






<<Показать меню

 Сообщений 12  Оценка 480   Оценить  +1  -1  

 Орфография

Руководство по стандартной библиотеке шаблонов (STL)

Авторы: Александр Степанов
Менг Ли

Перевод: Алексей Суханов
Андрей Кутырин

Григорий Александрович Милонов

Московский Государственный Институт Радиотехники, Электроники и Автоматики (Технический Университет)

- Введение
- Структура библиотеки
- Требования
- Основные компоненты
 - Операторы (Operators)
 - Пара (Pair)
- Итераторы
 - Итераторы ввода (Input iterators)
 - Итераторы вывода (Output iterators)
 - Последовательные итераторы (Forward iterators)
 - Двунаправленные итераторы (Bidirectional iterators)
 - Итераторы произвольного доступа (Random access iterators)
 - Теги итераторов (Iterator tags)
 - Операции с итераторами (Iterator operations)
- Функциональные объекты
 - Базовые классы (Base)
 - Арифметические операции (Arithmetic operations)
 - Сравнения (Comparisons)
 - Логические операции (Logical operations)
- Распределители
 - Требования распределителей (Allocator requirements)
 - Распределитель по умолчанию (The default allocator)
- Контейнеры
 - Последовательности (Sequences)
 - Ассоциативные контейнеры (Associative containers)
- Итераторы потоков
 - Итератор входного потока (Istream Iterator)
 - Итератор выходного потока (Ostream Iterator)
- Алгоритмы
 - Не меняющие последовательность операции (Non-mutating sequence operations)
 - Меняющие последовательность операции (Mutating sequence operations)
 - Операции сортировки и отношения (Sorting and related operations)
 - Обобщённые численные операции (Generalized numeric operations)
- Адаптеры
 - Адаптеры контейнеров (Container adaptors)
 - Адаптеры итераторов (Iterator adaptors)
 - Адаптеры функций (Function adaptors)
- Примитивы управления памятью (Memory Handling Primitives)
- Примеры программ с шаблонами

ВВЕДЕНИЕ

Стандартная Библиотека Шаблонов предоставляет набор хорошо сконструированных и согласованно работающих вместе обобщённых компонентов C++. Особая забота была проявлена для обеспечения того, чтобы все шаблонные алгоритмы работали не только со структурами данных в библиотеке, но также и с встроенными структурами данных C++. Например, все алгоритмы работают с обычными указателями. Ортогональный проект библиотеки позволяет программистам использовать библиотечные структуры данных со своими собственными алгоритмами, а библиотечные алгоритмы - со своими собственными структурами данных. Хорошо определённые семантические требования и требования сложности гарантируют, что компонент пользователя будет работать с библиотекой и что он будет работать эффективно. Эта гибкость обеспечивает широкую применимость библиотеки.

Другое важное соображение - эффективность. C++ успешен, потому что он объединяет выразительную мощность с эффективностью. Много усилий было потрачено, чтобы проверить, что каждый шаблонный компонент в библиотеке имеет обобщённую реализацию, которая имеет эффективность выполнения с разницей в пределах нескольких процентов от эффективности соответствующей программы ручной кодировки.

Третьим соображением в проекте была разработка библиотечной структуры, которая, будучи естественной и лёгкой для понимания, основана на прочной теоретической основе.

СТРУКТУРА БИБЛИОТЕКИ

Библиотека содержит пять основных видов компонентов:

- алгоритм (*algorithm*): определяет вычислительную процедуру.
- контейнер (*container*): управляет набором объектов в памяти.
- итератор (*iterator*): обеспечивает для алгоритма средство доступа к содержимому контейнера.
- функциональный объект (*function object*): инкапсулирует функцию в объекте для использования другими компонентами.
- адаптер (*adaptor*): адаптирует компонент для обеспечения различного интерфейса.

разделение позволяет нам уменьшить количество компонентов. Например, вместо написания функции поиска элемента для каждого вида контейнера мы обеспечиваем единственную версию, которая работает с каждым из них, пока удовлетворяется основной набор требований.

описание разъясняет структуру библиотеки. Если программные компоненты сведены в таблицу как трёхмерный массив, где одно измерение представляет различные типы данных (например, int, double), второе измерение представляет различные контейнеры (например, вектор, связный список, файл), а третье измерение представляет различные алгоритмы с контейнерами (например, поиск, сортировка, перемещение по кругу), если i , j и k - размеры измерений, тогда должно быть разработано $i * j * k$ различных версий кода. При использовании шаблонных функций, которые берут параметрами типы данных, нам нужно только $j * k$ версий. Далее, если заставим наши алгоритмы работать с различными контейнерами, то нам нужно просто $j + k$ версий. Это значительно упрощает разработку программ, а также позволяет очень гибким способом использовать компоненты в библиотеке вместе с определяемыми пользователем компонентами. Пользователь может легко определить специализированный контейнерный класс и использовать для него библиотечную функцию сортировки. Для сортировки пользователь может выбрать какую-то другую функцию сравнения либо через обычный указатель на сравнивающую функцию, либо через функциональный объект (объект, для которого определён *operator()*), который сравнивает. Если пользователю необходимо выполнить передвижение через контейнер в обратном направлении, то используется адаптер *reverse_iterator*.

Библиотека расширяет основные средства C++ последовательным способом, так что программисту на C/C++ легко начать пользоваться библиотекой. Например, библиотека содержит шаблонную функцию *merge* (слияние). Когда пользователю нужно два массива a и b объединить в c , то это может быть выполнено так:

```
int a[1000];
int b[2000];
int c[3000];
...
merge (a, a+1000, b, b+2000, c);
```

Когда пользователь хочет объединить вектор и список (оба - шаблонные классы в библиотеке) и поместить результат в заново распределённую неинициализированную память, то это может быть выполнено так:

```
vector<Employee> a;
list<Employee> b;
...
Employee* c = allocate(a.size() + b.size(), (Employee*) 0);
merge(a.begin(), a.end(), b.begin(), b.end(),
      raw_storage_iterator <Employee*, Employee> (c));
```

где *begin()* и *end()* - функции-члены контейнеров, которые возвращают правильные типы итераторов или указателя-подобных объектов, позволяющие *merge* выполнить задание, а *raw_storage_iterator* - адаптер, который позволяет алгоритмам помещать результаты непосредственно в неинициализированную память, вызывая соответствующий конструктор копирования.

Во многих случаях полезно перемещаться через потоки ввода-вывода таким же образом, как через обычные структуры данных. Например, если мы хотим объединить две структуры данных и затем сохранить их в файле, было бы хорошо избежать создания вспомогательной структуры данных для хранения результата, а поместить результат непосредственно в соответствующий файл. Библиотека обеспечивает и

istream_iterator, и *ostream_iterator* шаблонные классы, чтобы многие из библиотечных алгоритмов могли работать с потоками ввода-вывода, которые представляют однородные блоки данных. Далее приводится программа, которая читает файл, состоящий из целых чисел, из стандартного ввода, удаляя все числа, делящиеся на параметр команды, и записывает результат в стандартный вывод:

```
main(int argc, char** argv) {
    if(argc != 2) throw("usage: remove_if_divides integer\n ");
    remove_copy_if(istream_iterator<int>(cin), istream_iterator<int>(),
        ostream_iterator<int>(cout, "\n"),
        not1(bind2nd (modulus<int>(), atoi(argv[1]))));
}
```

работа выполняется алгоритмом *remove_copy_if*, который читает целые числа одно за другим, пока итератор ввода не становится равным *end-of-stream* (*конец-потока*) итератору, который создаётся конструктором без параметров. (Вообще все алгоритмы работают способом "отсюда досюда", используя два итератора, которые показывают начало и конец ввода.) Потом *remove_copy_if* записывает целые числа, которые выдерживают проверку, в выходной поток через итератор вывода, который связан с *cout*. В качестве предиката *remove_copy_if* использует функциональный объект, созданный из функционального объекта *modulus<int>*, который берёт *i* и *j* и возвращает *i % j* как бинарный предикат, и превращает в унарный предикат, используя *bind2nd*, чтобы связать второй параметр с параметром командной строки *atoi(argv[1])*. Потом отрицание этого унарного предиката получается с помощью адаптера функции *not1*.

более реалистичный пример - фильтрующая программа, которая берёт файл и беспорядочно перетасовывает его строки.

```
main(int argc, char**) {
    if(argc != 1) throw("usage: shuffle\n");
    vector<string> v;
    copy(istream_iterator<string>(cin),istream_iterator<string>(),
        inserter(v, v.end()));
    random_shuffle(v.begin(), v.end());
    copy(v.begin(), v.end(), ostream_iterator<string>(cout));
}
```

этом примере *copy* перемещает строки из стандартного ввода в вектор, но так как вектор предварительно не размещён в памяти, используется итератор вставки, чтобы вставить в вектор строки одну за другой. (Эта методика позволяет всем функциям копирования работать в обычном режиме замены также, как в режиме вставки.) Потом *random_shuffle* перетасовывает вектор, а другой вызов *copy* копирует его в поток *cout*.

ТРЕБОВАНИЯ

Для гарантии совместной работы различные компоненты библиотеки должны удовлетворять некоторым основным требованиям. Требования должны быть общими, насколько это возможно, так что вместо высказывания "класс *X* должен определить функцию-член *operator++()*", мы говорим "для любого объекта *x* класса *X* определён *++x*". (Не определено, является ли оператор членом или глобальной функцией.) Требования установлены в терминах чётких выражений, которые определяют допустимые условия типов, удовлетворяющих требованиям. Для каждого набора требований имеется таблица, которая определяет начальный набор допустимых выражений и их семантику. Любой обобщённый алгоритм, который использует требования, должен быть написан в терминах допустимых выражений для своих формальных параметров.

Если требуется, чтобы была операция линейного времени сложности, это значит - не хуже, чем линейного времени, и операция постоянного времени удовлетворяет требованию.

В некоторых случаях мы представили семантические требования, использующие код C++. Такой код предназначен как спецификация эквивалентности одной конструкции другой, не обязательно как способ, которым конструкция должна быть реализована (хотя в некоторых случаях данный код, однозначно, является оптимальной реализацией).

ОСНОВНЫЕ КОМПОНЕНТЫ

Этот раздел содержит некоторые основные шаблонные функции и классы, которые используются в остальной части библиотеки.

Операторы (Operators)

Чтобы избежать избыточных определений *operator!=* из *operator==* и *operator>*, *<=*, *>=* из *operator<*, библиотека обеспечивает следующее:

```
template <class T1, class T2>
inline bool operator!=(const T1& x, const T2& y) {
    return !(x == y);
}
```

```

template <class T1, class T2>
inline bool operator>(const T1& x, const T2& y) {
    return y < x;
}

template <class T1, class T2>
inline bool operator<=(const T1& x, const T2& y) {
    return !(y < x);
}

template <class T1, class T2>
inline bool operator>=(const T1& x, const T2& y) {
    return !(x < y);
}

```

Пара (Pair)

Библиотека включает шаблоны для разнородных пар значений.

```

template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
    pair() {}
    pair(const T1& x, const T2& y) : first(x), second(y) {}
};

template <class T1, class T2>
inline bool operator==(const pair<T1,T2>& x, const pair<T1,T2>& y) {
    return x.first == y.first && x.second == y.second;
}

template <class T1, class T2>
inline bool operator<(const pair<T1,T2>& x, const pair<T1,T2>& y) {
    return x.first < y.first
        || (!(y.first < x.first) && x.second < y.second);
}

```

Библиотека обеспечивает соответствующую шаблонную функцию *make_pair*, чтобы упростить конструкцию пар. Вместо выражения, например:

```
return pair<int, double>(5, 3.1415926); // явные типы,
```

можно написать

```
return make_pair(5, 3.1415926); // типы выводятся.

template <class T1, class T2>
inline pair<T1,T2> make_pair(const T1& x, const T2& y) {
    return pair<T1,T2>(x, y);
}

```

ИТЕРАТОРЫ

Итераторы - это обобщение указателей, которые позволяют программисту работать с различными структурами данных (контейнерами) единообразным способом. Чтобы создать шаблонные алгоритмы, которые правильно и эффективно работают с различными типами структур данных, нам нужно формализовать не только интерфейсы, но также семантику и предположения сложности итераторов. Итераторы - это объекты, которые имеют *operator**, возвращающий значение некоторого класса или встроенного типа *T*, называемого *значимым типом* (*value type*) итератора. Для каждого типа итератора *X*, для которого определено равенство, имеется соответствующий знаковый целочисленный тип, называемый *типом расстояния* (*distance type*) итератора.

Так как итераторы - обобщение указателей, их семантика - обобщение семантики указателей в C++. Это гарантирует, что каждая шаблонная функция, которая использует итераторы, работает с обычными указателями. Есть пять категорий итераторов в зависимости от операций, определённых для них: *ввода* (*input iterators*), *вывода* (*output iterators*), *последовательные* (*forward iterators*), *двунаправленные* (*bidirectional iterators*) и *произвольного доступа* (*random access iterators*.) Последовательные итераторы удовлетворяют всем требованиям итераторов ввода и вывода и могут использоваться всякий раз, когда определяется тот или другой вид. Двунаправленные итераторы удовлетворяют всем требованиям последовательных итераторов и могут использоваться всякий раз, когда определяется последовательный итератор. Итераторы произвольного доступа удовлетворяют всем требованиям двунаправленных итераторов и могут использоваться всякий раз, когда определяется двунаправленный итератор. Имеется дополнительный атрибут, который могли бы иметь последовательные, двунаправленные и произвольного доступа итераторы, то есть они могут быть *модифицируемые* (*mutable*) или *постоянные* (*constant*) в зависимости от того, ведёт ли себя результат *operator** как ссылка или как ссылка на константу. Постоянные итераторы не удовлетворяют требованиям итераторов вывода.

Таблица 1. Отношения среди категорий итераторов

				-->	Ввода
Произвольного доступа	-->	Двунаправленные	-->	Последовательные	---
				-->	Вывода

Точно также, как обычный указатель на массив гарантирует, что имеется значение указателя, указывающего за последний элемент массива, так и для любого типа итератора имеется значение итератора, который указывает за последний элемент соответствующего контейнера. Эти значения называются *закончными* (*past-the-end*) значениями. Значения итератора, для которых *operator** определён, называются *разыменовываемыми* (*dereferenceable*). Библиотека никогда не допускает, что закончные значения являются разыменовываемыми. Итераторы могут также иметь *исключительные* (*singular*) значения, которые не связаны ни с каким контейнером. Например, после объявления неинициализированного указателя *x* (например, *int* x;*), всегда должно предполагаться, что *x* имеет исключительное значение указателя. Результаты большинства выражений не определены для исключительных значений. Единственное исключение - присваивание неисключительного значения итератору, который имеет исключительное значение. В этом случае исключительное значение перезаписывается таким же образом, как любое другое значение. Разыменовываемые и закончные значения всегда являются неисключительными.

Итератор *j* называется *доступным* (*reachable*) из итератора *i*, если и только если имеется конечная последовательность применений *operator++* к *i*, которая делает *i==j*. Если *i* и *j* относятся к одному и тому же контейнеру, тогда или *j* доступен из *i*, или *i* доступен из *j*, или оба доступны (*i == j*).

Большинство алгоритмических шаблонов библиотеки, которые работают со структурами данных, имеют интерфейсы, которые используют диапазоны. Диапазон - это пара итераторов, которые указывают начало и конец вычисления. Интервал *[i,i)* - пустой диапазон; вообще, диапазон *[i,j)* относится к элементам в структуре данных, начиная с элемента, указываемого *i*, и до элемента, но не включая его, указываемого *j*. Диапазон *[i,j)* допустим, если и только если *j* доступен из *i*. Результат применения алгоритмов библиотеки к недопустимым диапазонам не определён.

Все категории итераторов требуют только те функции, которые осуществимы для данной категории со сложностью постоянного времени (амортизированные). Поэтому таблицы требований для итераторов не имеют столбца сложности.

В следующих разделах мы принимаем: *a* и *b* - значения *X*, *n* - значение типа расстояния *Distance*, *u*, *tmp* и *m* - идентификаторы, *r* и *s* - леводопустимые (lvalues) значения *X*, *t* - значение значимого типа *T*.

