3
ГЛАВА

Работа с контейнерами STL

Это первая из двух глав, представляющих рецепты, в которых используется стандартная библиотека шаблонов (Standard Template Library — STL). Из-за того, что STL — чрезвычайно обширная и важная часть C++, понадобились две главы. STL не только предлагает готовые решения наиболее трудных задач программирования, но также переопределяет способ подхода к решению многих задач общего характера. Например, вместо того, чтобы самому разрабатывать код связного списка, вы можете воспользоваться классом `list` из STL. Если вашей программе нужно ассоциировать ключ со значением и предоставить средства нахождения значения по заданному ключу, она может использовать для этого класс `map`. Поскольку STL предлагает надежные, отлаженные реализации наиболее часто используемых "механизмов данных", вы можете применять их везде, где это необходимо, не тратя время и силы на разработку ваших собственных.

Эта глава начинается с обзора STL, а затем представляет рецепты, демонстрирующие ядро STL: его контейнеры. В процессе она покажет, как итераторы используются для доступа и циклической обработки содержимого контейнера. Следующая глава покажет, как используются алгоритмы и некоторые другие ключевые компоненты STL.

Ниже перечислены рецепты, содержащиеся в этой главе.

- Базовые приемы использования последовательных контейнеров.
- Использование `vector`.
- Использование `deque`.
- Использование `list`.
- Использование адаптеров последовательных контейнеров: `stack`, `queue` и `priority_queue`.
- Хранение в контейнере определяемых пользователем объектов.
- Базовые приемы использования ассоциативных контейнеров.
- Использование `map`.
- Использование `multimap`.
- Использование `set` и `multiset`.

На заметку! *За углубленным описанием STL обращайтесь к моей книге* C++: базовый курс *(ИД "Вильямс", 2008 г.). Большинство обзора и описаний в этой главе заимствованы из этой работы. Тема STL также подробно раскрыта в моей книге* Полный справочник по C++ *(ИД "Вильямс", 2008 г.).*

Обзор STL

По своей сути STL — это сложный набор шаблонных классов и функций, реализующих многие популярные и часто используемые структуры данных и алгоритмы. Например, эта библиотека включает поддержку векторов, списков, очередей и стеков. Кроме того, она предоставляет множество алгоритмов, таких как сортировка, поиск и слияние, оперирующих этими структурами данных. Поскольку STL состоит из шаблонных классов и функций, структуры данных и алгоритмы могут применяться почти к любым типам данных. В этом, конечно же, часть ее мощи.

STL организована вокруг трех фундаментальных элементов: *контейнеров*, *алгоритмов* и *итераторов*. Говоря просто, алгоритмы применяются к контейнерам через итераторы. Более чем когда-либо дизайн и реализация этих средств определяют природу STL. В дополнение к контейнерам, алгоритмам и итераторам, STL полагается на поддержку несколько других стандартных элементов: *аллокаторы*, *адаптеры*, *функциональные объекты*, *предикаты*, *привязки* (binders) и *отрицатели* (negators). Краткое описание каждого приведено ниже.

Контейнеры

Как следует из названия, контейнер — это объект, который может содержать в себе другие объекты. Существует несколько разных типов контейнеров. Например, класс `vector` определяет динамический массив, `deque` создает двунаправленную очередь, а `list` представляет связный список. Эти контейнеры называются *последовательными контейнерами* (sequence containers), потому что в терминологии STL последовательность — это линейный список. STL также определяет *ассоциативные контейнеры* (associative containers), которые обеспечивают эффективное извлечение значений на основе ключей. Таким образом, ассоциативные контейнеры хранят пары "ключ/значение". Примером может служить `map`. Этот контейнер хранит пары "ключ/значение", в которых каждый ключ является уникальным. Это облегчает извлечение значения по заданному ключу.

Алгоритмы

Алгоритмы выполняют действия над контейнерами. Их возможности включают инициализацию, сортировку, поиск, слияние, замену и трансформацию содержимого контейнера. Многие алгоритмы оперируют *диапазонами* элементов в контейнере.

Итераторы

Итераторы — это объекты, которые ведут себя более или менее подобно указателям. Они предоставляют возможность выполнять циклическую обработку элементов контейнера — подобно тому, как вы используете указатель для организации цикла по массиву. Существует пять типов итераторов.

Итератор	Тип доступа
Произвольного доступа	Сохраняет и извлекает значения. Доступ к элементам — в произвольном порядке
Двунаправленный	Сохраняет и извлекает значения. Допускает перемещение вперед и назад
Прямой	Сохраняет и извлекает значения. Перемещение только вперед.
Входной	Извлекает, но не сохраняет значения. Перемещение только вперед
Выходной	Сохраняет, но не извлекает значения. Перемещение только вперед

Вообще итератор, имеющий более широкие возможности доступа, может применяться вместо итератора с меньшими возможностями. Например, прямой итератор может быть использован вместо входного итератора.

Итераторы обрабатываются подобно указателям. Обратные итераторы либо двунаправлены, либо произвольного доступа, которые перемещаются по последовательности в обратном направлении. Таким образом, если обратный итератор указывает на конец последовательности, то увеличение этого итератора на единицу переместит его на элемент, предшествующий конечному.

