Содержание

Введение

Представим себе данные как некую абстрактную последовательность. Вне зависимости от способа ее организации и типа данных нам требуются средства поэлементного просмотра последовательности и доступа к каждому ее элементу. Итератор обеспечивает доступ к данным контейнера.

Итератор — объект, позволяющий программисту перебирать все элементы коллекции без учёта особенностей её реализации. Предназначен итератор исключительно для последовательного доступа к элементам

Итератор — это обобщение понятия указателя для работы с различными структурами данных стандартным способом. Для того чтобы можно было реализовать алгоритмы, корректно и эффективно работающие с данными различной структуры, стандарт определяет не только интерфейс, но и требования ко времени доступа с помощью итераторов.

Поскольку итератор является обобщением понятия «указатель», семантика у них одинаковая, и все функции, принимающие в качестве параметра итераторы, могут также работать и с обычными указателями.

В стандартной библиотеке итераторы используются для работы с контейнерными классами, потоками и буферами потоков.

В итераторах используются понятия «текущий указываемый элемент» и «указать на следующий элемент». Доступ к текущему элементу последовательности выполняется аналогично указателям с помощью операций *(разыменование, разадрессация или снятие косвенности) и ->. Переход к следующему элементу - с помощью операции инкремента ++. Для всех итераторов определены также присваивание, проверка на равенство и неравенство.

Данные могут быть организованы различным образом - например, в виде массива, списка, вектора или дека. Для каждого вида последовательности требуется свой тип итератора, поддерживающего различные наборы операций. В соответствии с набором обеспечиваемых операций итераторы делятся на пять категорий, описанных в табл. 7. Выполнение приведенных в таблице операций обеспечивается за постоянное время.

Итераторы как интеллектуальные указатели

Часто бывает необходимо выполнить какую-то операцию над всеми элементами контейнера (или каким-то диапазоном данных). Например, это может быть операция вывода содержимого контейнера на экран или суммирования всех элементов. В обычных массивах в C++ для этого используется обращение в цикле, ко всем элементам с помощью указателя (или оператора [] (индексации), впрочем, за ним стоит все тот же механизм указателей). Например, в нижеследующем отрывке кода производится проход (итерация) по массиву типа float с выводом каждого элемент;

```
float* ptr = start_address;
for(int j = 0; j < SIZE; j++)
  cout << *ptr++;
```

Мы снимаем косвенность с указателя ptr с помощью оператора * для получения значения элемента, на который он ссылается, затем инкрементируем, используя ++. После этого значением указателя является адрес следующего элемента контейнера.

Недостатки обычных указателей

Несмотря на приведенный выше пример, использовать указатели с более сложными контейнерами, нежели простые массивы, довольно затруднительно. Во-первых, если элементы контейнера хранятся не последовательно в памяти, а фрагментировано, то методы доступа к ним значительно усложняются. Мы не можем в этом случае просто инкрементировать указатель для получения следующего значения. Например, при

движении от элемента к элементу в связном списке не можем по умолчанию предполагать, что следующий элемент является соседом предыдущего. Приходится идти по цепочке ссылок.

К тому же нам может понадобиться хранить адрес некоторого элемента контейнера в переменной-указателе, чтобы в будущем иметь возможность доступа к нему. Что случится со значением указателя, если мы вставим или удалим что-нибудь из середины контейнера? Он не сможет указывать на корректный адрес, если со структурой данных что-то произошло. Было бы, конечно, здорово, если бы не нужно было проверять все наши указатели после каждой вставки и удаления.

Одним из решений проблем такого рода является создание класса «интеллектуальных указателей». Объект такого класса обычно является оболочкой методов, работающих с обычными указателями. Операторы ++ и * перегружаются и поэтому знают, как необходимо работать с элементами контейнера, даже если они расположены не последовательно в памяти или изменяют свое местоположение. Вот как это может выглядеть на практике (приводится только схема работ)

```
class SmartPointer//Умный указатель
{
  private:
  float* p;  //обычный указатель
  public:
  float operator*()
  { }
  float operator++()
  { }
};
void mainO
{
. . .
  SmartPointer sptr = start_address;
  for(int j=0; j<SIZE; j++)
            cout << *sptr++;
}
```

На ком ответственность?

Интересен теперь такой вопрос: должен ли класс интеллектуальных указателей непременно включаться в контейнер, или он может быть отдельным классом? Подход, используемый в STL, заключается как раз в том, чтобы сделать интеллектуальные указатели полностью независимыми и даже дать им собственное имя — *итераторы*. (На самом деле, они представляют собой целое семейство шаблонных классов.) Для создания итераторов необходимо определять их в качестве объектов таких классов.

Итераторы в качестве интерфейса

Кроме того, что итераторы являются «умными указателями» на элементы контейнеров, они играют еще одну немаловажную роль в STL. Они определяют, какие алгоритмы, с какими контейнерами следует использовать. Почему это важно?

Дело в том, что теоретически вы, конечно, можете применить любой алгоритм к любому контейнеру. И, на самом деле, так зачастую можно и нужно делать. Это одно из достоинств STL. Но правил без исключений не бывает. Иногда оказывается, что некоторые алгоритмы ужасно неэффективны при работе с определенными типами контейнеров. Например, алгоритму sort() требуется произвольный доступ к тому контейнеру, который он пытается сортировать. В противном случае приходится проходить все элементы до тех пор, пока не найдется нужный, что, разумеется, займет немало времени, если контейнер солидных размеров. А для эффективной работы

алгоритму reverse() требуется иметь возможность обратной и прямой итерации, то есть прохождения контейнера, как в обратном, так и в прямом порядке.

