

ООП 2021
Лекция 13

Адаптеры итераторов.

`oopCpp@yandex.ru`

Типы адаптеров итераторов

Адаптеры - шаблонные классы, которые обеспечивают отображения интерфейса. Например, `insert_iterator` обеспечивает контейнер интерфейсом итератора вывода.

Иногда адаптеры итераторов называют адаптерами экземпляра для итераторов.

Для итераторов существуют два основных типа адаптеров:

- итераторы вставки (`insert`)
- обратные (`reverse`) итераторы.

Кроме того, имеются адаптеры потоковых итераторов. Со стандарта C++11 появился адаптер итератора перемещения.

Вставка

Итератор вставки **inserter** является общим в том смысле, что мы указываем позицию, в которой будет происходить вставка элементов. Если вставка в контейнер `L` должна происходить в конце, мы можем написать `L.end()`, как в следующей программе:

Пример

// Копирование вектора с помощью итератора вставки.

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;
int main() {
    int a [4] = {10, 20, 30, 40};
    vector<int> v(a, a+4);
    list<int> L(2, 123);
    copy( v.begin(), v.end(), inserter (L, L.end()));
    list<int>::iterator i;
    for (i=L.begin(); i != L.end(); ++i)
        cout *i << " "; // Вывод: 123 123 10 20 30 40
    cout << endl;
    return 0;
}
```

Еще пример

```
#include <iostream>
#include <iterator>
#include <list>
#include <algorithm>
int main () {
    std::list<int> foo, bar;
    for (int i=1; i<=5; i++)
        { foo.push_back(i); bar.push_back(i*10); }
    std::list<int>::iterator it = foo.begin();
    advance (it,3);
    std::copy ( bar.begin(), bar.end(), std::inserter (foo, it) );
    std::cout << "foo contains:";
    for ( std::list<int>::iterator it = foo.begin(); it!= foo.end(); ++it )
        std::cout << ' ' << *it;
    std::cout << '\n';    return 0; }
```

Output:

1 2 3 10 20 30 40 50 4 5

Поскольку вставка в конце контейнера является часто встречающейся операцией, для нее существует специальный итератор вставки, называемый **back_inserter**.

Приведенная выше программа работает точно так же, если мы заменим вызов алгоритма `copy` на вызов:

```
copy ( v.begin () , v.end() , back_inserter (L) ) ;
```

Так как **back_inserter** всегда добавляет элементы в конец, он требует в качестве единственного аргумента контейнер.

Пример back_inserter

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

int main () {
    std::vector<int> foo, bar;
    for (int i=1; i<=5; i++)
        { foo.push_back(i); bar.push_back(i*10); }
    std::copy (bar.begin(), bar.end(), back_inserter(foo));
    std::cout << "foo contains:";
    for ( std::vector<int>::iterator it = foo.begin(); it!= foo.end(); ++it )
        std::cout << ' ' << *it;
    std::cout << '\n';    return 0;
}
```

Output:

foo contains: 1 2 3 4 5 10 20 30 40 50

Еще один итератор вставки, **front_inserter**, работает специфическим образом: каждый вставляемый элемент помещается в начале контейнера, что приводит к тому, что значения следуют в обратном порядке. Например, если заменим в предыдущем примере `back_inserter` на `front_inserter`

`copy (v. begin () , v.end(), front_inserter (L)) ;`

будут выведены следующие значения:

50 40 30 20 10 1 2 3 4 5

Мы также можем использовать итераторы вставки другими способами.