Все итераторы должны поддерживать типы операций с указателями, допустимые для их категории. Например, класс входного итератора должен поддерживать операции `->`, `++`, `*`, `==` и `!=`. Более того, операция `*` не может использоваться для присваивания значения. В отличие от входного, итератор произвольного доступа должен поддерживать операции `->`, `+`, `++`, `-`, `--`, `*`, `<`, `>`, `<=`, `>=`, `-=`, `+=`, `==`, `!=` и `[]`. Вдобавок операция `*` должна позволять присваивание. Операции, поддерживаемые каждым типом итераторов, перечислены ниже.

Итератор	Тип доступа
Произвольного доступа	`*`, `->`, `=`, `+`, `-`, `++`, `--`, `[]`, `<`, `>`, `<=`, `>=`, `-=`, `+=`, `==`, `!=`
Двунаправленный	`*`, `->`, `=`, `++`, `--`, `==`, `!=`
Прямой	`*`, `->`, `=`, `++`, `==`, `!=`
Входной	`*`, `->`, `=`, `++`, `==`, `!=`
Выходной	`*`, `=`, `++`

При ссылках на различные типы итераторов в описаниях шаблонов будут использованы следующие термины.

Термин	Итератор
`BiIter`	Произвольного доступа
`ForIter`	Двунаправленный
`InIter`	Прямой
`OutIter`	Входной
`RandIter`	Выходной

Аллокаторы

Каждый контейнер имеет определенный для него *аллокатор* (allocator). Аллокатор управляет выделением памяти для контейнера. Аллокатором по умолчанию является объект класса `allocator`, но вы можете определять свои собственные аллокаторы, если это необходимо для специализированных приложений. Для большинства применений аллокатора по умолчанию вполне достаточно.

Функциональные объекты

Функциональные объекты — это экземпляры классов, определяющих `operator()`. Существует несколько предопределенных функциональных объектов, такие как `less()`, `greater()`, `plus()`, `minus()`, `multiplies()` и `divides()`. Возможно, наиболее широко используемым функциональным объектом является `less()`, определяющий, меньше ли один объект другого. Функциональные объекты могут использоваться вместо указателей на функции в алгоритмах STL. Функциональные объекты повышают эффективность некоторых типов операций и предлагают поддержку определенных операций, которые были бы невозможны с применением только указателей на функции.

Адаптеры

В наиболее общем смысле *адаптер* трансформирует одну вещь в другую. Существуют адаптеры контейнеров, адаптеры итераторов и адаптеры функций. Примером адаптера контейнера может служить `queue`, который адаптирует контейнер `queue` для использования в качестве стандартной очереди.

Предикаты

Несколько из алгоритмов и контейнеров используют специальный тип функций, именуемый *предикатом*. Существуют две вариации предикатов: унарные и бинарные. Унарный предикат принимает один аргумент. Бинарный предикат принимает два аргумента. Эти функции возвращают результат `true/false`, но точные условия, которые заставляют их возвращать `true` или `false`, определяются вами. В этой книге, когда требуется функция — унарный предикат, она обозначается как `UnPred`. Когда требуется бинарный предикат, он обозначается как `BinPred`. В бинарном предикате аргументы всегда следуют в порядке *первый, второй*. Как для унарного, так и бинарного предикатов аргументы будут содержать значения типа объектов, хранящихся в контейнере.

Некоторые алгоритмы используют специальный тип бинарного предиката, который сравнивает два элемента. *Функции сравнения* возвращают `true`, если их первый аргумент меньше второго. В этой книге функции сравнения обозначаются типом `Comp`.

Привязки и отрицатели

Две другие сущности, наполняющие STL — это привязки (binders) и *отрицатели* (negators). *Привязка* связывает аргумент с функциональным объектом. *Отрицатель* возвращает дополнение предиката. То и другое повышают разносторонность STL.

Контейнерные классы

В сердце STL находятся контейнеры. Они перечислены в табл. 3.1. Также в ней показаны заголовки, необходимые для использования каждого из этих контейнеров. Как и следовало ожидать, каждый контейнер обладает своими возможностями и атрибутами.

Таблица 3.1. Контейнеры, определенные в STL

Контейнер	Описание	Требуемый заголовок
`deque`	Двунаправленная очередь	`<deque>`
`list`	Линейный список	`<list>`
`map`	Хранит пары "ключ/значение", где каждый ключ ассоциирован только с одним значением	`<map>`
`multimap`	Хранит пары "ключ/значение", где каждый ключ может быть ассоциирован с двумя или более значениями	`<map>`
`multiset`	Множество, в котором каждый элемент не обязательно уникален	`<set>`
`priority_queue`	Очередь с приоритетами	`<deque>`
`queue`	Очередь	`<deque>`
`set`	Множество уникальных элементов	`<set>`
`stack`	Стек	`<stack>`
`vector`	Динамический массив	`<vector>`

Контейнеры реализованы посредством шаблонных классов. Например, ниже показана спецификация шаблона контейнера `deque`. Все контейнеры используют схожие спецификации:

```
template <class T, class Allocator = allocator<T> > class deque
```

Здесь обобщенный тип `T` специфицирует тип объектов, хранящихся в `deque`. Аллокатор, используемый `deque`, специфицирован `Allocator`; по умолчанию это стандартный класс для аллокаторов. Для подавляющего большинства приложений вы будете просто применять аллокатор по умолчанию, то же касается всего кода этой главы. Однако вы можете определять собственные типы аллокаторов, когда требуется специальная схема выделения памяти. Если вы не знакомы с аргументами по умолчанию шаблонов, просто знайте, что они работают почти так же, как аргументы функций по умолчанию. Если аргумент обобщенного типа не специфицирован явно при создании объекта, то используется тип по умолчанию.

