Тема 8. Функциональные объекты

C++ предоставляет некоторые средства для работы с функциями, как с отдельными сущностями наряду с переменными. Это совсем не означает, что C++ поддерживает парадигму функцинального программирования, наоборот, это совсем не так. Но введение отдельных функциональных объектов позволяет "отделить и обособить" вопросы вычисления значений для заданного набора входных аргументов от вопросов хранения данных и преобразования состояний объектов.

Простейшим представителем функционального объекта является указатель на функцию в стиле C. Также простейший пример использование — задание критерия сравнения элементов массива при выполнении сортировки (скажем, в функции qsort). Указатель на функцию может ссылаться на любую функцию, которая совпадает по своей сигнатуре с типом объявления указателя.

Но далее язык C++ расширяет и обобщает эту идею.

Функторы

Функтор — это класс, в котором определены один или несколько операторов (). Формальный пример может выглядеть, например, так

```
class  F1 {
public:
   int operator()(int x) { return x*x; }
};

class  F2 {
   int k;
public:
   F2(int _k) : k(_k) {}
   int operator()(int x) { return x + k; }
   int operator()(int x, int y) { return x + k*y; }
};
    .....
    F1 f1;
    F2 f2(5);
    int a = 1, b = 5, c = 10;
    a = f1(b);
    b = f2(a,c);
    .....
    int somefun(int a, F1 &f, F2 &g) {
       return f(a) + g(a,a);
    }
    .....
    c = somefun (3, f1, f2);
```

В первом случае функтор реализует "чистую" функцию, во втором случае результат зависит от текущего состояния данного экземпляна класса (что как раз и противоречит принципам функционального программирования).

Операторов () может быть определено в классе скольку угодно, главное, чтобы они различались по сигнатурам (как и любые другие методы с одинаковыми именами). Естественно, в классе могут быть и другие методы.

Лямбда функции

Это понятие позволяет быстро вставить в код некоторые операции, необходимые, например, для обработки наборов данных. Часто бывает, что нам нужна достаточно простая обособленная функция, например, сравнения проверки каких-то условий, для сравнения элементов и т.п.

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 (можно не писать, если естественно и очевидно);

noexcept — если присутствует, то функция не может вызывать исключения ;

тело функции — тоже понятно.

Несколько совсем простых примеров.

```
vector<int> a = {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto f = [](int k) { cout << k << " "; }
auto g = [](int k)->bool { return k%3; }

for (auto i=a.begin(); i!=a.end(); ++i) { *i = g(*i); }
cout << endl;

for (int j : a) { f(j); }
```

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

Забегая немного вперед, в примерах используется библиотечная функция `for_each`, которая применяет указанную функцию к последовательности элементо между двумя итераторами.

```
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);
```

Общий интерфейс функциональных объектов

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.

Язык C++ вводит для этих объектов универсальную их обертку std:: function, 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; }
};
// функция с функциональным параметром
void PrintAny(int i, std::function<void(int)> f)
{
    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(5, 110);       // другой временный класс Num(5)

    // 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;
}
```