Например, вместо

`L.push_front(111);`

`L.push_back(999);`

можно написать:

`*front_inserter (L)=111; *back_inserter (L)=999;`

Обратные итераторы

```
vector<int>::iterator vector<int>::reverse_iterator
```

```
vector<int>::const_iterator vector<int>::const_reverse_iterator
```

Обратный итератор может использоваться в следующем фрагменте для вывода всех элементов вектора `v` в обратном порядке:

```
vector<int>::reverse_iterator i;  
for ( i=v.rbegin() ; i != v.rend() ; ++ i)  
    cout << * i << " ";
```

Типы итераторов, имена которых начинаются с `const`, нужны, когда контейнер сам имеет атрибут `const`:

Пример

```
// const_iterator и const_reverse_iterator
#include <iostream>
#include <list>
using namespace std;

void showlist (const list<int> &x) {           // Вперед:
    list<int>::const_iterator i;
    for (i=x.begin(); i != x.end(); ++ i)
        cout << * i << " ";
    cout << endl;                             // Вывод: 10 20 30
                                           // Назад:
    list<int>::const_reverse_iterator j;
    for ( j=x.rbegin(); j != x.rend() ; ++j)
        cout << * j << " "; cout << endl;    // Вывод: 30 20 10
}
```

```
int main() {  
    list<int> L;  
    L.push_back(10); L.push_back(20); L.push_back(30);  
    showlist (L);  
    return 0;  
}
```

Чтобы удалить префикс `const_`, встречающийся два раза в функции **showlist**, мы должны также удалить ключевое слово `const`, имеющееся в первой строке этой функции.

Поскольку функция не модифицирует список, передаваемый ей в качестве параметра, хороший стиль программирования требует наличия **const** в этом примере.

Еще о реверсивных (обратных) итераторах

Реверсивный итератор адаптирует оператор `operator--` других итераторов таким образом, что когда используется оператор `operator++`, то на самом деле вызывается оператор `operator--`. Это позволяет перебирать элементы контейнера или последовательности в обратном порядке, не меняя код, который основывается на применении оператора `operator++`.

Например, если имеется функция, которая выводит в поток элементы некоторого контейнера посредством передачи в функцию диапазона элементов с помощью двух итераторов.

То, передавая реверсивные итераторы, можно, например, выводить элементы массива целых чисел в обратном порядке.

В следующем примере адаптер итераторов `std::reverse_iterator` адаптирует указатели типа `int*` для операций с ними в обратном порядке следования элементов массива. Он использует те методы указателей, которые уже определены и существуют для указателей.

(по материалу из <https://ru.stackoverflow.com/questions/479987/Что-такое-адаптер>)

```

#include <iostream>
#include <iterator>
template <class ForwardIterator>
std::ostream & display ( ForwardIterator first,      ForwardIterator last,
                        std::ostream &os = std::cout ) {
    for ( ; first != last; ++first )    {
        os << *first << ' ';
    }
    return os;
}
int main() {
    const size_t N = 10;
    int a[N] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    display( a, a + N ) << " и ";
    display( std::reverse_iterator<int *>( a + N ),
            std::reverse_iterator<int *>( a ) ) << std::endl;
}    // Вывод : 0 1 2 3 4 5 6 7 8 9 и 9 8 7 6 5 4 3 2 1 0

```

Адаптер итератора перемещения

`std::move_iterator`

Этот класс адаптирует итератор так, чтобы разыменовывание производило ссылки на `rvalue` (как если бы было бы применено `std::move`), а остальные операции ведут себя как в обычном итераторе.

Этот адаптер итератора хранит внутреннюю копию итератора (известного как базовый итератор), на которой отражаются все операции.

Копию базового итератора с текущим состоянием можно получить в любое время, вызвав элемент `base()`.

move_iterator

```
#include <iostream>
#include <iterator>
#include <vector>
#include <string>
#include <algorithm>
int main () {
    std::vector<std::string> foo (3);
    std::vector<std::string> bar {"one","two","three"};
    typedef std::vector<std::string>::iterator lter;
    std::copy ( std::move_iterator<lter>(bar.begin()),
                std::move_iterator<lter>(bar.end()),
                foo.begin() );
    bar.clear();
    std::cout << "foo:";
    for (std::string& x : foo) std::cout << ' ' << x;

    std::cout << '\n'; return 0;}