Каждый контейнерный класс включает несколько `typedef`, создающих набор стандартных имен типов. Несколько из этих имен `typedef` перечислено ниже.

`size_type`	Некоторый тип беззнакового целого
`reference`	Ссылка на элемент
`const_reference`	Константная ссылка на элемент
`iterator`	Итератор
`const_iterator`	Константный итератор
`reverse_iterator`	Обратный итератор

`const_reverse_iterator`	Константный обратный итератор
`value_type`	Тип значения, хранимого в контейнере; то же, что `T` для последовательных контейнеров
`allocator_type`	Тип аллокатора
`key_type`	Тип ключа

Как уже упоминалось, существуют две широких категории контейнеров: последовательные и ассоциативные. Последовательные контейнеры — это `vector`, `list` и `deque`. Ассоциативные контейнеры — это `map`, `multimap` и `multiset`. Последовательные контейнеры оперируют последовательностями, которые, по сути, являются линейными списками объектов. Ассоциативные контейнеры оперируют списками ключей. Ассоциативные контейнеры, реализующие карты (maps), оперируют парами "ключ/значение" и позволяют извлекать значение по заданному ключу.

Классы `stack`, `queue` и `priority_queue` называются адаптерами контейнеров, потому что используют (т.е. адаптируют) один из последовательных контейнеров, содержащий их элементы. Таким образом, для последовательных контейнеров подчеркивается функциональность, предоставленная `stack`, `queue` и `priority_queue`. С программистской точки зрения адаптеры контейнеров выглядят и ведут себя подобно другим контейнерам.

Общая функциональность

STL специфицирует набор требований, которым должны соответствовать все контейнеры. Специфицируя общую функциональность, STL гарантирует, что все контейнеры могут управляться алгоритмами и что все контейнеры могут быть использованы в понятной согласованной манере, не зависящей от деталей каждой конкретной реализации контейнера. В этом также проявляется сила STL.

Все контейнеры должны поддерживать операцию присваивания. Они должны также поддерживать все логические операции. Другими словами, все контейнеры должны поддерживать следующие операции:

```
=, ==, <, <=, !=, >, >=
```

Все контейнеры должны иметь конструктор, создающий пустой контейнер, и конструктор копирования. Они должны предусматривать деструктор, который освобождает всю память, использованную контейнером, и вызывать деструктор каждого элемента в контейнере.

Все контейнеры должны также предоставлять итераторы. Помимо прочих преимуществ, это гарантирует, что все контейнеры можно будет обрабатывать алгоритмами.

Все контейнеры должны предоставлять перечисленные ниже функции.

`iterator begin()`	Возвращает итератор, указывающий на первый элемент контейнера.
`const_iterator begin() const`	Возвращает константный итератор, указывающий на первый элемент контейнера.
`bool empty() const`	Возвращает `true`, если контейнер пуст.
`iterator end()`	Возвращает итератор, указывающий на позицию, следующую за последним элементом контейнера.

`const_iterator end() const`	Возвращает константный итератор, указывающий на позицию, следующую за последним элементом контейнера.
`size_type max_size() const`	Возвращает максимальное количество элементов, которые может содержать контейнер.
`size_type size() const`	Возвращает количество элементов, в текущий момент хранящихся в контейнере.
`void swap(`*`ContainerType`* `c)`	Обменивает между собой содержимое двух контейнеров.

Контейнер, поддерживающий двунаправленный доступ к его элементам, называется *обратимым* (reversible) *контейнером*. В дополнение к базовым требованиям к контейнерам, обратимый контейнер должен также предоставлять обратный итератор и следующие функции.

`reverse_iterator rbegin()`	Возвращает обратный итератор, указывающий на последний элемент контейнера.
`const_reverse_iterator rbegin() const`	Возвращает константный обратный итератор, указывающий на последний элемент контейнера.
`reverse_iterator rend()`	Возвращает обратный итератор, указывающий на позицию, предшествующую первому элементу в контейнере.
`const_reverse_iterator rend() const`	Возвращает константный обратный итератор, указывающий на позицию, предшествующую первому элементу в контейнере.

Требования к последовательному контейнеру

В дополнение к функциональности, общей для всех контейнеров, последовательные контейнеры добавляют перечисленные ниже функции.