Итераторы ввода (Input iterators)

Класс или встроенный тип *X* удовлетворяет требованиям итератора ввода для значимого типа *T*, если справедливы следующие выражения:

Таблица 2. Требования итератора ввода

выражение	возвращаемый тип	семантика исполнения	утверждение/примечание состояние до/после
<i>X(a)</i>			<i>X(a)</i> - копия <i>a</i> . примечание: предполагается деструктор.
<i>X u(a);</i> <i>X u = a;</i>			после: <i>u</i> - копия <i>a</i> .
<i>u = a</i>	<i>X&</i>		после: <i>u</i> - копия <i>a</i> .
<i>a == b</i>	обратимый в <i>bool</i>		если <i>a</i> - копия <i>b</i> , тогда <i>a == b</i> возвращает <i>true</i> . == - это отношение эквивалентности в области действия ==.
<i>a != b</i>	обратимый в <i>bool</i>	<i>!(a == b)</i>	
<i>*a</i>	обратимый в <i>T</i>		до: <i>a</i> - разыменовываемое. если <i>a</i> - копия <i>b</i> , то <i>*a</i> эквивалентно <i>*b</i> .
<i>++r</i>	<i>X&</i>		до: <i>r</i> - разыменовываемое. после: <i>r</i> - разыменовываемое или <i>r</i> - законченное.
<i>void r++</i>	<i>void</i>	<i>void ++r</i>	
<i>*r++</i>	<i>T</i>	<i>{ X tmp = r;</i> <i>++r;</i> <i>return tmp; }</i>	

ПРИМЕЧАНИЕ

Для итераторов ввода нет никаких требований на тип или значение $r++$ кроме требования, чтобы $*r++$ работал соответственным образом. В частности, $r == s$ не подразумевает, что $++r == ++s$. (Равенство не гарантирует свойство замены или ссылочной прозрачности.) Что касается $++r$, то нет больше никаких требований на значения любых копий r за исключением того, что они могут быть безопасно уничтожены или присвоены. После выполнения $++r$ не требуется, чтобы были копии (предыдущего) r в области $==$. Алгоритмы с итераторами ввода никогда не должны пытаться проходить через тот же самый итератор дважды. Они должны быть *однопроходными* (*single pass*) алгоритмами. *Не требуется, чтобы значимый тип T был леводопустимым типом (lvalue type)*. Эти алгоритмы могут использоваться с входными потоками как источниками входных данных через класс *istream_iterator*.

Итераторы вывода (Output iterators)

Класс или встроенный тип X удовлетворяет требованиям итератора вывода, если справедливы следующие выражения:

Таблица 3. Требования итератора вывода

выражение	возвращаемый тип	семантика исполнения	утверждение/примечание	состояние до/после
$X(a)$			$*a = t$ эквивалентно $*X(a) = t$. примечание: предполагается деструктор.	
$X\ u(a);$ $X\ u = a;$				
$*a = t$	результат не используется			
$++r$	$X\&$			
$r++$	X или $X\&$			

ПРИМЕЧАНИЕ

Единственное допустимое использование *operator** - на левой стороне выражения присваивания. *Присваивание через то же самое значение итератора происходит только однажды*. Алгоритмы с итераторами вывода никогда не должны пытаться проходить через тот же самый итератор дважды. Они должны быть *однопроходными* (*single pass*) алгоритмами. Равенство и неравенство не обязательно определены. Алгоритмы, которые берут итераторы вывода, могут использоваться с выходными потоками для помещения в них данных через класс *ostream_iterator*, также как с итераторами вставки и вставляющими указателями. В частности, следующие два условия должны соблюдаться: во-первых, через любое значение итератора должно выполняться присваивание до его увеличения (то есть, для итератора вывода i недопустима последовательность кода $i++$; $i++$); во-вторых, любое значение итератора вывода может иметь только одну активную копию в любое данное время (например, недопустима последовательность кода $i = j$; $*++i = a$; $*j = b$).

Последовательные итераторы (Forward iterators)

Класс или встроенный тип X удовлетворяет требованиям последовательного итератора, если справедливы следующие выражения:

Таблица 4. Требования последовательного итератора

выражение	возвращаемый тип	семантика исполнения	утверждение/примечание	состояние до/после
$X\ u;$			примечание: u может иметь исключительное значение. примечание: предполагается деструктор.	
$X()$			примечание: $X()$ может быть исключительным.	
$X(a);$			$a == X(a)$	
$X\ u(a);$ $X\ u = a;$		$X\ u; u = a;$	после: $u == a$.	
$a == b$	обратимый в <i>bool</i>		$==$ - это отношение эквивалентности.	
$a != b$	обратимый в <i>bool</i>	$!(a == b)$		
$r = a$	$X\&$.	после: $r == a$.	

<i>*a</i>	обратимый в <i>T</i>		до: <i>a</i> - разыменовываемое. <i>a == b</i> подразумевает <i>*a == *b</i> . Если <i>X</i> - модифицируемый, то <i>*a = t</i> - допустимо.
<i>++r</i>	<i>X&</i>		до: <i>r</i> - разыменовываемое. после: <i>r</i> - разыменовываемое или <i>r</i> - законченное. <i>r == s</i> и <i>r</i> - разыменовываемое подразумевает <i>++r == ++s</i> . <i>&r == &++r</i> .
<i>r++</i>	<i>X</i>	<pre>{ X tmp = r; ++ r; return tmp; }</pre>	

ПРИМЕЧАНИЕ

Тот факт, что *r == s* подразумевает *++r == ++s* (что неверно для итераторов ввода и вывода) и что удалено ограничение на число присваиваний через итератор (которое применяется к итераторам вывода), позволяет использование многопроходных однонаправленных алгоритмов с последовательными итераторами.

Двунаправленные итераторы (Bidirectional iterators)

Класс или встроенный тип *X* удовлетворяет требованиям двунаправленного итератора, если к таблице, которая определяет последовательные итераторы, мы добавим следующие строки:

Таблица 5. Требования двунаправленного итератора (в дополнение к последовательному итератору)

выражение	возвращаемый тип	семантика исполнения	утверждение/примечание	состояние до/после
<i>--r</i>	<i>X&</i>		до: существует <i>s</i> такое, что <i>r == ++s</i> . после: <i>s</i> - разыменовываемое. <i>--(++r) == r</i> . <i>--r == --s</i> подразумевает <i>r == s</i> . <i>&r == &--r</i> .	
<i>r--</i>	<i>X</i>	<pre>{ X tmp = r; --r; return tmp; }</pre>		

ПРИМЕЧАНИЕ

Двунаправленные итераторы позволяют алгоритмам перемещать итераторы назад также, как вперёд.

Итераторы произвольного доступа (Random access iterators)

Класс или встроенный тип *X* удовлетворяет требованиям итераторов произвольного доступа, если к таблице, которая определяет двунаправленные итераторы, мы добавим следующие строки:

Таблица 6: Требования итератора произвольного доступа (в дополнение к двунаправленному итератору)

выражение	возвращаемый тип	семантика исполнения	утверждение/примечание	состояние до/после
<i>r += n</i>	<i>X&</i>	<pre>{ Distance m = n; if(m >= 0) while(m--) ++r; else while(m++) --r; return r; }</pre>		
<i>a + n</i> <i>n + a</i>	<i>X</i>	<pre>{ X tmp = a; return tmp += n; }</pre>	<i>a + n == n + a</i> .	
<i>r -= n</i>	<i>X&</i>	<pre>return r += -n;</pre>		

$a - n$	X	$\{ X \ tmp = a;$ $\ return \ tmp \ -= \ n; \}$	
$b - a$	$Distance$		до: существует значение n типа $Distance$ такое, что $a + n = b$. $b == a + (b - a)$.
$a[n]$	обратимый в T	$*(a + n)$	
$a < b$	обратимый в $bool$	$b - a > 0$	$<$ - это отношение полного упорядочения
$a > b$	обратимый в $bool$	$b < a$	$>$ - это отношение полного упорядочения, противоположное $<$.
$a \geq b$	обратимый в $bool$	$!(a < b)$	
$a \leq b$	обратимый в $bool$	$!(a > b)$	

Теги итераторов (Iterator tags)

Чтобы осуществлять алгоритмы только в терминах итераторов, часто бывает необходимо вывести тип значения и тип расстояния из итератора. Для решения этой задачи требуется, чтобы для итератора i любой категории, отличной от итератора вывода, выражение $value_type(i)$ возвращало $(T^*)(0)$, а выражение $distance_type(i)$ возвращало $(Distance^*)(0)$. Для итераторов вывода эти выражения не требуются.

Примеры использования тегов итераторов

Для всех типов обычных указателей мы можем определить $value_type$ и $distance_type$ с помощью следующего:

```
template <class T>
inline T* value_type(const T*) { return (T*) (0); }

template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*) (0); }
```

Тогда, если мы хотим осуществить обобщённую функцию $reverse$, мы пишем следующее:

```
template <class BidirectionalIterator>
inline void reverse(BidirectionalIterator first, BidirectionalIterator last)
{
    _reverse(first, last, value_type(first), distance_type(first));
}
```

где $_reverse$ определена следующим образом:

```
template <class BidirectionalIterator, class T, class Distance>
void _reverse(BidirectionalIterator first, BidirectionalIterator last, T*,
              Distance*) {
    Distance n;
    distance(first, last, n); // смотри раздел "Операции с итераторами"
    --n;
    while (n > 0) {
        T tmp = *first;
        *first++ = *--last;
        *last = tmp;
        n -= 2;
    }
}
```

Если имеется дополнительный тип указателя $_huge$ такой, что разность двух указателей $_huge$ имеет тип $long\ long$, мы определяем:

```
template <class T>
```



```
inline T* value_type(const T _huge *) { return (T*) (0); }

template <class T>
inline long long* distance_type(const T _huge *) {
    return (long long*)(0);
}
```

Часто желательно для шаблонной функции выяснить, какова наиболее специфичная категория её итераторного аргумента, так чтобы функция могла выбирать наиболее эффективный алгоритм во время компиляции. Чтобы облегчить это, библиотека вводит классы *тегов категорий* (*category tag*), которые используются как теги времени компиляции для выбора алгоритма. Это следующие теги: *input_iterator_tag*, *output_iterator_tag*, *forward_iterator_tag*, *bidirectional_iterator_tag* и *random_access_iterator_tag*. Каждый итератор *i* должен иметь выражение *iterator_category(i)*, определённое для него, которое возвращает тег наиболее специфичной категории, который описывает его поведение. Например, мы определяем, что все типы указателей находятся в категории итераторов произвольного доступа:

```
template <class T>
inline random_access_iterator_tag iterator_category(const T*)
{
    return random_access_iterator_tag();
}
```

Определяемый пользователем итератор *BinaryTreeIterator* может быть включен в категорию двунаправленных итераторов следующим образом:

```
template <class T>
inline bidirectional_iterator_tag iterator_category(
    const BinaryTreeIterator<T>&)
{
    return bidirectional_iterator_tag();
}
```

Если шаблонная функция *evolve* хорошо определена для двунаправленных итераторов, но может быть осуществлена более эффективно для итераторов произвольного доступа, тогда реализация выглядит так:

```
template <class BidirectionalIterator>
inline void evolve(BidirectionalIterator first, BidirectionalIterator last)
{
    evolve(first, last, iterator_category(first));
}

template <class BidirectionalIterator>
void evolve(BidirectionalIterator first, BidirectionalIterator last,
    bidirectional_iterator_tag) {
    // ... более универсальный, но менее эффективный алгоритм
}

template <class RandomAccessIterator>
void evolve(RandomAccessIterator first, RandomAccessIterator last,
    random_access_iterator_tag) {
    // ... более эффективный, но менее универсальный алгоритм
}
```

Примитивы, определённые в библиотеке

Чтобы упростить задачу определения *iterator_category*, *value_type* и *distance_type* для определяемых пользователем итераторов, библиотека обеспечивает следующие предопределённые классы и функции:

```
// iterator tags (теги итераторов)

struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag {};
struct bidirectional_iterator_tag {};
struct random_access_iterator_tag {};
```

```

// iterator bases (базовые классы итераторов)

template <class T, class Distance = ptrdiff_t> struct input_iterator {};
struct output_iterator {};
// output_iterator не шаблон, потому что у итераторов вывода
// не определены ни значимый тип, ни тип расстояния.
template <class T, class Distance = ptrdiff_t>
    struct forward_iterator {};
template <class T, class Distance = ptrdiff_t>
    struct bidirectional_iterator {};
template <class T, class Distance = ptrdiff_t>
    struct random_access_iterator {};

// iterator_category (функции категорий итераторов)

template <class T, class Distance>
inline input_iterator_tag
iterator_category(const input_iterator<T, Distance>&) {
    return input_iterator_tag();
}
inline output_iterator_tag iterator_category(const output_iterator&) {
    return output_iterator_tag();
}
template <class T, class Distance>
inline forward_iterator_tag
iterator_category(const forward_iterator<T, Distance>&) {
    return forward_iterator_tag();
}
template <class T, class Distance>
inline bidirectional_iterator_tag
iterator_category(const bidirectional_iterator<T, Distance>&) {
    return bidirectional_iterator_tag();
}
template <class T, class Distance>
inline random_access_iterator_tag
iterator_category(const random_access_iterator<T, Distance>&) {
    return random_access_iterator_tag();
}
template <class T>
inline random_access_iterator_tag iterator_category(const T*) {
    return random_access_iterator_tag();
}

// value_type of iterator (функции значимого типа итераторов)

template <class T, class Distance>
inline T* value_type(const input_iterator<T, Distance>&) {
    return (T*) (0);
}
template <class T, class Distance>
inline T* value_type(const forward_iterator<T, Distance>&) {
    return (T*) (0);
}
template <class T, class Distance>
inline T* value_type(const bidirectional_iterator<T, Distance>&) {
    return (T*) (0);
}
template <class T, class Distance>
inline T* value_type(const random_access_iterator<T, Distance>&) {
    return (T*) (0);
}
template <class T>
inline T* value_type(const T*) { return (T*) (0); }

```

```
// distance_type of iterator (функции типа расстояния итераторов)

template <class T, class Distance>
inline Distance* distance_type(const input_iterator<T, Distance>&) {
    return (Distance*) (0);
}
template <class T, class Distance>
inline Distance* distance_type(const forward_iterator<T, Distance>&) {
    return (Distance*) (0);
}
template <class T, class Distance>
inline Distance* distance_type(const bidirectional_iterator<T, Distance>&) {
    return (Distance*) (0);
}
template <class T, class Distance>
inline Distance* distance_type(const random_access_iterator<T, Distance>&) {
    return (Distance*) (0);
}
}
template <class T>
inline ptrdiff_t* distance_type(const T*) { return (ptrdiff_t*) (0); }
```

Если пользователь хочет определить двунаправленный итератор для некоторой структуры данных, содержащей *double*, и такой, чтобы работал с большой (large) моделью памяти компьютера, то это может быть сделано таким определением:

```
class MyIterator : public bidirectional_iterator <double, long> {
// код, осуществляющий ++, и т.д.
};
```

Тогда нет необходимости определять *iterator_category*, *value_type*, и *distance_type* в *MyIterator*.

Операции с итераторами (Iterator operations)

Так как только итераторы произвольного доступа обеспечивают + и - операторы, библиотека предоставляет две шаблонные функции *advance* и *distance*. Эти функции используют + и - для итераторов произвольного доступа (и имеют, поэтому, сложность постоянного времени для них); для итераторов ввода, последовательных и двунаправленных итераторов функции используют ++, чтобы обеспечить реализацию со сложностью линейного времени. *advance* берет отрицательный параметр *n* только для итераторов произвольного доступа и двунаправленных итераторов. *advance* увеличивает (или уменьшает для отрицательного *n*) итераторную ссылку *i* на *n*. *distance* увеличивает *n* на число единиц, сколько требуется, чтобы дойти от *first* до *last*.

