

Технология ООП

Санкт-Петербургский государственный политехнический университет

11 октября 2011

Итераторы — это обобщенные указатели, которые позволяют программе C++ работать с различными структурами данных единообразно. Обеспечивают получение значений элементов и перемещение по последовательности. Чтобы использовать итераторы в C++ необходимо подключить заголовочный файл `<iterator>`.

Категории итераторов

Тип	Требования к операциям	Контейнеры
Input/Output	R/W,++	istream_iterator, ostream_iterator
Forward	R/W,++	vector, list
Bidirectional	R/W,++,—	list, set, map
Random access	R/W,++,—,rnd	vector, array, deque
Contiguous	R/W,++,—,rnd	array, vector, string

Цель — представить входные и выходные потоки как последовательности, чтобы можно было использовать потоки ввода/вывода в обобщенных алгоритмах, также как списки, вектора и т.п. Классы реализующие потоковые итераторы: `istream_iterator`, `ostream_iterator`.

Специфика:

- `operator++()` — реализован как заглушка;
- `operator=` выводит в поток;
- конструктор с одним или двумя параметрами — принимают объект потока вывода и разделитель.

Пример использования ostream_iterator

```
ostream_iterator<int> os(cout);  
*os = 1;  
//++os;  
*os = 2;
```

Итератор ввода `istream_iterator`

Специфика:

- `operator++()` — осуществляет ввод;
- default конструктор формирует признак конца ввода;
- конструктор с одним параметром принимает объект потокового ввода;
- разделители: пробел, табуляция, перевод строки игнорируются.

Пример использования `istream_iterator`

Защита от некорректного ввода:

```
vector<int> vec;  
copy(istream_iterator<int>(cin), istream_iterator<  
    int>(),  
      back_inserter(vec));  
  
cin.clear();  
while (cin.rdbuf()->in_avail() > 0) {  
    cin.ignore(); //cin.get()  
}
```

Адаптеры для итераторов

Существуют следующие варианты адаптеров:

- реверсивные итераторы;
- итераторы вставки расширяющие контейнер:
 - `back_insert_iterator` (использует `push_back`);
 - `front_insert_iterator` (использует `push_front`);
 - `insert_iterator` (использует `insert`).

Для формирования итераторов вставки STL предоставляются шаблоны глобальных функций:

- `back_inserter(cont& c);`
- `front_inserter(cont& c);`
- `inserter(cont& c, iterator it);`

Пример итератора вставки

```
vector<int> v; //пустой!  
back_insert_iterator<vector<int>> it =  
    back_inserter(v);  
*it = 1; //???
```



```
front_insert_iterator<vector<int>> it =  
    front_inserter(v); //???
```

Особенности использования вывода типов:

- тип переменной, объявленной как `auto`, определяется компилятором самостоятельно на основе того, чем эта переменная инициализируется;
- `auto` переменная не может быть создана без инициализатора;
- `auto` переменная не может хранить значения разных типов, следовательно после инициализации сменить тип переменной будет уже нельзя из-за статической типизации.

Ключевое слово `auto` было добавлено в C++11, старое ключевое слово из языка Си было удалено, для избежания путаницы.

Пример использования auto

```
auto n = 1; //так делать не стоит, т.к. тип не  
однозначен  
int main()  
{  
    map<const char*, int> months;  
    months["Январь"] = 31;  
    auto it = months.begin(); //auto => map<const  
        char*, int>::iterator  
    for(; it != months.end(); ++it)  
    {  
        cout << (*it).first << ": " << (*it).  
            second << endl;  
    }  
}
```

λ-выражениями(функциями) — называются безымянные локальные функции, которые можно создавать прямо внутри какого-либо выражения. В C++ — это краткая формы записи анонимных функциональных объектов. Синтаксис задания выражения:

```
[ список внешних переменных ] ( список параметров )  
    → <тип_возвращаемого_значения>  
{  
    <тело>  
}
```

Особенности λ -выражений

- выражения в C++ представляют собой анонимный класс (структуру) в котором перегружен константный `operator()`;
- всегда начинаются с `[]` (скобки могут быть непустыми);
- далее следует необязательный список параметров;
- параметры можно передавать по адресу и по значению;
- затем описывается непосредственно тело функции в `{ }`
- по умолчанию — функция возвращает `void`, либо программист может указать возвращаемый тип явно;
- тип возвращаемого λ -выражением значения может быть выведен на основе инструкции `return`.