```

Output: foo: one two three

std::make_move_iterator

```
#include <iostream>    // std::cout
#include <iterator>    // std::make_move_iterator
#include <vector>      // std::vector
#include <string>      // std::string
#include <algorithm>   // std::copy
int main () {
    std::vector<std::string> foo (3);
    std::vector<std::string> bar {"one","two","three"};
    std::copy ( make_move_iterator (bar.begin()),
                make_move_iterator (bar.end()),
                foo.begin() );
    bar.clear(); // бар теперь содержит неопределённые значения;
    std::cout << "foo:";
    for (std::string& x : foo) std::cout << ' ' << x;
    std::cout << '\n';
    return 0;
}
```

Output: foo: one two three

Адаптеры потоковых итераторов

Эти адаптеры делают удобным работу с потоками ввода и вывода, а также буферами потоков, представляя их в виде итераторов ввода и вывода: `std::istream_iterator`, `std::ostream_iterator`, `std::istreambuf_iterator` и `std::ostreambuf_iterator`.

В следующем примере показано, как тип `std::istream_iterator` можно использовать для чтения набора значений из стандартного потока ввода `std::cin`, а тип `std::ostream_iterator` - для записи этого набора после сортировки в строковый поток.

Пример

```
#include<vector>
#include<iostream>
#include<sstream>
#include<iterator>
#include<algorithm>
#include<string>
int _tmain(int argc, _TCHAR* argv[ ]) {
    std::vector<int> values;
    std::stringstream sstm;
    std::string s;
    std::copy(std::istream_iterator<int>(std::cin) // Прочитать значения
, std::istream_iterator<int>(), std::back_inserter(values));
    std::sort(values.begin(), values.end()); // Отсортировать значения
    std::copy(values.begin(), values.end() , std::ostream_iterator<int>(sstm, " ")); // Вывод
    s= sstm.str(); // передать из потока в строку
    std::cout<<s; // вывести строку
    return 0;
}
```

ostream_iterator и istream_iterator

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

```
int main () {
    std::vector<int> myvector;
    for (int i=1; i<10; ++i) myvector.push_back (i*10);

    std::ostream_iterator<int> out_it (std::cout, ", ");
    std::copy ( myvector.begin(), myvector.end(), out_it );
    return 0;
}
```

Output: 10, 20, 30, 40, 50, 60, 70, 80, 90,

Пример `istream_iterator`

```
#include <iostream>    // std::cin, std::cout
#include <iterator>    // std::istream_iterator
int main () {
    double value1, value2;
    std::cout << "Please, insert two values: ";
    std::istream_iterator <double> eos;           // концевой итератор
    std::istream_iterator <double> iit (std::cin); // stdin итератор

    if (iit != eos) value1=*iit;
        ++iit;
    if (iit !=eos) value2=*iit;

    std::cout << value1 << "*" << value2 << "=" << (value1*value2) << '\n';
    return 0;
}
```

Possible **output**:

```
Please, insert two values: 2 32
2*32=64
```

Чтение из файла

```
#include<vector>
#include<iostream>
#include<iterator>
#include <fstream>
using namespace std;
    typedef istream_iterator<int> istream_iter;
int main() {
    vector<int> a;
    ifstream file("example.txt"); // содержимое файла: 20 30 40 50
if (file.fail()) {
    cout << "Cannot open file example.txt.\n";
return 1;
}
    copy(istream_iter(file), istream_iter(),
        inserter(a, a.begin() )); // копируем из файла прямо в вектор
    copy(a.begin(), a.end(),
        ostream_iterator<int>(cout, " ")); // копируем из вектора прямо в cout
    cout << endl; return 0;
}
```

std::istreambuf_iterator

Класс **istreambuf_iterator** очень похож на `istream_iterator`. Однако вместо форматированного ввода произвольных типов он читает одиночные символы из потока ввода.

Это адаптер итератора ввода, и, как и `istream_iterator`, `istreambuf_iterator` достигает специального значения "за последним элементом", когда доходит до конца потока.

Например.

В буфер читаются все оставшиеся символы из стандартного потока ввода.

```
int main() {  
    istreambuf_iterator<char> first(cin);  
    istreambuf_iterator<char> end_of_stream;  
    vector<char> buffer (first, end_of_stream);  
}
```

std:: ostreambuf_iterator

Класс **ostreambuf_iterator** адаптер итератора вывода, который записывает СИМВОЛЫ В ПОТОК ВЫВОДА.