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n);

template <class InputIterator, class Distance>
inline void distance(InputIterator first, InputIterator last, Distance& n);
```

distance должна быть функцией 3-х параметров, сохраняющей результат в ссылке вместо возвращения результата, потому что тип расстояния не может быть выведен из встроенных итераторных типов, таких как *int**.

ФУНКЦИОНАЛЬНЫЕ ОБЪЕКТЫ

Функциональные объекты - это объекты, для которых определён *operator()*. Они важны для эффективного использования библиотеки. В местах, где ожидается передача указателя на функцию алгоритмическому шаблону, интерфейс установлен на приём объекта с определённым *operator()*. Это не только заставляет алгоритмические шаблоны работать с указателями на функции, но также позволяет им работать с произвольными функциональными объектами. Использование функциональных объектов вместе с функциональными шаблонами увеличивает выразительную мощность библиотеки также, как делает результирующий код более эффективным. Например, если мы хотим поэлементно сложить два вектора *a* и *b*, содержащие *double*, и поместить результат в *a*, мы можем сделать это так:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

Если мы хотим отрицать каждый элемент *a*, мы можем сделать это так:

```
transform(a.begin(), a.end(), a.begin(), negate<double>());
```

Соответствующие функции вставят сложение и отрицание.

Чтобы позволить адаптерам и другим компонентам манипулировать функциональными объектами, которые используют один или два параметра, требуется, чтобы они соответственно обеспечили определение типов (typedefs) *argument_type* и *result_type* для функциональных объектов, которые используют один параметр, и *first_argument_type*, *second_argument_type* и *result_type* для функциональных объектов, которые используют два параметра.

Базовые классы (Base)

Следующие классы предоставляются, чтобы упростить определение типов (typedefs) параметров и результата:

```
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```

Арифметические операции (Arithmetic operations)

Библиотека обеспечивает базовые классы функциональных объектов для всех арифметических операторов языка.

```
template <class T>
struct plus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x - y; }
};

template <class T>
struct times : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x * y; }
};

template <class T>
struct divides : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x / y; }
};

template <class T>
struct modulus : binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x % y; }
};

template <class T>
struct negate : unary_function<T, T> {
    T operator()(const T& x) const { return -x; }
};
```

Сравнения (Comparisons)

Библиотека обеспечивает базовые классы функциональных объектов для всех операторов сравнения языка

```
template <class T>
struct equal_to : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x == y; }
};

template <class T>
struct not_equal_to : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x != y; }
};

template <class T>
struct greater : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x > y; }
};

template <class T>
struct less : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x < y; }
};

template <class T>
struct greater_equal : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x >= y; }
};

template <class T>
struct less_equal : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x <= y; }
};
```

Логические операции (Logical operations)

```
template <class T>
struct logical_and : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x && y; }
};

template <class T>
struct logical_or : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x || y; }
};

template <class T>
struct logical_not : unary_function<T, bool> {
    bool operator()(const T& x) const { return !x; }
};
```

РАСПРЕДЕЛИТЕЛИ

Одна из общих проблем в мобильности - это способность инкапсулировать информацию относительно модели памяти. Эта информация включает типы указателей, тип их разности, тип размера объектов в этой модели памяти, также как её примитивы выделения и освобождения памяти.

STL принимается за эту проблему, обеспечивая стандартный набор требований для *распределителей* (*allocators*), являющихся объектами, которые инкапсулируют эту информацию. Все контейнеры в STL параметризованы в терминах распределителей. Это значительно упрощает задачу взаимодействия с многочисленными моделями памяти.

Требования распределителей (Allocator requirements)

В следующей таблице мы предполагаем, что X - класс распределителей для объектов типа T , a - значение X , n имеет тип $X::size_type$, p имеет тип $X::pointer$, r имеет тип $X::reference$ и s имеет тип $X::const_reference$.

Все операции с распределителями, как ожидается, сводятся к постоянному времени.

Таблица 7. Требования распределителей

выражение	возвращаемый тип	утверждение/примечание состояние до/после
$X::value_type$	T	
$X::reference$	леводопустимое значение T (lvalue of T)	
$X::const_reference$	const lvalue of T	
$X::pointer$	указатель на тип T	результатом $operator*$ для значений $X::pointer$ является $reference$.
$X::const_pointer$	указатель на тип $const\ T$	результат $operator*$ для значений $X::const_pointer$ - $const_reference$; это - тот же самый тип указателя, как $X::pointer$, в частности, $sizeof(X::const_pointer) == sizeof(X::pointer)$.
$X::size_type$	беззнаковый целочисленный тип	тип, который может представлять размер самого большого объекта в модели памяти.
$X::difference_type$	знаковый целочисленный тип	тип, который может представлять разность между двумя любыми указателями в модели памяти.
$X\ a;$		примечание: предполагается деструктор.
$a.address(r)$	указатель	$*(a.address(r)) == r$.
$a.const_address(s)$	$const_pointer$	$*(a.address(s)) == s$.
$a.allocate(n)$	$X::pointer$	память распределяется для n объектов типа T , но объекты не создаются. $allocate$ может вызывать соответствующее исключение.
$a.deallocate(p)$	результат не используется	все объекты в области, указываемой p , должны быть уничтожены до этого запроса.
$construct(p, a)$	$void$	после: $*p == a$.
$destroy(p)$	$void$	значение, указываемое p , уничтожается.
$a.init_page_size()$	$X::size_type$	возвращённое значение - оптимальное значение для начального размера буфера данного типа. Предполагается, что если k возвращено функцией $init_page_size$, t - время конструирования для T , и u - время, которое требуется для выполнения $allocate(k)$, тогда $k * t$ будет намного больше, чем u .
$a.max_size()$	$X::size_type$	наибольшее положительное значение $X::difference_type$

$pointer$ относится к категории модифицируемых итераторов произвольного доступа, ссылающихся на T . $const_pointer$ относится к категории постоянных итераторов произвольного доступа, ссылающихся на T . Имеется определённое преобразование из $pointer$ в $const_pointer$.

Для любого шаблона распределителя $Alloc$ имеется определение для типа $void$. У $Alloc<void>$ определены только конструктор, деструктор и $Alloc<void>::pointer$. Преобразования определены из любого $Alloc<T>::pointer$ в $Alloc<void>::pointer$ и обратно, так что для любого p будет $p == Alloc<T>::pointer(Alloc<void>::pointer(p))$.

Распределитель по умолчанию (The default allocator)

```
template <class T>
class allocator {
public:
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    allocator();
```

```
~allocator();
pointer address(reference x);
const_pointer const_address(const_reference x);
pointer allocate(size_type n);
void deallocate(pointer p);
size_type init_page_size();
size_type max_size();
};

class allocator<void> {
public:
    typedef void* pointer;
    allocator();
    ~allocator();
};
```

Предполагается, что в дополнение к *allocator* поставщики библиотеки обеспечивают распределители для всех моделей памяти.

КОНТЕЙНЕРЫ

Контейнеры - это объекты, которые содержат другие объекты. Они управляют размещением в памяти и освобождением этих объектов через конструкторы, деструкторы, операции вставки и удаления.

В следующей таблице мы полагаем, что *X* - контейнерный класс, содержащий объекты типа *T*, *a* и *b* - значения *X*, *u* - идентификатор, *r* - значение *X*&.

Таблица 8. Требования контейнеров

выражение	возвращаемый тип	семантика исполнения	утверждение/примечание состояние до/после	сложность
<i>X::value_type</i>	<i>T</i>			время компиляции
<i>X::reference</i>				время компиляции
<i>X::const_reference</i>				время компиляции
<i>X::pointer</i>	тип указателя, указывающий на <i>X::reference</i>		указатель на <i>T</i> в модели памяти, используемой контейнером	время компиляции
<i>X::iterator</i>	тип итератора, указывающий на <i>X::reference</i>		итератор любой категории, кроме итератора вывода.	время компиляции
<i>X::const_iterator</i>	тип итератора, указывающий на <i>X::const_reference</i>		постоянный итератор любой категории, кроме итератора вывода.	время компиляции
<i>X::difference_type</i>	знаковый целочисленный тип		идентичен типу расстояния <i>X::iterator</i> и <i>X::const_iterator</i>	время компиляции
<i>X::size_type</i>	беззнаковый целочисленный тип		<i>size_type</i> может представлять любое неотрицательное значение <i>difference_type</i>	время компиляции
<i>X u;</i>			после: <i>u.size()</i> == 0.	постоянная
<i>X()</i>			<i>X().size()</i> == 0.	постоянная
<i>X(a)</i>			<i>a</i> == <i>X(a)</i> .	линейная
<i>X u(a);</i> <i>X u == a;</i>		<i>X u; u = a;</i>	после: <i>u</i> == <i>a</i> .	линейная
<i>(&a)->~X()</i>	результат не используется		после: <i>a.size()</i> == 0. примечание: деструктор применяется к каждому элементу <i>a</i> , и вся память возвращается.	линейная
<i>a.begin()</i>	<i>iterator;</i> <i>const_iterator</i> для постоянного <i>a</i>			постоянная
<i>a.end()</i>	<i>iterator;</i> <i>const_iterator</i> для постоянного <i>a</i>			постоянная

<i>a == b</i>	обратимый в <i>bool</i>	<i>a.size() == b.size() && equal(a.begin(), a.end(), b.begin())</i>	== - это отношение эквивалентности. примечание: <i>equal</i> определяется в разделе алгоритмов.	линейная
<i>a != b</i>	обратимый в <i>bool</i>	<i>!(a == b)</i>		линейная
<i>r = a</i>	<i>X&</i>	<i>if(&r != &a) { (&r)-> X::~~X(); new (&r) X(a); return r; }</i>	после: <i>r == a</i> .	линейная
<i>a.size()</i>	<i>size_type</i>	<i>size_type n = 0; distance(a.begin(), a.end(), n); return n;</i>		постоянная
<i>a.max_size()</i>	<i>size_type</i>		<i>size()</i> самого большого возможного контейнера.	постоянная
<i>a.empty()</i>	обратимый в <i>bool</i>	<i>a.size() == 0</i>		постоянная
<i>a < b</i>	обратимый в <i>bool</i>	<i>lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())</i>	до: < определён для значений <i>T</i> . < - отношение полного упорядочения. <i>lexicographical_compare</i> определяется в разделе алгоритмов.	линейная
<i>a > b</i>	обратимый в <i>bool</i>	<i>b < a</i>		линейная
<i>a <= b</i>	обратимый в <i>bool</i>	<i>!(a > b)</i>		линейная
<i>a >= b</i>	обратимый в <i>bool</i>	<i>!(a < b)</i>		линейная
<i>a.swap(b)</i>	<i>void</i>	<i>swap(a, b)</i>		постоянная

Функция-член *size()* возвращает число элементов в контейнере. Её семантика определяется правилами конструкторов, вставок и удалений.

begin() возвращает итератор, ссылающийся на первый элемент в контейнере. *end()* возвращает итератор, который является законечным.

Если тип итератора контейнера принадлежит к категории двунаправленных итераторов или итераторов произвольного доступа, то контейнер называется *reversible* (обратимым) и удовлетворяет следующим дополнительным требованиям:

Таблица 9. Требования обратимых контейнеров (в дополнение к контейнерам)

выражение	возвращаемый тип	семантика исполнения	сложность
<i>X::reverse_iterator</i>		<i>reverse_iterator<iterator, value_type, reference, difference_type></i> для итератора произвольного доступа. <i>reverse_bidirectional_iterator<iterator, value_type, reference, difference_type></i> для двунаправленного итератора	время компиляции
<i>X::const_reverse_iterator</i>		<i>reverse_iterator<const_iterator, value_type, const_reference, difference_type></i> для итератора произвольного доступа. <i>reverse_bidirectional_iterator<const_iterator, value_type, const_reference, difference_type></i> для двунаправленного итератора.	время компиляции
<i>a.rbegin()</i>	<i>reverse_iterator;</i> <i>const_reverse_iterator</i> для постоянного <i>a</i>	<i>reverse_iterator(end())</i>	постоянная

постоянная

Последовательность - это вид контейнера, который организует конечное множество объектов одного и того же типа в строгом линейном порядке. Библиотека обеспечивает три основных вида последовательных контейнеров: *vector* (вектор), *list* (список) и *deque* (двусторонняя очередь). Она также предоставляет контейнерные адаптеры, которые облегчают создание абстрактных типов данных, таких как стеки или очереди, из основных видов последовательностей (или из других видов последовательностей, которые пользователь может сам определить).

Сложности выражений зависят от последовательностей.

Таблица 10. Требования последовательностей (в дополнение к контейнерам)

$X(n, t)$ $X\ a(n, t);$	после: $size() == n$. создаёт последовательность с n копиями t .
$X(i, j)$ $X\ a(i, j);$	после: $size() ==$ расстоянию между i и j . создаёт последовательность, равную диапазону $[i, j)$.
$a.insert(p, t)$ <i>iterator</i>	вставляет копию t перед p . возвращаемое значение указывает на вставленную копию.
$a.insert(p, n, t)$ результат не используется	вставляет n копий t перед p .
$a.insert(p, i, j)$ результат не используется	вставляет копии элементов из диапазона $[i, j)$ перед p .
$a.erase(q)$ результат не используется	удаляет элемент, указываемый q .
$a.erase(q1, q2)$ результат не используется	удаляет элементы в диапазоне $[q1, q2)$.

Типы *iterator* и *const_iterator* для последовательностей должны быть, по крайней мере, из категории последовательных итераторов.

Таблица 11. Необязательные операции последовательностей

<i>a.front()</i>	<i>reference;</i> <i>const_reference</i> для постоянного <i>a</i>	<i>*a.begin()</i>	<i>vector, list, deque</i>
<i>a.back()</i>	<i>reference;</i> <i>const_reference</i> для постоянного <i>a</i>	<i>*a(--end())</i>	<i>vector, list, deque</i>
<i>a.push_front(t)</i>	<i>void</i>	<i>a.insert(a.begin(), t)</i>	<i>list, deque</i>
<i>a.push_back(t)</i>	<i>void</i>	<i>a.insert(a.end(), t)</i>	<i>vector, list, deque</i>
<i>a.pop_front()</i>	<i>void</i>	<i>a.erase(a.begin())</i>	<i>list, deque</i>
<i>a.pop_back()</i>	<i>void</i>	<i>a.erase(-- a.end())</i>	<i>vector, list, deque</i>
<i>a[n]</i>	<i>reference;</i> <i>const_reference</i> для постоянного <i>a</i>	<i>*(a.begin() + n)</i>	<i>vector, deque</i>

Вектор (Vector)

vector - вид последовательности, которая поддерживает итераторы произвольного доступа. Кроме того, он поддерживает операции вставки и удаления в конце с постоянным (амортизированным) временем; вставка и удаление в середине занимают линейное время. Управление памятью обрабатывается автоматически, хотя для улучшения эффективности можно давать подсказки.

```
template <class T, template <class U> class Allocator = allocator>
class vector {
public:

    // определения типов (typedefs):

    typedef iterator;
    typedef const_iterator;
    typedef Allocator<T>::pointer pointer;
    typedef Allocator<T>::reference reference;
    typedef Allocator<T>::const_reference const_reference;
    typedef size_type;
    typedef difference_type;
    typedef T value_type;
    typedef reverse_iterator;
    typedef const_reverse_iterator;

    // размещение/освобождение (allocation/deallocation):

    vector();
    vector(size_type n, const T& value = T());
    vector(const vector<T, Allocator>& x);
    template <class InputIterator>
    vector(InputIterator first, InputIterator last);
    ~vector();
    vector<T, Allocator>& operator=(const vector<T, Allocator>& x);
    void reserve(size_type n);
    void swap(vector<T, Allocator>& x);

    // средства доступа (accessors):

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin();
    reverse_iterator rend();
    const_reverse_iterator rend();
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    bool empty() const;
    reference operator[](size_type n);
    const_reference operator[](size_type n) const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;

    // вставка/стирание (insert/erase):

    void push_back(const T& x);
    iterator insert(iterator position, const T& x = T());
    void insert(iterator position, size_type n, const T& x);
    template <class InputIterator>
    void insert(iterator position, InputIterator first, InputIterator last);
    void pop_back();
    void erase(iterator position);
```

```

    void erase(iterator first, iterator last);
};

template <class T, class Allocator>
bool operator==(const vector<T, Allocator>& x,
               const vector<T, Allocator>& y);

template <class T, class Allocator>
bool operator<(const vector<T, Allocator>& x,
              const vector<T, Allocator>& y);

```

iterator - это итератор произвольного доступа, ссылающийся на *T*. Точный тип зависит от исполнения и определяется в *Allocator*.

const_iterator - это постоянный итератор произвольного доступа, ссылающийся на *const T*. Точный тип зависит от исполнения и определяется в *Allocator*. Гарантируется, что имеется конструктор для *const_iterator* из *iterator*.

size_type - беззнаковый целочисленный тип. Точный тип зависит от исполнения и определяется в *Allocator*.

difference_type - знаковый целочисленный тип. Точный тип зависит от исполнения и определяется в *Allocator*.

Конструктор `template <class InputIterator> vector(InputIterator first, InputIterator last)` делает только *N* вызовов конструктора копирования *T* (где *N* - расстояние между *first* и *last*) и никаких перераспределений, если итераторы *first* и *last* относятся к последовательной, двунаправленной или произвольного доступа категориям. Он делает, самое большее, *2N* вызовов конструктора копирования *T* и *logN* перераспределений, если они - только итераторы ввода, так как невозможно определить расстояние между *first* и *last* и затем сделать копирование.

