5. Итераторы

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 эти средства просмотра.

Итератор – это обобщение понятия указателя для работы с различными структурами данных стандартным способом.

В стандартной библиотеке итераторы используются для работы с контейнерными классами (однонаправленными и двунаправленными списками, упорядоченными и неупорядоченными множествами, стеками, очередями и др.), потоками, строками (string), обычными массивами в стиле языка Си и т.п.

Отличие от обычных указателей заключается в том, что итератор является *интеллектуальным указателем*, т.е. может обходить более сложные структуры данных. Внутреннее поведение итераторов зависит от структуры данных, по которой они перемещаются. По этой причине каждый контейнерный тип предусматривает свой собственный вид итераторов. В результате итераторы имеют общий интерфейс, но разные типы.

В каждом контейнере объявляются два типа итераторов.

1. Итератор container::iterator перемещается по элементам в режиме чтения/записи.

2. Итератор container::const_iterator перемещается по элементам только в режиме чтения.

Константные итераторы используются тогда, когда изменять значения соответствующих элементов контейнера нет необходимости.

Например, в классе list определения могут иметь следующий вид:

```
template <typename T>
class list {
    public:
        typedef . . . iterator;
        typedef . . . const_iterator;
        . . . . .
};
```

Точный тип iterator и const_iterator определяется реализацией.

5.1. Категории итераторов

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

Данные могут быть организованы различным образом – например, в виде массива, списка или дерева. Для каждого вида последовательности требуется свой тип итератора, поддерживающий различные наборы операций. В соответствии с набором обеспечиваемых операций итераторы делятся на пять категорий, приведенных в таблице.

Операции	Категория				
	вывода (output)	ввода (input)	прямой (forward)	двунаправленный (bidirectional)	произвольного доступа (random-access)
Чтение		=*p	=*p	=*p	=*p
Доступ		->	->	->	-> [ ]
Запись	*p=		*p=	*p=	*p=
Итерация	++	++	++	++ --	++ -- + - += -=
Сравнение		= = !=	= = !=	= = !=	= = != < > >= <=

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.

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

• итератор не был инициализирован;

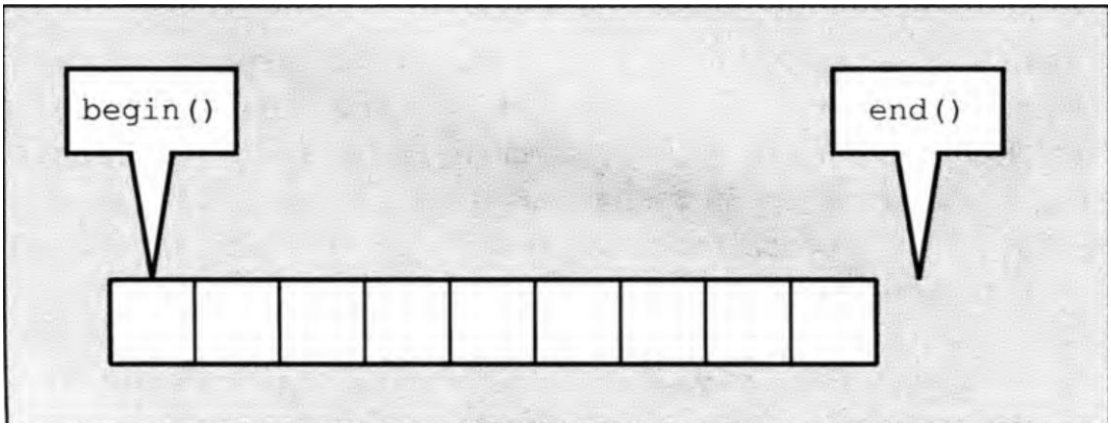
• контейнер, с которым он связан, изменил размеры или уничтожен;

• итератор указывает на конец последовательности.

Конец последовательности представляется как указатель на (воображаемый) 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. Для того, что узнать, указывает ли итератор на некоторый элемент или нет, его стандартным образом сравнивают с концом данной последовательности, а не путем выявления нулевого элемента.

Для получения итераторов на первый и следующий за последним элементом некоторой последовательности (коллекции) с именем coll служат операции coll.begin() и coll.end(), а для получения константных итераторов – coll.cbegin() и coll.cend(). Также имеются глобальные функции std::begin()/std::end(), std::cbegin()/std::cend(), применимые, например, также для массивов языка Си.



Таким образом, диапазон элементов представляет собой полуоткрытый интервал, стандартная математическая форма записи которого имеет вид: [begin, end).

Рассмотрим специфику отдельных категорий итераторов.

К итераторам вывода не применяются операции сравнения. Невозможно проверить, является ли итератор вывода корректным и была ли запись успешной. Можно только записывать значения. Как правило, конец записи определяется дополнительным требованием к конкретным итераторам вывода.

Итераторы ввода могут читать элементы только один раз. Следовательно, если вы копируете итератор ввода и выполняете чтение с помощью оригинального итератора и его копии, то можете получить разные значения.

Для итераторов ввода операции = = и != выполняют лишь проверку, равен ли итератор итератору элемента, следующего за последним. Это необходимо потому, что операции над итераторами ввода обычно выполняются следующим образом:

```
// pos - итератор ввода текущего элемента,
// end - итератор ввода элемента, следующего за последним
    while (pos != end) {
    . . . . .   // доступ только для чтения с помощью *pos
    ++pos;
    }
```

Нет гарантии, что два разных итератора ввода, которые оба не равны итератору end, при сравнении окажутся разными, если они ссылаются на разные позиции.

Для двух однонаправленных итераторов, ссылающихся на один и тот же элемент, гарантируется, что операция = = возвращает значение true и что после инкремента обоих итераторов они будут ссылаться на один и тот же элемент.

Пример. Однонаправленные итераторы.

```
// pos1, pos2 - однонаправленные итераторы
    pos1 = pos2 = begin;  // оба итератора ссылаются на один и тот же элемент
    if (pos1 != end) {
        ++pos1;        // итератор pos1 смещается на один элемент вперед
        while (pos1 != end) {
            if (*pos1 = = *pos2) {
                . . . . . // обработка соседних дубликатов
                ++pos1;
                ++pos2;
            }
        }
    }
```

Итераторы произвольного доступа обладают всеми возможностями двунаправленных итераторов и произвольным доступом к элементам. Следовательно, они реализуют операции арифметики итераторов (по аналогии с арифметикой обычных указателей). Иначе говоря, к ним можно прибавлять и вычитать смещения, вычислять их разности и сравнивать итераторы с помощью операторов сравнения, например < и >.

Замечание. Здесь и далее мы иллюстрируем использование итераторов на примере строки типа string, которая рассматривается как контейнер, содержащий отдельные символы. В связи с этим заметим, что, хотя такой взгляд на строки и возможен, реализации типа string оптимизируются для работы со строкой целиком, а не с ее отдельными элементами.

Пример. Следующая программа иллюстрирует специальные возможности итераторов произвольного доступа.

```
void main()
{
string coll="1234567";
// выводим на экран количество элементов,
// вычисляя расстояние между концом и началом строки
// ПРИМЕЧАНИЕ: к итераторам применяется оператор -
cout << "number/distance: " << coll.end()-coll.begin() << endl;
// выводим на экран все элементы
// ПРИМЕЧАНИЕ: к итераторам применяется оператор <, а не !=
string::iterator pos;// объявляем итератор для типа string
for (pos=coll.begin(); pos<coll.end(); ++pos) cout << *pos << ' ';
```

```
cout << endl;
// выводим на экран все элементы
// ПРИМЕЧАНИЕ: к итераторам применяется оператор [], а не *
for (int i=0; i<coll.size(); ++i) cout << coll.begin()[i] << ' '; // эквивалентно coll[i]
cout << endl;
// выводим на экран каждый второй элемент
// ПРИМЕЧАНИЕ: используется оператор +=
for (pos = coll.begin(); pos < coll.end()-1; pos += 2) cout << *pos << ' ';
cout << endl;
}
```

Программа выводит следующий результат.

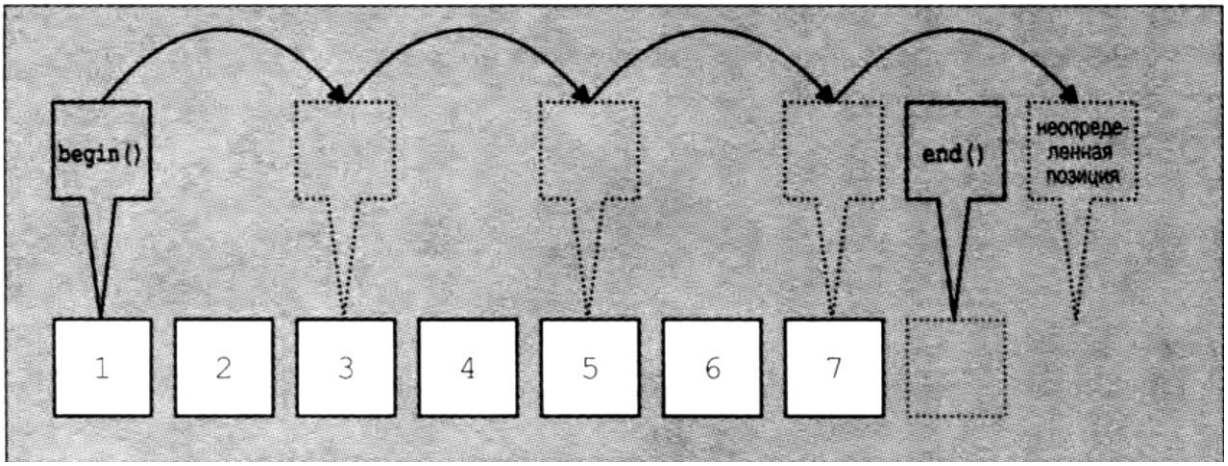
```
number/distance: 7
1 2 3 4 5 6 7
1 2 3 4 5 6 7
1 3 5
```

Отметим, что следующее выражение в последнем цикле требует, чтобы строка coll содержала хотя бы один элемент:

```
pos < coll.end()-1
```

Если коллекция пустая, выражение coll.end()-1 может вернуть позицию, предшествующую coll.begin(). Сравнение может по-прежнему выполняться, но, строго говоря, перемещение итератора на позицию, предшествующую началу контейнера, приводит к непредсказуемым последствиям. Аналогично выражение pos+=2 может привести к непредсказуемым последствиям, если оно перемещает итератор за позицию end(). Таким образом, изменение последнего цикла на следующий фрагмент может оказаться очень опасным, поскольку приводит к непредсказуемым последствиям, если коллекция содержит нечетное количество элементов (см. рисунок).

```
for (pos = coll.begin(); pos < coll.end(); pos+= 2) cout << *pos << ' ';
```



5.2. Вспомогательные функции для работы с итераторами

Стандартная библиотека содержит несколько вспомогательных функций для работы с итераторами: advance(), next(), prev(), distance() и iter_swap(). Первые четыре функции наделяют все итераторы некоторыми возможностями, которыми обладают только итераторы произвольного доступа: перемещаться на один элемент вперед (или назад) и вычислять разность между итераторами. Последняя вспомогательная функция позволяет обменивать значения двух итераторов.

Функция advance().

Функция advance() переносит итератор вперед на указанное аргументом количество позиций. Следовательно, эта функция позволяет итератору перемещаться вперед (или назад) на несколько элементов.

advance (pos, n)

• Переносит итератор ввода pos на n элементов вперед (или назад).

• Для двунаправленных итераторов и итераторов произвольного доступа число n может быть отрицательным, задавая перемещение назад.

• Функция advance() не проверяет, заходит ли она за позицию end() последовательности (она не может это проверить, потому что итераторы вообще не знают ничего о контейнерах, которые они обходят). Следовательно, вызов этой функции может привести к непредсказуемым последствиям, потому что результат применения операции ++ к концу последовательности не определен.

Эта функция всегда использует наилучшую реализацию, зависящую от категории итератора. Для итераторов произвольного доступа она просто выполняет вызов pos+=n. Следовательно, для таких итераторов функция advance() имеет константную сложность. Для всех других итераторов она n раз выполняет оператор ++pos (или --pos, если n отрицательное). Таким образом, для всех других категорий итераторов функция advance() имеет линейную сложность.

Пример. Использование функции advance().

```
void main()
{
string coll="1234567";
string::iterator pos = coll.begin();
// выводим на экран текущий элемент
cout << *pos << endl;
// перемещаемся на три элемента вперед
advance (pos, 3);
// выводим на экран текущий элемент
cout << *pos << endl;
// перемещаемся на один элемент назад
advance (pos, -1);
// выводим на экран текущий элемент
cout << *pos << endl;
}
```

В этой программе функция advance() позволяет переместить итератор pos на три элемента вперед и на один элемент назад. Следовательно, результат работы программы будет таким:

1
4
3

Функции next() и prev().

В соответствии со стандартом C++11 две эти вспомогательные функции позволяют перемещать итератор в следующую или предыдущую позицию.

Функции next (pos), next (pos, n):

• Возвращают позицию прямого итератора pos, в которую он попал бы, если бы переместился на одну или n позиций вперед.

• Для двунаправленных итераторов и итераторов произвольного доступа число n может быть отрицательным и приводить к перемещению в предыдущие позиции.

• При выполнении функции выполняется вызов advance(pos, n) для внутреннего временного объекта.