`void clear()`	Удаляет все элементы из контейнера.
`iterator erase(iterator i)`	Удаляет элемент, на который указывает *`i`*. Возвращает итератор, указывающий на элемент, находящийся после удаленного.
`iterator erase(iterator start, iterator end)`	Удаляет элементы в диапазоне, указанном *`start`* и *`end`*. Возвращает итератор, указывающий на элемент, находящийся после последнего удаленного.
`iterator insert(iterator i, const T &val)`	Вставляет *`val`* непосредственно перед элементом, специфицированным *`i`*. Возвращает итератор, указывающий на вставленный элемент.
`void insert(iterator i, size_type num, const T &val)`	Вставляет *`num`* копий *`val`* непосредственно перед элементом, специфицированным *`i`*.
`template <class InIter> void insert(iterator i, InIter start, InIter end)`	Вставляет последовательность, определенную *`start`* и *`end`*, непосредственно перед элементом, специфицированным *`i`*.

STL также определяет набор функций для последовательных контейнеров, которые необязательны, но часто реализуются.

Эти функции описаны ниже.

`reference at(size_type idx)`	Возвращает ссылку на элемент, специфицированный *idx*.
`const_reference at(size_type idx) const`	Возвращает константную ссылку на элемент, специфицированный *idx*.
`reference back()`	Возвращает ссылку на последний элемент в контейнере.
`const_reference back() const`	Возвращает константную ссылку на последний элемент в контейнере.
`reference front()`	Возвращает ссылку на первый элемент в контейнере.
`const_reference front() const`	Возвращает константную ссылку на первый элемент в контейнере.
`reference operator[ ] (size_type idx)`	Возвращает ссылку на элемент, специфицированный *idx*.
`const_reference operator[ ] (size_type idx) const`	Возвращает константную ссылку на элемент, специфицированный *idx*.
`void pop_back()`	Удаляет последний элемент в контейнере.
`void pop_front()`	Удаляет первый элемент в контейнере.
`void push_back(const T &val)`	Добавляет элемент со значением, специфицированным *val*, в конец контейнера.
`void push_front(const T &val)`	Добавляет элемент со значением, специфицированным *val*, в начало контейнера.

Последовательные контейнеры должны также применять конструкторы, которые позволяют контейнеру инициализироваться парой итераторов или заданным количеством указанного элемента. Конечно, последовательный контейнер волен реализовывать дополнительную функциональность.

Требования к ассоциативному контейнеру

В дополнение к функциональности, необходимой для всех контейнеров, к ассоциативным контейнерам предъявляется несколько дополнительных требований. Во-первых, все ассоциативные контейнеры должны поддерживать следующие функции.

`void clear()`	Удаляет все элементы из контейнера.
`size_type count(const key_type &k) const`	Возвращает количество вхождений *k* в контейнер.
`void erase(iterator i)`	Удаляет элемент, указанный *i*.
`void erase(iterator start, iterator end)`	Удаляет элементы в диапазоне от *start* до *end*.
`size_type erase(const key_type &k)`	Удаляет элементы, имеющие ключи со значением *k*. Возвращает количество удаленных элементов.

`pair<iterator, iterator> equal_range(const key_type &k)`	Возвращает пару итераторов, указывающих на верхнюю и нижнюю границу диапазона элементов контейнера, соответствующих указанному ключу.
`pair<const_iterator, const_iterator> equal_range(const key_type &k) const`	Возвращает пару константных итераторов, указывающих на верхнюю и нижнюю границу диапазона элементов контейнера, соответствующих указанному ключу.
`iterator find(const key_type &k)`	Возвращает итератор, указывающий на специфицированный ключ. Если ключ не найден, то возвращается итератор, указывающий на конец контейнера.
`const_iterator find(const key_type &k) const`	Возвращает константный итератор, указывающий на специфицированный ключ. Если ключ не найден, то возвращается итератор, указывающий на конец контейнера.
`pair<iterator, bool> insert(const value_type &val)`	Вставляет *val* в контейнер. Если контейнер требует уникальных ключей, то *val* вставляется только в случае, если он еще там не существует. Если элемент вставлен, возвращается `pair<iterator, true>`. В противном случае возвращается `pair<iterator, false>`.
`iterator insert(iterator start, const value_type &val)`	Вставляет *val*. Поиск правильной точки вставки начинается с элемента, специфицированного *start*. Для контейнеров, требующих уникальных ключей, элементы вставляются, только если они еще не существуют. Возвращается итератор, указывающий на элемент.
`template <class InIter> void insert(InIter start, InIter end)`	Вставляет диапазон элементов. Для контейнеров, требующих уникальных ключей, элементы вставляются, только если они еще не существуют в контейнере.
`key_compare key_comp() const`	Возвращает функциональный объект, сравнивающий два ключа.
`iterator lower_bound(const key_type &k)`	Возвращает итератор, указывающий на первый элемент с ключом, эквивалентным или большим *k*.
`const_iterator lower_bound(const key_type &k) const`	Возвращает константный итератор, указывающий на первый элемент с ключом, эквивалентным или большим *k*.
`iterator upper_bound(const key_type &k)`	Возвращает итератор, указывающий на первый элемент с ключом, большим *k*.
`const_iterator upper_bound(const key_type &k) const`	Возвращает константный итератор, указывающий на первый элемент с ключом, большим *k*.
`value_compare value_comp() const`	Возвращает функциональный объект, сравнивающий два значения.