Итераторы предлагают очень элегантный выход из таких неловких положений. Они позволяют определять соответствие алгоритмов контейнерам. Как уже отмечалось, итераторы можно представлять себе в виде кабеля, соединяющего, например, компьютер и принтер. Один конец «вставляется» в контейнер, другой — в алгоритм. Не все кабели можно воткнуть в любой контейнер, и не все кабели можно воткнуть в любой алгоритм. Если вы попытаетесь использовать слишком мощный для данного контейнера алгоритм, то просто вряд ли сможете найти подходящий итератор для их соединения.

Сколько типов итераторов (кабелей) необходимо для работы? Получается, что всего пять. На рис. 3 показаны эти пять категорий в порядке, соответствующем возрастанию их сложности (на начальном уровне «входные» и «выходные» итераторы одинаково просты).

Если алгоритму требуется только лишь продвинуться на один шаг по контейнеру для осуществления последовательного чтения (но не записи!), он может использовать «входной» итератор для связывания себя с контейнером. В реальной практике входные итераторы используются не с контейнерами, а при чтении из файлов или из потока cin.

Если же алгоритму требуется продвинуться вперед на один шаг по контейнеру для осуществления последовательной записи, он может использовать «выходной» итератор для связывания себя с контейнером. Выходные итераторы используются при записи в файлы или в поток cout.

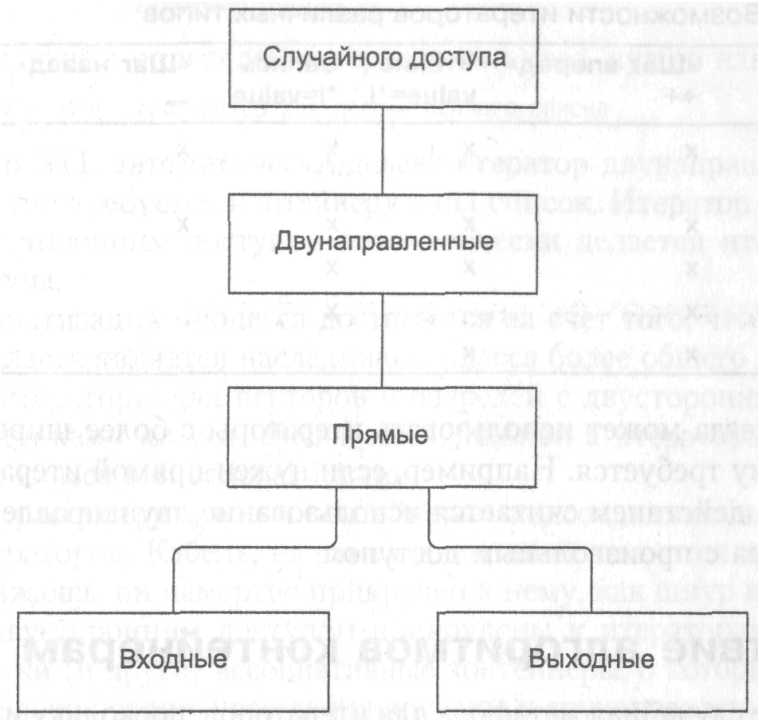

Рис. 3. Категории итераторов

Если алгоритму нужно продвигаться вперед, но при этом совершать как запись, так и чтение, ему придется использовать «прямой» итератор.

В том случае, когда алгоритму требуется продвигаться и вперед, и назад по контейнеру, он использует двунаправленный итератор.

Наконец, если алгоритму нужен незамедлительный доступ к произвольному элементу контейнера безо всяких поэлементных продвижений, используете итератор «произвольного доступа». Он чем-то напоминает массив своей возможностью обращения непосредственно к указанному элементу. Строго определены итераторы, которые могут изменяться с помощью арифметических операций, таких, как

iter2 = iterl + 7:

В табл. 7 показано, какие операции, какими итераторами поддерживаются. Как видите, все итераторы поддерживают оператор ++ для продвижения вперед по контейнеру. Входной итератор (данные получаем из итератора) может использоваться с оператором * справа от знака равенства (но не слева!):

value = *iter;

Выходные итераторы (данные передаём в итератор) могут использоваться с оператором *, но только стоящим слева от знака равенства:

*iter = value;

Прямой итератор поддерживает и запись, и чтение, а двунаправленный итератор может быть как инкрементирован, так и декрементирован. Итератор произвольного доступа поддерживает оператор [] (как и простые арифметические операции + и -) для скоростного обращения к любому элементу.

Таблица 7. Возможности итераторов различных типов

Тип итератора	**«Шаг вперёд ++»**	**Чтение value=*i**	**Запись *i = value**	**«Шаг назад --»**	**Произвольный доступ[n]**
Произвольного доступа	x	x	x	x	x
Двунаправленный	x	x	x	x	
Прямой	x	x	x		
Выходной	x		x		
Входной	x	x			

Алгоритм всегда может использовать итераторы с более широкими возможностями, чем ему требуется. Например, если нужен прямой итератор, совершенно нормальным действием считается использование двунаправленного итератора или итератора с произвольным доступом.

Как видно из таблицы, прямой итератор поддерживает все операции входных и выходных итераторов и может использоваться везде, где требуются входные или выходные итераторы. Двунаправленный итератор поддерживает все операции прямого, а также декремент, и может использоваться везде, где требуется прямой итератор. Итератор произвольного доступа поддерживает все операции двунаправленного, а кроме того, переход к произвольному элементу последовательности и сравнение итераторов.