• Функция next() не проверяет факт выхода за позицию end() последовательности. Следовательно, именно вызывающая сторона должна гарантировать корректность результата.

Функции prev (pos), prev (pos, n):

• Возвращают позицию двунаправленного итератора pos, в которую он попал бы, если бы переместился на одну или n позиций назад.

• Для перехода на последующие позиции число n может быть отрицательным.

• При выполнении функции выполняется вызов advance(pos, -n) для внутреннего временного объекта.

• Функция prev() не проверяет факт выхода за позицию begin() последовательности. Следовательно, именно вызывающая сторона должна гарантировать корректность результата.

Эти функции позволяют, например, обойти коллекцию, проверяя значения следующих элементов:

```
pos = coll.begin();
while (pos != coll.end() && next(pos) != coll.end()) {
. . . . .
++pos;
}
```

Это особенно полезно, учитывая, что прямые и двунаправленные итераторы не предусматривают операции + и -. В противном случае нам всегда требовался бы временный объект:

```
pos = coll.begin();
nextPos = pos;
```

```
++nextPos;
while (pos != coll.end() && nextPos != coll.end()) {
. . . . .
++pos; ++nextPos;
}
```

или код, использующий исключительно итераторы произвольного доступа:

```
pos = coll.begin();
while (pos != coll.end() && pos+1 != coll.end()) {
. . . . .
++pos;
}
```

Не забудьте проверить, что позиция является корректной, перед тем как использовать ее (по этой причине мы сначала проверяем, не равен ли итератор pos итератору coll.end (), прежде чем перейти на следующую позицию).

Функция distance().

Функция distance() предназначена для вычисления разности между двумя итераторами.

Функция distance (pos1, pos2):

• Возвращает расстояние между итераторами ввода pos1 и pos2.

• Оба итератора должны ссылаться на элементы, принадлежащие одной и той же последовательности.

• Если итераторы не являются итераторами произвольного доступа, то позиция итератора pos2 должна быть достижимой из позиции итератора pos1; иначе говоря, итератор pos2 должен иметь ту же позицию или позицию, расположенную правее.

• Тип возвращаемого значения – это тип разности, соответствующий типу итератора.

Эта функция выбирает наилучшую реализацию в соответствии с категорией итератора. Для итераторов произвольного доступа эта функция просто возвращает значение выражения pos2 - pos1. Таким образом, для таких итераторов функция distance() имеет константную сложность. Для всех других категорий итераторов выполняется многократный инкремент итератора pos1, пока он не достигнет позиции итератора pos2, и после этого возвращается количество выполненных инкрементов. Следовательно, для всех остальных категорий итераторов функция distance() имеет линейную сложность. Итак, функция distance() имеет низкую производительность для всех итераторов, кроме итераторов произвольного доступа.

Фрагмент

```
cout << distance(coll.begin(),coll.end()) << endl;
```

дает довольно тривиальный пример использования функции distance() для вычисления размера коллекции.

При вычислении разности между итераторами, которые не являются ите-

раторами произвольного доступа, будьте внимательны. Первый итератор должен ссылаться на элемент, расположенный до, а не после элемента, на который ссылается второй итератор. В противном случае последствия будут непредсказуемыми. Если вам неизвестно, какой из итераторов предшествует другому, следует вычислить расстояние между ними и началом контейнера, а затем вычислить разность между этими расстояниями.

Функция iter_swap().

Это простая вспомогательная функция, предназначенная для обмена значений, на которые ссылаются два итератора.

Функция iter_swap (pos1, pos2)

• Обменивает значения, на которые ссылаются итераторы pos1 и pos2.

• Итераторы не обязаны иметь одинаковый тип. Однако значения должны допускать присваивание.

5.3. Адаптеры итераторов

5.3.1. Обратные итераторы

Обратный итератор reverse_iterator переопределяет операции инкремента и декремента так, что их семантика становится обратной. Следовательно, если использовать этот итератор вместо обычного, алгоритмы будут выполняться в обратном направлении.

Большинство контейнерных классов стандартной библиотеки, а также строки обеспечивают возможность использования обратных итераторов для обхода своих элементов. Рассмотрим следующий пример:

```
void main()
{
string coll="1234567";
string::iterator pos;
for (pos=coll.begin(); pos != coll.end(); ++pos)
        cout << *pos << ' ';
cout << endl;
string::reverse_iterator rpos; // определяем обратный итератор
for (rpos=coll.rbegin(); rpos != coll.rend(); ++rpos)
        cout << *rpos << ' ';
cout << endl;
}
```

Функции-члены rbegin() и rend() возвращают обратный итератор. Аналогично функциям begin() и end(), эти итераторы определены на полуоткрытом диапазоне. Однако они действуют в обратном направлении.

• Функция-член rbegin() возвращает позицию первого элемента при обратном обходе. Следовательно, она возвращает позицию последнего элемента.

• Функция-член rend() возвращает позицию, расположенную за последним элементом при обратном обходе. Следовательно, она возвращает позицию, предшествующую первому элементу.

Результат работы программы выглядит следующим образом:

```
1 2 3 4 5 6 7
7 6 5 4 3 2 1
```

Также имеются соответствующие функции-члены crbegin() и crend(), возвращающие обратные итераторы только для чтения, и глобальные функции std::rbegin()/std::rend(), std::crbegin()/std::crend().

Итераторы и обратные итераторы.

Обычные итераторы можно превратить в обратные. Естественно, эти итераторы должны быть двунаправленными, но обратите внимание, что логическая позиция итератора при его преобразовании перемещается. Рассмотрим следующую программу:

```
void main()
{
string coll="123456789";
string::iterator pos= coll.begin()+4;
cout << "pos: " << *pos << endl;
string::reverse_iterator rpos(pos);
cout << "rpos: " << *rpos << endl;
}
```

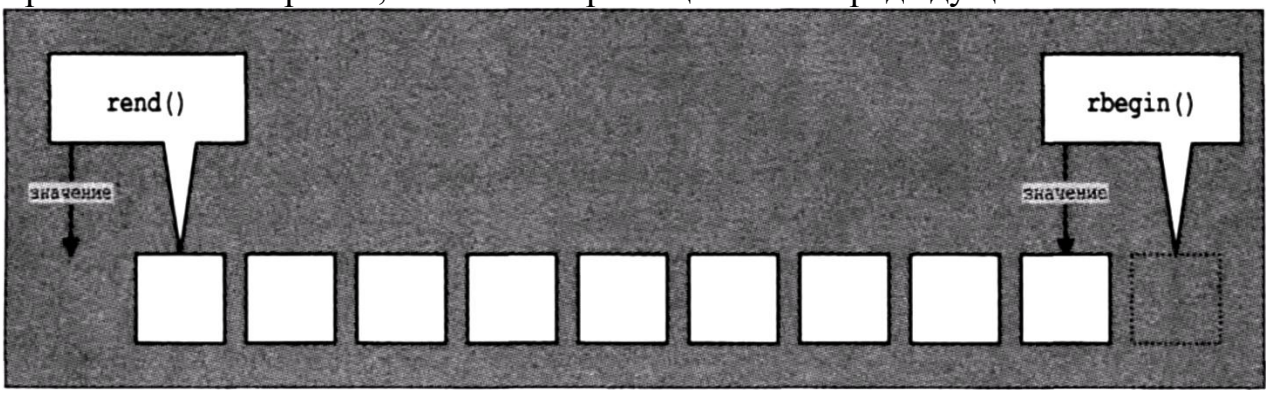
Эта программа выводит на экран следующий результат:

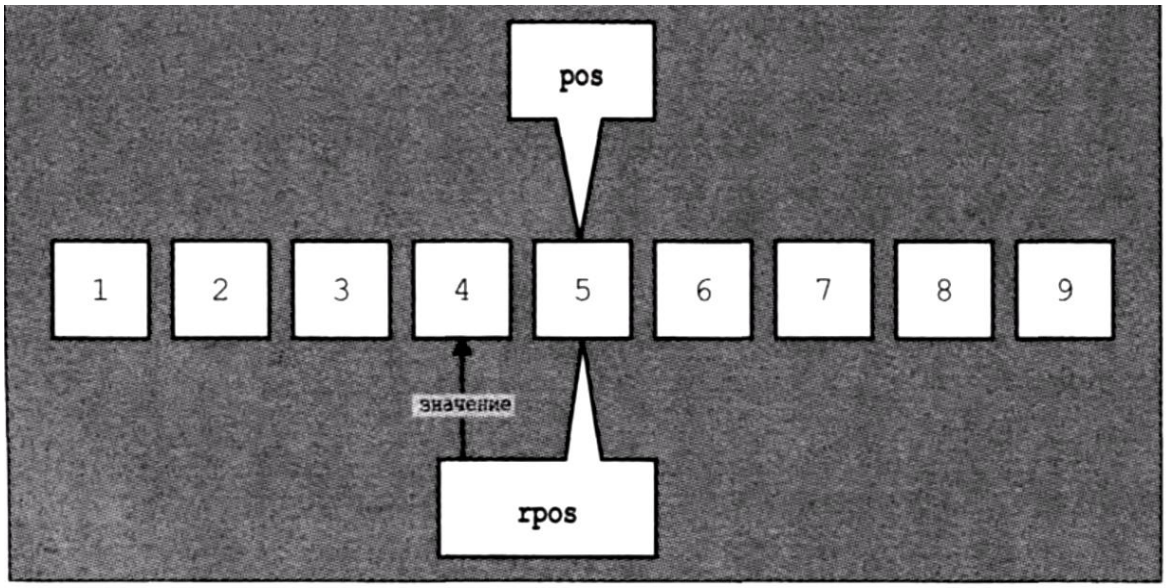
```
pos: 5
rpos: 4
```

Таким образом, при выводе значения итератора и его преобразовании в обратный итератор это значение изменилось. Это не ошибка, а особенность! Это поведение является следствием того факта, что диапазоны являются полуоткрытыми. Для того чтобы определить все элементы в контейнере, необходимо использовать позицию, расположенную за последним элементом. Однако для обратного итератора этой позицией является позиция, предшествующая первому элементу. К сожалению, эта позиция может не существовать. Контейнеры не обязаны гарантировать, что позиция, предшествующая первому элементу, является корректной. Представьте себе, что обычные строки и массивы тоже контейнеры, а язык программирования не гарантирует, что адресация массивов начинается с нулевого адреса.

В результате разработчики обратных итераторов использовали трюк: они "физически" изменили на противоположный "принцип полуоткрытости". Физически в диапазоне определены обратные итераторы, начало не включено в диапазон, а конец включен. Однако логически итераторы работают как обычно. Следовательно, существует различие между физической позицией, определя-

ющей элемент, на который ссылается итератор, и логической позицией, определяющей значение, на которое ссылается итератор (см. рисунки). Вопрос заключается в том, что происходит во время превращения итератора в обратный итератор? Какую позицию сохраняет итератор: логическую (значение) или физическую (элемент)? Как показывает предыдущий пример, имеет место второй вариант. Таким образом, значение перемещается на предыдущий элемент.






```
void main()
{
string coll="1234567";
string::iterator pos1=coll.begin()+1, pos2=coll.end()-1;
string::reverse_iterator rpos1(pos2), rpos2(pos1);
for (; pos1 != pos2; ++pos1) cout << *pos1 << " ";
cout << endl;
for (; rpos1 != rpos2; ++rpos1) cout << *rpos1 << " ";
cout << endl;
}
```

Итераторы pos1 и pos2 определяют полуоткрытый диапазон, включая элемент со значением 2, но исключая элемент со значением 7. Когда итератор, описывающий этот диапазон, превращается в обратный, диапазон остается корректным и может обрабатываться в обратном порядке. Таким образом, про-

грамма выводит на экран следующие результаты:
2 3 4 5 6
6 5 4 3 2

Константные итераторы могут преобразовываться в тип const_reverse_iterator.

Обратное преобразование обратных итераторов с помощью функции-члена base().

Обратные итераторы можно превратить обратно в обычные. Для этого обратные итераторы снабжены функцией-членом base().
Рассмотрим пример использования функции base():

```
void main()
{
string coll="123456789";
string::iterator pos = coll.begin()+4;
cout << "pos: " << *pos << endl;
string::reverse_iterator rpos(pos);
cout << "rpos: " << *rpos << endl;
// превращаем обратный итератор в обычный
string::iterator rrpos;
rrpos = rpos.base();
cout << "rrpos: " << *rrpos << endl;
}
```

Эта программа выводит на экран следующие строки:
pos: 5
rpos: 4
rrpos: 5

Таким образом, преобразование итератора с помощью функции base() эквивалентно его превращению в обратный итератор. Иначе говоря, физическая позиция (элемент итератора) сохраняется, а логическая (значение элемента) перемещается.

5.3.2. Итераторы вставки

Итераторы вставки – это адаптеры итераторов, преобразовывающие присвоение нового значения во вставку этого нового значения. Используя итераторы вставки, алгоритмы могут вставлять элементы, а не заменять их. Все итераторы вставки относятся к категории итераторов вывода. Таким образом, они обеспечивают только возможность присваивать новые значения.

Операции над итераторами вставки:

Выражение	**Результат**
*iter	Фиктивный (возвращает iter)
iter = value	Вставляет значение

++iter	Фиктивный (возвращает iter)
iter++	Фиктивный (возвращает iter)

Таким образом, чтобы вставить новый элемент с помощью итератора вставки, можно писать iter = value, а не *iter = value. Однако это касается деталей реализации итераторов ввода. Правильным выражением для присваивания нового значения является *iter = value.