Обратите внимание, что некоторые из функций возвращают объект `pair`. Это класс, инкапсулирующий два объекта. Для ассоциативных контейнеров — карт `value_type` представляет `pair`, который инкапсулирует ключ и значение. Класс `pair` детально описан в разделе "Базовые приемы использования ассоциативных контейнеров".

Ассоциативные контейнеры должны предоставлять конструкторы, которые позволяют контейнеру инициализироваться элементами, специфицированными парой итераторов. Они также должны поддерживать конструкторы, которые позволяют специфицировать функцию сравнения, используемую для сравнения двух ключей. Разумеется, ассоциативный контейнер может реализовать и дополнительную функциональность.

Проблемы производительности

Существует еще один важный аспект STL, который добавляет мощи и применимости: гарантии производительности. Хотя производитель компилятора волен реализовать механизмы, положенные в основу каждого контейнера и алгоритма, собственным способом, все реализации должны отвечать требованиям производительности, специфицированным STL. Определены следующие основные категории производительности:

- постоянная;
- линейная;
- логарифмическая.

Поскольку разные контейнеры хранят свое содержимое по-разному, они предоставляют разные гарантии производительности. Например, вставка в середину контейнера типа `vector` занимает линейное время. В отличие от этого, вставка в `list` требует постоянного времени. Разные алгоритмы также могут вести себя по-разному. Например, алгоритм `sort()` требует времени, пропорционального $N \log N$, а алгоритм `find()` выполняется за постоянное время.

В некоторых случаях говорят, что операции занимают *амортизированное постоянное время*. Этот термин используется для описания ситуации, при которых операция обычно требует постоянного времени, но иногда немного больше. (Например, вставка в конец вектора обычно происходит за постоянное время, но если при этом должна распределяться память, то вставка потребует линейного времени.) Если такие случаи происходят достаточно редко, то можно считать, что они амортизируются количеством операций, занимающих краткое время.

Вообще спецификация STL требует, чтобы контейнеры и алгоритмы были реализованы с применением приемов, гарантирующих (кратко говоря) оптимальную производительность времени выполнения. Это важно, поскольку гарантирует вам, как программисту, то, что строительные блоки STL соответствуют определенному уровню эффективности, независимо от того, какую реализацию STL вы используете. Без такой гарантии производительность основанного на STL кода целиком зависела бы от индивидуальной реализации и варьировалась в широких пределах.

Базовые приемы использования последовательных контейнеров

Ключевые ингредиенты		
Заголовки	**Классы**	**Функции**
<vector>	vector	iterator begin() void clear() bool empty() const iterator end() iterator erase(iterator *i*) iterator insert(iterator *i*, const T &*val*) reverse_iterator rbegin() reverse_iterator rend() size_type size() const void swap(vector<T, Allocator> &*ob*)
<vector>	vector	template <class T, class Allocator> bool operator==(const vector<T, Allocator> &*leftop*, const vector<T, Allocator> &*rightop*) template <class T, class Allocator> bool operator<(const vector<T, Allocator> &*leftop*, const vector<T, Allocator> &*rightop*) template <class T, class Allocator> bool operator>(const vector<T, Allocator> &*leftop*, const vector<T, Allocator> &*rightop*)

Все последовательные контейнеры разделяют общую функциональность. Например, все они позволяют добавлять элементы к контейнеру, удалять элементы из контейнера или выполнять циклы по контейнеру посредством итератора. Все поддерживают операцию присваивания и логические операции, и все последовательные контейнеры конструируются сходным образом. Настоящий рецепт описывает эту общую функциональность, демонстрируя базовые приемы, которые применимы ко всем последовательным контейнерам. Этот рецепт показывает, как:

- создать последовательный контейнер;
- добавить элементы в контейнер;
- определить размер контейнера;
- использовать итератор для выполнения цикла по контейнеру;
- присвоить один контейнер другому;
- определять эквивалентность одного контейнера другому;
- удалять элементы из контейнера;
- менять одни элементы в контейнере другими;
- определять признак пустоты контейнера.

Данный рецепт использует контейнерный класс vector, но в нем только те методы, что являются общими для всех последовательных контейнеров. Таким образом, одни и те же общие принципы могут быть применимы к любому типу последовательного контейнера.

Необходимые шаги

Создание и использование последовательного контейнера предусматривает выполнение перечисленных ниже шагов.

1. Создать экземпляр нужного контейнера. В этом рецепте используется vector, но вместо него можно подставить любой последовательный контейнер.
2. Добавить элементы в контейнер вызовом insert().
3. Получить количество элементов в контейнере вызовом size().
4. Определить, пуст ли контейнер (т.е. факт отсутствия в нем элементов), вызовом empty().
5. Удалить элементы из контейнера вызовом erase().
6. Удалить все элементы из контейнера вызовом clear().
7. Получить итератор, указывающий на начало последовательности вызовом begin(). Получить итератор, указывающий на позицию, находящуюся за концом последовательности, вызовом end().
8. Для обратимых последовательных контейнеров получить обратный итератор, указывающий на конец последовательности, вызовом rbegin(). Получить обратный итератор, указывающий на позицию, предшествующую началу последовательности, вызовом rend().
9. Выполнить цикл по элементам контейнера посредством итератора.
10. Обменять содержимое одного контейнера с содержимым другого посредством swap().
11. Определить, когда один контейнер эквивалентен, меньше или больше другого.