Можно сказать, что итераторы образуют иерархию, на верхнем уровне которой находятся итераторы произвольного доступа. Чем выше уровень итератора, тем более высокие функциональные требования предъявляются к контейнеру, для которого используется итератор. Например, для списков итераторами произвольного доступа пользоваться нельзя, поскольку список не поддерживает требуемый набор операций итератора.

Итераторные классы и функции описаны в заголовочном файле <iterator>. При использовании стандартных контейнеров этот файл подключается автоматически.

Итераторы могут быть константными. Константные итераторы используются тогда, когда изменять значения соответствующих элементов контейнера нет необходимости.

Итератор может быть действительным (когда он указывает на какой-либо элемент) или недействительным. Итератор может стать недействительным в следующих случаях:

итератор не был инициализирован;

контейнер, с которым он связан, изменил размеры или уничтожен;

итератор указывает на конец последовательности.

Конец последовательности представляется как указатель на элемент, следующий за последним элементом последовательности. Этот указатель всегда существует. Такой подход позволяет не рассматривать пустую последовательность как особый случай. Понятия «нулевой итератор» не существует.

Соответствие алгоритмов контейнерам

Кабель — это не случайная метафора для итераторов, поскольку именно итераторы осуществляют связь между алгоритмами и контейнерами. Сейчас мы сосредоточим внимание на двух концах нашего воображаемого кабеля: на контейнерах и алгоритмах.

Монтаж «кабеля» в контейнерный конец

Если ограничиться рассмотрением только основных контейнеров STL, придется констатировать тот факт, что можно обойтись лишь двумя категориями итераторов. Как показано в табл. 8, векторам и очередям с двусторонним доступом вообще все равно, какой итератор используется, а списки, множества, мультимножества, отображения и мультиотображения воспринимают все, кроме итераторов произвольного доступа.

Таблица 8. Типы итераторов, поддерживаемые контейнерами

Тип итератора	Вектор	Список	Deque	Мно-жество	Мульти-множество	Отобра-жение	Мультиото-бражение
Произвольного доступа	x		x				
Двунаправленный	x	x	x	x	x	x	x
Прямой	x	x	x	x	x	x	x
Входной	x	x	x	x	x	x	x
Выходной	x	x	x	x	x	x	x

Теперь разберемся, как же STL подключает правильный итератор к контейнеру? При определении итератора нужно указывать, для какого типа контейнера его следует использовать. Например, если вы определили список значений типа int:

```
list<int> iList;        //список int
```

тогда для того, чтобы определить итератор для него, нужно написать:

```
list<int>::iterator iter; //итератор для целочисленного списка
```

После этого STL автоматически делает итератор двунаправленным, так как именно такой тип требуется контейнеру типа список. Итератор для вектора очереди с двусторонним доступом автоматически делается итератором произвольного доступа.

Такая автоматизация процесса достигается за счет того, что класс итератора конкретного класса является наследником класса более общего итератора. Так и выходит, что итераторы для векторов и очередей с двусторонним доступом являются наследниками класса random_access_iterator, а итераторы для списков - наследниками класса bidirectional_iterator.

Сейчас мы рассмотрим, как к контейнерам подсоединяются «концы» *наших* «кабелей»-итераторов. Кабель, на самом деле, не вставляется в контейнер, фигурально выражаясь, он намертво прикручен к нему, как шнур к утюгу. Вектор и очереди с двусторонним доступом прикручены к итераторам произвольного доступа, а списки (и

другие ассоциативные контейнеры, о которых мы еще будем говорить несколько позднее) всегда прикручены к двунаправленным итераторам.

Монтаж «кабеля» в алгоритм

Теперь мы посмотрим, что происходит по другую сторону нашей воображаемой проволоки, соединяющей алгоритмы и контейнеры. Итак, что творится со стороны алгоритмов? Разумеется, каждому алгоритму, в зависимости от того, чем он занимается, требуется свой тип итератора. Если нужно обращаться к произвольным элементам контейнера, потребуется итератор произвольного доступа. Если же нужно последовательно передвигаться от элемента к элементу, то подойдет менее мощный прямой итератор. В табл. 9 показаны примеры алгоритмов и требующихся им итераторов.

Таблица 9. Типы итераторов, требующиеся представленным алгоритмам

Алгоритм	**Входной**	**Выходной**	**Прямой**	**Двунаправленный**	**Произвольного доступа**
for_each	x				
find	x				
count	x				
copy	x	x			
replace			x		
unique			x		
reverse				x	
sort					x
nth_element					x
merge	x	x			
accumulate	x				

Опять-таки, несмотря на то, что каждому алгоритму требуется определенный уровень возможностей, более мощные итераторы тоже будут работать нормально. Алгоритму replace() требуется прямой итератор, но он будет работать и с двунаправленным итератором, и с итератором произвольного доступа.

Теперь представим, что из разъемов кабеля торчат ножки, как на силовом кабеле компьютера. Это, кстати, показано на рис. 4. Те кабели, которым требуются итераторы произвольного доступа, будут иметь по 5 ножек, те, которым требуются двунаправленные итераторы, — 4 ножки. Наконец, те, которым требуются прямые итераторы, будут иметь 3 ножки и т. д.