Аналогично операция инкремента реализуется как фиктивная операция, просто возвращающая *this. Следовательно, модифицировать позицию итератора вставки невозможно.

Стандартная библиотека C++ содержит три вида итераторов вставки: итераторы вставки в конец, итераторы вставки в начало и общие итераторы вставки (обычно называемые просто итераторами вставки). Они различаются позициями, в которые выполняется вставка. Фактически каждый из них использует определенную функцию-член стандартных контейнеров (push_back(), push_front(), insert() соответственно), если она есть у контейнеров, иначе соответствующего итератора нет. Следовательно, итератор вставки всегда должен быть инициализирован своим контейнером.

Каждый вид итератора вставки имеет удобную функцию для его создания и инициализации.

Имя	**Класс**	**Создание**
Итератор вставки в конец	back_insert_iterator<type>	back_inserter(cont)
Итератор вставки в начало	front_insert_iterator<type>	front_inserter(cont)
Общий итератор вставки	insert_iterator<type>	inserter(cont,pos)

Итераторы вставки в начало и конец должны инициализироваться контейнером в момент своего создания. Функции front_inserter() и back_inserter() обеспечивают удобный способ для этого.

Общий итератор вставки инициализируется двумя значениями: контейнером и позицией для вставки. Функция-член inserter() обеспечивает удобный способ для создания и инициализации общего итератора вставки.

Итераторы вставки в начало и конец указывают соответственно на начало и конец последовательности, куда и вставляют элементы. Общий итератор вставки всегда указывает на позицию того **элемента**, который указан при его создании, поэтому все элементы вставляются **перед ним**.

Пример. Использование итераторов вставки для строк типа string.

Во-первых, заметим, что строки не имеют операции добавления элементов в начало push_front(), поэтому у них нет и соответствующего итератора вставки. Однако, функция-член insert(pos, ch) позволяет вставку символа ch в произвольную позицию pos, которой может быть и первая позиция. Значит, принципиально такой итератор можно создать, хотя и путем отхода от описанного выше правила реализации такого итератора (и оставляя в стороне вопрос о

его эффективности).

Поэтому первым делом рассмотрим специализацию шаблона front_insert_iterator для типа string. Это позволит нам, помимо прочего, заглянуть внутрь итераторов – изучить, как они могут быть устроены.

```
template<> class front_insert_iterator<string>{
protected:
	string& str; // строка, в которую вставляются элементы
public:
// конструктор
	explicit front_insert_iterator (string& s) : str(s) { }
// оператор присваивания - вставляет символ ch в начало строки
	front_insert_iterator<string>&
	operator= (const char& ch) {
		str.insert(str.begin(), ch); // используем функцию-член insert
		return *this;
	}
// разыменование - это фиктивная операция, возвращающая сам итератор
	front_insert_iterator<string>& operator*() {
		return *this;
	}
// инкремент - это фиктивная операция, возвращающая сам итератор
// префиксный инкремент
	front_insert_iterator<string>& operator++() {
		return *this;
	}
// постфиксный инкремент (отличается наличием фиктивного аргумента)
	front_insert_iterator<string>& operator++(int) {
		return *this;
	}
};
// вспомогательная функция для создания итератора вставки
	inline front_insert_iterator<string> front_inserter(string& s)
	{
		return front_insert_iterator<string>(s);
	}
```

Теперь воспользуемся всеми итераторами вставки, в том числе, и только что написанным.

```
void main()
{
string coll;
string::iterator pos;
back_insert_iterator<string> iter1(coll);
*iter1='3'; // * - фиктивная операция
```

```
iter1++; // ++ - фиктивная операция
*iter1='4';
back_inserter(coll)='5';
back_inserter(coll)='6';
for (pos=coll.begin(); pos != coll.end(); ++pos) cout << *pos << ' ';
cout << endl;
insert_iterator<string> iter2(coll, coll.begin());
*iter2='1';
iter2++;
*iter2='2';
inserter(coll, coll.end())='7';
inserter(coll, coll.end())='8';
for (pos=coll.begin(); pos != coll.end(); ++pos) cout << *pos << ' ';
cout << endl;
front_insert_iterator<string> iter3(coll);
*iter3='1';
iter3++;
*iter3='2';
front_inserter(coll)='7';
front_inserter(coll)='8';
for (pos=coll.begin(); pos != coll.end(); ++pos) cout << *pos << ' ';
}
```

Результат работы программы выглядит следующим образом:

```
3 4 5 6
1 2 3 4 5 6 7 8
8 7 2 1 1 2 3 4 5 6 7 8
```

Отметим, что несколько раз последовательно использованный итератор вставки в начало (как стандартный, так и наш собственный) вставляет элементы в обратном порядке. Это объясняется тем, что следующий элемент всегда вставляется перед предыдущим.

5.3.3. Итераторы потоков

Итератор потока – это адаптер итератора, позволяющий использовать поток в качестве источника или получателя данных для алгоритмов стандартной библиотеки. В частности, итератор потока ввода можно использовать для чтения элементов из потока ввода, а итератор потока вывода – для записи элементов в поток вывода.

Особой формой итератора потока является итератор буфера потока, который можно использовать для чтения или записи в буфер потока.

Итераторы потока вывода.

Итераторы потока вывода записывают присвоенные значения в поток

вывода. С помощью итераторов потока вывода алгоритмы могут записывать данные непосредственно в потоки.

Операции над итераторами вывода.

Выражение	**Результат**
ostream_iterator<type> iter(ostream)	Создает итератор iter потока вывода для потока ostream, будут выводиться элементы типа type
ostream_iterator<type> iter(ostream, delim)	Создает итератор iter потока вывода для потока ostream с разделителем, заданным строкой delim (строка delim имеет тип const char*), будут выводиться элементы типа type
*iter	Фиктивная операция (возвращает итератор iter)
iter = value	Записывает значение в поток ostream: ostream<<value (после значения выводится разделитель delim, если таковой задан)
++iter	Фиктивная операция (возвращает итератор iter)
iter++	Фиктивная операция (возвращает итератор iter)

При создании итератора потока вывода необходимо указать поток вывода, в который будут записываться данные. С помощью необязательного аргумента можно указать строку, которая будет использоваться как разделитель между отдельными значениями. Без разделителя элементы будут просто непосредственно следовать друг за другом.

Пример. Использование итераторов потоков вывода.

```
void main()
{
// создаем итератор потока вывода для потока cout
// значения разделяются символом перехода на новую строку
ostream_iterator<int> intWriter(cout,"\n");
// записываем элементы с помощью интерфейса итераторов
*intWriter = 42;
intWriter++;
*intWriter = 77;
intWriter++;
*intWriter = -5;
}
```

Результат работы этой программы выглядит следующим образом:

```
42
77
```

-5

Отметим, что разделитель имеет тип const char*. Следовательно, если передать объект типа string, необходимо вызывать его функцию-член c_str(), чтобы преобразовать его в правильный тип. Например:

```
string delim;
. . . . .
ostream_iterator<int>(cout, delim.c_str ());
```

Итераторы потоков ввода.

Итераторы потоков ввода – это противоположность итераторов потоков вывода. Итератор потока ввода читает элементы из потока ввода. Однако итераторы потоков ввода немного сложнее итераторов потоков вывода (поскольку чтение, как обычно, немного сложнее, чем запись).

В момент создания итератор потока ввода инициализируется потоком ввода, из которого он будет читать данные. Затем с помощью обычного интерфейса итераторов ввода итератор потока ввода читает элемент за элементом, используя оператор >>. Однако чтение может завершиться неудачей (из-за конца файла или ошибки). Для решения этих проблем можно использовать **итератор конца потока**, созданный с помощью конструктора по умолчанию итератора потока ввода. Если чтение завершается неудачей, итератор потока ввода превращается в итератор конца потока. Следовательно, после каждой попытки чтения необходимо сравнивать итератор потока ввода с итератором конца потока, чтобы убедиться, что итератор имеет корректное значение.

Операции над итераторами ввода.

Выражение	**Действие**
istream_iterator<type> eof()	Создает итератор eof конца потока
istream_iterator<type> iter(istream)	Создает итератор iter потока ввода для потока istream (и может считывать первое значение)
*iter	Возвращает считанное ранее значение (считывает первое значение, если оно не было считано конструктором)
iter->member	Возвращает член ранее прочитанного значения, если таковой имеется
++iter	Считывает следующее значение и возвращает его позицию
iter++	Считывает следующее значение и возвращает итератор для предыдущего значения
iter1 == iter2	Проверяет, равны ли итераторы iter1 и iter2
iter1 != iter2	Проверяет, не равны ли итераторы iter1 и iter2

Отметим, что конструктор итератора потока ввода открывает поток и обычно считывает первый элемент. В противном случае он не мог бы вернуть первый элемент при выполнении операции * после инициализации. Однако реализации могут отложить чтение первого значения до первого вызова операции

*. Таким образом, итератор потока ввода не следует определять до того, как он потребуется.

Два потоковых итератора ввода считаются равными, если

• они являются итераторами конца потока и, следовательно, больше не могут читать данные, или

• оба итератора могут читать и использовать один и тот же поток.

Пример. Операции, предусмотренные для потоковых итераторов ввода.

```
void main()
{
// создаем итератор потока ввода, читающий целые числа из потока cin
istream_iterator<int> intReader(cin);
// создаем итератор конца потока
istream_iterator<int> intReaderEOF;
// читаем данные с помощью потокового итератора ввода, пока это возможно,
// записываем их дважды
        while(intReader != intReaderEOF) {
                cout << "once: " << *intReader << endl;
                cout << "once again: " << *intReader << endl;
                ++intReader;
        }
}
```

Если выполнить эту программы с входными данными

1 2 3 f 4

то результат ее работы будет выглядеть так:

```
once: 1
once again: 1
once: 2
once again: 2
once: 3
once again: 3
```

Легко видеть, что ввод символа f завершает работу программы. Из-за ошибки формата поток теряет корректное состояние. Следовательно, итератор потока ввода intReader равен итератору конца потока intReaderEOF. В результате условие цикла становится равным false.

Контейнеры стандартной библиотеки имеют конструкторы, позволяющие инициализацию указанием начального и конечного итераторов. Этими итераторами могут быть и итераторы потока ввода, что позволяет ввести объект из потока, например, из файла.

Пример. Инициализация строки содержимым файла.

```
ifstream inpf ("in.txt"); // входной файл
inpf >> noskipws; // не пропускать символы-разделители
istream_iterator<char> si(inpf), eof; // eof - итератор конца потока
string ss(si, eof); // конструктор с инициализацией
```

5.4. Диапазонные циклы for

Стандарт C++11 предоставляет упрощенное средство перебора всех элементов в заданном диапазоне, массиве или коллекции – использование новой формы цикла for, называемой в других языках программирования foreach. Общая синтаксическая конструкция этого цикла имеет следующий вид:

```
for (decl : coll) {
операторы }
```

где decl – объявление каждого элемента перебираемой коллекции coll, и к каждому элементу применяются указанные операторы. Например, следующий цикл применяет к каждому значению передаваемого списка инициализации оператор, выводящий это значение в стандартный поток вывода cout:

```
for (int i : {2, 3, 5, 7, 9, 13, 17, 19})
        cout << i << endl;
```

Для умножения каждого элемента elem вектора vec на 3 можно написать следующий код:

```
vector<double> vec;
. . . . .
for (auto& elem : vec) elem *= 3;
```

Здесь важно подчеркнуть, что переменная elem объявлена как ссылка, потому что в противном случае операторы в теле цикла for применяются к локальным копиям элементов в векторе (что иногда бывает полезным).

Обобщенная функция для вывода всех элементов коллекции может быть реализована следующим образом:

```
template <typename T>
void printElements (const T& coll)
{
        for (const auto& elem : coll)
                        cout << elem << endl;
}
```

Здесь переменная elem также объявлена как ссылка (причем, константная), для того чтобы избежать вызова копирующего конструктора и деструктора для каждого элемента.