Обсуждение

Хотя внутреннее устройство STL довольно сложно, использовать STL на самом деле достаточно просто. Во многих отношениях самое сложное в STL — это решение относительно того, какой контейнер применять. Каждый контейнер имеет определенные преимущества и недостатки. Например, vector очень хорош для произвольного доступа, когда требуется объект, подобный массиву, и не предполагается большое количество операций вставки и удаления. Контейнер list обеспечивает недорогие операции вставки и удаления, но медленный поиск. Двунаправленная очередь поддерживается контейнером deque. В этом рецепте vector используется для демонстрации базовых операций с последовательным контейнером, но программа также будет работать и с list, и с deque. В этом состоит одно из главных преимуществ STL: все последовательные контейнеры поддерживают базовый уровень общей функциональности.

Спецификация шаблона для vector показана здесь:

```
template <class T, class Allocator = allocator<T> > class vector
```

Здесь T — тип хранимых в контейнере данных, а Allocator специфицирует аллокатор, по умолчанию — стандартный. Чтобы использовать vector, вы должны включить заголовок <vector>.

Класс vector поддерживает несколько конструкторов. Два, используемых в данном рецепте, обязательны для всех последовательных контейнеров. Вот они:

```
explicit vector(const Allocator &alloc = Allocator() )
vector(const vector<T, Allocator> &ob)
```

Первая форма конструирует пустой вектор. Вторая форма — копирующий конструктор vector.

После создания контейнера объект может быть добавлен в него. Один способ, который работает со всеми последовательными контейнерами, предусматривает вызов insert(). Все последовательные контейнеры поддерживают, как минимум, три версии insert(). Здесь используется следующая:

```
iterator insert(iterator i, const T &val)
```

Она вставляет *val* в вызывающий контейнер в точке, специфицированной *i*. Функция возвращает итератор, указывающий на вставленный элемент. Последовательный контейнер будет автоматически расти по мере необходимости, когда элементы добавляются к нему.

Вы можете удалить один или более элементов из последовательного контейнера вызовом erase(). Эта функция имеет, как минимум, две формы. В настоящем рецепте используется следующая:

```
iterator erase(iterator i)
```

Она удаляет элемент, на который указывает *i*. Функция возвращает итератор, указывающий на элемент, который находится после удаленного. Чтобы удалить все элементы в контейнере, вызовите функцию clear(). Она показана ниже:

```
void clear()
```

Вы можете определять количество элементов в контейнере вызовом size(). Чтобы определить, пуст ли контейнер, вызывайте empty(). Обе функции показаны здесь:

```
bool empty() const
size_type size() const
```

Вы можете получить итератор, указывающий на начало последовательности вызовом begin(). Итератор, указывающий на позицию, следующую за последним элементом, получается вызовом end(). Эти функции показаны ниже:

```
iterator begin()
iterator end()
```

Существуют также const-версии этих функций.

Чтобы объявить переменную, которая будет использоваться в качестве итератора, вы должны специфицировать тип итератора контейнера. Например, ниже объявлен итератор, который может указывать на элементы внутри vector<double>:

```
vector<double>::iterator itr;
```

Важно подчеркнуть, что end() *не* возвращает итератор, указывающий на последний элемент контейнера. Вместо этого он возвращает итератор, который указывает на позицию, *следующую за* последним элементом.

Таким образом, на последний элемент в контейнере указывает end()-1. Это средство позволяет писать очень эффективные алгоритмы, которые проходят циклом по всем элементом контейнера, включая последний. Когда итератор принимает значение end(), вы знаете, что все элементы были обработаны. Например, вот цикл, проходящий по всем элементам последовательного контейнера по имени cont:

```
for(itr = cont.begin(); itr != cont.end(); ++itr) // ...
```

Цикл продолжается до тех пор, пока itr не станет равным cont.end(). Таким образом, все элементы в cont будут обработаны.

Как уже объяснялось, обратимый контейнер — это такой, по элементам которого можно проходить в обратном порядке (от конца к началу). Все встроенные последовательные контейнеры обратимы. Для обратимого контейнера можно получить обратный итератор, указывающий на конец последовательности, с помощью вызова rbegin(). Итератор, указывающий на позицию, предшествующую первому элементу последовательности, получается вызовом rend(). Эти функции показаны ниже:

```
reverse_iterator rbegin()
reverse_iterator rend()
```

Существуют также const-версии этих функций. Обратный итератор объявлен как обычный итератор. Например:

```
vector<double>::reverse_iterator ritr;
```

Вы можете использовать обратный итератор для прохода по элементам вектора в обратном порядке. Например, имея обратный итератор по имени ritr, вот как будет выглядеть цикл прохода по всем элементам в обращаемом последовательном контейнере по имени cont, от конца к началу:

```
for(ritr = cont.rbegin(); ritr != cont.rend(); ++ritr) // ...
```

Обратный итератор ritr начинается с элемента, указанного rbegin(), который является последним элементом последовательности. Цикл продолжается до тех пор, пока ritr не станет эквивалентным rend(), указывая на позицию, непосредственно предшествующую началу последовательности. (Иногда полезно воспринимать rbegin() и rend() как итераторы, указывающие на начало и конец обратной последовательности.) Каждый раз, когда обратный итератор увеличивается, он указывает на предшествующий элемент, а когда уменьшается — на следующий.