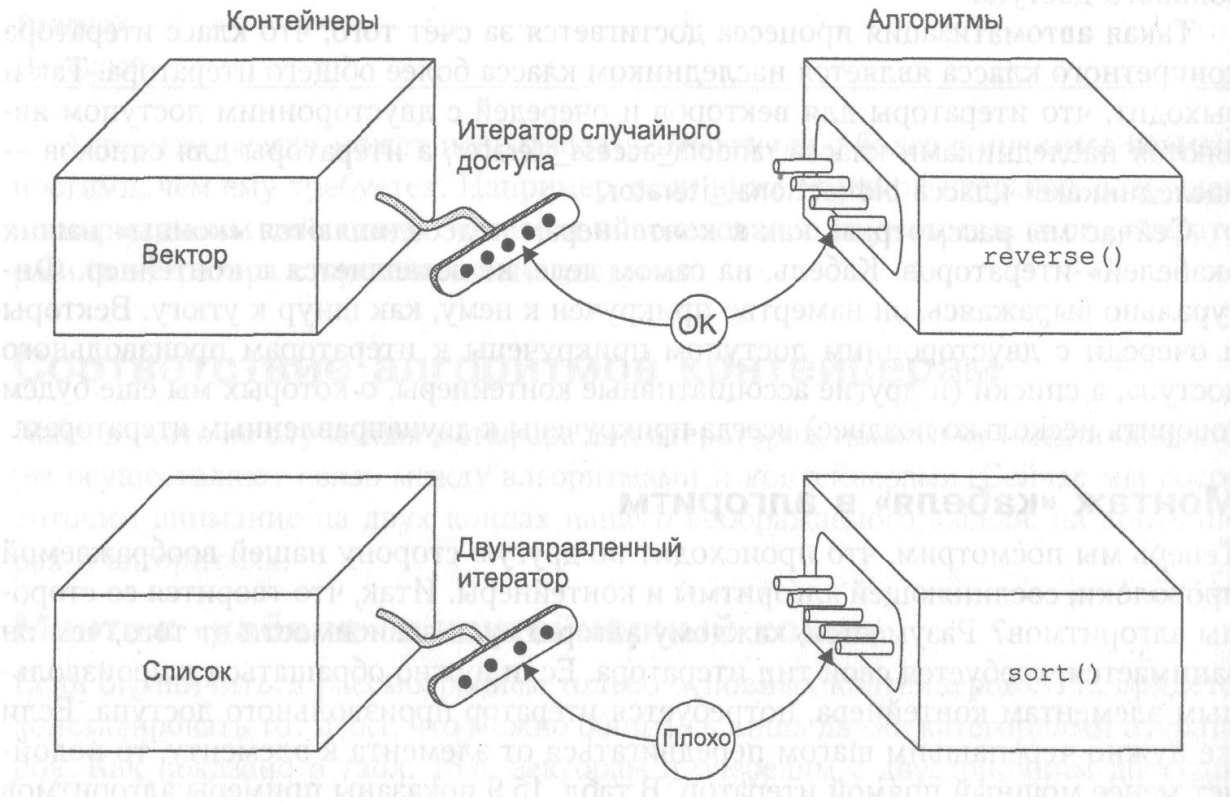

Рис. 4. Итераторы, соединяющие контейнеры и алгоритмы

Соответственно, алгоритмический конец «кабеля» имеет некоторое число отверстий под ножки. 5-пиновый кабель можно вставить и в 5-пиновый алгоритм, и в 4-пиновый и т. д. Наоборот — не получится. То есть, переводя на язык терминов STL, нельзя использовать двунаправленные итераторы с алгоритмами, которым требуется произвольный доступ. Таким образом, векторы и очереди с двусторонним доступом, использующие итераторы произвольного доступа, могут сочетаться законным браком с любым алгоритмом, а списки и другие ассоциативные контейнеры с двунаправленным итератором — только с менее мощными алгоритмами.

О чем рассказывают таблицы

Из приведенных выше табл. 8 и 9 можно узнать, с какими алгоритмами будет работать тот или иной контейнер. Например, из табл. 9 видно, что алгоритму sort() требуется итератор произвольного доступа. Таблица 8 показывает, что единственными контейнерами, которые поддерживают такие итераторы, являются векторы и очереди с двусторонним доступом. Бесполезно даже пытаться применить sort() к спискам, множествам, отображениям и т. д.

Любой алгоритм, которому не требуется итератор произвольного доступа, будет работать с любым типом контейнера STL, поскольку все контейнеры пользуют двунаправленные итераторы, находящиеся лишь на одну ступень ниже уровня произвольного доступа. (Если бы в STL были однонаправленные списки, можно было бы пользоваться прямым итератором, но нельзя было бы применить, например, алгоритм reverse().)

Как видите, относительно малое число алгоритмов реально нуждаются в итераторах произвольного доступа. Поэтому все-таки большинство алгоритмов будет работать с большинством контейнеров.

Перекрытие методов и алгоритмов

Иногда приходится выбирать между методами и алгоритмами с одинаковыми

именами. Например, алгоритму find() требуется только входной итератор, поэтому он может использоваться с любым контейнером. Но у множеств и отображений есть свой собственный метод find() (чего нет у последовательных контейнеров). Какую же версию find() следует использовать? В общем случае, если имеется метод, дублирующий своим названием алгоритм, так это сделано потому, что алгоритм в данном случае оказывается неэффективным. Поэтому, наверное, при наличии такого выбора следует обратиться к методу.

Работа с итераторами

Использовать итераторы гораздо проще, чем рассказывать о них. Некоторые примеры их применения мы уже видели, когда значения итераторов возвращались методами begin() и end() контейнеров. Мы не обращали внимания на этот факт, полагая, что они работают, как указатели. Теперь посмотрим, как реально используются итераторы с этими и другими функциями.