В общем случае диапазонный цикл for, объявленный как

```
for (decl : coll) {
операторы
}
```

эквивалентен следующей конструкции, если объект coll имеет члены begin() и end():

```
{
```

```
        for (auto _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos) {
                decl = *_pos;
                операторы
        }
}
```

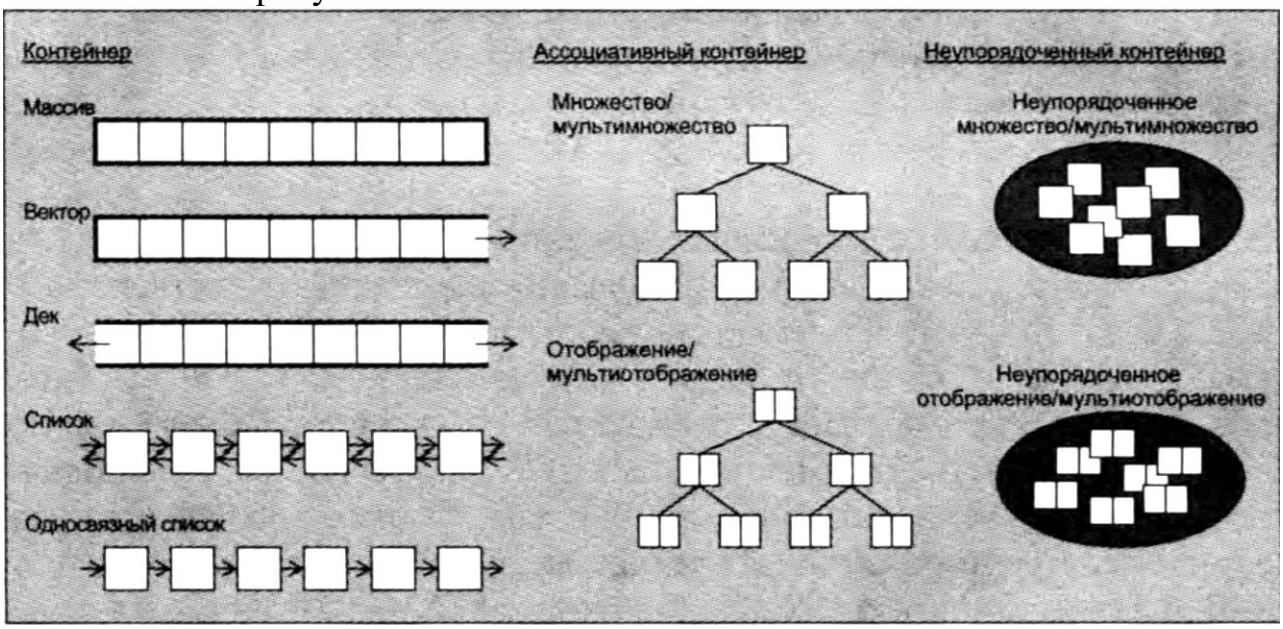
или, если таких членов нет, следующей конструкции, в которой используются глобальные функции begin() и end(), принимающие объект coll как аргумент:

```
{
        for   (auto_pos=begin(coll), _end=end(coll); _pos!=_end; ++_pos) {
                decl = *_pos;
                операторы
        }
}
```

6. Введение в контейнеры STL

6.1. Контейнеры и адаптеры контейнеров

Контейнерные классы, или контейнеры, управляют коллекциями элементов. Для разных целей в библиотеке STL предусмотрены разные контейнеры, как показано на рисунке.



Существуют три разновидности контейнеров.

1. Последовательные контейнеры – это упорядоченные коллекции, в которых каждый элемент занимает определенную позицию. Эта позиция зависит от времени и места вставки, но не зависит от значения элемента. Например, если вставить шесть элементов в упорядоченную коллекцию, добавляя каждый элемент в конец коллекции, то эти элементы будут следовать в точном порядке их вставки. Библиотека STL содержит пять стандартных контейнерных классов: массив (array) моделирует статический массив, вектор (vector) моделирует

динамический массив, дек (deque), двусвязный список (list) и односвязный список forward_list.

2. Ассоциативные контейнеры – это упорядоченные коллекции, в которых позиция элемента зависит от его значения (или ключа, если элемент представляет собой пару "ключ-значение") в соответствии с определенным критерием сортировки. Если вставить шесть элементов в такую коллекцию, то порядок их следования будет определен их значениями. В этом случае порядок вставки элементов не имеет значения. Библиотека STL содержит четыре стандартных ассоциативных контейнерных класса:

– Множество set – коллекция, в которой элементы сортируются в соответствии со своими значениями. Каждый элемент входит в коллекцию только один раз, так что дубликаты не допускаются.

– Мультимножество multiset – это множество, в котором разрешены дубликаты. Таким образом, мультимножество может содержать несколько элементов, имеющих одинаковые значения.

– Отображение map содержит пары "ключ-значение". У каждого элемента есть ключ, использующийся для сортировки, и значение. Каждый ключ может входить в отображение только один раз.

– Мультиотображение multimap – это отображение, в котором разрешены дубликаты.

Множество можно рассматривать как особую разновидность отображения, в котором значение идентично ключу.

Под парами "ключ-значение" понимаются объекты шаблонного класса **pair**, который описан в заголовочном файле <utility>. Основные операции с парами представлены в таблице.

Операция	Действие
pair<T1, T2> p	p – пара с переменными-членами типов T1 и T2, инициализированными значением по умолчанию
pair<T1, T2> p(v1, v2)	p – пара с переменными-членами типов T1 и T2, инициализированными значениями v1 и v2 соответственно
pair<T1, T2> p = {v1, v2}	Эквивалент p(v1, v2). В C++17 указание типов шаблонных параметров необязательно, если их можно вывести из инициализирующих значений: pair p = {v1, v2}
make_pair(v1, v2)	Возвращает пару, инициализированную значениями v1 и v2. Тип пары выводится из типов значений v1 и v2
p.first	Возвращает открытую переменную-член first пары p
p.second	Возвращает открытую переменную-член second пары p
p1 опсравн p2	Операторы сравнения (<, >, <=, >=). Пара p1 меньше

	пары p2, если p1.first < p2.first или p1.first = = p2.first && p1.second < p2.second
p1 = = p2, p1 != p2	Две пары равны, если их первый и второй члены соответственно равны. При сравнении используется оператор = = хранимых элементов

3. Неупорядоченные ассоциативные контейнеры – это неупорядоченные коллекции, в которых позиция элемента не имеет значения. В этом случае смысл имеет только один вопрос: принадлежит ли конкретный элемент такой коллекции. Ни порядок вставки, ни значение вставленного элемента не влияет на его позицию. Его позиция со временем может изменяться. Таким образом, если вставить в такую коллекцию шесть элементов, то их порядок будет неопределенным и со временем может измениться. Библиотека STL содержит четыре стандартных неупорядоченных контейнерных классов: неупорядоченное множество unordered_set, неупорядоченное мультимножество unordered_multiset, неупорядоченное отображение unordered_map и неупорядоченное мультиотображение unordered_multimap.

Далее под ассоциативным контейнером подразумевается упорядоченный ассоциативный контейнер, а термин "неупорядоченные контейнеры" употребляется без промежуточного слова "ассоциативные".

Три категории контейнеров, введенные выше, представляют собой логические группы, зависящие от определения порядка следования элементов. Они резко отличаются друг от друга и имеют совершенно разные реализации, не являющиеся производными друг от друга:

– Последовательные контейнеры обычно реализуются как массивы или связанные списки.

– Ассоциативные контейнеры, как правило, реализуются как бинарные деревья.

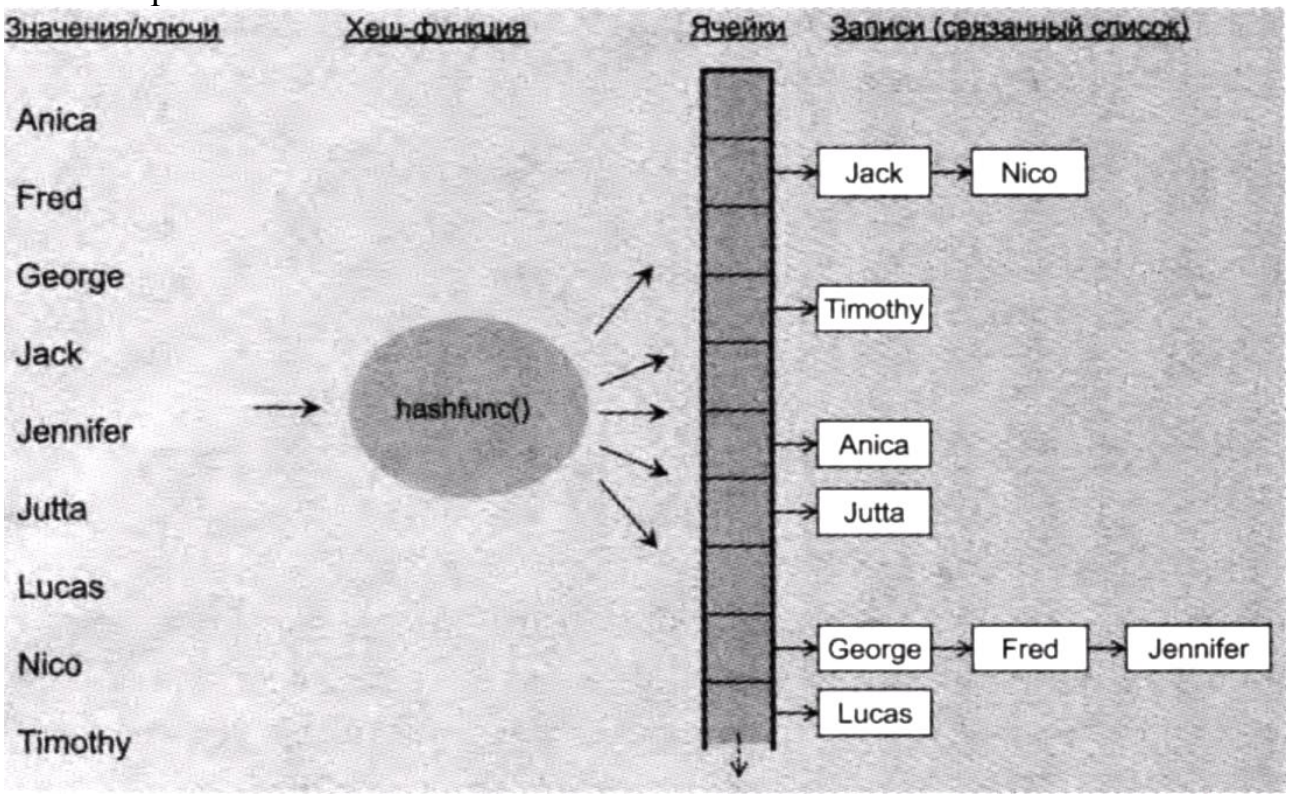
– Неупорядоченные контейнеры обычно реализуются как хеш-таблицы.

Таким образом, по существу, неупорядоченный контейнер – это массив связанных списков (см. рисунок). Позиция элемента в массиве вычисляется с помощью хеш-функции. Цель заключается в том, чтобы каждый элемент имел свою позицию, обеспечивающую к нему быстрый доступ, при условии быстрого вычисления хеш-функции. Однако, поскольку быстрые идеальные хеш-функции не всегда возможны или могут потребовать огромный объем памяти для массива, разрешается хранить в одной и той же позиции несколько элементов. По этой причине элементами массива являются связанные списки, позволяющие хранить в одной позиции массива несколько элементов контейнера.

Строго говоря, в стандартной библиотеке C++ не определена конкретная реализация ни одного контейнера. Однако поведение и сложность, определенные стандартом, не оставляют большого выбора вариантов. По этой причине на практике реализации различаются лишь незначительными деталями.

Адаптеры контейнеров.

Кроме основных контейнерных классов, в стандартной библиотеке C++ предусмотрены адаптеры контейнеров, представляющие собой стандартные контейнеры, предоставляющие ограниченный интерфейс для специальных нужд. Эти адаптеры контейнеров реализуются на основе фундаментальных контейнерных классов.



– Стек управляет своими элементами по принципу LIFO (последним вошел – первым вышел).

– Очередь управляет своими элементами по принципу FIFO (первым вошел – первым вышел).

– Очередь с приоритетами – это контейнер, в котором элементы имеют разные приоритеты. Приоритет зависит от критерия сортировки, который может задавать программист (по умолчанию используется операция <).

Адаптеры контейнеров исторически являются частью библиотеки STL. Однако с точки зрения программирования они представляют собой всего лишь специальную категорию контейнерных классов, использующих общую архитектуру контейнеров, итераторов и алгоритмов библиотеки STL.

Кроме того, некоторые классы обеспечивают контейнерный интерфейс: например, классы строк string, битовые множества bitset и класс для эффективной работы с массивами чисел valarray.

Пример. Вектор управляет элементами, хранящимися в динамическом массиве. Он обеспечивает произвольный доступ к элементам, т.е. к каждому элементу можно обратиться напрямую по соответствующему индексу. Добавление и удаление элементов происходит в конце массива и выполняется быстро. Однако вставка элемента в середину или в начало массива занимает достаточно много времени, потому что все последующие элементы должны переме-

ститься, чтобы освободить место для нового элемента, сохраняя порядок следования.

В следующем примере определяется вектор целых чисел, выполняются вставка шести элементов и вывод элементов вектора на экран:

```
#include <vector>
. . . . .
int main()
{
vector<int> coll;   // вектор для целочисленных элементов
// добавляем элементы со значениями от 1 до 6
for (int i=1; i<=6; ++i)
        coll.push_back(i);
// выводим на экран все элементы и пробел
for (int i=0; i<coll.size(); ++i)
        cout << coll[i] << ' ';
cout << endl;
}
```

Заголовочный файл для векторов включается с помощью директивы

```
#include <vector>
```

Следующее объявление создает вектор элементов типа int:

```
vector<int> coll;
```

Этот вектор не инициализирован никакими значениями, поэтому конструктор, заданный по умолчанию, создает его в виде пустой коллекции. Функция push_back() добавляет элемент в контейнер:

```
coll.push_back(i);
```

Эта функция-член есть во всех последовательных контейнерах, в которых возможно добавление элемента за достаточно быстрое время.

Функция-член size() возвращает количество элементов, содержащихся в контейнере:

```
for (int i=0; i<coll.size(); ++i)
```

Функция size() есть в любом контейнерном классе, за исключением односвязных списков (класса forward_list).

Используя операцию индексирования [ ], можно получить доступ к отдельному элементу вектора:

```
cout << coll[i] << ' ';
```

Здесь элементы выводятся в стандартный выходной поток данных, поэтому результат работы программы выглядит следующим образом:

```
1 2 3 4 5 6
```

6.2. Общие возможности контейнеров

Основные возможности контейнеров. Рассмотрим общие возможности контейнерных классов STL. Большая часть этих возможностей – требования,

которые в принципе должен удовлетворять каждый контейнер STL. Перечислим три основных возможности.