В отличие от **ostream_iterator**, вместо обобщенного форматированного вывода произвольных типов он записывает одиночные символы с помощью функции члена `sputc` класса `streambuf`.

Например:

```
int main() {  
    string s = "This is a test\n";  
    copy(s.begin(), s.end(), ostreambuf_iterator <char> (cout) );  
}
```

Пример

```
#include <iostream>    // std::cin, std::cout
#include <iterator>    // std::istreambuf_iterator
#include <string>      // std::string
int main () {
    std::istreambuf_iterator<char> eos;                // end-of-range iterator
    std::istreambuf_iterator<char> iit (std::cin.rdbuf()); // stdin iterator
    std::string mystring;
    std::cout << "Please, enter your name: ";
    while (iit != eos && *iit != '\n') mystring += *iit++;

    std::cout << "Your name is " << mystring << ".\n";
    return 0;
}
```

Possible output:

Please, enter your name: Andrey

Your name is Andrey

Пример

```
#include <iostream>    // std::cin, std::cout
#include <iterator>    // std::ostreambuf_iterator
#include <string>      // std::string
#include <algorithm>   // std::copy

int main () {
    std::string mystring ("Some text here...\n");

    std::ostreambuf_iterator<char> out_it (std::cout);    // stdout iterator

    std::copy ( mystring.begin(), mystring.end(), out_it);

    return 0;
}
```

Пример из книги С. Мейерса

Совет 29. Рассмотрите возможность использования **istreambuf_iterator** при посимвольном вводе (“Эффективное использование STL”).

Предположим, вы хотите скопировать текстовый файл в объект `string`. На первый взгляд следующее решение выглядит вполне разумно:

```
ifstream inputFile ( "interestingData.txt");  
    // Прочитать inputFile в fileData  
string fileData ( istream_iterator<char>(inputFile) , istream_iterator<char>() );
```

Но вскоре выясняется, что приведенный синтаксис не копирует в строку пропуски (`whitespace`), входящие в файл.

Это объясняется тем, что **istream_iterator** производит непосредственное чтение функциями `operator`, а эти функции по умолчанию не читают пропуски.

Чтобы сохранить пропуски, входящие в файл, достаточно включить режим чтения пропусков сбросом флага **skipws** для входного потока:

```
ifstream inputfile ("interestingData.txt");
```

```
inputFile.unset(ios::skipws); // Включить режим чтения пропусков в inputFile
```

```
string fileData( istream_iterator<char>(inputFile) , istream_iterator<char>() );
```

Теперь все символы inputFile копируются в fileData.

Кроме того, может выясниться, что копирование происходит не так быстро, как хотелось бы.

Функции `operator<<`, от которых зависит работа `istream_iterator`, производят форматный ввод, а это означает, что каждый вызов сопровождается многочисленными служебными операциями.

Более эффективное решение основано на использовании итератора **istreambuf_iterator**. Итераторы **istreambuf_iterator** работают аналогично `istream_iterator`, но если объекты `istream_iterator<char>` читают отдельные символы из входного потока оператором `<>`, то объекты **istreambuf_iterator** обращаются прямо к **буферу потока** и непосредственно читают следующий символ (выражаясь точнее, объект `istreambuf_iterator<char>` читает следующий символ из входного потока `S` вызовом

```
s.rdbuf ( )->sgetc( );
```

```
ifstream inputfile ( "interestingData.txt");  
string fileData ( istreambuf_iterator<char>(inputfile ),  
                  istreambuf_iterator<char>() );
```

На этот раз сбрасывать флаг `skipws` не нужно, итераторы **istreambuf_iterator** никогда не пропускают символы при вводе и просто возвращают следующий символ из буфера.