Функция-член *capacity* (ёмкость) возвращает размер распределённой памяти в векторе. Функция-член *reserve* - директива, которая сообщает *vector*(вектору) запланированное изменение размера, так чтобы он мог соответственно управлять распределением памяти. Это не изменяет размер последовательности и занимает, самое большее, линейное время от размера последовательности. Перераспределение в этом случае происходит тогда и только тогда, когда текущая ёмкость меньше, чем параметр *reserve*. После *reserve* ёмкость (*capacity*) больше или равна параметру *reserve*, если происходит перераспределение; а иначе равна предыдущему значению *capacity*. Перераспределение делает недействительными все ссылки, указатели и итераторы, ссылающиеся на элементы в последовательности. Гарантируется, что нет никакого перераспределения во время вставок, которые происходят после того, как *reserve* выполняется, до времени, когда размер вектора достигает размера, указанного *reserve*.

insert (вставка) вызывает перераспределение, если новый размер больше, чем старая ёмкость. Если никакого перераспределения не происходит, все итераторы и ссылки перед точкой вставки остаются справедливыми. Вставка единственного элемента в вектор линейна относительно расстояния от точки вставки до конца вектора. Амортизированная сложность во время жизни вектора, вставляющего единственный элемент в свой конец, постоянна. Вставка множественных элементов в вектор с единственным вызовом вставляющей функции-члена линейна относительно суммы числа элементов плюс расстояние до конца вектора. Другими словами, намного быстрее вставить много элементов в середину вектора сразу, чем делать вставку по одному элементу. Шаблонная вставляющая функция-член предраспределяет достаточно памяти для вставки, если итераторы *first* и *last* относятся к последовательной, двунаправленной или произвольного доступа категориям. Иначе функция вставляет элементы один за другим и не должна использоваться для вставки в середину векторов.

erase (стирание) делает недействительными все итераторы и ссылки после пункта стирания. Деструктор *T* вызывается столько раз, каково число стёртых элементов, а оператор присваивания *T* вызывается столько раз, каково число элементов в векторе после стёртых элементов.

Чтобы оптимизировать распределение места, даётся определение для *bool*.

```

class vector<bool, allocator> {
public:

    // битовая ссылка (bit reference):

    class reference {
    public:
        ~reference();
        operator bool() const;
        reference& operator=(const bool x);
        void flip();           // инвертирует бит (flips the bit )
    };

    // определения типов (typedefs):

    typedef bool const_reference;
    typedef iterator;
    typedef const_iterator;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef bool value_type;

```

```

    typedef reverse_iterator;
    typedef const_reverse_iterator;

// размещение/освобождение (allocation/deallocation):

    vector();
    vector(size_type n, const bool& value = bool());
    vector(const vector<bool, allocator>& x);
    template <class InputIterator>
    vector(InputIterator first, InputIterator last);
    ~vector();
    vector<bool, allocator>& operator=(const vector<bool,
        allocator>& x);
    void reserve(size_type n);
    void swap(vector<bool, allocator>& x);

// средства доступа (accessors):

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin();
    reverse_iterator rend();
    const_reverse_iterator rend();
    size_type size() const;
    size_type max_size() const;
    size_type capacity() const;
    bool empty() const;
    reference operator[](size_type n);
    const_reference operator[](size_type n) const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;

// вставка/стирание (insert/erase):

    void push_back(const bool& x);
    iterator insert(iterator position, const bool& x = bool());
    void insert(iterator position, size_type n, const bool& x);
    template <class InputIterator>
    void insert(iterator position, InputIterator first, InputIterator last);
    void pop_back();
    void erase(iterator position);
    void erase(iterator first, iterator last);
};

void swap(vector<bool, allocator>::reference x,
    vector<bool, allocator>::reference y);

bool operator==(const vector<bool, allocator>& x,
    const vector<bool, allocator>& y);

bool operator<(const vector<bool, allocator>& x,
    const vector<bool, allocator>& y);

```

reference - класс, который имитирует поведение ссылок отдельного бита в *vector<bool>*.

Ожидается, что каждое исполнение обеспечит определение *vector<bool>* для всех поддерживаемых моделей памяти.

Сейчас невозможно шаблонизировать определение. То есть мы не можем написать:

```
template <template <class U> class Allocator == allocator>
```

```
class vector<bool, Allocator> { /* ... */ };
```

Поэтому обеспечивается только *vector<bool, Allocator>*.

Список (List)

list - вид последовательности, которая поддерживает двунаправленные итераторы и позволяет операции вставки и стирания с постоянным временем в любом месте последовательности, с управлением памятью, обрабатываемым автоматически. В отличие от векторов и двусторонних очередей, быстрый произвольный доступ к элементам списка не поддерживается, но многим алгоритмам, во всяком случае, только и нужен последовательный доступ.

```
template <class T, template <class U> class Allocator = allocator>
class list {
public:
```

```
// определения типов:
```

```
typedef iterator;
typedef const_iterator;
typedef Allocator<T>::pointer pointer;
typedef Allocator<T>::reference reference;
typedef Allocator<T>::const_reference const_reference;
typedef size_type;
typedef difference_type;
typedef T value_type;
typedef reverse_iterator;
typedef const_reverse_iterator;
```

```
// размещение/удаление:
```

```
list()
list(size_type n, const T& value = T());
template <class InputIterator>
list(InputIterator first, InputIterator last);
list(const list<T, Allocator>& x) ;
~list();
list<T, Allocator>& operator=(const list<T,Allocator>& x);
void swap(list<T, Allocator>& x);
```

```
// средства доступа:
```

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin();
reverse_iterator rend();
const_reverse_iterator rend();
bool empty() const;
size_type size() const;
size_type max_size() const;
reference front();
const_reference front() const;
reference back();
const_reference back() const;
```

```
// вставка/стирание:
```

```
void push_front(const T& x);
void push_back(const T& x);
iterator insert(iterator position, const T& x = T());
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
```

```

void insert(iterator position, InputIterator first, InputIterator last);
void pop_front();
void pop_back();
void erase(iterator position);
void erase(iterator first, iterator last);

// специальные модифицирующие операции со списком:

void splice(iterator position, list<T, Allocator>& x);
void splice(iterator position, list<T, Allocator>& x,
            iterator i);
void splice(iterator position, list<T, Allocator>& x,
            iterator first, iterator last);
void remove(const T& value);
template <class Predicate> void remove_if(Predicate pred);
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate
            binary_pred);
void merge(list<T, Allocator>& x);
template <class Compare> void merge(list<T, Allocator>& x,
            Compare comp);
void reverse();
void sort();
template <class Compare> void sort(Compare comp);
};

template <class T, class Allocator>
bool operator==(const list<T, Allocator>& x, const list<T,
            Allocator>& y);

template <class T, class Allocator>
bool operator<(const list<T, Allocator>& x, const list<T,
            Allocator>& y);

```

iterator - двунаправленный итератор, ссылающийся на *T*. Точный тип зависит от исполнения и определяется в *Allocator*.

const_iterator - постоянный двунаправленный итератор, ссылающийся на *const T*. Точный тип зависит от исполнения и определяется в *Allocator*. Гарантируется, что имеется конструктор для *const_iterator* из *iterator*.

size_type - беззнаковый целочисленный тип. Точный тип зависит от исполнения и определяется в *Allocator*.

difference_type - знаковый целочисленный тип. Точный тип зависит от исполнения и определяется в *Allocator*.

insert не влияет на действительность итераторов и ссылок. Вставка единственного элемента в список занимает постоянное время, и ровно один раз вызывается конструктор копирования *T*. Вставка множественных элементов в список зависит линейно от числа вставленных элементов, а число вызовов конструктора копирования *T* точно равно числу вставленных элементов.

erase делает недействительными только итераторы и ссылки для стёртых элементов. Стирание единственного элемента - операция постоянного времени с единственным вызовом деструктора *T*. Стирание диапазона в списке занимает линейное время от размера диапазона, а число вызовов деструктора типа *T* точно равно размеру диапазона.

Так как списки позволяют быструю вставку и стирание в середине списка, то некоторые операции определяются специально для них:

list обеспечивает три операции стыковки, которые разрушительно перемещают элементы из одного списка в другой:

void splice(iterator position, list<T, Allocator>& x) вставляет содержимое *x* перед *position*, и *x* становится пустым. Требуется постоянное время. Результат не определён, если *&x == this*.

void splice(iterator position, list<T, Allocator>& x, iterator i) вставляет элемент, указываемый *i*, из списка *x* перед *position* и удаляет элемент из *x*. Требуется постоянное время. *i* - допустимый разыменовываемый итератор списка *x*. Результат не изменяется, если *position == i* или *position == ++i*.

void splice(iterator position, list<T, Allocator>& x, iterator first, iterator last) вставляет элементы из диапазона *[first, last)* перед *position* и удаляет элементы из *x*. Требуется постоянное время, если *&x == this*; иначе требуется линейное время. *[first, last)* - допустимый диапазон в *x*. Результат не определён, если *position* - итератор в диапазоне *[first, last)*.

remove стирает все элементы в списке, указанном итератором списка *i*, для которого выполняются следующие условия: **i == value*, *pred(*i) == true*. *remove* устойчиво, то есть относительный порядок элементов, которые не удалены, тот же самый, как их относительный порядок в первоначальном списке. Соответствующий предикат применяется точно *size()* раз.

unique стирает все, кроме первого элемента, из каждой последовательной группы равных элементов в списке. Соответствующий бинарный предикат применяется точно *size()* - 1 раз.

merge сливает список аргумента со списком (предполагается, что оба сортированы). Слияние устойчиво, то есть для равных элементов в двух списках элементы списка всегда предшествуют элементам из списка аргумента. *x* пуст после слияния. Выполняется, самое большее, *size() + x.size() - 1* сравнений.

reverse переставляет элементы в списке в обратном порядке. Операция линейного времени.

sort сортирует список согласно *operator<* или сравнивающему функциональному объекту. Она устойчива, то есть относительный порядок равных элементов сохраняется. Выполняется приблизительно $N \log N$ сравнений, где N равно *size()*.

Двусторонняя очередь (Deque)

deque - вид последовательности, которая, подобно вектору, поддерживает итераторы произвольного доступа. Кроме того она поддерживает операции вставки и стирания в начале или в конце за постоянное время; вставка и стирание в середине занимают линейное время. Как с векторами, управление памятью обрабатывается автоматически.

```
template <class T, template <class U> class Allocator = allocator>
class deque {
public:

// typedefs:

    typedef iterator;
    typedef const_iterator;
    typedef Allocator<T>::pointer pointer;
    typedef Allocator<T>::reference reference;
    typedef Allocator<T>::const_reference const_reference;
    typedef size_type;
    typedef difference_type;
    typedef T value_type;
    typedef reverse_iterator;
    typedef const_revcrse_iterator;

// размещение/удаление:

    deque();
    deque(size_type n, const T& value = T());
    deque(const deque<T, Allocator>& x);
    template <class InputIterator>
    deque(InputIterator first, InputIterator last);
    ~deque();
    deque<T, Allocator>& operator=(const deque<T, Allocator>& x);
    void swap(deque<T, Allocator>& x);

// средства доступа:

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;
    reverse_iterator rbegin();
    const_reverse_iterator rbegin();
    reverse_iterator rend();
    const_reverse_iterator rend();
    size_type size() const;
    size_type max_size() const;
    bool empty() const;
    reference operator[](size_type n);
    const_reference operator[](size_type n) const;
    reference front();
    const_reference front() const;
    reference back();
    const_reference back() const;

// вставка/стирание:

    void push_front(const T& x);
```

```
void push_back(const T& x);
iterator insert(iterator position, const T& x = T());
void insert(iterator position, size_type n, const T& x);
template <class InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);
void pop_front();
void pop_back();
void erase(iterator position);
void erase(iterator first, iterator last);
};

template <class T, class Allocator>
bool operator==(const deque<T, Allocator>& x, const deque<T,
    Allocator>& y);

template <class T, class Allocator>
bool operator<(const deque<T, Allocator>& x, const deque<T,
    Allocator>& y);
```

iterator - итератор произвольного доступа, ссылающийся на *T*. Точный тип зависит от исполнения и определяется в *Allocator*.

const_iterator - постоянный итератор произвольного доступа, ссылающийся на *const T*. Точный тип зависит от исполнения и определяется в *Allocator*. Гарантируется, что имеется конструктор для *const_iterator* из *iterator*.

size_type - беззнаковый целочисленный тип. Точный тип зависит от исполнения и определяется в *Allocator*.

difference_type - знаковый целочисленный тип. Точный зависит от исполнения и определяется в *Allocator*.

insert (вставка) в середину двусторонней очереди делает недействительными все итераторы и ссылки двусторонней очереди. *insert* и *push* (помещение) с обоих концов двусторонней очереди делают недействительными все итераторы двусторонней очереди, но не влияют на действительность всех ссылок на двустороннюю очередь. В худшем случае вставка единственного элемента в двустороннюю очередь занимает линейное время от минимума двух расстояний: от точки вставки - до начала и до конца двусторонней очереди. Вставка единственного элемента либо в начало, либо в конец двусторонней очереди всегда занимает постоянное время и вызывает единственный запрос конструктора копии *T*. То есть двусторонняя очередь особенно оптимизирована для помещения и извлечения элементов в начале и в конце.

erase (стирание) в середине двусторонней очереди делает недействительными все итераторы и ссылки двусторонней очереди. *erase* и *pop* (извлечение) с обоих концов двусторонней очереди делают недействительными только итераторы и ссылки на стёртый элемент. Число вызовов деструктора равно числу стёртых элементов, а число вызовов оператора присваивания равно минимуму из числа элементов перед стёртыми элементами и числа элементов после стёртых элементов.

Ассоциативные контейнеры (Associative containers)

Ассоциативные контейнеры обеспечивают быстрый поиск данных, основанных на ключах. Библиотека предоставляет четыре основных вида ассоциативных контейнеров: *set* (множество), *multiset* (множество с дубликатами), *map* (словарь) и *multimap* (словарь с дубликатами).

Все они берут в качестве параметров *Key* (ключ) и упорядочивающее отношение *Compare*, которое вызывает полное упорядочение по элементам *Key*. Кроме того, *map* и *multimap* ассоциируют произвольный тип *T* с *Key*. Объект типа *Compare* называется *сравнивающим объектом* (*comparison object*) контейнера.

В этом разделе, когда мы говорим о равенстве ключей, мы подразумеваем отношение эквивалентности, обусловленное сравнением и *не (not) operator==* для ключей. То есть считается, что два ключа *k1* и *k2* являются равными, если для сравнивающего объекта *comp* истинно *comp(k1, k2) == false && comp(k2, k1) == false*.

Ассоциативный контейнер поддерживает *уникальные ключи* (*unique keys*), если он может содержать, самое большее, один элемент для каждого значения ключа. Иначе он поддерживает *равные ключи* (*equal keys*). *set* и *map* поддерживают уникальные ключи. *multiset* и *multimap* поддерживают равные ключи.

Для *set* и *multiset* значимый тип - тот же самый, что и тип ключа. Для *map* и *multimap* он равен *pair<const Key, T>*.

iterator ассоциативного контейнера относится к категории двунаправленного итератора. *insert* не влияет на действительность итераторов и ссылок контейнера, а *erase* делает недействительными только итераторы и ссылки на стёртые элементы.

В следующей таблице обозначается: *X* - класс ассоциативного контейнера, *a* - значение *X*, *a_uniq* - значение *X*, когда *X* поддерживает уникальные ключи, а *a_eq* - значение *X*, когда *X* поддерживает многократные ключи, *i* и *j* удовлетворяют требованиям итераторов ввода и указывают на элементы *value_type*, [*i*, *j*) - допустимый диапазон, *p* - допустимый итератор для *a*, *q* - разыменовываемый итератор для *a*, [*q1*, *q2*) - допустимый диапазон в *a*, *t* - значение *X::value_type* и *k* - значение *X::key_type*.

Таблица 12. Требования ассоциативных контейнеров (в дополнение к контейнерам)

выражение	возвращаемый тип	утверждение/примечание состояние до/после	сложность
-----------	------------------	---	-----------

время компиляции
 время компиляции
 время компиляции
 постоянная
 постоянная
 вообще $N \log N$ (N -
 линейная, если $[i, j]$
 $value_comp()$
 то же, что выше
 постоянная
 постоянная
 логарифмическая
 логарифмическая
 вообще логарифми
 постоянной, если t
 вообще $N \log(size())$
 $);$
 линейная, если $[i, j]$
 $value_comp()$
 $\log(size()) + count(k$
 сводится к постоян
 $\log(size()) + N$, где N
 логарифмическая
 $\log(size()) + count(k$
 логарифмическая
 логарифмическая

	<code>pair<iterator, iterator>;</code>		
<code>a.equal_range(k)</code>	<code>pair<const_iterator, const_iterator></code> для константы <code>a</code>	эквивалент <code>make_pair(lower_bound(k), upper_bound(k))</code> .	логарифмическая

Основным свойством итераторов ассоциативных контейнеров является то, что они выполняют итерации через контейнеры в порядке неубывания ключей, где неубывание определено сравнением, которое использовалось для их создания. Для любых двух разыменованных итераторов i и j таких, что расстояние от i до j является положительным, `value_comp(*j, *i) == false`. Для ассоциативных контейнеров с уникальными ключами выдерживается более сильное условие `value_comp(*i, *j) == true`.

Множество (Set)

`set` - это ассоциативный контейнер, который поддерживает уникальные ключи (не содержит ключи с одинаковыми значениями) и обеспечивает быстрый поиск ключей.