1. Все контейнеры поддерживают семантику значений, а не ссылочную семантику. При вставке контейнеры копируют и/или перемещают элементы, а не работают со ссылками на них. Таким образом, теоретически каждый элемент контейнера STL должен допускать копирование и перемещение. Если объекты, которые необходимо сохранить в контейнере, не имеют открытого копирующего конструктора или их копирование затруднительно (например, потому, что оно занимает слишком много времени), к ним можно применять только операции перемещения или же хранить в контейнере указатели.

2. Элементы в контейнере хранятся в определенном порядке. Каждый контейнерный тип имеет операции, возвращающие итераторы для перебора элементов. Это основной интерфейс для алгоритмов STL. Таким образом, если не выполняются операции вставки или удаления, при многократном переборе элементов порядок их следования не изменяется. Это относится даже к неупорядоченным контейнерам, если не выполняются операции добавления или удаления элементов или не инициируется внутренняя реорганизация контейнера.

3. В общем случае операции контейнеров не являются безопасными в том смысле, что они не проверяют все возможные ошибки. Вызывающая сторона должна гарантировать, что параметры операции соответствуют требованиям, установленным для этой операции. Нарушение этих требований, например, некорректный индекс, приводит к неопределенному поведению, т.е. случиться может все, что угодно.

4. Обычно библиотека STL не генерирует исключений. Если пользовательские операции, вызванные контейнерами STL, генерируют исключение, ситуация изменяется.

Требования к элементам контейнеров.

Элементы контейнеров должны удовлетворять определенным требованиям, поскольку контейнеры обрабатывают их определенным образом. Опишем эти требования и обсудим последствия того факта, что контейнеры неявно создают копии своих элементов.

Контейнеры, итераторы и алгоритмы библиотеки STL являются шаблонами. Вследствие этого они могут работать как со стандартными, так и с пользовательскими типами. Однако операции, которые они вызывают, накладывают на них определенные ограничения. Элементы контейнеров STL должны удовлетворять трем основным требованиям.

1. Элемент должен допускать копирование или перемещение. Таким образом, тип элемента явно или неявно должен иметь копирующий или перемещающий конструктор.

2. Сгенерированная копия должна быть эквивалентной оригиналу. Это значит, что любая проверка равенства должна возвращать одинаковые результаты как для оригинала, так и для копии.

3. Элемент должен допускать (перемещающее) присваивание. Контейнеры и алгоритмы используют операции присваивания для перезаписи старых элементов новыми.

4. Элемент должен допускать удаление с помощью деструктора. Контейнеры должны удалять внутренние копии элементов, когда эти элементы удаляются из контейнера. Таким образом, деструктор не должен быть закрытым. Кроме того, как обычно в языке C++, деструктор не должен генерировать исключение; в противном случае возникает неопределенное поведение.

Эти три операции неявно генерируются для любого класса. Таким образом, класс автоматически удовлетворяет перечисленным требованиям, если в нем не определены специальные версии этих операций и нет специальных функций-членов, нарушающих правильную работу этих операций.

Элементы контейнеров должны также удовлетворять следующим требованиям.

– Для некоторых функций-членов последовательных контейнеров должен быть доступным конструктор по умолчанию. Например, можно создать непустой контейнер или увеличить количество элементов в контейнере, не имея представления о том, какое значения должны принимать новые значения. Эти элементы создаются без аргументов с помощью вызова конструктора, заданного по умолчанию для данного типа.

– Для некоторых операций необходимо определять проверку равенства с помощью операции = =. Это особенно необходимо при поиске элементов. Однако для неупорядоченных контейнеров можно предусмотреть собственное определение эквивалентности, если их элементы не поддерживают операцию = =.

– Элементы ассоциативных контейнеров должны поддерживать операции критерии сортировки. По умолчанию в качестве таковой используется операция <.

– Элементы неупорядоченных контейнеров должны предусматривать хеш-функцию и критерий эквивалентности.

6.3. Операции над контейнерами

Стандарт содержит список общих требований, предъявляемых к контейнерам, которые применяются ко всем контейнерам STL. Однако вследствие разнообразия контейнеров, предусмотренных в стандарте C++11, допускаются исключения, позволяющие некоторым контейнерам не выполнять все требования, а также дополнительные операции, предусмотренные во всех контейнерах. В таблице перечислены операции, общие для (почти) всех контейнеров. Столбец "Требование" содержит индикатор того, что операции являются частью общих требований, предъявляемых к контейнерам. Напомним, что имя rv используется для rvalue-ссылок (семантика перемещения).

Операция	Требование	Действие
ContType c	Да	Конструктор по умолчанию; создает пустой контейнер, не содержащий ни одного элемента (контейнер array<> содержит элементы, заданные по умолчанию)
ContType c(c2)	Да	Копирующий конструктор; создает новый контейнер, являющийся копией контейнера c2 (все элементы копируются)
ContType c = c2	Да	Копирующий конструктор; создает новый контейнер, являющийся копией контейнера c2 (все элементы копируются)
ContType c(rv)	Да	Перемещающий конструктор; создает новый контейнер, получающий содержание от контейнера rv (к классу array<> это не относится)
ContType c = rv	Да	Перемещающий конструктор; создает новый контейнер, получающий содержание от контейнера rv (к классу array<> это не относится)
ContType c (beg, end)		Создает контейнер и инициализирует его копиями всех элементов интервала [beg,end) (к классу array<> это не относится)
ContType c (initlist)		Создает контейнер и инициализирует его копиями значения из списка инициализации initlist (к классу array<> это не относится)
ContType c = initlist		Создает контейнер и инициализирует его копиями значения из списка инициализации initlist
c.~ContType()	Да	Удаляет все элементы и освобождает память, если это возможно
c.empty()	Да	Возвращает признак того, что контейнер пуст (эквивалент size() = = 0, но может работать быстрее)
c.size()	Да	Возвращает текущее количество элементов (к классу forward_list<> это не относится)
c.max_size()	Да	Возвращает максимально возможное количество элементов
c1 = = c2	Да	Возвращает признак того, что контейнер c1 равен контейнеру c2
c1 != c2	Да	Возвращает признак того, что контейнер c1 не равен контейнеру c2 (эквивалент !(c1 = = c2))

c1 < c2		Возвращает признак того, что контейнер c1 меньше контейнера c2 (это не относится к неупорядоченным контейнерам)
c1 > c2		Возвращает признак того, что контейнер c1 больше контейнера c2 (эквивалент c2 < c1; это не относится к неупорядоченным контейнерам)
c1 <= c2		Возвращает признак того, что контейнер c1 меньше или равен контейнеру c2 (эквивалент !(c2<c1); это не относится к неупорядоченным контейнерам)
c1 >= c2		Возвращает признак того, что контейнер c1 больше или равен контейнеру c2 (эквивалент !(c1<c2); это не относится к неупорядоченным контейнерам)
c = c2	Да	Присваивает все элементы контейнера c2 контейнеру c
c = rv	Да	Присваивает с перемещением все элементы контейнера rv контейнеру c (это не относится к классу array<>)
c = initlist		Присваивает все элементы списка инициализации initlist (это не относится к классу array<>)
c1.swap(c2)	Да	Обменивает данные контейнеров cl и c2
swap(c1, c2)	Да	Обменивает данные контейнеров cl и c2
c.begin()	Да	Возвращает итератор, установленный на первый элемент
c.end()	Да	Возвращает итератор, установленный на позицию, следующую за последним элементом
c.cbegin()	Да	Возвращает константный итератор, установленный на первый элемент
c.cend()	Да	Возвращает константный итератор, установленный на позицию, следующую за последним элементом
c.clear()		Удаляет все элементы (опустошает контейнер; это не относится к классу array<>)

Инициализация.

Каждый контейнерный класс имеет конструктор по умолчанию, копирующий конструктор и деструктор. Контейнер можно инициализировать элементами из заданного интервала и, по стандарту C++11, с помощью списка инициализации.

Конструктор для списка инициализации обеспечивает удобный способ

задания начальных значений. Это особенно полезно при инициализации константных контейнеров.

```
// инициализация вектора конкретными значениями
const vector<int> v1 = {1, 2, 3, 5, 7, 11, 13, 17, 21};
// то же самое с другим синтаксисом
const vector<int> v2 {1, 2, 3, 5, 7, 11, 13, 17, 21};
```

Использование списка инициализации для контейнеров класса array<> регламентируется отдельными правилами.

Конструктор для заданного интервала обеспечивает возможность инициализации контейнера элементами другого контейнера, массива в стиле языка Си или из потока ввода. Этот конструктор является шаблонным членом, поэтому не только контейнер, но и тип его элементов могут отличаться от данного класса, при условии, что существует автоматическое преобразование из типа элементов контейнера-источника в тип элементов контейнера-приемника. Например, существуют следующие возможности.

– Можно инициализировать контейнер элементами другого контейнера.

```
list<int> l;   //  l - список целых чисел
. . . . .
// копируем в вектор все элементы списка как объекты типа float
vector<float> c(l.begin(), l.end());
```

– По стандарту C++11 можно также переместить элементы, используя итератор перемещения – адаптер итератора, который превращает любое обращение к элементу в операцию перемещения. Обычный итератор преобразуется в итератор перемещения при вызове библиотечной функции make_move_iterator(), которая получает итератор и возвращает итератор перемещения.

```
list< string> l;   //  l - список строк
. . . . .
// перемещаем все элементы списка в вектор
vector< string> c(make_move_iterator(l.begin()), make_move_iterator(l.end()));
```

– Можно инициализировать контейнер элементами обычного массива в стиле языка Си.

```
int carray[ ] = {2, 3, 17, 33, 45, 77};
. . . . .
// копируем во множество все элементы массива в стиле языка Си
set<int> c(begin(carray), end(carray));
```

– Начиная со стандарта C++11 итераторы std::begin() и std::end() для массивов в стиле языка Си определены в заголовочном файле <iterator>.

– Можно инициализировать контейнер из потока ввода.

```
// считываем все целочисленные элементы в дек из потока ввода
deque<int> c{istream_iterator<int>(cin), istream_iterator<int>()};
```

В принципе эти приемы позволяют присваивать и вставлять элементы из другого интервала. Однако для этих операций точный интерфейс либо отлича-

ется наличием дополнительных аргументов, либо не предусматривается для всех контейнерных классов.

В заключение отметим, что стандарт C++11 позволяет использовать конструктор перемещения для инициализации контейнера.

```
vector<int> v1;
. . . . .
// для замены копирования перемещением используем
// библиотечную функцию std::move, возвращающую rvalue
vector<int> v2 = move(v1);
```

В результате контейнер v2 содержит элементы контейнера v1, а содержимое контейнера v1 становится неопределенным. Этот конструктор значительно повышает быстродействие программы, поскольку элементы перемещаются с помощью перестановки указателей, а не поэлементного копирования. Таким образом, если скопированный контейнер больше не нужен, следует использовать конструктор перемещения.

Присваивание и функция swap().

Во время присваивания контейнеров все элементы контейнера-источника копируются в контейнер-приемник, а старые элементы контейнера-приемника удаляются из него. Таким образом, присваивание контейнеров является относительно затратным.

По стандарту C++11 вместо этого можно использовать семантику перемещающего присваивания. Все контейнеры реализуют операцию перемещающего присваивания (причем класс array<> делает это неявно). При этом они просто переставляют указатели в памяти, а не копируют все значения. Точное поведение в этой ситуации не определено, но гарантированная константная сложность этой операции приводит к реализации, описанной ниже. Стандартная библиотека C++ просто обеспечивает, что после перемещающего присваивания контейнер, стоящий в левой части оператора присваивания, содержит элементы контейнера, стоящего в правой части этого оператора. После этого содержимое контейнера, стоящего в правой части операции, становится неопределенным.

Если после присваивания содержимое контейнера, стоящего в правой части оператора, больше не используется, то для повышения быстродействия следует применять перемещающее присваивание.

Все контейнеры имеют функцию-член swap() для обмена содержимого двух контейнеров. Фактически эта операция сводится к простой перестановке внутренних указателей, ссылающихся на данные (элементы, распределители, критерий сортировки). Таким образом, функция swap() гарантированно обеспечивает константную, а не линейную сложность копирующего присваивания. Итераторы и ссылки на элементы контейнера следуют за переставляемыми элементами. Следовательно, после выполнения функции swap() итераторы и ссылки по-прежнему будут ссылаться на элементы, на которые они ссылались

раньше, однако в другом контейнере.

Отметим, что с контейнерами типа array<> функция swap() работает несколько иначе. Поскольку мы не можем просто переставить внутренние указатели, функция swap() имеет линейную сложность, а после ее выполнения итераторы и ссылки ссылаются на тот же контейнер, но на другие элементы.

Операции над размерами.

Почти для всех контейнерных классов предусмотрены три операции над размерами.

1. Функция empty() возвращает признак того, что количество элементов равно нулю (begin() = = end()). Эту функцию целесообразно применять вместо оператора size() = = 0, потому что она реализована более эффективно, чем функция size(), однако, функция size() не работает с односвязными списками.

2. Функция size() возвращает текущее количество элементов контейнера. Эта операция не поддерживается классом forward_list<>, потому что в этом случае она не имела бы константной сложности.