Доступ к данным

В контейнерах, связанных с итераторами произвольного доступа (векторах и очередях с двусторонним доступом), итерация производится очень просто с помощью оператора []. Такие контейнеры, как списки, которые не поддерживают произвольный доступ, требуют особого подхода. В предыдущих примерах мы пользовали «деструктивное чтение» (чтение с разрушением данных) для вывода и одновременного выталкивания элементов итератора. Так было в программах LIST, LISTPLUS. Более практичным действием было бы определение итератора для контейнера. Рассмотрим приведенный ниже пример.

```
//--------------------------------------------------------------
// Листинг 10.cpp: определяет точку входа для консольного приложения.
// итератор и цикл for для вывода данных
//
#include "stdafx.h"
#include <iostream>
#include <list>
#include <algorithm>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
  SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
  int arr[] = { 2, 4, 6, 8 };
  list<int> theList;

  for(int k = 0; k < 4; k++)        //заполнить список элементами
    theList.push_back( arr[k] );    // массива

  list<int>::iterator iter;          //итератор для целочисленного
                                      //списка
  for(iter = theList.begin(); iter != theList.end(); iter++)
    cout << *iter << ' ';           //вывести список
  cout << endl;
```

```
  system("pause");
  return 0;
}
//------------------------------------------------------------
```

Программа просто выводит содержимое контейнера theList:

2 4 6 8

Мы определяем итератор типа list<int>, соответствующий типу контейнера. Как и в случае с переменной-указателем, до начала использования итератора нужно задать его значение. В цикле for происходит его инициализация значением метода theList.begin(), это указатель на начало контейнера. Итератор может быть инкрементирован оператором ++ для прохождения по элементам, а оператором * можно снять с него косвенность для получения значения того элемента, на который он указывает. Можно даже сравнивать его с чем-нибудь с помощью оператора != и выходить из цикла при достижении итератором конца контейнера (theList.end()).

Вот что будет, если использовать цикл while вместо for:

```
iter = theLst.begin();
while (iter != theList.end() ) cout << *iter++ << ' ';
```

Синтаксис *iter++ такой же, как если бы это был указатель, а не итератор.

С помощью итератора можно вывести значения заданного диапазона:

```
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <list>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	//Заносим в список: 2, 4, 6, 8, 10, 12, 14
	list<int> theList = { 2, 4, 6, 8, 10, 12, 14 };
	// Итератор.
	list<int>::iterator iter,iter_beg = theList.begin(), iter_end = theList.begin();
	//Продвигаем итератор на 2 позиции.
	advance(iter_beg, 2);
	//Продвигаем итератор на 5 позиции.
	advance(iter_end, 5);
	//Выводим список с 3 элемента по 6:  6, 8, 10
	for (iter = iter_beg; iter != iter_end; iter++)
		cout << *iter << ", ";
	cout << endl;
	system("pause");
	return 0;
}
```

```
6, 8, 10,
Для продолжения нажмите любую клавишу . . .
```

Вставка данных

Похожий код можно использовать для размещения данных среди существующих элементов контейнера, как показано в следующем примере:

```
//------------------------------------------------------------
```

```
// Листинг 11.cpp: определяет точку входа для консольного приложения.
// Итератор используется для заполнения контейнера данными
//
#include "stdafx.h"
#include <iostream>
#include <list>
#include <algorithm>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
// пустой список для хранения 5 значений типа int
   list<int> iList(5);
                              //
// итератор
   list<int>::iterator it;
   int data = 0;

// заполнение списка данными
   for(it = iList.begin(); it != iList.end(); it++)
      *it = data += 2;

// вывод списка
   for(it = iList.begin(); it != iList.end(); it++)
      cout << *it << ' ';
   cout << endl;
   system("pause");
   return 0;
}

//------------------------------------------------------------
```

Первый цикл заполняет контейнер целыми значениями 2, 4, 6, 8, 10, демонстрируя, что перегружаемая операция * может стоять и слева, и справа от знака равенства. Второй цикл предназначен для вывода этих значений.

Алгоритмы и итераторы

Как уже говорилось, алгоритмы используют итераторы в качестве параметров (иногда еще в виде возвращаемых значений). Пример Листинг 1 показывает применение алгоритма find() к списку (мы знаем, что это возможно, потому данному алгоритму требуется только входной итератор).

```
//------------------------------------------------------------
// Листинг 1.cpp: определяет точку входа для консольного приложения.
// find() возвращает итератор списка
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <list>
```

```
#include "windows.h"
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
// Пустой список для 5 значений int
   list<int> theList(5);
// Итератор для списка
   list<int>::iterator iter;
   int data = 0;
// заполнение списка данными: 2, 4, 6, 8, 10
   for(iter = theList.begin(); iter != theList.end(); iter++)
      *iter = data += 2;      //
// поиск числа 8
   iter = find(theList.begin(), theList.end(), 8);
   if( iter != theList.end() )
      cout << "Да " << *iter;
   else
      cout << "\n Нет. \n";
   cout << endl;
   system("pause");
   return 0;
}
//-------------------------------------------------------------
```

Алгоритм find() имеет три параметра. Первые два — это значения итераторов, определяющие диапазон элементов, по которым может производиться поиск, третий — искомое значение. Контейнер заполняется теми же значениями *2, 4, 6,* 8, 10, что и в предыдущем примере. Затем используется алгоритм find() для нахождения числа 8. Если он возвращает значение theList.end(), мы понимаем, что достигли конца контейнера, а искомый элемент не найден. В противном случае алгоритм возвратит итератор на элемент с искомым значением 8, это означает, что элемент найден. Тогда на экран выводится надпись:

Да 8.