Примеры лямбда-выражений

Использование λ -выражений вместо функциональных объектов:

```
class PrintCube {
    public:
        void operator()(int x) const { cout << (x*x*x)
            << '\n'; }
};

int main()
{
    int ar[] = {5, -1, 4, -7, 3};
    for_each(ar, ar + sizeof(ar)/sizeof(int),
        PrintCube() );

    //Использование лямбда-выражений
    for_each(ar, ar + sizeof(ar)/sizeof(int), [](
        int x){cout<< (x*x*x)<< '\n';} );
}
```

В данном примере метод ничего не возвращает и принимает по значению очередной элемент последовательности.

Примеры лямбда-выражений с выводом типа

Задание: скопировать только отрицательные значения из `ar`.
Для неявного формирования типа возвращаемого значения в лямбда-функции должна быть только одна инструкция `return`, либо несколько инструкций с одинаковыми типами.

```
vector<int> v;  
copy_if(ar, ar + sizeof(ar)/sizeof(int), ???, [](  
    int x){ return x<0; }  
);
```

```
copy_if(ar, ar + sizeof(ar)/sizeof(int),  
    back_inserter(v), [](int x) -> bool { return x  
    < 0; }  
);
```

```
copy_if(ar, ar + sizeof(ar)/sizeof(int),  
    back_inserter(v),  
    [](int x) -> bool { if(x == 0) return true; else  
        return 0.0; } //???)  
);
```

Копирование через функциональный объект

Задание: требуется скопировать только те значения, которые попадают в указанный диапазон.

```
class Range { //Функциональный объект  
    int lower, upper;  
    public:  
        Range(int l, int u):lower(l), upper(u) {}  
        bool operator ()(int x) const { return (x >  
            lower) && (x < upper); }  
};
```

```
copy_if(ar, ar + sizeof(ar)/sizeof(int),  
        back_inserter (v), Range(lower, upper) );
```

```
int lower = 0, upper = 10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter (v),
        [lower, upper] (int _x) -> bool {
            return _x > lower && _x < upper;
        }
);
```

Модификация параметров

```
int ar[] = {5, -1, 4, -7, 3};  
for_each(ar, ar + sizeof(ar)/sizeof(int),  
        [](int& x){ x++; }  
);  
  
for_each(ar, ar + sizeof(ar)/sizeof(int),  
        [](int x){ x++; }  
); //???
```

Модификация переменных объекта

Модификация переменных анонимного функционального объекта внутри λ -функции. Задание: при каждом копировании увеличиваем верхнюю границу диапазона.

```
int ar[] = {5, -1, 4, -7, 3, 11};
int lower = 0, upper = 10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter(v),
        [lower, upper](int _x) -> bool {
            if(_x > lower && _x < upper) {
                upper++; //ошибка!
                return true;
            }
            else return false;
        }
);
```

Модификация переменных анонимного функционального объекта

```
int ar[] = {5, -1, 4, -7, 3, 11};
int lower=0, upper=10;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter(v),
        [lower, upper](int _x) mutable -> bool {
            if(_x > lower && _x < upper) {
                upper++;
                return true;
            }
            else return false;
        }
);
```

Формирование адресов внешних переменных

- $[x, y]$ — захват x и y по значению;
- $[\&x, \&y]$ — захват x и y по ссылке;
- $[x, \&y]$ — захват x по значению, а y — по ссылке;
- $[=]$ — все из внешнего контекста по значению;
- $[\&]$ — все из внешнего контекста все по ссылке;
- $[=, \&x]$ — захват x по ссылке, все остальные по значению;
- $[this]$ — захват всех переменных объекта.

Формирование в анонимном объекте адресов внешних переменных

Задание: Посчитать количество скопированных элементов.

```
int ar[] = {5, -1, 4, -7, 3, 11};
int lower = 0, upper = 10, count = 0;
vector<int> v;
copy_if(ar, ar + sizeof(ar)/sizeof(int),
        back_inserter(v),
        [lower, upper, &count](int _x) mutable -> bool
        {
            if(_x > lower && _x < upper) {
                count++; return true;
            }
            else return false;
        }
);
```

Исключения в лямбда-выражениях

Спецификация исключений в λ -функции:

- не генерирует исключения: `[] (int x) throw() { ... }`
- не генерирует исключения: `[] (int x) noexcept { ... }`
- генерирует `bad_alloc`: `[] (int x) throw(std::bad_alloc) { ... }`