3. Функция max_size() возвращает максимальное количество элементов, которое может содержать контейнер. Это значение зависит от реализации. Например, вектор обычно содержит все элементы в отдельном блоке памяти, поэтому могут возникнуть ограничения, связанные с организацией памяти в компьютере. Если таких ограничений нет, функция max_size() обычно возвращает максимальное значение для типа индекса.

Сравнения.

Во всех контейнерах, за исключением неупорядоченных, определены обычные операторы сравнения = =, ! =, <, < =, > и > =, подчиняющиеся трем правилам.

1. Оба контейнера должны быть однотипными.
2. Два контейнера считаются равными, если их элементы равны и располагаются в одинаковом порядке. Для проверки равенства элементов используется оператор = =.
3. Для проверки того, какой из двух контейнеров меньше используется лексикографическое сравнение.

Лексикографическое сравнение означает, что последовательности сравниваются поэлементно, пока не произойдет одно из следующих событий:

– если два элемента не равны, результат их сравнения является результатом общего сравнения;

– если одна последовательность больше не содержит элементов, то исчерпанная последовательность считается меньше другой;

– если обе последовательности больше не содержат элементов, то они считаются равными.

Для неупорядоченных контейнеров определены только операции = = и ! =. Первая возвращает значение true, если каждому элементу в одном контей-

нере соответствует равный ему элемент в другом контейнере. В этом случае порядок значения не имеет.

Поскольку операции <, < =, > и > = не определены для неупорядоченных контейнеров, общим требованием к контейнерам является предоставление только операторов = = и ! =.

Для сравнения контейнеров разных типов следует использовать специальные алгоритмы сравнения.

Доступ к элементам.

Все контейнеры реализуют интерфейс итераторов, т.е. поддерживают диапазонный цикл for. Таким образом, в соответствии со стандартом C++11 проще всего получить доступ ко всем элементам следующим образом:

for (const auto& elem : coll) cout << elem << endl;

Для того чтобы иметь возможность манипулировать элементами, следует пропустить ключевое слово const:

for (auto& elem : coll) elem = . . . ;

Для того чтобы оперировать позициями (например, иметь возможность вставлять, удалять и перемещать элементы), всегда можно использовать итераторы, возвращаемые функциями cbegin() и cend() и предназначенные только для чтения:

for (auto pos=coll.cbegin(); pos!=coll.cend(); ++pos) cout << *pos << endl;

и итераторы, возвращаемые функциями begin() и end(), и предназначенные для чтения и для записи:

for (auto pos=coll.begin(); pos!=coll.end(); ++pos) *pos = . . . ;

До появления стандарта C++11 программист был обязан, а теперь просто может явно объявить тип итератора только для чтения:

colltype::const_iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) . . .

или для записи:

colltype::iterator pos;
for (pos=coll.begin(); pos!=coll.end(); ++pos) . . .

Все контейнеры, за исключением векторов и деков, гарантируют, что итераторы и ссылки на элементы остаются корректными после удаления любых остальных элементов. Для векторов корректными остаются только элементы, расположенные до точки удаления.

После удаления всех элементов с помощью функции clear() в векторах, деках и строках итератор, возвращенный функциями end() и cend(), может стать некорректным.

После вставки элементов только списки, односвязные списки и ассоциативные контейнеры гарантируют, что итераторы и ссылки на элементы останутся корректными. Для векторов эта гарантия имеет место, только если вставки не превышают емкость контейнера. Для неупорядоченных контейнеров эта гарантия распространяется в основном на ссылки, а для итераторов она дей-

ствует, только если не выполнялось повторное хеширование.

6.4. Типы контейнера

Все контейнеры предоставляют определения общих типов, перечисленных в таблице.

Тип	Требование	Действие
size_type	Да	Целочисленный тип без знака для значений размера
difference_type	Да	Целочисленный тип со знаком для разности между значениями
value_type	Да	Тип элементов
reference	Да	Тип ссылок на элементы
const_reference	Да	Тип константных ссылок на элементы
iterator	Да	Тип итераторов
const_iterator	Да	Тип итераторов только для чтения
pointer		Тип указателей на элементы
const_pointer		Тип указателей на элементы, предназначенные только для чтения

7. Функциональные объекты STL и лямбда-выражения

7.1. Концепция функциональных объектов

Функциональным объектом (функтором) будем назвать объекты классов, содержащих операцию operator(). Класс, в котором определена такая операция, называется функциональным.

Отметим, что в соответствии со стандартом C++11 функциональным объектом считается любой объект, который может быть использован в качестве вызова функции. Таким образом, указатели на функции, объекты классов, содержащие операцию operator() или преобразование в указатель на функцию, а также лямбда-выражения являются функциональными объектами. Однако мы будем называть их «вызываемыми объектами», оставив термин «функциональный объект» за объектами класса, содержащего операцию operator().

Функциональное поведение – это нечто, что можно вызывать с помощью пары скобок и передачи аргументов.

```
function(argl, arg2); // вызов функции
```

Если требуется, чтобы объекты вели себя подобным образом, необходимо сделать возможным вызов этих объектов с помощью пары скобок и передачи аргументов. Для этого достаточно объявить операцию operator() с соответствующими типами параметров.

```
class X { public:
// определение операции "вызов функции":
return-value operator()(argument1, argument2) const;
. . . . .
};
```

Теперь объекты этого класса можно использовать как вызов функции

```
X fo;
. . . . .
fo(argl, arg2);  // вызов operator()  из объекта-функции fo
```

Этот вызов эквивалентен конструкции.

```
fo.operator()(argl,arg2);   // вызов operator() из функционального объекта fo
```

От функционального класса не требуется наличия других полей и методов:

```
class if_greater{
public:
        int operator()(int a, int b) const {
        return a > b;
        }
};
. . . . .
if_greater x;
cout << x(1, 5) << endl; // результат: 0
```

```
cout << if_greater()(5, 1) << endl; // результат: 1
```

Во втором операторе вывода выражение if_greater() используется для вызова конструктора по умолчанию класса if_greater. Результатом выполнения этого выражения является объект класса if_greater. Далее, как и в предыдущем случае, для этого объекта вызывается функция с двумя аргументами, записанными в круглых скобках.

Операцию () можно определять только как метод класса. Можно определить перегруженные операции вызова функции с различным количеством аргументов.

Функциональный объект можно рассматривать как обычную функцию, записанную необычным способом.

Несмотря на свою сложность, это определение имеет три важных преимущества.

1. Функциональный объект может быть более интеллектуальным, обладать большими возможностями, потому что у него есть состояние (данные, определенные в классе и влияющие на поведение функционального объекта). Фактически вы можете иметь два экземпляра одного и того же класса функционального объекта, которые могут одновременно иметь разные состояния. Обычные функции такими возможностями не обладают.

2. Каждый функциональный объект имеет свой тип. Следовательно, его тип можно передавать как шаблонный параметр, чтобы описать определенное поведение, и вы получите отличающиеся друг от друга контейнерные типы с разными функциональными объектами.

3. Функциональный объект обычно работает быстрее, чем указатель на функцию.

Рассмотрим один пример, в котором для контейнера требуется задать нестандартный критерий сортировки.

Пример. Функциональный объект как критерий сортировки.

Программистам часто нужны упорядоченные коллекции элементов некоторого класса (например, коллекция объектов класса Person). Однако использовать для сортировки этих объектов оператор < иногда нежелательно, а иногда просто невозможно. Вместо этого желательно упорядочить эти объекты в соответствии с критерием сортировки, заданным некоторой функцией-членом. В этой ситуации может помочь функциональный объект. Рассмотрим следующий пример:

```
class Person  {
public:
        string firstname() const;
        string lastname() const;
. . . . .
};
// класс функционального объекта
// операция () возвращает результат проверки того, что первый объект класса
```

```
// Person меньше второго объекта
class PersonSortCriterion {
public:
        bool operator()(const Person& p1, const Person& p2) const {
                return p1.lastname() <p2.lastname() ||
                        (p1.lastname() = = p2.lastname()   &&
                         p1.firstname() < p2 .firstname());
        }
};
int main()
{
// создаем множество со специальным критерием сортировки
set<Person, PersonSortCriterion> coll; // первый параметр шаблона - тип
                                       // элементов, второй параметр - критерий сортировки
. . . . .
// что-то делаем с его элементами
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
. . . . .
}
. . . . .
}
```

Напомним, что в ассоциативных контейнерах, в том числе во множествах (set), элементы хранятся в упорядоченном виде, критерий сортировки при этом может быть задан вторым шаблонным параметром.

Множество coll использует специальный критерий сортировки PersonSortCriterion, определенный как класс функциональных объектов. В классе PersonSortCriterion операция operator() определена так, что она сравнивает два объекта класса Person по их фамилиям, а если они совпадают, по их именам. Конструктор множества coll автоматически создает экземпляр класса PersonSortCriterion, так что элементы упорядочиваются именно по этому критерию сортировки.

Отметим, что критерий сортировки PersonSortCriterion является типом. Следовательно, его можно использовать как шаблонный параметр множества. Это было бы невозможно, если бы мы реализовали критерий сортировки с помощью обычной функции.

Все множества с таким критерием сортировки имеют свой собственный тип. Их нельзя комбинировать или присваивать множествам, имеющим "обычный" или другой пользовательский критерий сортировки. Однако, можно определить функциональные объекты, задающие разные критерии сортировки с одним и тем же типом (см. пример в п. 8.9).

7.2. Стандартные функциональные объекты и функциональные адаптеры

Стандартная библиотека C++ содержит много стандартных функциональных объектов и адаптеров, позволяющих создавать более сложные функциональные объекты.

Эта возможность называется функциональной композицией. Она основана на фундаментальных функциональных объектах и адаптерах, описанных в этом пункте. Для использования этих функциональных объектов и адаптеров в программу следует включить заголовочный файл <functional>.

7.2.1. Стандартные функциональные объекты

Все предопределенные функциональные объекты перечислены в таблице.

Выражение	Результат
negate<type>()	– param
plus<type>()	param1 + param2
minus<type>()	param1 – param2
multiplies<type>()	param1 * param2
divides<type>()	param1 / param2
modulus<type>()	param1 % param2
equal to<type>()	param1 = = param2
not_equal to<type>()	parami1 ! = param2
less<type>()	param1 < param2
greater<type>()	param1 > param2
less_equal<type>()	param1 < = param2
greater_equal<type>()	par ami > = param2
logical_not<type>()	! param
logical_and<type>()	param1 && param2
logical_or<type>()	param1 | | param2
bit_and<type>()	param1 & param2
bit_or<type>()	paraml | param2
bit_xor<type>()	param1 ^ param2

Функциональный объект less<> – это критерий, заданный по умолчанию и предназначенный для упорядочения и сравнения объектов с помощью функций сортировки и ассоциативных контейнеров. Таким образом, операции сортировки, заданные по умолчанию, всегда создают возрастающий порядок (ele-

ment < nextElement). Функциональный объект equal_to<> – это критерий эквивалентности по умолчанию для неупорядоченных контейнеров.

7.2.2. Функциональные адаптеры

Функциональный адаптер – это функциональный объект, предназначенный для создания композиций, состоящих из функциональных объектов, определенных значений и специальных функций.

В таблице перечислены основные функциональные адаптеры, предусмотренные в стандартной библиотеке C++.

Выражение	Результат
bind(op , args...)	Связывает список аргументов args с операцией op
mem_fn(op)	Вызывает операцию op() как функцию-член объекта или объекта, на который ссылается указатель
not_fn(op)	Возвращает предикат, являющийся отрицаниием предиката op

Связыватель (привязка) bind().

Наиболее важным адаптером является bind(), он позволяет делать следующее:

– адаптировать и создавать новые функциональные объекты из существующих или стандартных функциональных объектов;

– вызывать глобальные функции;

– вызывать функции-члены объектов, указателей на объекты и интеллектуальных указателей на объекты (рассматриваться не будут).

Функция bind() получает вызываемый объект и создает новый вызываемый объект, который адаптирует список параметров исходного объекта.

Общая форма вызова функции bind() такова:

```
auto  Новый_Вызываемый_Объект =  bind(Вызываемый_Объект,
                                      список_аргументов);
```

Здесь Новый_Вызываемый_Объект – это новый вызываемый объект, а список_аргументов – разделяемый запятыми список аргументов, соответствующих параметрам переданного вызываемого объекта Вызываемый_Объект. Таким образом, когда происходит вызов объекта Новый_Вызываемый_Объект, он вызывает Вызываемый_Объект, передавая аргументы из списка список_аргументов.

Аргументы из списка список_аргументов могут включать имена в формате _n, где n – целое число. Эти аргументы – *заполнители*, представляющие параметры объекта Новый_Вызываемый_Объект. Они располагаются вместо аргументов, которые будут переданы объекту Новый_Вызываемый_Объект. Число n является позицией параметра вновь созданного вызываемого объекта: _1 – первый параметр, _2 – второй и т.д. Заполнители _1, _2, ... определены в про-

странстве имен std::placeholders. Таким образом, чтобы не использовать всегда спецификацию
placeholders::_n
целесообразно в программы включать строку
using namespace std::placeholders; // для использования заполнителей

Пример. Предположим, что f() – вызываемый объект с пятью параметрами, g – вызываемый объект, получающий два аргумента:
auto g = bind(f, a, b, _2, c, _1);

Вызов функции bind() создает новый вызываемый объект, получающий два аргумента, представленные заполнителями _1 и _2. Новый вызываемый объект передает собственные аргументы как пятый и третий аргументы вызываемому объекту f(). Первый, второй и четвертый аргументы вызываемого объекта f() связаны с переданными значениями a, b и c соответственно.