Можно ли с помощью значения итератора выяснить, где именно в контейнере расположено число 8? Видимо, да. Можно найти смещение относительно начала (iter—theList.begin()). Однако это не является легальным использованием итератора для списков, поскольку итератор должен быть двунаправленным, а значит, над ним нельзя выполнять никакие арифметические операции. Так можно делать с итераторами случайного доступа, например используемыми с векторами и очередями. То есть, если вы ищете значение в векторе v, а не в списке theList, перепишите последнюю часть ITERFIND:

```
iter = find(v.begin(), v.end(), 8);
if( iter != v.end() )
  cout << "\nЧисло 8 расположено по смещению " << (iter - v.begin() );
else
  cout << "\nЧисло 8 не найдено.\n":
```

В результате будет выведено сообщение:

Число 8 расположено по смещению 3

Вот еще пример, в котором алгоритм использует итераторы в качестве аргументов. Здесь алгоритм copy() используется с вектором. Пользователь определяет диапазон размещений, которые должны быть скопированы из одного вектора в другой. Программа осуществляет это копирование. Диапазон вычисляется с помощью итераторов.

```
//-------------------------------------------------------------
// Листинг 2.cpp: определяет точку входа для консольного приложения.
// использование итераторов с алгоритмом copy()
//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <vector>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   int beginRange, endRange;
   int arr[] = { 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 };
// Инициализированный вектор.
   vector<int> v1(arr, arr+10);
//Неинициализированный вектор.
   vector<int> v2(10);
   cout << "Введите диапазон копирования (пример: 2 5): ";
   cin >> beginRange >> endRange;

   vector<int>::iterator iter1 = v1.begin() + beginRange;
   vector<int>::iterator iter2 = v1.begin() + endRange;
   vector<int>::iterator iter3;
// Копировать диапазон из v1 в v2.
   iter3 = copy( iter1, iter2, v2.begin() );
// (iter3 -> последний скопированный элемент).
// Итерация по диапазону.
   iter1 = v2.begin();
// Вывести значения из v2.
   while(iter1 != iter3)
      cout << *iter1++ << ' ';
   cout << endl;
   system("pause");
   return 0;
}
//-------------------------------------------------------------
```

Вот один из примеров взаимодействия с программой:

```
Введите диапазон копирования (пример: 2 5): 2 5
15 17 19
```

Мы не выводим целиком вектор, только скопированный диапазон значений. К счастью, copy() возвращает итератор, указывающий на последний элемент (вернее, на элемент, располагающийся следом за последним) из тех, которые были скопированы в

контейнер назначения (v2 в данном случае). Программа использует это значение в цикле while для вывода только нужной части элементов.

Специализированные итераторы

В этом разделе мы рассмотрим два специализированных типа итераторов: адаптеры итераторов, которые могут довольно необычным способом изменять поведение обычных итераторов, и потоковые итераторы, благодаря которым входные и выходные потоки могут вести себя как итераторы.

Адаптеры итераторов

В STL различают три варианта модификации обычного итератора. Это *обратный итератор, итератор вставки* и *итератор неинициализированного хранения.* Обратный итератор позволяет проходить контейнер в обратном направлении. Итератор вставки создан для изменения поведения различных алгоритмов, так как copy() и merge(), чтобы они именно вставляли данные, а не перезаписывали; существующие. Итератор неинициализированного хранения позволяет хранить данные в участке памяти, который еще не инициализирован. Он используется в довольно специфических случаях, поэтому здесь мы рассматривать его применение не будем.

Обратные итераторы

Допустим, вам нужно производить итерацию по контейнеру в обратном порядке от конца к началу. Вы можете попробовать написать такой текст:

```
list<int>::iterator iter;   //обычный итератор
iter = iList.end();    //начать в конце
whilet (iter != iList.begin()     )    // перейти в начало
cout << *iter-- << '     ';   // декрементировать итератор
```

К сожалению, это работать не будет (прежде всего, у вас получится некорректный диапазон: от n до 1, а не от n -1 до 0).

Для осуществления реверсного прохода контейнеров используются обратные итераторы. В следующем примере показано применение этого специализированного итератора для вывода содержимого списка в обратном порядке.

```
//-----------------------------------------------------------------
// Листинг 3.cpp: определяет точку входа для консольного приложения.
// демонстрация обратного итератора
//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <list>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
// Массив типа int.
   int arr[] = { 2, 4, 6, 8, 10 };
//Список целых чисел.
```

```
   list<int> theList;
// Перенести содержимое массива в список
   for(int j = 0; j < 5; j++)
      theList.push_back( arr[j] );
// Обратный итератор.
   list<int>::reverse_iterator revit;
// Реверсная итерация по списку с выводом на экран.
   revit = theList.rbegin();
   while( revit != theList.rend() )
      cout << *revit++ << ' ';
   cout << endl;
   system("pause");
   return 0;
}
//--------------------------------------------------------------
```

Результаты работы программы:

10 8 6 4 2

При использовании обратного итератора следует использовать методы rbegin() и rend() (не стоит пытаться использовать их с обычным итератором). Интересно, что прохождение контейнера начинается с конца, при этом вызывается метод rbegin(). К тому же значение итератора нужно инкрементировать по мере продвижения от элемента к элементу! Не пытайтесь декрементировать его; revit— выдаст совсем не тот результат, который вы ожидаете. При использовании reverse_ iterator вы движетесь от rbegin() к rend(), инкрементируя итератор.