```
template <class Key, class Compare = less<Key>,
          template <class U> class Allocator = allocator>
class set {
public:

// typedefs:
    typedef Key key_type;
    typedef Key value_type;
    typedef Allocator<Key>::pointer pointer;
    typedef Allocator<Key>::reference reference;
    typedef Allocator<Key>::const_reference const_reference;
    typedef Compare key_compare;
    typedef Compare value_compare;
    typedef iterator;
    typedef iterator const_iterator;
    typedef size_type;
    typedef difference_type;
    typedef reverse_iterator;
    typedef const_reverse_iterator;

// allocation/deallocation:
    set(const Compare& comp = Compare());
    template <class InputIterator>
    set(InputIterator first, InputIterator last,
        const Compare& comp = Compare());
    set(const set<Key, Compare, Allocator>& x);
    ~set();
    set<Key, Compare, Allocator>& operator=(const set<Key, Compare,
        Allocator>& x);
    void swap(set<Key, Compare, Allocator>& x);

// accessors:
    key_compare key_comp() const;
    value_compare value_comp() const;
    iterator begin() const;
    iterator end() const;
    reverse_iterator rbegin() const;
    reverse_iterator rend() const;
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

// insert/erase
    pair<iterator, bool> insert(const value_type& x);
    iterator insert(iterator position, const value_type& x);
    template <class InputIterator>
```

```

    void insert(InputIterator first, InputIterator last);
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// set operations:

    iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;
    iterator lower_bound(const key_type& x) const;
    iterator upper_bound(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(const set<Key, Compare, Allocator>& x,
               const set<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<(const set<Key, Compare, Allocator>& x,
              const set<Key, Compare, Allocator>& y);

```

iterator - постоянный двунаправленный итератор, указывающий на *const value_type*. Точный тип зависит от реализации и определяется в *Allocator*.

const_iterator - тот же самый тип, что и *iterator*.

size_type - целочисленный тип без знака. Точный тип зависит от реализации и определяется в *Allocator*.

difference_type - целочисленный тип со знаком. Точный тип зависит от реализации и определяется в *Allocator*.

Множество с дубликатами (Multiset)

multiset - это ассоциативный контейнер, который поддерживает равные ключи (возможно, содержит множественные копии того же самого значения ключа) и обеспечивает быстрый поиск ключей.

```

template <class Key, class Compare = less<Key>,
         template <class U> class Allocator = allocator>
class multiset {
public:

// typedefs:

    typedef Key key_type;
    typedef Key value_type;
    typedef Allocator<Key>::pointer pointer;
    typedef Aliocator<Key>::reference reference;
    typedef Allocator<Key>::const_reference const_reference;
    typedef Compare key_compare;
    typedef Compare value_compare;
    typedef iterator;
    typedef iterator const_iterator;
    typedef size_type;
    typedef difference_type;
    typedef reverse_iterator;
    typedef const_reverse_iterator;

// allocation/deallocation:

    multiset(const Compare& comp = Compare());
    template <class InputIterator>
    multiset(InputIterator first, InputIterator last,
             const Compare& comp == Compare());
    multiset(const multiset<Key, Compare, Allocator>& x);
    ~multiset();

```

```

    multiset<Key, Compare, Allocator>& operator=(const multiset<Key,
        Compare, Allocator>& x);
    void swap(multiset<Key, Compare, Allocator>& x);

// accessors:

    key_compare key_comp() const;
    value_compare value_comp() const;
    iterator begin() const;
    iterator end() const;
    reverse_iterator rbegin();
    revferse_iterator rend();
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

// insert/erase:

    iterator insert(const value_type& x);
    iterator insert(iterator position, const value_type& x);
    template <class InputIterator>
    void insert(InputIterator first, InputIterator last);
    void erase(iterator position);
    size_type erase(const key_type& x);
    void erase(iterator first, iterator last);

// multiset operations:

    iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;
    iterator lower_bound(const key_type& x) const;
    iterator upper_bound(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x) const;
};

template <class Key, class Compare, class Allocator>
bool operator==(const multiset<Key, Compare, Allocator>& x,
    const multiset<Key, Compare, Allocator>& y);

template <class Key, class Compare, class Allocator>
bool operator<(const multiset<Key, Compare, Allocator>& x,
    const multiset<Key, Compare, Allocator>& y);

```

iterator - постоянный двунаправленный итератор, указывающий на *const value_type*. Точный тип зависит от реализации и определяется в *Allocator*.

const_iterator - тот же самый тип, что и *iterator*.

size_type - целочисленный тип без знака. Точный тип зависит от реализации и определяется в *Allocator*.

difference_type - целочисленный тип со знаком. Точный тип зависит от реализации и определяется в *Allocator*.

Словарь (Map)

map - ассоциативный контейнер, который поддерживает уникальные ключи (не содержит ключи с одинаковыми значениями) и обеспечивает быстрый поиск значений другого типа *T*, связанных с ключами.

```

template <class Key, class T, class Compare = less<Key>,
    template <class U> class Allocator = allocator>
class map {
public:

// typedefs:

    typedef Key key_type;

```

```

typedef pair<const Key, T> value_type;
typedef Compare key_compare;
class value_compare
    : public binary_function<value_type, value_type, bool> {
friend class map;
protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
public:
    bool operator()(const value_type& x, const value_type& y) {
        return comp(x.first, y.first);
    }
};
typedef iterator;
typedef const_iterator;
typedef Allocator<value_type>::pointer pointer;
typedef Allocator<value_type>::reference reference;
typedef Allocator<value_type>::const_reference const_reference;
typedef size_type;
typedef difference_type;
typedef reverse_iterator;
typedef const_reverse_iterator;

// allocation/deallocation:

map(const Compare& comp = Compare());
template <class InputIterator>
map(InputIterator first, InputIterator last,
    const Compare& comp = Compare());
map(const map<Key, T, Compare, Allocator>& x);
~map();
map<Key, T, Compare, Allocator>&
    operator=(const map<Key, T, Compare, Allocator>& x);
void swap(map<Key, T, Compare, Allocator>& x);

// accessors:

key_compare key_comp() const;
value_compare value_comp() const;
iterator begin()
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin();
reverse_iterator rend();
const_reverse_iterator rend();
bool empty() const;
size_type size() const;
size_type max_size() const;
Allocator<T>::reference operator[](const key_type& x);

// insert/erase:

pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

// map operations:

iterator find(const key_type& x);

```



```

    const_iterator find(const key_type& x) const;
    size_type count(const key_type& x) const;
    iterator lower_bound(const key_type& x);
    const_iterator lower_bound(const key_type& x) const;
    iterator upper_bound(const key_type& x);
    const_iterator upper_bound(const key_type& x) const;
    pair<iterator, iterator> equal_range(const key_type& x);
    pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const map<Key, T, Compare, Allocator>& x,
               const map<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const map<Key, T, Compare, Allocator>& x,
              const map<Key, T, Compare, Allocator>& y);

```

iterator - двунаправленный итератор, указывающий на *value_type*. Точный тип зависит от реализации и определяется в *Allocator*.

const_iterator - постоянный двунаправленный итератор, указывающий на *const value_type*. Точный тип зависит от реализации и определяется в *Allocator*. Гарантируется, что имеется конструктор для *const_iterator* из *iterator*.

size_type - целочисленный тип без знака. Точный тип зависит от реализации и определяется в *Allocator*.

difference_type - целочисленный тип со знаком. Точный тип зависит от реализации и определяется в *Allocator*.

В дополнение к стандартному набору методов ассоциативных контейнеров, *map* обеспечивает операцию *Allocator<T>::reference operator[](const key_type&)*. Для словаря *m* и ключа *k* запись *m[k]* семантически эквивалентна *((m.insert(make_pair(k, T()))).first).second*.

Словарь с дубликатами (Multimap)

multimap - ассоциативный контейнер, который поддерживает равные ключи (возможно, содержит множественные копии того же самого значения ключа) и обеспечивает быстрый поиск значений другого типа *T*, связанных с ключами.

```

template <class Key, class T, class Compare = less<Key>,
         template <class U> class Allocator = allocator>
class multimap {
public:

    // typedefs:

    typedef Key key_type;
    typedef pair<const Key, T> value_type;
    typedef Compare key_compare;
    class value_compare
    : public binary_function<value_type, value_type, bool> {
    friend class multimap;
    protected:
        Compare comp;
        value_compare(Compare c) : comp(c) {}
    public:
        bool operator()(const value_type& x, const value_type& y) {
            return comp(x.first, y.first);
        }
    };
    typedef iterator;
    typedef const_iterator;
    typedef Allocator<value_type>::pointer pointer;
    typedef Allocator<value_type>::reference reference;
    typedef Allocator<value_type>::const_reference const_reference;
    typedef size_type;
    typedef difference_type;

```

```

typedef reverse_iterator;
typedef const_reverse_iterator;

// allocation/deallocation:

multimap(const Compare& comp = Compare());
template <class InputIterator>
multimap(InputIterator first, InputIterator last,
         const Compare& comp = Compare());
multimap(const multimap<Key, T, Compare, Allocator>& x);
~multimap();
multimap<Key, T, Compare, Allocator>&
operator=(const multimap<Key, T, Compare, Allocator>& x);
void swap(multimap<Key, T, Compare, Allocator>& x);

// accessors:

key_compare key_comp() const;
value_compare value_comp() const;
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin();
reverse_iterator rend();
const_reverse_iterator rend();
bool empty() const;
size_type size() const;
size_type max_size() const;

// insert/erase:

iterator insert(const value_type& x);
iterator insert(iterator position, const value_type& x);
template <class InputIterator>
void insert(InputIterator first, InputIterator last);
void erase(iterator position);
size_type erase(const key_type& x);
void erase(iterator first, iterator last);

// multimap operations:

iterator find(const key_type& x);
const_iterator find(const key_type& x) const;
size_type count(const key_type& x) const;
iterator lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;
iterator upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x) const;
pair<iterator, iterator> equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
};

template <class Key, class T, class Compare, class Allocator>
bool operator==(const multimap<Key, T, Compare, Allocator>& x,
               const multimap<Key, T, Compare, Allocator>& y);

template <class Key, class T, class Compare, class Allocator>
bool operator<(const multimap<Key, T, Compare, Allocator>& x,
              const multimap<Key, T, Compare, Allocator>& y);

```

iterator - двунаправленный итератор, указывающий на *value_type*. Точный тип зависит от реализации и определяется в *Allocator*.

const_iterator - постоянный двунаправленный итератор, указывающий на *value_type*. Точный тип зависит от реализации и определяется в *Allocator*. Гарантируется, что имеется конструктор для *const_iterator* из

iterator.

size_type - целочисленный тип без знака. Точный тип зависит от реализации и определяется в *Allocator*.

difference_type - целочисленный тип со знаком. Точный тип зависит от реализации и определяется в *Allocator*.

ИТЕРАТОРЫ ПОТОКОВ

Чтобы шаблоны алгоритмов могли работать непосредственно с потоками ввода-вывода, предусмотрены соответствующие шаблонные классы, подобные итераторам. Например,

```
partial_sum_copy(istream_iterator<double>(cin), istream_iterator<double>(),
    ostream_iterator<double>(cout, "\n"));
```

читает файл, содержащий числа с плавающей запятой, из *cin* и печатает частичные суммы в *cout*.

Итератор входного потока (Istream Iterator)

istream_iterator<T> читает (используя *operator>>*) последовательные элементы из входного потока, для которого он был создан. После своего создания итератор каждый раз при использовании ++ читает и сохраняет значение *T*. Если достигнут конец потока (*operator void* ()* в потоке возвращает *false*), итератор становится равным значению *end-of-stream* (*конец-потока*). Конструктор без параметров *istream_iterator()* всегда создаёт итераторный объект конца потокового ввода, являющийся единственным законным итератором, который следует использовать для конечного условия. Результат *operator** для конца потока не определён, а для любого другого значения итератора возвращается *const T&*.

Невозможно записывать что-либо с использованием входных итераторов. Основная особенность входных итераторов - тот факт, что операторы ++ не сохраняют равенства, то есть *i == j* не гарантирует вообще, что ++ *i* == ++ *j*. Каждый раз, когда ++ используется, читается новое значение. Практическое следствие этого факта - то, что входные итераторы могут использоваться только для однопроходных алгоритмов, что действительно имеет здравый смысл, так как многопроходным алгоритмам всегда более соответствует использование структур данных в оперативной памяти.

Два итератора *конец-потока* всегда равны. Итератор *конец-потока* не равен *не-конец-потока* итератору. Два *не-конец-потока* итератора равны, когда они созданы из того же самого потока.

```
template <class T, class Distance = ptrdiff_t>
class istream_iterator : public input_iterator<T, Distance> {
friend bool operator==(const istream_iterator<T, Distance>& x,
    const istream_iterator<T, Distance>& y);
public:
    istream_iterator();
    istream_iterator(istream& s);
    istream_iterator(const istream_iterator<T, Distance>& x);
    ~istream_iterator();
    const T& operator*() const;
    istream_iterator<T, Distance>& operator++();
    istream_iterator<T, Distance> operator++(int);
};

template <class T, class Distance>
bool operator==(const istream_iterator<T, Distance>& x,
    const istream_iterator<T, Distance>& y);
```

Итератор выходного потока (Ostream Iterator)

ostream_iterator<T> записывает (используя *operator<<*) последовательные элементы в выходной поток, из которого он был создан. Если он был создан с параметром конструктора *char**, эта строка, называемая *строкой разделителя* (*delimiter string*), записывается в поток после того, как записывается каждое *T*. Невозможно с помощью выходного итератора получить значение. Его единственное использование - выходной итератор в ситуациях, подобных нижеследующему:

```
while (first != last) *result++ = *first++;
```

ostream_iterator определён как:

```
template <class T>
```

```

class ostream_iterator : public output_iterator {
public:
    ostream_iterator(ostream& s);
    ostream_iterator(ostream& s, const char* delimiter);
    ostream_iterator(const ostream_iterator<T>& x);
    ~ostream_iterator();
    ostream_iterator<T>& operator=(const T& value);
    ostream_iterator<T>& operator*();
    ostream_iterator<T>& operator++();
    ostream_iterator<T>& operator++(int);
};

```

АЛГОРИТМЫ

Все алгоритмы отделены от деталей реализации структур данных и используют в качестве параметров типы итераторов. Поэтому они могут работать с определяемыми пользователем структурами данных, когда эти структуры данных имеют типы итераторов, удовлетворяющие предположениям в алгоритмах.

Для некоторых алгоритмов предусмотрены и оперативные и копирующие версии. Решение, включать ли копирующую версию, было обычно основано на рассмотрении сложности. Когда стоимость выполнения операции доминирует над стоимостью копии, копирующая версия не включена. Например, *sort_copy* не включена, так как стоимость сортировки намного значительнее, и пользователи могли бы также делать *copy* перед *sort*. Когда такая версия предусмотрена для какого-то алгоритма *algorithm*, он называется *algorithm_copy*. Алгоритмы, которые берут предикаты, оканчиваются суффиксом *_if* (который следует за суффиксом *_copy*).

Класс *Predicate* используется всякий раз, когда алгоритм ожидает функциональный объект, при применении которого к результату разыменования соответствующего итератора возвращается значение, обратимое в *bool*. Другими словами, если алгоритм берёт *Predicate pred* как свой параметр и *first* как свой параметр итератора, он должен работать правильно в конструкции *if (pred(*first)) {...}*. Предполагается, что функциональный объект *pred* не применяет какую-либо непостоянную функцию для разыменованного итератора.

Класс *BinaryPredicate* используется всякий раз, когда алгоритм ожидает функциональный объект, который при его применении к результату разыменования двух соответствующих итераторов или к разыменованию итератора и типа *T*, когда *T* - часть сигнатуры, возвращает значение, обратимое в *bool*. Другими словами, если алгоритм берёт *BinaryPredicate binary_pred* как свой параметр и *first1* и *first2* как свои параметры итераторов, он должен работать правильно в конструкции *if (binary_pred(*first, *first2)) {...}*. *BinaryPredicate* всегда берёт тип первого итератора как свой первый параметр, то есть в тех случаях, когда *T value* - часть сигнатуры, он должен работать правильно в контексте *if (binary_pred (*first, value)) {...}*. Ожидается, что *binary_pred* не будет применять какую-либо непостоянную функцию для разыменованных итераторов.

В описании алгоритмов операторы *+* и *-* используются для некоторых категорий итераторов, для которых они не должны быть определены. В этих случаях семантика *a+n* такая же, как семантика *{X tmp = a; advance(tmp, n); return tmp;}*, а семантика *a-b* такая же, как семантика *{Distance n; distance(a, b, n); return n;}*.

Не меняющие последовательность операции (Non-mutating sequence operations)

Операции с каждым элементом (For each)