Таким образом, использование этого адаптера bind() преобразует вызов g(_1, _2) в вызов f(a, b, _2, c, _1). Например, вызов g (X, Y) приводит к вызову f(a, b, Y, c, X).

Как правило, связыватели используются для указания параметров при использовании стандартных функциональных объектов, предусмотренных в стандартной библиотеке C++.

Пример.

```
int main()
{
auto plus10 = bind(plus<int>(), _1, 10); // param1 + 10
cout << "+10: " << plus10(7) << endl;

auto plus10times2 = bind(multiplies<int>(),
                         bind(plus<int>(), _1, 10),
                         2);
cout << "+10 *2: " << plus10times2(7) << endl;

auto pow3 = bind(multiplies<int>(),
                 bind(multiplies<int>(), _1, _1),
                 _1);
cout << "x*x*x:  " << pow3(7) << endl;

auto inversDivide = bind(divides<double>(), _2, _1);
cout << "invdiv: " << inversDivide(49,7) << endl;
}
```

Здесь определены четыре разных связывателя, представляющие собой функциональные объекты. Например, связыватель plus10, определенный как
bind(std::plus<int>(), _1, 10);
представляет собой функциональный объект, который автоматически вызывает функциональный объект plus<> (т.е. операцию +), передавая ему заполнитель

_1 в качестве первого параметра/операнда и 10 в качестве второго параметра/операнда. Следовательно, для любого аргумента, передаваемого в это выражение, данный функциональный объект будет возвращать значение этого аргумента, увеличенное на 10.

Связыватель также можно вызывать непосредственно. Например, выражение

```
cout << bind(plus<int>(), _1, 10)(32) << endl;
```

запишет 42 в стандартный поток вывода. Если передать этот функциональный объект алгоритму, алгоритм может применить его к каждому элементу, с которым он работает:

```
// добавляем 10 к каждому элементу
transform (coll.begin(), coll.end(), //   источник
coll.begin(),                        //   получатель
bind(plus<int>(), _1, 10)); //   операция
```

Алгоритм стандартной библиотеки transform выполняет заданную операцию над каждым элементом заданной коллекции.

Аналогично можно определить связыватель, представляющий критерий сортировки.

Например, для поиска первого элемента, превышающего 42, можно связать функциональный объект greater<>, переданный в качестве первого аргумента, с числом 42, указанным в качестве второго аргумента.

```
// ищем первый элемент > 42
auto pos =  find_if (coll.begin(), coll.end(),      //   диапазон
                     bind(greater<int>(),_1, 42)) // предикат
```

Алгоритм стандартной библиотеки find_if выполняет поиск значения, соответствующего заданному предикату.

Предикаты – это функции или функциональные объекты, возвращающие булево значение (значение, допускающее преобразование в тип bool). Однако, предикаты, как правило, не должны изменять свое состояние и модифицировать передаваемые аргументы при вызове функции.

Отметим, что для используемого стандартного функционального объекта всегда необходимо задавать тип аргумента. Если типы не совпадают, выполняется преобразование или возникает ошибка компиляции.

Остальные инструкции в этой программе показывают, что связыватели могут быть вложенными и создавать еще более сложные функциональные объекты. Например, следующее выражение определяет функциональный объект, добавляющий число 10 к передаваемому аргументу, а затем умножающий его на 2:

```
bind(  multiplies<int>(), // (param1+10)*2
       bind(plus<int>(), _1, 10),
       2);
```

Как видим, выражения вычисляются в направлении изнутри наружу. Аналогично можно возвести значение в куб, объединив два объекта multi-

plies<> с тремя заполнителями передаваемого аргумента.

```
bind( multiplies<int>(),   //   (param1*param1)*param1
        bind(multiplies<int>(), _1, _1)
        _1);
```

Последнее выражение определяет функциональный объект, в котором аргументы для деления переставлены местами. Таким образом, он делит второй аргумент на первый.

```
bind(divides<double>(), _2, _1); // param2/param1
```

Итак, программа выводит на экран следующие результаты:

```
+10: 17
+10 *2:   34
x*x*x: 343
invdiv: 0.142857
```

Пример. Следующая программа демонстрирует, как связыватель bind() позволяет вызвать произвольную функцию-член класса.

```
class Person  {
private:
        string name;
public:
        Person (const string& n) : name(n) { }
        void print() const {cout << name << endl;}
        void print2(const string& prefix) const {
                cout << prefix << name << endl;
        }
. . . . .
};
int main()
{
vector<Person> coll = {Person("Tick"), Person("Trick"), Person("Track")};
// вызываем функцию-член print() каждого объекта класса Person
// алгоритм for_each стандартной библиотеки вызывает для каждого элемента
// последовательности заданную функцию (не обязательно являющуюся членом
// класса)
for_each (coll.begin(), coll.end(), // диапазон
            bind(&Person::print, _1)) ; // функция
cout << endl;
// вызываем функцию-член print2() каждого объекта класса Person
// с дополнительным аргументом
for_each (coll.begin(), coll.end(),
            bind(&Person::print2, _1, "Person: "));
cout << endl;
// вызываем функцию-член print2() временного объекта класса Person
```

```
bind(&Person::print2, _1, "This is: ")(Person("nico"));
}
```

Здесь связыватель

```
bind(&Person::print,_1);
```

определяет функциональный объект, вызывающий функцию-член param1.print() переданного объекта класса Person. Иначе говоря, поскольку первый аргумент является функцией-членом, второй аргумент определяет объект, для которого эта функция вызывается. Этой функции-члену передаются все дополнительные аргументы. Это значит, что связыватель

```
bind(&Person::print2, _1, "Person: ")
```

определяет функциональный объект, вызывающий функцию-член param1.print2 ("Person: ") для любого переданного объекта класса Person.

Здесь передаваемые объекты являются членами вектора coll, но в принципе объекты можно передавать непосредственно. Например:

```
Person n("nico");
bind(&Person::print2, _l, "This is: ")(n);
```

вызывает

```
n.print2("This  is: ").
```

Результаты работы этой программы имеют следующий вид:

```
Tick
Trick
Track
Person: Tick
Person: Trick
Person: Track
This is: nico
```

Отметим, что с помощью связывателя можно также вызывать модифицирующие функции-члены.

```
class Person {
public:
. . . . .
        void setName (const string& n) {name = n;}
};
vector<Person> coll;
. . . . .
for_each (coll.begin(), coll.end(), // присваиваем всем одинаковые имена
            bind(&Person::setName, _1, "Paul"));
```

Возможен также вызов виртуальных функций-членов. Если связывается метод базового класса, а объект принадлежит производному классу, то будет вызвана правильная виртуальная функция производного класса.

Адаптер mem_fn().

Для вызова функций-членов можно также использовать адаптер mem_fn(), в котором можно не указывать заполнитель для объекта, для которого вызывается функция-член:

```
mem_fn(&Person::print)(n); // вызывает n.print()
mem_fn(&Person::print2)(n,"Person: "); // вызывает n.print2("Person: ")
```

Таким образом, можно вызывать

```
for_each (coll.begin(), coll.end(), mem_fn(&Person::print));
```

Однако для связывания дополнительного аргумента с функциональным объектом снова необходимо использовать связыватель bind():

```
for_each (coll.begin(), coll.end(),
            bind(mem_fn(&Person::print2), _1, "Person: "));
```

Связывание с данными-членами класса.

Привязку можно установить и к данным-членам. Рассмотрим следующий пример:

```
map<string,int> coll; // отображение целых чисел, ассоциированных со строками
. . . . .
// накапливаем все значения (вторые члены элементов)
// вызывается алгоритм accumulate с 4 параметрами, производящий над третьим
// параметром и очередным элементом последовательности заданную операцию
// и сохраняющую результат в третьем параметре
int sum = accumulate (    coll.begin(),   coll.end(), // диапазон
                          0,  // третий параметр
                          bind(plus<int>(), _1, // заданная операция
                          bind(&map<string,int>::value_type::second, _2))
);
```

Здесь вызывается алгоритм accumulate() для суммирования всех значений всех элементов. Однако, поскольку мы используем отображение, в котором элементами являются пары "ключ-значение", для доступа к значению элемента мы с помощью связывателя

```
bind(&map<string,int>::value_type::second,_2)
```

связываем второй аргумент каждого вызова операции с его членом second.

Пользовательские функциональные объекты для функциональных адаптеров.

Связыватели можно использовать и с пользовательскими функциональными объектами. Следующий пример демонстрирует полное описание функционального объекта, возводящего первый аргумент в степень, равную значению второго аргумента.

```
template <typename T1, typename T2>
struct fopow
{
```

```
    T1 operator() (T1 base, T2 exp) const { return pow(base,exp);}
};
```

Отметим, что первый аргумент и возвращаемое значение имеют одинаковый тип T1, а показатель степени имеет другой тип, T2. Заметим, что функция pow перегружена не для всех сочетаний T1 и T2.

Следующая программа демонстрирует применение пользовательского функционального объекта fopow<>().

```
int main()
{
vector<int> coll = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// выводим на экран число 3, возведенное в степени,
// равные значениям всех элементов
transform (coll.begin(), coll.end(),    //   источник
ostream_iterator<float>(cout," "),     //   получатель
bind(fopow<float,int>(), 3, _1));        //   операция
cout << endl;
// выводим на экран все элементы, возведенные в степень 3
transform (coll.begin(), coll.end(),                    // источник
ostream_iterator<float>(cout," "),  // получатель
bind(fopow<float,int>(), _1, 3));     // операция
cout << endl;
}
```

Программа выводит на экран следующие числа:

```
3 9 27 81 243 729 2187 6561 19683
1 8 27 64 125 216 343 512 729
```

Универсальный инвертор not_fn.

Универсальный инвертор **not_fn** появился в стандарте C++17. Он принимает предикат и возвращает новый предикат. Когда возвращенный предикат вызывается с некоторыми аргументами, он передает их в исходный предикат и логически инвертирует результат.

```
bool less_10(int x){
            return x < 10;
}
int main(){
vector<int> coll = {1, 3, 7, 77, 10, 6};
auto n = count_if(coll.begin(), coll.end(),     //   диапазон
                  not_fn(less_10));   // предикат
cout << n << endl;
}
```

Алгоритм стандартной библиотеки count_if подсчитывает количество объектов в коллекции, соответствующих заданному предикату. Таким образом, программа выводит на экран число 2.

7.3. Лямбда-выражения

Лямбда-выражение (или лямбда-функция) является видом вызываемого объекта.

Лямбда-выражения появились в стандарте C++11. Они представляют собой мощный и очень удобный механизм локальной функциональности, особенно полезный для уточнения деталей алгоритмов и функций-членов контейнеров. Если алгоритму требуется передать индивидуальное поведение, достаточно описать его как любую другую функцию именно в том месте, где оно требуется.

Лямбда-выражения часто демонстрируют преимущество описания локальной функциональности по сравнению с ранее использовавшимися для этого функциональными объектами и адаптерами, такими как bind(). Тем не менее, лямбда-выражения не всегда являются предпочитительными по сравнению с ними.

Форма лямбда-выражений.

Лямбда-выражение представляет собой вызываемый блок кода. Его можно считать безымянной встраиваемой функцией. Подобно любой функции, у лямбда-выражений есть тип возвращаемого значения, список параметров и тело функции. В отличие от функции, лямбда-выражения могут быть определены внутри функции. Форма лямбда-выражений такова:

[список захвата] (список параметров)

-> тип возвращаемого значения {тело функции}

Здесь список захвата (зачастую пустой) – это список локальных переменных, определенных в функции, содержащей лямбда-выражение; тип возвращаемого значения, список параметров и тело функции – те же самые, что и у любой обычной функции.

Для указания типа возвращаемого значения используется новая синтаксическая конструкция языка C++, которая применяется и для обычных функций. Но обычно тип возвращаемого значения указывать приходится редко.

Список параметров и тип возвращаемого значения могут отсутствовать, но список захвата и тело функции должны быть всегда:

```
auto f = [ ]{ return 42; };
```

Здесь f определено как вызываемый объект, не получающий никаких аргументов и возвращающий значение 42. Вызов лямбда-выражений происходит таким же способом, что и вызов функций, – при помощи оператора вызова:

```
cout << f() << endl;   //   выводит   42
```

Пропуск круглых скобок и списка параметров в лямбда-выражении эквивалентен определению пустого списка параметров. Следовательно, когда происходит вызов лямбда-выражения f, список аргументов оказывается пустым. Если пропущен тип возвращаемого значения, то выведенный тип возвращаемого значения лямбда-выражения будет зависеть от кода в теле функции.

Передача аргументов лямбда-выражению.

Подобно вызовам обычных функций, аргументы вызова лямбда-выражения используются для инициализации его параметров. Как обычно, типы аргумента и параметра должны совпадать. В отличие от обычных функций, у лямбда-выражений не может быть аргументов по умолчанию. Поэтому у вызова лямбда-выражения всегда столько аргументов, сколько и параметров. Как только параметры инициализируются, выполняется тело лямбда-выражения.

Пример. Для примера передачи аргументов рассмотрим лямбда-выражение, предназначенное для использования при сортировке строк:

```
[ ] (const string &a, const string &b){return a.size() < b.size();}
```

Пустой список захвата означает, что это лямбда-выражение не будет использовать локальных переменных из окружающей функции. Параметры лямбда-выражения будут ссылками на константные строки. Тело лямбда-выражения сравнивает размеры строк и возвращает логическое значение, зависящее от соотношения размеров.