Итераторы вставки

Некоторые алгоритмы, например copy(), очень любят перезаписывать существующие данные в контейнере назначения. Например, так происходит в следующей программе. В ней копируется содержимое одной очереди с двусторонним доступом в другую.

```
//--------------------------------------------------------------
//демонстрация обычного копирования контейнера deque
// Листинг 4.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <deque>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   int arr1[] = { 1, 3, 5, 7, 9 };
   int arr2[] = { 2, 4, 6, 8, 10 };
   deque<int> d1;
   deque<int> d2;
```

```
//Перенос из массивов в очереди.
   for(int j = 0; j < 5; j++)
      {
           d1.push_back( arr1[j] );
           d2.push_back( arr2[j] );
      }
//Копирование из d1 в d2.
   copy( d1.begin(), d1.end(), d2.begin() );
// Вывод d2.
   for(int k = 0; k < d2.size(); k++)
      cout << d2[k] << ' ';
   cout << endl;
   system("pause");
   return 0;
}
//-------------------------------------------------------------
```

Вот что в результате стало с очередью:

1 3 5 7 9

Содержимое d2 было перезаписано содержимым d1, поэтому в d2 мы не находим хранившихся ранее данных. Часто именно это и требуется по логике работы программы, но иногда требуется, чтобы алгоритм copy() не записывал, а вставлял данные в контейнер. Такого поведения алгоритма можно добиться с помощью *итератора вставки.* У него есть три интересных инструмента:

- back_inserter вставляет новые элементы в конец;
- front_inserter вставляет новые элементы в начало;
- inserter вставляет новые элементы в указанное место.

Программа DINSITER показывает, как осуществить вставку в конец контейнера.

```
//-------------------------------------------------------------
// Очереди и итераторы вставки
// Листинг 5.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iostream>
#include <algorithm>
#include <deque>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   int arr1[] = { 1, 3, 5, 7, 9 };  //Инициализация d1.
   int arr2[] = {2, 4, 6};          //Инициализация d2.
   deque<int> d1;
   deque<int> d2;
//Перенести данные из массивов в очереди с двусторонним доступом.
   for(int i = 0; i < 5; i++)
     d1.push_back( arr1[i] );
```

```
   for(int j = 0; j < 3; j++)
      d2.push_back( arr2[j] );
 //Копировать d1 в конец d2.
   copy( d1.begin(), d1.end(), back_inserter(d2) );
//Вывести d2.
   cout << "d2: ";
   for(int k = 0; k < d2.size(); k++)
      cout << d2[k] << ' ';
   cout << endl;
   system("pause");
   return 0;
}
//--------------------------------------------------------------
```

Для вставки новых элементов в конец используется метод push_back(). При этом новые элементы из исходного контейнера d1 вставляются в конец d2, следом за существующими элементами. Контейнер d1 остается без изменений. Программа выводит на экран содержимое d2:

d2: 2 4 6 1 3 5 7 9

Если бы мы указали в программе, что элементы нужно вставлять в начало

```
copy( d1.begin(), d1.end(), front_inserter(d2) );
```

тогда новые элементы оказались бы перед существующими данными d2. Механизм, который стоит за этим, это просто метод push_front(), вставляющий элементы в начало и переворачивающий порядок их следования. В этом случае содержимое d2 было бы таким:

d2: 9 7 5 3 1 2 4 6

Можно вставлять данные и в произвольное место контейнера, используя версию inserter итератора. Например, вставим новые данные в начало d2:

```
copy( d1.begin(), d1.end(), inserter(d2, d2.begin() );
```

Первым параметром inserter является имя контейнера, в который нужно переносить элементы, вторым — итератор, указывающий позицию, начиная с которой должны вставляться данные. Так как inserter использует метод insert(), то порядок следования элементов не изменяется. Результирующий контейнер будет содержать следующие данные:

1 3 5 7 9 2 4 6

Изменяя второй параметр inserter, мы можем указывать произвольное место контейнера.

Обратите внимание на то, что front_inserter не может использоваться с векторами, потому что у последних отсутствует метод push_front(); доступ возможен только к концу такого контейнера.

Потоковые итераторы

Потоковые итераторы позволяют интерпретировать файлы и устройства ввода/вывода (потоки cin и cout), как итераторы. А значит, можно использовать файлы и устройства ввода/вывода в качестве параметров алгоритмов! (Еще одно подтверждение гибкости применения итераторов для связи алгоритмов и контейнеров.)

Основное предназначение входных и выходных итераторов — поддержка классов потоковых итераторов. С их помощью можно осуществлять применение соответствующих алгоритмов напрямую к потокам ввода/вывода.

Потоковые итераторы — это, на самом деле, объекты шаблонных классов разных типов ввода/вывода.

Существует два потоковых итератора: ostream_iterator и istream_iterator. Рассмотрим их по порядку.

Класс ostream_iterator

Объект этого класса может использоваться в качестве параметра любого алгоритма, который имеет дело с выходным итератором. В следующем небольшом примере мы используем его как параметр copy().

```
//------------------------------------------------------------
// Демонстрация ostream_iterator
// Листинг 6.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iterator>
#include <iostream>
#include <algorithm>
#include <list>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   int arr[] = { 10, 20, 30, 40, 50 };
   list<int> theList;

   for(int j = 0; j < 5; j++)                          //перенести
массив в список
      theList.push_back( arr[j] );

   ostream_iterator<int> ositer(cout, ", ");     //итератор ostream

   cout << "theList: ";
   copy(theList.begin(), theList.end(), ositer); //вывод  списка
                                                 //
   cout << endl;
   system("pause");
   return 0;
}
//------------------------------------------------------------
```

Мы определяем итератор ostream для чтения значений типа int. Двумя параметрами конструктора являются поток, в который будут записываться значения, и строка, которая будет выводиться вслед за каждым из них. Значением потока обычно является имя файла или cout; в данном случае это cout. При записи в этот поток может использоваться строка-разделитель, состоящая из любых символов. Здесь мы используем запятую и пробел.

Алгоритм copy() копирует содержимое списка в поток cout. Итератор выходного потока используется в качестве его третьего аргумента и является именем объекта назначения.