Содержимое двух последовательных контейнеров можно поменять местами вызовом функции swap(). Ниже показан способ, определенный для vector:

```
void swap(vector<T, Allocator> &ob)
```

Содержимое вызывающего контейнера обменивается с содержимым *ob*.

Пример

Следующий пример демонстрирует применение базовых операций с последовательным контейнером.

```
// Демонстрация использования базовых операций с последовательным контейнером.
//
// В этом примере используется вектор, но те же приемы могут
// применяться к любому последовательному контейнеру.

#include <iostream>
#include <vector>

using namespace std;

void show(const char *msg, vector<char> vect);

int main() {
  // Объявление пустого вектора, который может хранить объекты char.
  vector<char> v;
```

```
// Объявление итератора для vector<char>.
vector<char>::iterator itr;

// Получить итератор, указывающий на начало v.
itr = v.begin();

// Вставить символы в v. Возвращается итератор,
// указывающий на вставленный объект.
itr = v.insert(itr, 'A');
itr = v.insert(itr, 'B');
v.insert(itr, 'C');

// Отобразить содержимое v.
show("Содержимое v: ", v);

// Объявление обратного итератора.
vector<char>::reverse_iterator ritr;

// Использовать обратный итератор для отображения v в обратном порядке.
cout << "А вот v в обратном порядке: ";
for(ritr = v.rbegin(); ritr != v.rend(); ++ritr)
  cout << *ritr << " ";
cout << "\n\n";

// Создать другой вектор, идентичный первому.
vector<char> v2(v);
show("Содержимое v2: ",v2);
cout << "\n";

// Показать размер v, представляющий количество
// элементов,в настоящий момент находящихся в v.
cout << "Размер v равен " << v.size() << "\n\n";

// Сравнить два контейнера.
if(v == v2) cout << "v и v2 эквивалентны.\n\n";

// Вставить дополнительные символы в v и v2.
// На этот раз вставлять в конец.
cout << "Вставка дополнительных символов в v и v2.\n";
v.insert(v.end(), 'D');
v.insert(v.end(), 'E');
v2.insert(v2.end(), 'X');
show("Содержимое v: ", v);
show("Содержимое v2: ", v2);
cout << "\n";

// Определить, меньше ли v чем v2.
// Сравнение лексикографическое.
// Поэтому первый несовпадающий элемент определяет,
// какой контейнер меньше другого.
if(v < v2) cout << "v меньше, чем v2.\n\n";

// Теперь вставить Z в начало v.
cout << "Вставка Z в начало v.\n";
v.insert(v.begin(), 'Z');
show("Содержимое v: ", v);
cout << "\n";

// Теперь снова сравнить v и v2.
if(v > v2) cout << "Теперь v больше, чем v2.\n\n";
```

```
  // Удалить первый элемент из v2.
  v2.erase(v2.begin());
  show("v2 после удаления первого элемента: ", v2);
  cout << "\n";

  // Создать другой вектор.
  vector<char> v3;
  v3.insert(v3.end(), 'X');
  v3.insert(v3.end(), 'Y');
  v3.insert(v3.end(), 'Z');
  show("Содержимое v3: ", v3);
  cout << "\n";

  // Обменять содержимым v и v3.
  cout << "Обмен v и v3.\n";
  v.swap(v3);
  show("Содержимое v: ", v);
  show("Содержимое v3: ", v3);
  cout << "\n";

  // Очистить v.
  v.clear();
  if(v.empty()) cout << "v теперь пуст.";

  return 0;
}

// Отображение содержимого vector<char>
// с использованием итератора.
void show(const char *msg, vector<char> vect) {
  vector<char>::iterator itr;

  cout << msg;
  for(itr=vect.begin(); itr != vect.end(); ++itr)
    cout << *itr << " ";
  cout << "\n";
}
```

Вывод примера представлен ниже.

```
Содержимое v: C B A
A вот v в обратном порядке: A B C

Содержимое v2: C B A

Размер v равен 3

v и v2 эквивалентны.

Вставка дополнительных символов в v и v2.
Содержимое v: C B A D E
Содержимое v2: C B A X

v меньше, чем v2.

Вставка Z в начало v.
Содержимое v: Z C B A D E

Теперь v больше, чем v2.

v2 после удаления первого элемента:

Содержимое v3: X Y Z
```

```
Обмен v и v3.
Содержимое v: X Y Z
Содержимое v3: Z C B A D E

v теперь пуст.
```

Хотя большая часть программы самоочевидна, есть несколько интересных моментов, которые требуют более пристального рассмотрения. Для начала обратите внимание, в объявлениях контейнеров, используемых в программе (v, v2 и v3), аллокаторы не специфицированы. Как объяснялось, для большинства применений STL аллокатор по умолчанию будет правильным выбором.