```

template <class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);

```

for_each применяет *f* к результату разыменования каждого итератора в диапазоне *[first, last)* и возвращает *f*. Принято, что *f* не применяет какую-то непостоянную функцию к разыменованному итератору. *f* применяется точно *last-first* раз. Если *f* возвращает результат, результат игнорируется.

Найти (Find)

```

template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);

template <class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);

```

find возвращает первый итератор *i* в диапазоне *[first, last)*, для которого соблюдаются следующие соответствующие условия: **i == value*, *pred (*i) == true*. Если такой итератор не найден, возвращается *last*. Соответствующий предикат применяется точно *find(first, last, value) - first* раз.

Найти рядом (Adjacent find)

```
template <class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                             BinaryPredicate binary_pred);
```

adjacent_find возвращает первый итератор *i* такой, что *i* и *i+1* находятся в диапазоне *[first, last)* и для которого соблюдаются следующие соответствующие условия: **i == *(i + 1)*, *binary_pred(*i, *(i + 1)) == true*. Если такой итератор *i* не найден, возвращается *last*. Соответствующий предикат применяется, самое большее, *max((last - first) - 1, 0)* раз.

Подсчет (Count)

```
template <class InputIterator, class T, class Size>
void count(InputIterator first, InputIterator last, const T& value, Size& n);

template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last, Predicate pred, Size& n);
```

count добавляет к *n* число итераторов *i* в диапазоне *[first, last)*, для которых соблюдаются следующие соответствующие условия: **i == value*, *pred(*i) == true*. Соответствующий предикат применяется точно *last - first* раз.

count должен сохранять результат в параметре ссылки вместо того, чтобы возвращать его, потому что тип размера не может быть выведен из встроенных типов итераторов, как, например, *int**.

Отличие (Mismatch)

```
template <class InputIterator1, class InputIterator2>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
                                             InputIterator1 last1, InputIterator2 first2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
                                             InputIterator1 last1, InputIterator2 first2,
                                             BinaryPredicate binary_pred);
```

mismatch возвращает пару итераторов *i* и *j* таких, что *j == first2 + (i - first1)* и *i* является первым итератором в диапазоне *[first1, last1)*, для которого следующие соответствующие условия выполнены: *!(*i == *(first2 + (i - first1)))*, *binary_pred(*i, *(first2 + (i - first1))) == false*. Если такой итератор *i* не найден, пара *last1* и *first2 + (last1 - first1)* возвращается. Соответствующий предикат применяется, самое большее, *last1 - first1* раз.

Сравнение на равенство (Equal)

```
template <class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

template <class InputIterator1, class InputIterator2, class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
           BinaryPredicate binary_pred);
```

equal возвращает *true*, если для каждого итератора *i* в диапазоне *[first1, last1)* выполнены следующие соответствующие условия: **i == *(first2 + (i - first1))*, *binary_pred(*i, *(first2 + (i - first1))) == true*. Иначе *equal* возвращает *false*. Соответствующий предикат применяется, самое большее, *last1 - first1* раз.

Поиск подпоследовательности (Search)

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                       ForwardIterator2 first2, ForwardIterator2 last2);
```

```
template <class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                      ForwardIterator2 first2, ForwardIterator2 last2,
                      BinaryPredicate binary_pred);
```

search находит подпоследовательность равных значений в последовательности. *search* возвращает первый итератор *i* в диапазоне $[first1, last1 - (last2 - first2))$ такой, что для любого неотрицательного целого числа *n*, меньшего чем *last2 - first2*, выполнены следующие соответствующие условия: $*(i + n) == *(first2 + n)$, $binary_pred(*(i + n), *(first2 + n)) == true$. Если такой итератор не найден, возвращается *last1*. Соответствующий предикат применяется, самое большее, $(last1 - first1) * (last2 - first2)$ раз. Квадратичное поведение, однако, является крайне маловероятным.

Меняющие последовательность операции (Mutating sequence operations)

Копировать (Copy)

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                  OutputIterator result);
```

copy копирует элементы. Для каждого неотрицательного целого числа *n* < (*last - first*) выполняется присваивание $*(result + n) = *(first + n)$. Точно делается *last - first* присваиваний. Результат *copy* не определён, если *result* находится в диапазоне $[first, last)$.

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                    BidirectionalIterator1 last, BidirectionalIterator2 result);
```

copy_backward копирует элементы в диапазоне $[first, last)$ в диапазон $[result - (last - first), result)$, начиная от *last-1* и продолжая до *first*. Его нужно использовать вместо *copy*, когда *last* находится в диапазоне $[result - (last - first), result)$. Для каждого положительного целого числа *n* ≤ (*last - first*) выполняется присваивание $*(result - n) = *(last - n)$. *copy_backward* возвращает *result - (last - first)*. Точно делается *last - first* присваиваний. Результат *copy_backward* не определён, если *result* находится в диапазоне $[first, last)$.

Обменять (Swap)

```
template <class T>
void swap(T& a, T& b);
```

swap обменивает значения, хранимые в двух местах.

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

iter_swap обменивает значения, указанные двумя итераторами *a* и *b*.

```
template <class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                           ForwardIterator1 last1, ForwardIterator2 first2);
```

Для каждого неотрицательного целого числа *n* < (*last1 - first1*) выполняется перестановка: $swap(*(first1 + n), *(first2 + n))$. *swap_ranges* возвращает *first2 + (last1 - first1)*. Выполняется точно *last1 - first1* перестановок. Результат *swap_ranges* не определён, если два диапазона $[first1, last1)$ и $[first2, first2 + (last1 - first1))$ перекрываются.

Преобразовать (Transform)

```
template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                      OutputIterator result, UnaryOperation op);
```



```
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, OutputIterator result,
                        BinaryOperation binary_op);
```

transform присваивает посредством каждого итератора *i* в диапазоне *[result, result + (last1 - first1))* новое соответствующее значение, равное *op(* (first1 + (i - result))* или *binary_op(*(first1 + (i - result)), *(first2 + (i - result)))*. *transform* возвращает *result + (last1 - first1)*. Применяются *op* или *binary_op* точно *last1 - first1* раз. Ожидается, что *op* и *binary_op* не имеют каких-либо побочных эффектов. *result* может быть равен *first* в случае унарного преобразования или *first1* либо *first2* в случае бинарного.

Заменить (Replace)

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T& old_value,
            const T& new_value);

template <class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last, Predicate pred,
               const T& new_value);
```

replace заменяет элементы, указанные итератором *i* в диапазоне *[first, last)*, значением *new_value*, когда выполняются следующие соответствующие условия: **i == old_value, pred(*i) == true*. Соответствующий предикат применяется точно *last - first* раз.

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
                           OutputIterator result, const T& old_value, const T& new_value);

template <class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
                              OutputIterator result, Predicate pred, const T& new_value);
```

replace_copy присваивает каждому итератору *i* в диапазоне *[result, result + (last - first))* значение *new_value* или **(first + (i - result))* в зависимости от выполнения следующих соответствующих условий: **(first + (i - result)) == old_value, pred(*(first + (i - result))) == true*. *replace_copy* возвращает *result + (last - first)*. Соответствующий предикат применяется точно *last - first* раз.

Заполнить (Fill)

```
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);

template <class OutputIterator, class Size, class T>
OutputIterator fill_n(OutputIterator first, Size n, const T& value);
```

fill присваивает значения через все итераторы в диапазоне *[first, last)* или *[first, first + n)*. *fill_n* возвращает *first + n*. Точно делается *last - first* (или *n*) присваиваний.

Породить (Generate)

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
             Generator gen);

template <class OutputIterator, class Size, class Generator>
OutputIterator generate_n(OutputIterator first, Size n, Generator gen);
```

generate вызывает функциональный объект *gen* и присваивает возвращаемое *gen* значение через все итераторы в диапазоне *[first, last)* или *[first, first + n)*. *gen* не берёт никакие параметры. *generate_n* возвращает *first + n*. Точно выполняется *last - first* (или *n*) вызовов *gen* и присваиваний.

Удалить (Remove)


```
template <class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
    const T& value);

template <class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
    Predicate pred);
```

remove устраняет все элементы, указываемые итератором *i* в диапазоне *[first, last)*, для которых выполнены следующие соответствующие условия: **i == value*, *pred (*i) == true*. *remove* возвращает конец возникающего в результате своей работы диапазона. *remove* устойчив, то есть относительный порядок элементов, которые не удалены, такой же, как их относительный порядок в первоначальном диапазоне. Соответствующий предикат применяется точно *last - first* раз.

```
template <class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
    OutputIterator result, const T& value);

template <class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
    OutputIterator result, Predicate pred);
```

remove_copy копирует все элементы, указываемые итератором *i* в диапазоне *[first, last)*, для которых не выполнены следующие соответствующие условия: **i == value*, *pred (*i) == true*. *remove_copy* возвращает конец возникающего в результате своей работы диапазона. *remove_copy* устойчив, то есть относительный порядок элементов в результирующем диапазоне такой же, как их относительный порядок в первоначальном диапазоне. Соответствующий предикат применяется точно *last - first* раз.

Убрать повторы (Unique)

```
template <class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
    BinaryPredicate binary_pred);
```

unique устраняет все, кроме первого, элементы из каждой последовательной группы равных элементов, указываемые итератором *i* в диапазоне *[first, last)*, для которых выполнены следующие соответствующие условия: **i == *(i - 1)* или *binary_pred(*i, *(i - 1)) == true*. *unique* возвращает конец возникающего в результате диапазона. Соответствующий предикат применяется точно *(last - first) - 1* раз.

```
template <class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutputIterator,
    class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
    OutputIterator result, BinaryPredicate binary_pred);
```

unique_copy копирует только первый элемент из каждой последовательной группы равных элементов, указываемых итератором *i* в диапазоне *[first, last)*, для которых выполнены следующие соответствующие условия: **i == *(i - 1)* или *binary_pied(*i, *(i - 1)) == true*. *unique_copy* возвращает конец возникающего в результате диапазона. Соответствующий предикат применяется точно *(last - first) - 1* раз.

Расположить в обратном порядке (Reverse)

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first,
    BidirectionalIterator last);
```

Для каждого неотрицательного целого числа *i* $i \leq (last - first)/2$ функция *reverse* применяет перестановку ко всем парам итераторов *first + i*, *(last - i) - 1*. Выполняется точно *(last - first)/2* перестановок.

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                           BidirectionalIterator last, OutputIterator result);
```

reverse_copy копирует диапазон $[first, last)$ в диапазон $[result, result + (last - first))$ такой, что для любого неотрицательного целого числа $i < (last - first)$ происходит следующее присваивание: $*(result + (last - first) - i) = *(first + i)$. *reverse_copy* возвращает $result + (last - first)$. Делается точно $last - first$ присваиваний. Результат *reverse_copy* не определён, если $[first, last)$ и $[result, result + (last - first))$ перекрываются.

Переместить по кругу (Rotate)

```
template <class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

Для каждого неотрицательного целого числа $i < (last - first)$ функция *rotate* помещает элемент из позиции $first + i$ в позицию $first + (i + (last - middle)) \% (last - first)$. $[first, middle)$ и $[middle, last)$ - допустимые диапазоны. Максимально выполняется $last - first$ перестановок.

```
template <class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                           ForwardIterator last, OutputIterator result);
```

rotate_copy копирует диапазон $[first, last)$ в диапазон $[result, result + (last - first))$ такой, что для каждого неотрицательного целого числа $i < (last - first)$ происходит следующее присваивание: $*(result + (i + (last - middle)) \% (last - first)) = *(first + i)$. *rotate_copy* возвращает $result + (last - first)$. Делается точно $last - first$ присваиваний. Результат *rotate_copy* не определён, если $[first, last)$ и $[result, result + (last - first))$ перекрываются.

Перетасовать (Random shuffle)

```
template <class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                    RandomNumberGenerator& rand);
```

random_shuffle переставляет элементы в диапазоне $[first, last)$ с равномерным распределением. Выполняется точно $last - first$ перестановок. *random_shuffle* может брать в качестве параметра особый генерирующий случайное число функциональный объект *rand* такой, что *rand* берёт положительный параметр n типа расстояния *RandomAccessIterator* и возвращает случайно выбранное значение между 0 и $n-1$.

Разделить (Partitions)

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                               BidirectionalIterator last, Predicate pred);
```

partition помещает все элементы в диапазоне $[first, last)$, которые удовлетворяют *pred*, перед всеми элементами, которые не удовлетворяют. Возвращается итератор i такой, что для любого итератора j в диапазоне $[first, i)$ будет $pred(*j) == true$, а для любого итератора k в диапазоне $[i, last)$ будет $pred(*k) == false$. Делается максимально $(last - first)/2$ перестановок. Предикат применяется точно $last - first$ раз.

```
template <class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last, Predicate pred);
```

stable_partition помещает все элементы в диапазоне $[first, last)$, которые удовлетворяют *pred*, перед всеми элементами, которые не удовлетворяют. Возвращается итератор i такой, что для любого итератора j в диапазоне $[first, i)$ будет $pred(*j) == true$, а для любого итератора k в диапазоне $[i, last)$ будет $pred(*k) == false$. Относительный порядок элементов в обеих группах сохраняется. Делается максимально $(last - first) * \log(last - first)$ перестановок, но только линейное число перестановок, если имеется достаточная дополнительная память. Предикат применяется точно $last - first$ раз.

Операции сортировки и отношения (Sorting and related operations)

Все операции в этом разделе имеют две версии: одна берёт в качестве параметра функциональный объект типа *Compare*, а другая использует *operator<* .

Compare - функциональный объект, который возвращает значение, обратимое в *bool*. *Compare comp* используется полностью для алгоритмов, принимающих отношение упорядочения. *comp* удовлетворяет стандартным аксиомам для полного упорядочения и не применяет никакую непостоянную функцию к разыменованному итератору. Для всех алгоритмов, которые берут *Compare*, имеется версия, которая использует *operator<* взамен. То есть *comp(*i, *j) == true* по умолчанию для **i < *j == true*.

Последовательность сортируется относительно компаратора *comp*, если для любого итератора *i*, указывающего на элемент в последовательности, и любого неотрицательного целого числе *n* такого, что *i + n* является допустимым итератором, указывающим на элемент той же самой последовательности, *comp(*(i + n), *i) == false*.

В описаниях функций, которые имеют дело с упорядочивающими отношениями, мы часто используем представление равенства, чтобы описать такие понятия, как устойчивость. Равенство, к которому мы обращаемся, не обязательно *operator==*, а отношение равенства стимулируется полным упорядочением. То есть два элемента *a* и *b* считаются равными, если и только если *!(a < b) && !(b < a)*.

Сортировка (Sort)

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

sort сортирует элементы в диапазоне *[first, last)*. Делается приблизительно *NlogN* (где *N* равняется *last - first*) сравнений в среднем. Если режим наихудшего случая важен, должны использоваться *stable_sort* или *partial_sort*.

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

stable_sort сортирует элементы в диапазоне *[first, last)*. Он устойчив, то есть относительный порядок равных элементов сохраняется. Делается максимум *N(logN)²* (где *N* равняется *last - first*) сравнений; если доступна достаточная дополнительная память, тогда это - *NlogN*.

