + y + z; } );
for_each(a.begin(), a.end(), [=](int &k) mutable { k += ++x + y + z; } );
```

```
    for_each(a.begin(), a.end(), [](int k) { cout << k << " "; } ); cout << endl;
    cout << x << endl;

    // [&]  все переменные захватываются по ссылке
    for_each(a.begin(), a.end(), [&](int &k) { k += x + y + z; } );
    for_each(a.begin(), a.end(), [&](int &k) { k += ++x + y + z; } );
    for_each(a.begin(), a.end(), [](int k) { cout << k << " "; } ); cout << endl;
    cout << x << endl;

    // [x, y]  захват x и y по значению
    // [&x]    захват только x по ссылке
    // [x, &z] захват x по значению, a z по ссылке
    // [=, &x, &y]  захват всех по значению, кроме x, y, которые по ссылке
    // [&, z]  захват всех по ссылке, кроме z по значению

    auto f = [](int k) { cout << k << endl; };
    for_each(a.begin(), a.end(), f);
```

Функциональные объекты

На данный момент мы встретились с тремя типами функциональных объектов в C++. Это обычные функции (доступные через указатель на функцию), функторы, используемые через оператор (), и лямбда функции, которые в основном определяются непосредственно в месте их использования. Однако все эти объекты имеют одинаковый смысл — функции, которую можно вызвать с некоторым набором параметров и получить от нее возвращаемое значение.

Язык C++ вводит для этих обектов единый тип std:: fyunction (по сути, некоторую универсальную их обертку), что позволяет любой из них использовать в одном стиле.

Вот пример:

```
#include <functional>
#include <iostream>

void Print(int i)  // просто функция
{
    std::cout << i << std::endl;
}
struct Funct   // функтор
{
    void operator()(int i) const { std::cout << i << std::endl; }
};
struct Num    // класс с функцией-членом
{
    int n;
    Num(int nn) { n = nn; }
    void PrintSum(int i) const { std::cout << n + i << std::endl; }
};
void PrintAny(int i, std::function<void(int)> f) // как параметр
{
    f(i);
}

int main()
{
```

```
    // просто функция
    std::function<void(int)> fun_print = Print;
    fun_print(123);

    // функтор
    std::function<void(int)> funct_print = Funct();
    funct_print(456);

    // лямбда функция
    std::function<void(int)> lam_print = [](int i) { std::cout << i << std::endl; };
    lam_print(789);

    // функция член-класса  (при вызове должен существовать (быть создан) класс)
    std::function<void(const Num&, int)> num_print0 = &Num::PrintSum;
    const Num numnum(1);
    num_print0(numnum, 110);
    num_print0(1, 110);

    // функция член-класса, привязанная к конкретному экземпляру класса
    using std::placeholders::_1;  // см. инструкцию к std::bind
    std::function<void(int)> num_print1 = std::bind(&Num::PrintSum, numnum, _1);
    num_print1(221);
    std::function<void(int)> num_print2 = std::bind(&Num::PrintSum, &numnum, _1);
    num_print2(332);

    std::cout << std::endl;

    // передача в качестве параметра
    PrintAny(100, fun_print);
    PrintAny(200, funct_print);
    PrintAny(300, lam_print);
    PrintAny(400, num_print1);
    PrintAny(500, num_print2);

    // рекурсия в лямбда функции :)))
    // тут надо объявить явно так как auto работать не будет
    std::function<int(int)> fac = [&](int n) {return (n<2) ? 1 : n*fac(n-1); };
    std::cout << "10! = " << fac(10) << std::endl;
}
```

И еще один пример с иллюстрацией создания функционального объекта.

```
#include <functional>
#include <iostream>

struct Funct2   // функтор с нетривиальным внутренним состоянием
{
    int z;   // внутренняя переменная (состояние)
    Funct2(int zz) { z = zz; }
    // несколько разных операторов (), зависящих от состояния
    void operator()(int i) const { std::cout << i*z << std::endl; }
    void operator()(int i, int j) const { std::cout << (i+j)*z << std::endl; }
};

int main()
```

```
{
    // функтор - инициализация с разными конструкторами
    std::function<void(int)> f_print1 = Funct2(10);
    std::function<void(int,int)> f_print2 = Funct2(100);
    auto f_print0 = Funct2(1000);

    // вызов по сигнатуре - OK
    f_print1(5);
    f_print2(5, 6);

    // ошибка компиляции - противоречие с сигнатурой:
    // f_print2(5);
    // f_print1(5, 6);

    // а здесь все OK
    f_print0(5);
    f_print0(5, 6);

    return 0;
}
```