Мои опыты

// рекомендую со следующей программой поработать самостоятельно

// в ней разные варианты - изучите

```
#include <windows.h>
```

```
#include <iostream>
```

```
#include <vector>
```

```
#include <fstream>
```

```
#include <iterator>
```

```
#include <string>
```

```
#include <list>
```

```
#include <memory>
```

```
using namespace std;
```

```
template <class Function>
```

```
__int64 time_call(Function&& f) // для подсчета времени
```

```
{
```

```
    __int64 begin = GetTickCount();
```

```
    f();
```

```
    return GetTickCount() - begin;
```

```
}
```

```
template< typename Fwd>
void printData (Fwd first, Fwd last, char* delim =",", ostream& out = cout ){
    while (first != last) {
        out << *first;
        if (++first != last)
            out << delim << ' ';
    }
    out<<endl;
}

template<typename C>
void printAllData (const C& c, char* delim=",", ostream& out = cout){
    printData( c.begin( ), c.end( ), delim, out);
}
```

```

int _tmain(int argc, _TCHAR* argv[ ]){
    int i;
    setlocale(LC_ALL, "ru");
    cout<<"введите вариант алгоритма"<< endl;
    cin>>i;
    const int COUNT =10000;
if(i==0){
    ifstream inputfile ;
    __int64 begin = GetTickCount();
        string fileData;
    for(int i=0;i< COUNT; i++){
        inputfile.open ( "data.txt");
            fileData = string( istream_iterator<char>( inputfile) ,
                istream_iterator<char>( ) );

        inputfile.close();
    }
    wcout<< L"serial: " << GetTickCount() - begin<< endl;;
        cout<< fileData;
    } // 0

```

```

else if(i==1){
    string fileData ;
    __int64 begin = GetTickCount(); // сравнение времени выполнения
    for(int i=0;i< COUNT; i++){
        fileData = string( istreambuf_iterator<char>( ifstream ( "data.txt" ) ,
            istreambuf_iterator<char>() );
    }
        wcout<< L"serial: " << GetTickCount() - begin<< endl;
        cout<< fileData;
} // 1
else if(i==2){
    string fileData ;
    __int64 begin = GetTickCount(); // использование умного указателя (след. сем.)
    for( int i=0;i< COUNT; i++){
        fileData = string( ( istreambuf_iterator<char>( *( unique_ptr <ifstream>( new
            ifstream("data.txt") )) ) ),
            istreambuf_iterator<char>() ); // ВЫВОД
    }
        wcout<< L"serial: " << GetTickCount() - begin<< endl;
        cout<< fileData;
} // 2

```



```

else if(i==3) {
    string fileData ;
    ifstream dataFile ( "data.txt"); // многострочное чтение файла
    typedef int    T ;
    istream_iterator<T> dataBegin (dataFile);
    istream_iterator<T> dataEnd;
    list<T> data(dataBegin, dataEnd);
        printAllData ( data," ");
} // 3
else if(i==4) {
    // одной строкой читаем собственный файл в строку и выводим на консоль
    cout<< string( ( istreambuf_iterator<char>( *( unique_ptr <ifstream>
                ( new ifstream(__FILE__ ) ) ) ), // __FILE__
                istreambuf_iterator<char>( ) );
}

```

```

else if(i==5) {
    // одной строкой читаем файл и сразу кладем токены в список
    list< string> data( (istream_iterator< string> ( ifstream ( "data.txt" ) )),
                      istream_iterator< string>() );
    printAllData ( data,"  "); // и печатаем из списка
}
else {
    cout<< string ( ( istreambuf_iterator<char>( ifstream ( __FILE__ ) ) ), // __FILE__
                  istreambuf_iterator<char>() ); // читаем файл с помощью istreambuf_iterator
    cout<< string("");
}
return 0;
}

```

Домашнее задание на неделю

Проект 36.

В потоке прикреплен, кроме лекции, **файл** печати на плоттер HP, соответственно формата HPGL.

Необходимо прочитать графические объекты из этого файла и занести их в базу данных. При этом содержащие не более 2 точек объекты создать одного типа, а с большим количеством точек – другого типа. Наследовать эти классы от некоего базового для обоих класса.

Далее, объекты с превышающим 2 числом точек переместить в область отрицательных по X координат.

Распечатать в консольном окне информацию о всех объектах: тип и координаты.