Затем обратите внимание, как объявлен итератор itr в следующем операторе:

```
vector<char>::iterator itr;
```

Здесь объявляется итератор, который может быть использован с объектами типа vector<char>. Каждый класс контейнера создает typedef для iterator. Итераторы для других типов векторов или других контейнеров объявляются таким же общим способом. Например:

```
vector<double>::iterator itrA;
deque<string>::iterator itrB;
```

Здесь itrA — итератор, который может применяться с контейнерами vector<double>, а itrB — с контейнерами типа deque<string>. Вообще вы должны объявить итератор таким образом, чтобы он соответствовал как типу контейнера, так и типу объектов, хранящихся в контейнере. То же касается и обратных итераторов.

Затем вызовом begin() получается итератор, указывающий на начало контейнера, и следующая последовательность вызовов insert() помещает элементы в v:

```
itr = v.insert(itr, 'A');
itr = v.insert(itr, 'B');
v.insert(itr, 'C');
```

Каждый вызов вставляет значение непосредственно перед элементом, на который указывает итератор, переданный в itr. Возвращается итератор, указывающий на вставленный элемент. Таким образом, эти три вызова приводят к тому, что в v содержится последовательность CBA.

Теперь взгляните на функцию show(). Она используется для отображения содержимого vector<char>. Обратите особое внимание на следующий цикл:

```
for(itr=vect.begin(); itr != vect.end(); ++itr)
  cout << *itr << " ";
```

Он проходит по всему вектору, переданному в vect, начиная с первого элемента и останавливаясь после достижения последнего элемента. Напомню, что end() возвращает итератор, указывающий на элемент, который находится за концом контейнера. Таким образом, когда itr равен vect.end(), то конец контейнера достигнут. Такого рода циклы очень широко применяются при работе с STL. Также обратите внимание, как разыменовывается itr с использованием операции * — точно так же, как разыменовывается указатель. Вообще итераторы работают подобно указателям и обрабатываются, по сути, аналогичным образом.

Далее отметьте, как два контейнера сравниваются операциями == и <. Для последовательных контейнеров сравнения элементов осуществляются в лексикографическом порядке. Хотя термин “лексикографический” буквально означает “словарный порядок”, его значение обобщено в отношении STL. При сравнении два контейнера считаются эквива-

лентными, если они содержат одинаковое количество элементов, в одинаковом порядке, и все соответствующие элементы в двух контейнерах эквивалентны. Иначе результат лексикографического сравнения определяется сравнением первых, не совпадающих элементов. Например, имея следующие две последовательности:

```
seq1: 7, 8, 9
seq2: 7, 8, 11
```

seq1 меньше, чем seq2, потому что первое несоответствие встречается между 9 и 11, и 9 < 11. Поскольку сравнение лексикографическое, seq1 меньше, чем seq2, даже несмотря на то, что длина seq1 возрастает до 7, 8, 9, 10, 11, 12. Первая несовпадающая пара (в данном случае 9 и 11) определяют результат.

Варианты и альтернативы

В дополнение к версии insert(), использованной в данном рецепте, все последовательные контейнеры поддерживают две формы, показанные ниже:

```
void insert(iterator i, size_type num, const T &val)
template <class InIter> void insert(iterator i, InIter start, InIter end)
```

Первая форма вставляет *num* копий *val* непосредственно перед элементом, специфицированным *i*. Вторая форма вставляет последовательность, простирающуюся от *start* до *end* - 1, непосредственно перед элементом, специфицированным *i*. Обратите внимание, что *start* и *end* не обязаны указывать на вызывающий контейнер. Таким образом, эта форма может быть использована для вставки элементов из одного контейнера в другой.

Более того, контейнеры не обязаны быть одного вида. До тех пор, пока элементы совместимы, вы можете вставлять элементы, например, из deque в list.

Существует и вторая форма erase(), которая поддерживается всеми последовательными контейнерами. Вот она:

```
iterator erase(iterator start, iterator end)
```

Эта версия удаляет элементы в диапазоне от *start* до *end* - 1 и возвращает итератор, указывающий на элемент, находящийся после последнего удаленного элемента.

В дополнение к операциям ==, < и >, все последовательные контейнеры поддерживают логические операции <=, >= и !=.

Найти максимальное количество элементов, которое может вмещать контейнер, можно с помощью функции max_size(), показанной ниже:

```
size_type max_size() const
```

Следует понимать, что максимальный размер будет варьироваться в зависимости от типа данных, содержащихся в контейнере. Также разные типы контейнеров могут (и, вероятно, будут) иметь разные максимальные емкости.

Как упоминалось, предыдущий пример работает со всеми последовательными контейнерами. Чтобы доказать это, попробуйте подставить list или deque вместо vector. Как вы увидите, программа выдаст тот же результат, что и раньше. Конечно, выбор правильного контейнера — важное условие успешного использования STL. Напомню, что разные контейнеры имеют разные гарантии производительности. Например, вставка в list происходит за постоянное время. Вставка в середину vector требует линейного времени, но вставка в конец происходит за постоянное время (если не требуется перераспределение памяти). Вообще, если нет веских причин для выбора одного контейнера вместо другого, то обычно vector — наилучший выбор, потому что он реализует то, что, по сути, является динамическим массивом (см. рецепт “Использование vector”).