Используем это лямбда-выражение для вызова библиотечного алгоритма сортировки коллекции строк:

```
sort(coll.begin(), coll.end(), // диапазон
                [ ](const string &a, const string &b) // критерий сортировки
                                {return a.size() < b.size();}
);
```

Когда функция sort() будет сравнивать два элемента, она вызовет данное лямбда-выражение.

7.3.1. Захват значений лямбда-выражениями

Использование списка захвата.

Когда лямбда-выражение определено в функции, оно способно использовать локальные переменные этой функции, заранее определив, какие из них предстоит использовать. Лямбда-выражение определяет подлежащие использованию локальные переменные, включив их в список захвата. Список захвата предписывает лямбда-выражению включить информацию, необходимую для доступа к этим переменным, в само лямбда-выражение. Лямбда-выражение может использовать локальную переменную окружающей функции, только если она присутствует в ее списке захвата.

Список захвата определяется в начинающих лямбда-выражение квадратных скобках [ ], где располагается разделяемый запятыми список имен, определенных в окружающей функции:

```
int sz=10;
[sz](const string &a){return a.size() >= sz;};
```

Поскольку данное лямбда-выражение захватывает переменную sz, ее можно использовать в теле лямбда-выражения. Тело лямбда-выражения сравнивает размер переданной строки с захваченным значением переменной sz.

Используя это лямбда-выражение, можно найти первый элемент коллекции строк, размер которого не меньше sz:

```
// получить итератор на первый элемент, размер которого >= sz
int sz=10;
auto pos = find_if(coll.begin(), coll.end(),
                   [sz](const string &a){return a.size() >= sz;});
```

Вызов функции find_if () возвращает итератор на первый элемент, длина которого не меньше sz, или на элемент coll.end(), если такового элемента не существует.

Список захвата используется только для локальных нестатических переменных; лямбда-выражения могут непосредственно использовать статические локальные переменные и переменные, объявленные вне функции.

Например, можно вывести элементы нашей коллекции в стандартный поток вывода cout.

```
for_each(coll.begin(), coll.end(), [ ](const string &s){cout << s << " ";});
cout << endl;
```

В данном случае имя cout является не локальным, а определено в заголовке iostream. Пока заголовок iostream находится в области видимости функции, в которой определено данное лямбда-выражение, оно может использовать имя cout.

При определении лямбда-выражения компилятор создает новый (безымянный) класс, соответствующий этому лямбда-выражению. Создание этих классов рассматривается ниже, а пока следует понять, что при передаче лямбда-выражения функции определяется новый тип и создается его объект. Безымянный объект этого созданного компилятором типа и передается как аргумент. Аналогично при использовании ключевого слова auto для определения переменной, инициализированной лямбда-выражением, определяется объект типа, созданного из этого лямбда-выражения.

По умолчанию созданный из лямбда-выражения класс содержит переменные-члены, соответствующие захваченным переменным лямбда-выражения. Подобно переменным-членам любого класса, переменные-члены лямбда-выражения инициализируются при создании его объекта.

Подобно передаче параметров, переменные можно захватывать по значению или по ссылке. В таблице приведены различные способы создания списка захвата.

Выражение	Результат
[ ]	Пустой список захвата. Лямбда-выражение не может использовать локальные переменные из содержащей функции.
[names]	names – разделяемый запятыми список имен, локальных для содержащей функции. По умолчанию переменные в списке захвата копируются. Имя, которому предшествует знак &, захватывается по ссылке

[&]	Неявный захват по ссылке. Сущности из содержащей функции используются в теле лямбда-выражения по ссылке
[=]	Неявный захват по значению. Сущности из содержащей функции используются в теле лямбда-выражения как копии
[&, identifier_list]	identifier_list – разделяемый запятыми список любого количества переменных из содержащей функции. Эти переменные захватываются по значению; любые неявно захваченные переменные захватывается по ссылке. Именам в списке identifier_list не могут предшествовать символы &
[=, reference_list]	Переменные, включенные в список reference_list, захватываются по ссылке; любые неявно захваченные переменные захватывается по значению. Имена в списке reference_list не могут включать указатель this и должны предваряться символом &

Захват по значению.

До сих пор у использованных лямбда-выражений захват переменных осуществлялся по значению. Подобно передаче по значению параметров, захват переменной по значению подразумевает ее копирование. Но, в отличие от параметров, копирование значения при захвате осуществляется при создании лямбда-выражения, а не при его вызове.

```
void fcn1() {
int v1 = 42; // локальная переменная
// копирует v1 в вызываемый объект f
auto f = [v1]{ return v1;};
v1 = 0;
auto j = f() ; // j = 42; f получит копию v1 на момент создания
}
```

Поскольку значение копируется при создании лямбда-выражения, последующие изменения захваченной переменной никак не влияют на соответствующее значение в лямбда-выражении.

Захват по ссылке.

Можно также определять лямбда-выражения, захватывающие переменные по ссылке. Например:

```
void fcn2 () {
int v1 = 42; // локальная переменная
// объект f2 содержит ссылку на v1
auto f2 = [&v1]{return v1;};
v1 = 0;
auto j = f2(); // j = 0
}
```

Символ & перед v1 указывает, что переменная v1 должна быть захвачена как ссылка. Захваченная по ссылке переменная действует так же, как любая другая ссылка. При использовании переменной в теле лямбда-выражения фактически применяется объект, с которым связана эта ссылка. В данном случае, когда лямбда-выражение возвращает v1, возвращается значение объекта, на который ссылается переменная v1.

Захват ссылок имеет те же проблемы и ограничения, что и возвращение ссылок. При захвате переменной по ссылке следует быть уверенным, что объект, на который она ссылается, существует на момент выполнения лямбда-выражения. Переменные, захваченные лямбда-выражением, являются локальными, они перестают существовать сразу, как только функция завершится.

Неявный захват.

Вместо предоставления явного списка переменных содержащей функции, которые предстоит использовать, можно позволить компилятору самостоятельно вывести используемые переменные из кода тела лямбда-выражения. Чтобы заставить компилятор самостоятельно вывести список захвата, в нем используется символ & или =. Символ & указывает, что предполагается захват по ссылке, а символ = – что значения захватываются по значению. Например, передаваемое функции find_if () лямбда-выражение можно переписать так:

```
// sz неявно захватывается по значению
int sz=10;
auto pos = find_if(coll.begin(), coll.end(),
                   [=](const string &a){return a.size() >= sz;});
```

Если одни переменные необходимо захватить по значению, а другие по ссылке, вполне можно совместить явный и неявный захваты.

При совмещении неявного и явного захвата первым элементом в списке захвата должен быть символ & или =. Эти символы задают режим захвата по умолчанию: по ссылке или по значению соответственно.

Изменяемые лямбда-выражения.

По умолчанию лямбда-выражение не может изменить значение переменной, которую она копирует по значению. Чтобы изменить значение захваченной переменной, за списком параметров должно следовать ключевое слово mutable. Изменяемые лямбда-выражения не могут пропускать список параметров:

```
void fcn3 () {
int v1 = 42; // локальная переменная
// f может изменить значение захваченных переменных
auto f = [v1]() mutable {return ++v1;};
v1 = 0;
auto j = f(); // j = 43, v1 = 0
}
```

Может ли захваченная по ссылке переменная быть изменена, зависит

только от того, ссылается ли она на константный или неконстантный тип:

```
void   fcn4 () {
int v1 = 42; // локальная переменная
// v1 - ссылка на неконстантную переменную, эту переменную можно изменить
// в f2 при помощи ссылки
auto f2 = [&v1]{return ++v1;};
v1 = 0;
auto j = f2(); // j = v1 = 1
}
```

7.3.2. Лямбда-выражения как функциональные объекты

Написанное лямбда-выражение компилятор преобразует в безымянный объект безымянного класса. Классы, созданные из лямбда-выражения, содержат перегруженный оператор вызова функции. Рассмотрим, например, лямбда-выражение, передававшееся как последний аргумент алгоритма sort():

```
sort(coll.begin(),coll.end(),
        [ ](const string &a, const string &b){return a.size() < b.size();}
);
```

Это действует как безымянный объект класса, который выглядел бы примерно так:

```
class ShorterString {
public:
        bool operator() (const string &s1, const string &s2) const
                                                {return s1.size() < s2.size();}
};
```

У этого класса есть один член, являющийся оператором вызова функции, получающим две строки и сравнивающий их длины. Список параметров и тело функции те же, что и у лямбда-выражения. Как уже упоминалось выше, по умолчанию лямбда-выражения не могут изменять свои захваченные переменные. В результате по умолчанию оператор вызова функции в классе, созданном из лямбда-выражения, является константной функцией-членом. Если лямбда-выражение объявляется как mutable, то оператор вызова не будет константным.
Вызов алгоритма sort() можно переписать так, чтобы использовать этот класс вместо лямбда-выражения:

```
sort(coll.begin(), coll.end(), ShorterString());
```

Третий аргумент – созданный алгоритмом sort бузымянный объект класса ShorterString. Код в алгоритме sort() будет вызывать этот объект каждый раз, когда он сравнивает две строки. При вызове объекта будет выполнено тело его операции operator(), возвращающего значение true, если размер первой строки будет меньше, чем второй.

Классы, представляющие лямбда-выражения с захваченными переменными.

Как уже упоминалось, при захвате лямбда-выражением переменной по ссылке разработчик должен сам гарантировать существование переменной, на которую ссылается ссылка, во время выполнения лямбда-выражения. Поэтому компилятору разрешено использовать ссылку непосредственно, не сохраняя ее как переменную-член в созданном классе.

Переменные, которые захватываются по значению, напротив, копируются в лямбда-выражение. В результате классы, созданные из лямбда-выражений, переменные которых захватываются по значению, имеют переменные-члены, соответствующие каждой такой переменной. У этих классов есть также конструктор для инициализации этих переменных-членов значениями захваченных переменных.

В одном из примеров предыдущего пункта лямбда-выражение использовалось для поиска первой строки, длина которой была больше или равна заданному значению:

```
auto pos = find_if(coll.begin(),coll.end(),
            [sz](const string &a){return a.size() >= sz;});
```

Созданный класс выглядел бы примерно так:

```
class SizeComp{
    int sz; // переменная-член для каждой переменной, захваченной по
            // значению
public:
    SizeComp(int n): sz(n) { } // параметр для каждой захваченной переменной
//оператор вызова с тем же типом возвращаемого значения, параметрами
// и телом, как у лямбда-выражения:
    bool operator() (const string &s) const {return s.size() >= sz;}
};
```

В отличие от класса ShorterString, у этого класса есть переменная-член и конструктор для ее инициализации. У этого синтезируемого класса нет стандартного конструктора; чтобы использовать этот класс, следует передать аргумент:

```
find_if(coll.begin(), coll.end(), SizeComp(sz));
```

У классов, созданных из лямбда-выражения, нет конструктора по умолчанию, оператора присваивания по умолчанию, но есть стандартный деструктор. Будет ли у класса стандартный конструктор копирования/перемещения, зависит обычно от способа и типа захватываемых переменных-членов.

Тип лямбда-выражения.

Иногда необходимо указать тип лямбда-выражения. Например, оно требуется при передаче лямбда-функции в качестве критерия сортировки или хеш-функции для ассоциативных или неупорядоченных контейнеров.

Для этого можно использовать ключевое слово decltype, с помощью ко-

торого можно позволить компилятору самому распознать тип выражения (причем не обязательно лямбда-выражения).

Рассмотрим, например, использование лямбда-функции для определения критерия сортировки для ассоциативных массивов:

```
auto cmp = [ ](const Person& p1, const Person& p2){
                    return p1.lastname() < p2.lastname() ||
                           (p1.lastname() = = p2.lastname() &&
                                  p1.firstname() < p2.firstname());
};
. . . . .
set<Person, decltype(cmp)> coll(cmp);
```

Поскольку в объявлении объекта класса set необходимо определить тип лямбда-функции, приходится использовать операцию decltype, возвращающую тип лямбда-объекта. Отметим, что необходимо также передать лямбда-объект в конструктор coll; в противном случае класс coll будет вынужден вызвать конструктор по умолчанию для передаваемого критерия сортировки, в то время как лямбда-функции не могут иметь конструкторов по умолчанию и операций присваивания. Данный пример показывает, что для критерия сортировки класс, определяющий функциональные объекты, может оказаться более интуитивно понятным, чем лямбда-функция.

В качестве альтернативы для указания общего типа, предназначенного для функционального программирования, можно использовать шаблонный класс std::function<>.

```
function<int(int,int)> returnLambda()
{
    return [ ](int x, int y){return x*y; };
}
int main()
{
auto lf = returnLambda();
cout << lf(6,7) << endl;
}
```

Ниже приведен результат работы этой программы.

```
42
```

В общем случае при использовании кода

```
function <F> f { g }
```

вызов f(args) будет означать g(args), где args – один или несколько аргументов, а F представляет собой тип g.

Поскольку function представляет собой шаблон, можно определять переменные типа function<T> и присваивать этим переменным вызываемые объекты, например:

```
int f1 (double) ;
function<int (double)> fct { f1 };// Инициализация функцией f1
```

```
int x = fct(2.3) ; // Вызов f1(2.3)
function<int (double)> fun ; // fun может хранить любую
fun = fl ; // функцию int(double)
```