```
template <class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                 RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                 RandomAccessIterator last, Compare comp);
```

partial_sort помещает первые *middle - first* отсортированных элементов из диапазона *[first, last)* в диапазон *[first, middle)*. Остальная часть элементов в диапазоне *[middle, last)* помещена в неопределённом порядке. Берётся приблизительно *(last - first) * log(middle - first)* сравнений.

```
template <class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first,
                                     InputIterator last, RandomAccessIterator result_first,
                                     RandomAccessIterator result_last);

template <class InputIterator, class RandomAccessIterator,
         class Compare>
RandomAccessIterator partial_sort_copy(InputIterator first,
                                     InputIterator last, RandomAccessIterator result_first,
                                     RandomAccessIterator result_last, Compare comp);
```

partial_sort_copy помещает первые *min(last - first, result_last - result_first)* отсортированных элементов в диапазон *[result_first, result_first + min(last - first, result_last - result_first))*. Возвращается или *result_last*, или *result_first + (last - first)*, какой меньше. Берётся приблизительно *(last - first) * log(min(last - first, result_last - result_first))* сравнений.

N-й элемент (Nth element)

```
template <class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

После операции *nth_element* элемент в позиции, указанной *nth*, является элементом, который был бы в той позиции, если бы сортировался целый диапазон. Также для любого итератора *i* в диапазоне *[first, nth)* и любого итератора *j* в диапазоне *[nth, last)* считается, что *!(*i* > *j*)* или *comp(*i*, *j*) == false*. Операция линейна в среднем.

Двоичный поиск (Binary search)

Все алгоритмы в этом разделе - версии двоичного поиска. Они работают с итераторами не произвольного доступа, уменьшая число сравнений, которое будет логарифмическим для всех типов итераторов. Они особенно подходят для итераторов произвольного доступа, так как эти алгоритмы делают логарифмическое число шагов в структуре данных. Для итераторов не произвольного доступа они выполняют линейное число шагов.

```
template <class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                           const T& value);

template <class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first,
                           ForwardIterator last, const T& value, Compare comp);
```

lower_bound находит первую позицию, в которую *value* может быть вставлено без нарушения упорядочения. *lower_bound* возвращает самый дальний итератор *i* в диапазоне *[first, last)* такой, что для любого итератора *j* в диапазоне *[first, i)* выполняются следующие соответствующие условия: **j < value* или *comp(*j, value) == true*. Делается максимум *log(last - first) + 1* сравнений.

```
template <class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first,
                           ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first,
                           ForwardIterator last, const T& value, Compare comp);
```

upper_bound находит самую дальнюю позицию, в которую *value* может быть вставлено без нарушения упорядочения. *upper_bound* возвращает самый дальний итератор *i* в диапазоне *[first, last)* такой, что для любого итератора *j* в диапазоне *[first, i)* выполняются следующие соответствующие условия: *!(value < *j)* или *comp(value, *j) == false*. Делается максимум *log(last - first) + 1* сравнений.

```
template <class ForwardIterator, class T>
ForwardIterator equal_range(ForwardIterator first,
                           ForwardIterator last, const T& value);

template <class ForwardIterator, class T, class Compare>
ForwardIterator equal_range(ForwardIterator first,
                           ForwardIterator last, const T& value,
                           Compare comp);
```

equal_range находит самый большой поддиапазон *[i, j)* такой, что значение может быть вставлено по любому итератору *k* в нём. *k* удовлетворяет соответствующим условиям: *!(*k* < value) && !(value < *k)* или *comp(*k, value) == false && comp(value, *k) == false*. Делается максимум *2 * log(last - first) + 1* сравнений.

```
template <class ForwardIterator, class T>
ForwardIterator binary_search(ForwardIterator first,
                             ForwardIterator last, const T& value);
```

```
template <class ForwardIterator, class T, class Compare>
ForwardIterator binary_search(ForwardIterator first,
                             ForwardIterator last, const T& value,
```

binary_search возвращает истину, если в диапазоне *[first, last)* имеется итератор *i*, который удовлетворяет соответствующим условиям: *!(*i < value) && !(value < *i)* или *comp(*i, value) == false && comp(value, *i) == false*. Делается максимум $\log(last - first) + 2$ сравнений.

Объединение (Merge)

```
template <class InputIterator1, class Input Iterator2,
         class OutputIterator>
OutputIterator merge(InputIterator1 first1,
                    InputIterator1 last1, InputIterator2 first2,
                    InputIterator2 last2, OutputIterator result);

template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1,
                    InputIterator1 last1, InputIterator2 first2,
                    InputIterator2 last2, OutputIterator result,
                    Compare comp);
```

merge объединяет два сортированных диапазона *[first1, last1)* и *[first2, last2)* в диапазон *[result, result + (last1 - first1) + (last2 - first2))*. Объединение устойчиво, то есть для равных элементов в двух диапазонах элементы из первого диапазона всегда предшествуют элементам из второго. *merge* возвращает *result + (last1 - first1) + (last2 - first2)*. Выполняется максимально $(last1 - first1) + (last2 - first2) - 1$ сравнений. Результат *merge* не определён, если возникающий в результате диапазон перекрывается с любым из первоначальных диапазонов.

```
template <class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first,
                  BidirectionalIterator middle,
                  BidirectionalIterator last,
                  Compare comp);
```

inplace_merge объединяет два сортированных последовательных диапазона *[first, middle)* и *[middle, last)*, помещая результат объединения в диапазон *[first, last)*. Объединение устойчиво, то есть для равных элементов в двух диапазонах элементы из первого диапазона всегда предшествуют элементам из второго. Когда доступно достаточно дополнительной памяти, выполняется максимально $(last - first) - 1$ сравнений. Если никакая дополнительная память не доступна, может использоваться алгоритм со сложностью $O(N \log N)$.

Операции над множеством для сортированных структур (Set operations on sorted structures)

Этот раздел определяет все основные операции над множеством для сортированных структур. Они даже работают с множествами с дубликатами, содержащими множественные копии равных элементов. Семантика операций над множеством обобщена на множества с дубликатами стандартным способом, определяя объединение, содержащее максимальное число местонахождений каждого элемента, пересечение, содержащее минимум, и так далее.

```
template <class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2,
         class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
              InputIterator2 first2, InputIterator2 last2,
              Compare comp);
```

includes возвращает *true*, если каждый элемент в диапазоне *[first2, last2)* содержится в диапазоне *[first1, last1)*. Иначе возвращается *false*. Выполняется максимально $((last1 - first1) + (last2 - first2)) * 2 - 1$ сравнений.

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_union(InputIterator1 first1,
                       InputIterator1 last1, InputIterator2 first2,
                       InputIterator2 last2, OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1,
                       InputIterator1 last1, InputIterator2 first2,
                       InputIterator2 last2, OutputIterator result,
                       Compare comp);
```

set_union создаёт сортированное объединение элементов из двух диапазонов. Он возвращает конец созданного диапазона. *set_union* устойчив, то есть, если элемент присутствует в обоих диапазонах, он копируется из первого диапазона. Выполняется максимально $((last1 - first1) + (last2 - first2)) * 2 - 1$ сравнений. Результат *set_union* не определён, если возникающий в результате диапазон перекрывается с любым из первоначальных диапазонов.

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1,
                              InputIterator1 last1, InputIterator2 first2,
                              InputIterator2 last2, OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1,
                              InputIterator1 last1, InputIterator2 first2,
                              InputIterator2 last2, OutputIterator result,
                              Compare comp);
```

set_intersection создаёт сортированное пересечение элементов из двух диапазонов. Он возвращает конец созданного диапазона. Гарантируется, что *set_intersection* устойчив, то есть, если элемент присутствует в обоих диапазонах, он копируется из первого диапазона. Выполняется максимально $((last1 - first1) + (last2 - first2)) * 2 - 1$ сравнений. Результат *set_intersection* не определён, если возникающий в результате диапазон перекрывается с любым из первоначальных диапазонов.

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_difference(InputIterator1 first1,
                             InputIterator1 last1, InputIterator2 first2,
                             InputIterator2 last2, OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1,
                             InputIterator1 last1, InputIterator2 first2,
                             InputIterator2 last2, OutputIterator result,
                             Compare comp);
```

set_difference создаёт сортированную разность элементов из двух диапазонов. Он возвращает конец созданного диапазона. Выполняется максимально $((last1 - first1) + (last2 - first2)) * 2 - 1$ сравнений. Результат *set_difference* не определён, если возникающий в результате диапазон перекрывается с любым из первоначальных диапазонов.

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator>
OutputIterator set_symmetric_difference(InputIterator1 first1,
                                       InputIterator1 last1, InputIterator2 first2,
                                       InputIterator2 last2, OutputIterator result);
```

```
template <class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1,
```



```
InputIterator1 last1, InputIterator2 first2,
InputIterator2 last2, OutputIterator result,
Compare comp);
```

set_symmetric_difference создаёт отсортированную симметричную разность элементов из двух диапазонов. Он возвращает конец созданного диапазона. Выполняется максимально $((last1 - first1) + (last2 - first2)) * 2 - 1$ сравнений. Результат *set_symmetric_difference* не определён, если возникающий в результате диапазон перекрывается с любым из первоначальных диапазонов.

Операции над пирамидами (Heap operations)

Пирамида - специфическая организация элементов в диапазоне между двумя итераторами произвольного доступа $[a, b)$. Два её ключевые свойства: (1) $*a$ - самый большой элемент в диапазоне, (2) $*a$ может быть удалён с помощью *pop_heap* или новый элемент добавлен с помощью *push_heap* за $O(\log N)$ время. Эти свойства делают пирамиды полезными для приоритетных очередей. *make_heap* преобразовывает диапазон в пирамиду, а *sort_heap* превращает пирамиду в отсортированную последовательность.

```
template <class RandomAccessIterator>
void push_heap(RandomAccessIterator first,
RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

push_heap полагает, что диапазон $[first, last - 1)$ является соответствующей пирамидой, и надлежащим образом помещает значение с позиции $last - 1$ в результирующую пирамиду $[first, last)$. Выполняется максимально $\log(last - first)$ сравнений.

```
template <class RandomAccessIterator>
void pop_heap(RandomAccessIterator first,
RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first,
RandomAccessIterator last,
Compare comp);
```

pop_heap полагает, что диапазон $[first, last)$ является соответствующей пирамидой, затем обменивает значения в позициях $first$ и $last - 1$ и превращает $[first, last - 1)$ в пирамиду. Выполняется максимально $2 * \log(last - first)$ сравнений.

```
template <class RandomAccessIterator>
void make_heap(RandomAccessIterator first,
RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first,
RandomAccessIterator last,
Compare comp);
```

make_heap создает пирамиду из диапазона $[first, last)$. Выполняется максимально $3 * (last - first)$ сравнений.

```
template <class RandomAccessIterator>
void sort_heap(RandomAccessIterator first,
RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first,
RandomAccessIterator last, Compare comp);
```

sort_heap сортирует элементы в пирамиде $[first, last)$. Выполняется максимально $N \log N$ сравнений, где N равно $last - first$. *sort_heap* не устойчив.

Минимум и максимум (Minimum and maximum)


```

template <class T>
const T& min(const T& a, const T& b);

template <class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);

template <class T>
const T& max(const T& a, const T& b);

template <class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);

```

min возвращает меньшее, а *max* большее. *min* и *max* возвращают первый параметр, когда их параметры равны.

```

template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first,
                           ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first,
                           ForwardIterator last, Compare comp);

```

max_element возвращает первый такой итератор *i* в диапазоне *[first, last)*, что для любого итератора *j* в диапазоне *[first, last)* выполняются следующие соответствующие условия: *!(*i < *j)* или *comp(*i, *j) == false*. Выполняется точно *max((last - first) - 1, 0)* соответствующих сравнений.

```

template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first,
                           ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first,
                           ForwardIterator last, Compare comp);

```

min_element возвращает первый такой итератор *i* в диапазоне *[first, last)*, что для любого итератора *j* в диапазоне *[first, last)* выполняются следующие соответствующие условия: *!(*j < *i)* или *comp(*j, *i) == false*. Выполняется точно *max((last - first) - 1, 0)* соответствующих сравнений.

Лексикографическое сравнение (Lexicographical comparison)

```

template <class InputIterator1, class InputIterator2>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2);

template <class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                             InputIterator2 first2, InputIterator2 last2, Compare comp);

```

lexicographical_compare возвращает *true*, если последовательность элементов, определённых диапазоном *[first1, last1)*, лексикографически меньше, чем последовательность элементов, определённых диапазоном *[first2, last2)*. Иначе он возвращает ложь. Выполняется максимум $2 * \min((last1 - first1), (last2 - first2))$ сравнений.

Генераторы перестановок (Permutation generators)

```

template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                     BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,

```

```
BidirectionalIterator last, Compare comp);
```

next_permutation берёт последовательность, определённую диапазоном *[first, last)*, и трансформирует её в следующую перестановку. Следующая перестановка находится, полагая, что множество всех перестановок лексикографически сортировано относительно *operator<* или *comp*. Если такая перестановка существует, возвращается *true*. Иначе он трансформирует последовательность в самую маленькую перестановку, то есть сортированную по возрастанию, и возвращает *false*. Максимально выполняется $(last - first)/2$ перестановок.

```
template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last);

template <class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
    BidirectionalIterator last, Compare comp);
```

prev_permutation берёт последовательность, определённую диапазоном *[first, last)*, и трансформирует её в предыдущую перестановку. Предыдущая перестановка находится, полагая, что множество всех перестановок лексикографически сортировано относительно *operator<* или *comp*. Если такая перестановка существует, возвращается *true*. Иначе он трансформирует последовательность в самую большую перестановку, то есть сортированную по убыванию, и возвращает *false*. Максимально выполняется $(last - first)/2$ перестановок.

Обобщённые численные операции (Generalized numeric operations)

Накопление (Accumulate)

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
    BinaryOperation binary_op);
```

accumulate подобен оператору APL *reduction* и функции Common Lisp *reduce*, но он избегает трудности определения результата уменьшения для пустой последовательности, всегда требуя начальное значение. Накопление выполняется инициализацией сумматора *acc* начальным значением *init* и последующим изменением его $acc = acc + *i$ или $acc = binary_op(acc, *i)$ для каждого итератора *i* в диапазоне *[first, last)* по порядку. Предполагается, что *binary_op* не вызывает побочных эффектов.

Скалярное произведение (Inner product)

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
    class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
    InputIterator2 first2, T init,
    BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

inner_product вычисляет свой результат, инициализируя сумматор *acc* начальным значением *init* и затем изменяя его $acc = acc + (*i1) * (*i2)$ или $acc = binary_op1(acc, binary_op2(*i1, *i2))$ для каждого итератора *i1* в диапазоне *[first, last)* и итератора *i2* в диапазоне *[first2, first2 + (last - first))* по порядку. Предполагается, что *binary_op1* и *binary_op2* не вызывают побочных эффектов.

Частичная сумма (Partial sum)

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);
```

partial_sum присваивает каждому итератору *i* в диапазоне $[result, result + (last - first))$ значение, соответственно равное $((...(*first + *(first + 1)) + ...) + *(first + (i - result)))$ или *binary_op(binary_op(..., binary_op(*first, *(first + 1)), ...), *(first + (i - result)))*. Функция *partial_sum* возвращает *result + (last - first)*. Выполняется *binary_op* точно $(last - first) - 1$ раз. Ожидается, что *binary_op* не имеет каких-либо побочных эффектов. *result* может быть равен *first*.

Смежная разность (Adjacent difference)

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
    OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
    OutputIterator result, BinaryOperation binary_op);
```

adjacent_difference присваивает каждому элементу, указываемому итератором *i* в диапазоне $[result + 1, result + (last - first))$ значение, соответственно равное $*(first + (i - result)) - *(first + (i - result) - 1)$ или *binary_op(*(first + (i - result)), *(first + (i - result) - 1))*. Элемент, указываемый *result*, получает значение **first*. Функция *adjacent_difference* возвращает *result + (last - first)*. Применяется *binary_op* точно $(last - first) - 1$ раз. Ожидается, что *binary_op* не имеет каких-либо побочных эффектов. *result* может быть равен *first*.

АДАПТЕРЫ

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

Адаптеры контейнеров (Container adaptors)

Часто бывает полезно обеспечить ограниченные интерфейсы контейнеров. Библиотека предоставляет *stack*, *queue* и *priority_queue* через адаптеры, которые могут работать с различными типами последовательностей.