На экране в результате работы программы мы увидим:
theList: 10, 20, 30, 40, 50,

В следующем примере показано, как использовать этот подход для записи содержимого контейнера в файл.

```
//-------------------------------------------------------------
// Демонстрация работы ostream_iterator с файлами
// Листинг 7.cpp: определяет точку входа для консольного приложения.
#include "stdafx.h"
#include <iterator>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <list>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   int arr[] = { 11, 21, 31, 41, 51 };
   list<int> theList;

   for(int j = 0; j < 5; j++)            //Передача данных
      theList.push_back( arr[j] );   //из массива в список
   ofstream outfile("ITER.DAT");     //создание файлового объекта

   ostream_iterator<int> ositer(outfile, " "); //итератор
                                       //записать список в файл
   copy(theList.begin(), theList.end(), ositer);
   cout << endl;
   system("pause");
   return 0;
}
//-------------------------------------------------------------
```

Необходимо определить файловый объект класса ofstream и ассоциировать его с конкретным файлом (в программе ассоциируем его с файлом ITER.DAT). При записи в файл желательно использовать удобочитаемые разделители. Здесь мы между элементами вставляем просто символ пробела.

Вывод на экран в этой программе не производится, но с помощью любого текстового редактора можно просмотреть содержимое файла ITER.DAT: **11 21 31 41 51**

Класс istream_iterator

Объект этого класса может использоваться в качестве параметра любого алгоритма, работающего с входным итератором. На примере программы INITER покажем, как такие объекты могут являться сразу двумя аргументами алгоритма copy(). Введенные с клавиатуры (поток cin) числа в формате с плавающей запятой сохраняются в контейнере типа список.

```
//-------------------------------------------------------------------
-------
```

```
// Демонстрация istream_iterator
// Листинг 8.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iterator>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <list>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   list<float> fList(5);               // неинициализированный список

   cout << "\nВведите 5 чисел (тип float): ";
                                       // итераторы istream
   istream_iterator<float> cin_iter(cin);   // cin
   istream_iterator<float> end_of_stream;   //eos (конец потока)
                                       // копировать из cin в fList
   copy( cin_iter, end_of_stream, fList.begin() );

   cout << endl;                       // вывести fList
   ostream_iterator<float> ositer(cout, "--");
   copy(fList.begin(), fList.end(), ositer);
   cout << endl;
   system("pause");
   return 0;
}
//--------------------------------------------------------------------
-------
```

Взаимодействие программы с пользователем может выглядеть, например следующим образом:

```
Введите 5 чисел (типа float): 1.1 2.2 3.3 4.4 5.5
1.1--2.2--3.3--4.4--5.5--
```

Для copy() необходимо указывать и верхний, и нижний предел диапазона значений, поскольку данные, приходящие с cin, являются исходными, а не конечными. Программа начинается с того, что мы соединяем istream_iterator с потоком cin, который определен с помощью конструктора с одним параметром как с iter. Но что у нас с концом диапазона? Используемый по умолчанию конструктор класса stream_iterator без аргументов выполняет здесь особую роль. Он всегда создает объект istream_iterator для указания конца потока.

Как пользователь сообщает об окончании ввода данных? Нажатием комбинации клавиш Ctrl+Z, затем Enter в результате чего в поток передается стандартный символ конца файла. Иногда требуется нажать Ctrl+Z несколько раз. Нажатие Enter приведет к установке признака окончания файла, хотя и поставит ограничитель чисел.

ostream_iterator используется для вывода содержимого списка. Впрочем, для этих

целей можно использовать множество других инструментов.

Любые операции вывода на экран, даже такие, как просто вывод строки «Введите пять чисел в формате float», необходимо выполнять не только до использования итератора istream, но даже до его определения, поскольку с того момента как он определяется в программе, вывод на экран оказывается недоступен: программа ожидает ввода данных.

В следующем примере вместо потока cin используется входной файл, из которого берутся данные для программы. Пример также демонстрирует работу с алгоритмом copy().

```
//-------------------------------------------------------------------
-------
// Демонстрация работы istream_iterator с файлами
// Листинг 9.cpp: определяет точку входа для консольного приложения.
//
#include "stdafx.h"
#include <iterator>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <list>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);
   list<int> iList;                // пустой список
   ifstream infile("ITER.DAT");    // создать входной файловый объект
                                   // (файл ITER.DAT должен уже
существовать)
                                   // итераторы istream
   istream_iterator<int> file_iter(infile);  // файл
   istream_iterator<int> end_of_stream;      // eos (конец потока)
                                   // копировать данные из входного
файла в iList
   copy( file_iter, end_of_stream, back_inserter(iList) );

   cout << endl;                   // вывести iList
   ostream_iterator<int> ositer(cout, "--");
   copy(iList.begin(), iList.end(), ositer);
   cout << endl;
   system("pause");
   return 0;
}
//-------------------------------------------------------------------
-------
```

Результат работы программы:

11--21--31--31--41--51--

Объект ifstream используется для представления файла ITER.DAT, который к моменту запуска программы должен уже физически существовать на диске и содержать

необходимые данные (в программе Листинг 7 мы его, если помните, создавали).

Вместо cout, как и в случае с итератором istream в программе INITER, мы пользуемся объектом класса ifstream под названием infile. Для обозначения конца потока используется все тот же объект.

Еще одно изменение, сделанное в этой программе, заключается в том, что мы используем back_inserter для вставки данных в iList. Таким образом, этот контейнер можно сделать изначально пустым, не задавая его размеров. Так имеет смысл делать при чтении входных данных, поскольку заранее неизвестно, сколько элементов будет введено.