Стек (Stack)

Любая последовательность, поддерживающая операции *back*, *push_back* и *pop_back*, может использоваться для модификации *stack*. В частности, могут использоваться *vector*, *list* и *deque*.

```
template <class Container>
class stack {
friend bool operator==(const stack<Container>& o, const stack<Container>& y);
friend bool operator<(const stack<Container>& o, const stack<Container>& y);
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& top() { return c.back(); }
    const value_type& top() const { return c.back(); }
    void push(const value_type& o) { c.push_back(o); }
    void pop() { c.pop_back(); }
};

template <class Container>
bool operator==(const stack<Container>& o, const stack<Container>& y)
    { return o.n == o.n; }

template <class Container>
bool operator<(const stack<Container>& o, const stack<Container>& y)
    { return o.n < o.n; }
```

Например, *stack<vector<int> >* - целочисленный стек, сделанный из *vector*, а *stack<deque<char> >* - символьный стек, сделанный из *deque*.

Очередь (Queue)

Любая последовательность, поддерживающая операции *front*, *push_back* и *pop_front*, может использоваться для модификации *queue*. В частности, могут использоваться *list* и *deque*.

```
template <class Container>
class queue {
    friend bool operator==(const queue<Container>& o, const queue<Container>& y);
    friend bool operator<(const queue<Container>& o, const queue<Container>& y);
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
public:
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    value_type& front() { return c.front(); }
    const value_type& front() const { return c.front(); }
    value_type& back() { return c.back(); }
    const value_type& back() const { return c.back(); }
    void push(const value_type& o) { c.push_back(o); }
    void pop() { c.pop_front(); }
};

template <class Container>
bool operator==(const queue<Container>& o, const queue<Container>& y)
{ return x.c == y.c; }

template <class Container>
bool operator<(const queue<Container>& o, const queue<Container>& y)
{ return x.c < y.c; }
```

Очередь с приоритетами (Priority queue)

Любая последовательность, с итератором произвольного доступа и поддерживающая операции *front*, *push_back* и *pop_front*, может использоваться для модификации *priority_queue*. В частности, могут использоваться *vector* и *deque*.

```
template <class Container, class Compare = less<Container::value_type> >
class priority_queue {
public:
    typedef Container::value_type value_type;
    typedef Container::size_type size_type;
protected:
    Container c;
    Compare comp;
public:
    priority_queue(const Compare& x = Compare()) : c(), comp(x) {}
    template <class InputIterator>
    priority_queue(InputIterator first, InputIterator last,
        const Compare& x = Compare()) : c(first, last), comp(x)
        { make_heap(c.begin(), c.end(), comp); }
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    const value_type& top() const { return c.front(); }
    void push(const value_type& x) {
        c.push_back(x);
        push_heap(c.begin(), c.end(), comp);
    }
    void pop() {
        pop_heap(c.begin(), c.end(), comp);
        c.pop_back();
    }
}; // Никакое равенство не обеспечивается
```

Адаптеры итераторов (Iterator adaptors)

Обратные итераторы (Reverse iterators)

Двунаправленные итераторы и итераторы произвольного доступа имеют соответствующие адаптеры обратных итераторов, которые выполняют итерации через структуру данных в противоположном направлении. Они имеют те же самые сигнатуры, как и соответствующие итераторы. Фундаментальное соотношение между обратным итератором и его соответствующим итератором i установлено тождеством $\&*(reverse_iterator(i)) == \&*(i - 1)$. Это отображение продиктовано тем, что, в то время как после конца массива всегда есть указатель, может не быть допустимого указателя перед началом массива.

```
template <class BidirectionalIterator, class T, class Reference = T&,
         class Distance = ptrdiff_t>
class reverse_bidirectional_iterator
: public bidirectional_iterator<T, Distance> {
    typedef reverse_bidirectional_iterator<BidirectionalIterator, T,
        Reference, Distance> self;

    friend bool operator==(const self& o, const self& y);
protected:
    BidirectionalIterator current;
public:
    reverse_bidirectional_iterator() {}
    reverse_bidirectional_iterator(BidirectionalIterator o)
        : current(o) {}
    BidirectionalIterator base() { return current; }
    Reference operator*() const {
        BidirectionalIterator tmp = current;
        return *--tmp;
    }
    self& operator++() {
        --current;
        return *this;
    }
    self operator++(int) {
        self tmp = *this;
        --current;
        return tmp;
    }
    self& operator--() {
        ++current;
        return *this;
    }
    self operator--(int) {
        self tmp = *this;
        ++current;
        return tmp;
    }
};

template <class BidirectionalIterator, class T, class Reference,
         class Distance>
inline bool operator==(
    const reverse_bidirectional_iterator<BidirectionalIterator, T,
        Reference, Distance>& x,
    const reverse_bidirectional_iterator<BidirectionalIterator,
        T, Reference, Distance>& y){
    return x.current == y.current;
}

template <class RandomAccessIterator, class T, class Reference = T&,
         class Distance = ptrdiff_t>
class reverse_iterator : public random_access_iterator<T, Distance>{
    typedef reverse_iterator<RandomAccessIterator, T, Reference,
        Distance> self;
    friend bool operator==(const self& x, const self& y);
    friend bool operator<(const self& x, const self& y);
    friend Distance operator-(const self& x, const self& y);
```

```

    friend self operator+(Distance n, const self& x);
protected:
    RandomAccessIterator current;
public:
    reverse_iterator() {}
    reverse_iterator(RandomAccessIterator x) : current (x) {}
    RandomAccessIterator base() { return current; }
    Reference operator*() const {
        RandomAccessIterator tmp = current;
        return *--tmp;
    }
    self& operator++() {
        --current;
        return *this;
    }
    self operator++(int) {
        self tmp = *this;
        --current;
        return tmp;
    }
    self& operator--() {
        ++current;
        return *this;
    }
    self operator--(int) {
        self tmp = *this;
        ++current;
        return tmp;
    }
    self operator+(Distance n) const {
        return self (current - n);
    }
    self& operator+=(Distance n) {
        current -= n;
        return *this;
    }
    self operator-(Distance n) const {
        return self(current + n);
    }
    self operator-=(Distance n) {
        current += n;
        return *this;
    }
    Reference operator[](Distance n) { return *(*this + n);}
};

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline bool operator==(
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x,
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& y) {
    return x.current == y.current;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline bool operator<(
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x,
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& y) {
    return y.current < x.current;
}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline Distance operator-(
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& o,
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& y) {
    return y.current - x.current;
}

```

```

}

template <class RandomAccessIterator, class T, class Reference, class Distance>
inline reverse_iterator<RandomAccessIterator, T, Reference, Distance> operator+(
    Distance n,
    const reverse_iterator<RandomAccessIterator, T, Reference, Distance>& x) {
    return reverse_iterator<RandomAccessIterator, T, Reference, Distance>
        (x.current - n);
}

```

Итераторы вставки (Insert iterators)

Чтобы было возможно иметь дело с вставкой таким же образом, как с записью в массив, в библиотеке обеспечивается специальный вид адаптеров итераторов, называемых *итераторами вставки* (*insert iterators*). С обычными классами итераторов

```
while (first != last) *result++ = *first++;
```

вызывает копирование диапазона *[first, last)* в диапазон, начинающийся с *result*. Тот же самый код с *result*, являющимся итератором вставки, вставит соответствующие элементы в контейнер. Такой механизм позволяет всем алгоритмам копирования в библиотеке работать в *режиме вставки* (*insert mode*) вместо обычного режима наложения записей.

Итератор вставки создаётся из контейнера и, возможно, одного из его итераторов, указывающих, где вставка происходит, если это ни в начале, ни в конце контейнера. Итераторы вставки удовлетворяют требованиям итераторов вывода. *operator** возвращает непосредственно сам итератор вставки. Присваивание *operator=(const T& x)* определено для итераторов вставки, чтобы разрешить запись в них, оно вставляет *x* прямо перед позицией, куда итератор вставки указывает. Другими словами, итератор вставки подобен курсору, указывающему в контейнер, где происходит вставка. *back_insert_iterator* вставляет элементы в конце контейнера, *front_insert_iterator* вставляет элементы в начале контейнера, а *insert_iterator* вставляет элементы, куда итератор указывает в контейнере. *back_inserter*, *front_inserter* и *inserter* - три функции, создающие итераторы вставки из контейнера.

```

template <class Container>
class back_insert_iterator : public output_iterator {
protected:
    Container& container;
public:
    back_insert_iterator(Container& x) : container(x) {}
    back_insert_iterator<Container>&
    operator=(const Container::value_type& value) {
        container.push_back(value);
        return *this;
    }
    back_insert_iterator<Container>& operator*() { return *this; }
    back_insert_iterator<Container>& operator++() { return *this; }
    back_insert_iterator<Container>& operator++(int) { return *this; }
};

template <class Container>
back_insert_iterator<Container> back_inserter(Container& x) {
    return back_insert_iterator<Container>(x);
}

template <class Container>
class front_insert_iterator : public output_iterator {
protected:
    Container& container;
public:
    front_insert_iterator(Container& x) : container (x) {}
    front_insert_iterator<Container>&
    operator=(const Container::value_type& value) {
        container.push_front(value);
        return *this;
    }
    front_insert_iterator<Container>& operator*() { return *this; }
    front_insert_iterator<Container>& operator++() { return *this; }
    front_insert_iterator<Container>& operator++(int) { return *this; }
};

```



```

template <class Container>
front_insert_iterator<Container> front_inserter(Container& x) {
    return front_insert_iterator<Container>(x);
}

template <class Container>
class insert_iterator : public output_iterator {
protected:
    Container& container;
    Container::iterator iter;
public:
    insert_iterator(Container& x, Container::iterator i)
        : container(x), iter(i) {}
    insert_iterator<Container>& operator=(const Container::value_type& value) {
        iter = container.insert(iter, value);
        ++iter;
        return *this;
    }
    insert_iterator<Container>& operator*() { return *this; }
    insert_iterator<Container>& operator++() { return *this; }
    insert_iterator<Container>& operator++(int) { return *this; }
};

template <class Container, class Iterator>
insert_iterator<Container> inserter(Container& x, Iterator i) {
    return insert_iterator<Container>(x, Container::iterator(i));
}

```

Адаптеры функций (Function adaptors)

Функциональные адаптеры работают только с классами функциональных объектов с определёнными типами параметров и типом результата.

Отрицатели (Negators)

Отрицатели *not1* и *not2* берут унарный и бинарный предикаты соответственно и возвращают их дополнения.

```

template <class Predicate>
class unary_negate : public unary_function<Predicate::argument_type, bool> {
protected:
    Predicate pred;
public:
    unary_negate(const Predicate& x) : pred(x) {}
    bool operator()(const argument_type& x) const { return !pred(x); }
};

template <class Predicate>
unary_negate<Predicate> not1(const Predicate& pred) {
    return unary_negate<Predicate>(pred);
}

template <class Predicate>
class binary_negate : public binary_function<Predicate::first_argument_type,
    Predicate::second_argument_type, bool> {
protected:
    Predicate pred;
public:
    binary_negate(const Predicate& x) : pred(x) {}
    bool operator()(const first_argument_type& x,
        const second_argument_type& y) const {
        return !pred(x, y);
    }
}

```

```
};

template <class Predicate>
binary_negate<Predicate> not2(const Predicate& pred) {
    return binary_negate<Predicate>(pred);
}
```

Привязки (Binders)

Привязки *bind1st* и *bind2nd* берут функциональный объект *f* двух параметров и значение *x* и возвращают функциональный объект одного параметра, созданный из *f* с первым или вторым параметром соответственно, связанным с *x*.

```
template <class Predicate>
class binder1st : public unary_function {
protected:
    Operation op;
    Operation::first_argument_type value;
public:
    binder1st(const Operation& x, const Operation::first_argument_type& y)
        : op(x), value(y) {}
    result_type operator()(const argument_type& x) const {
        return op(value, x);
    }
};

template <class Operation, class T>
binder1st<Operation> bind1st(const Operation& op, const T& x) {
    return binder1st<Operation>(op, Operation::first_argument_type(x));
}

template <class Operation>
class binder2nd : public unary_function<Operation::first_argument_type,
    Operation::result_type> {
protected:
    Operation op;
    Operation::second_argument_type value;
public:
    binder2nd(const Operation& x, const Operation::second_argument_type& y)
        : op(x), value(y) {}
    result_type operator()(const argument_type& x) const {
        return op(x, value);
    }
};

template <class Operation, class T>
binder2nd<Operation> bind2nd(const Operation& op, const T& x) {
    return binder2nd<Operation>(op, Operation::second_argument_type(x));
}
```

Например, *find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5))* находит первое целое число в векторе *v* большее, чем 5; *find_if(v.begin(), v.end(), bind1st(greater<int>(), 5))* находит первое целое число в *v* меньшее, чем 5.

Адаптеры указателей на функции (Adaptors for pointers to functions)

Чтобы позволить указателям на (унарные и бинарные) функции работать с функциональными адаптерами, библиотека обеспечивает следующее:

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
protected:
    Result (*ptr)(Arg);
public:
```

```

    pointer_to_unary_function() {}
    pointer_to_unary_function(Result (*x)(Arg)) : ptr(x) {}
    Result operator()(Arg x) const { return ptr(x); }
};

template <class Arg, class Result>
pointer_to_unary_function<Arg, Result> ptr_fun(Result (*x)(Arg)) {
    return pointer_to_unary_function<Arg, Result>(x);
}

template
class pointer_to_binary_function : public binary_function {
protected:
    Result (*ptr)(Arg1, Arg2);
public:
    pointer_to_binary_function() {}
    pointer_to_binary_function(Result (*x)(Arg1, Arg2)) : ptr(o) {}
    Result operator()(Arg1 x, Arg2 y) const { return ptr(x, y); }
};

template <class Arg1, class Arg2, class Result>
pointer_to_binary_function<Arg1, Arg2, Result>
ptr_fun(Result (*x)(Arg1, Arg2)) {
    return pointer_to_binary_function<Arg1, Arg2, Result>(x);
}

```

Например, *replace_if(v.begin(), v.end(), not1(bind2nd(ptr_fun(strcmp), "C")), "C++")* заменяет все "C" на "C++" в последовательности *v*.

Системы трансляции, которые имеют множественный указатель на типы функций, должны обеспечить дополнительные шаблоны функций *ptr_fun*.

ПРИМИТИВЫ УПРАВЛЕНИЯ ПАМЯТЬЮ (MEMORY HANDLING PRIMITIVES)

Чтобы получать типичный указатель на неинициализированный буфер памяти данного размера, определена следующая функция:

```

template <class T>
inline T* allocate(ptrdiff_t n, 0*); // n >= 0

```

Размер (в байтах) распределённого буфера - не меньше *n*sizeof(T)*.

Для каждой модели памяти имеется соответствующий шаблон функции *allocate*, определённый с типом первого параметра, являющимся типом расстояния указателей в модели памяти.

Например, если система трансляции поддерживает *_huge* указатели с типом расстояния *long long*, обеспечивается следующая шаблонная функция:

```

template <class T>
inline T _huge* allocate(long long n, T _huge *);

```

Также обеспечиваются следующие функции:

```

template <class T>
inline void deallocate(T* buffer);

template <class T1, class T2>
inline void construct(T1* p, const T2& value) { new (p) T1(value);}

template <class T>
inline void destroy(T* pointer) { pointer->~T();}

```

deallocate освобождает буфер, выделенный *allocate*. Для каждой модели памяти имеются соответствующие шаблоны функций *deallocate*, *construct* и *destroy*, определённые с типом первого параметра, являющимся типом указателя в модели памяти.

```
template <class T>
pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n, T*);

template <class T>
void return_temporary_buffer(T* p);
```

get_temporary_buffer ищет наибольший буфер, не больше чем $n \cdot \text{sizeof}(T)$, и возвращает пару, состоящую из адреса и размера (в единицах $\text{sizeof}(T)$) буфера. *return_temporary_buffer* возвращает буфер, выделенный *get_temporary_buffer*.

ПРИМЕРЫ ПРОГРАММ С ШАБЛОНАМИ

Эти примеры демонстрируют использование нового продукта *STL<ToolKit>* от компании *ObjectSpace*. *STL<ToolKit>* - это самый простой способ использования *STL*, который работает на большинстве комбинаций платформ/компиляторов, включая *cfront*, *Borland*, *Visual C++*, *Set C++*, *ObjectCenter* и последние компиляторы от *Sun&HP*.

Примеры...

К сожалению, все наши попытки найти авторов этого перевода были тщетны. Тем не менее, мы сочли необходимым опубликовать этот материал ввиду скудности информации по данной тематике на русском языке.

<<Показать меню

 Сообщений **12**

 Оценка **480**

 Оценить

