Контейнер

Контейнер это тип данных, позволяющий хранить в себе объекты других типов. В языках программирования со статической типизацией (например, Java, C++) все объекты, находящиеся в контейнере, должны иметь одинаковый тип.

Каждый контейнер реализует определенный интерфейс для работы со своим содержимым. (Упрощенно, интерфейс — это набор методов.) При этом реализация каждого контейнера опирается на конкретную структуру данных, что накладывает ограничения на способы использования этого контейнера. Незнание этих ограничений приводит к неверному или неэффективному использованию контейнеров.

Контейнер в C++ — это шаблонный тип. После имени типа контейнера в угловых скобках указывается тип содержимого. Например, `set<string>` — множество строк или `queue<int>` — очередь целых чисел. Элементами контейнера могут быть другие контейнеры, что позволяет создавать сложные структуры данных. Например, `vector<set<string>>` — вектор из множеств строк или `map<string,vector<int>>` — отображение в котором ключем является строка, а значением — вектор целых чисел.

Работа с контейнерами в C++ осуществляется через итераторы. Каждый контейнер предоставляет итераторы на начало и конец последовательности с использованием методов `begin()` и `end()`. Следует обратить внимание на то, что `end()` возвращает итератор на фиктивный элемент — заглушку конца последовательности. Доступ к этому элементу запрещен. В пустом контейнере эта заглушка также присутствует и `begin()` в этом случае возвращает итератор на нее.

Таким образом, стандартный цикл, пробегающий все элементы произвольного контейнера `X` выглядит следующим образом.

```
for (auto it=X.begin(); it!=X.end(); it++) {
    ...
}
```

В связи с тем, что такая конструкция является очень часто используемой, для нее существует упрощенная запись.

```
for (auto a:X) {
    ...
}
```

Здесь `a` уже не итератор, а сам объект из контейнера.

Если контейнер поддерживает изменение объектов в месте их хранения, то можно использовать эту конструкцию в следующем виде.

```
for (auto &a:X) {
    a=...
}
```

Например, фрагмент программы для чтения последовательности целых чисел в вектор может выглядеть так.

```
int n;
cin>>n;
vector<int> X(n);
for (auto &a:X)
    cin>>a;
```

Кроме того, контейнеры предоставляют реверсивные итераторы `rbegin()` и `rend()`. Они позволяют просматривать объекты контейнера с конца. Например, фрагмент программы

```
for (auto it=X.rbegin(); it!=X.rend(); it++)
    cout<<*it<<' ';
```

Выведет все элементы контейнера `X` в обратном порядке.

Наконец, существуют константные версии итераторов `cbegin()`, `cend()`, `crbegin()`, `crend()`, которые позволяют читать итерируемый объект, но не допускают изменять его.

Для определения количества элементов в контейнере можно использоать метод `X.size()`. Метод `X.empty()` возвращает истину, если контейнер пуст.

Контейнеры сохраняющие взаимное расположение элементов

Контейнеры с произвольным доступом к элементам

Контейнеры этого вида предоставляют итераторы произвольного доступа. Это означает, что итераторы способны смещаться на произольное число элементов за время $O(1)$. (Упрощенно, за одно действие). Кроме того, итераторы этого типа позволяют определять количество элементов между двумя итераторами и их взаимное расположение (операции `-`, `<` и `>`). Для доступа к элементам этих контейнеров можно использовать оператор индексации `[]`, который работает со сложностью $O(1)$.

К контейнерам этого вида относятся: `array`, `vector`, `deque`.

Контейнер `array` является реализацией обычного массива. Количество элементов должно быть известно на момент трансляции, и указывается вторым параметром шаблона. Например, `array<int,100>` Операции, требующие изменения размера массива не поддерживаются.

Контейнер `vector` близок по реализации к обыкновенному массиву, но поддерживает резервирование памяти под операции изменяющие его размер. Операции, требующие изменения размера, производятся за счет резерва памяти. Когда резервная память заканчивается, происходит реаллокация (перемещение) элементов вектора с новым резервированием. Это позволяет выполнять увеличение размера вектора в среднем со сложностью $O(1)$. В связи с тем, что после увеличения размера вектора может происходить реаллокация элементов, все итераторы после выполнения операций, приводящих к увеличению размера вектора становятся недопустимыми и их требуется получать заново. Операция индексации в векторе выполняется с той же скоростью, как и в массиве или незначительно медленнее, поэтому вектор является самой распространенным контейнером с произвольным доступом к элементам. Простой массив используется реже, поскольку в большинстве случаев количество элементов контейнера определяется в момент выполнения программы.

Контейнер `deque` (дек) обычно реализуется как "двухуровневый"массив. Все элементы дека разбиваются на блоки фиксированного размера. При выполнении операции индексации вычисляется сначала номер блока, далее позиция элемента внутри блока. В силу этого операция индексации работает в деке медленнее чем в векторе, хотя ее сложность и остается в пределах $O(1)$. Резервирование памяти в деке также имеет место, однако размер резерва определяется размером крайних блоков, которые могут быть не заполнены до конца. При этом изменение размеров дека, в отличии от вектора, не приволит к реаллокации элементов, поскольку оно происходит за счет добавления новых блоков. Таким образом, при изменении размера дека итераторы остаются допустимыми.

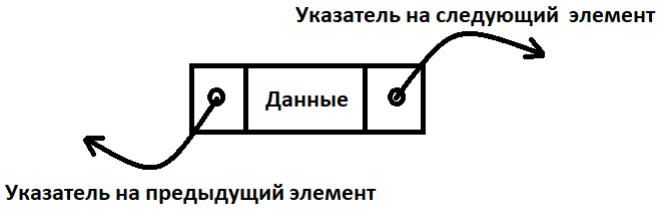
Контейнеры вектор и дек поддерживают операции вставки элемента или диапазона элементов в любое место вектора. `X.insert(pos,value)` или `X.insert(pos,first,last)`. Здесь `pos` это итератор, указывающий на позицию вставки, `value` — вставляемый элемент, `first` и `last` — итераторы, задающие диапазон вставляемых элементов. Вставка элементов в середину вектора или дека приводит к перемещению элементов, расположенных после позиции вставки, поэтому выполняется с линейной сложностью $O(n)$, что в большинстве случаев приводит к недопустимым затратам времени. Вместе с тем, вставка и удаление элементов в хвост вектора или дека не приводит к смещению остальных элементов, поэтому имеет среднюю сложность $O(1)$ и часто используется. Для вставки и удаления элементов в хвост вектора или дека используются методы `X.push_back(value)` и `X.pop_back()`.

Ключевым отличием дека от вектора с точки зрения использования является возможность быстрой вставки и удаления элемента не только в конец дека, но и в его начало. В силу этого дек обладает также методами `X.push_front(value)` и `X.pop_front()`, которые выполняются со средней сложностью $O(1)$.

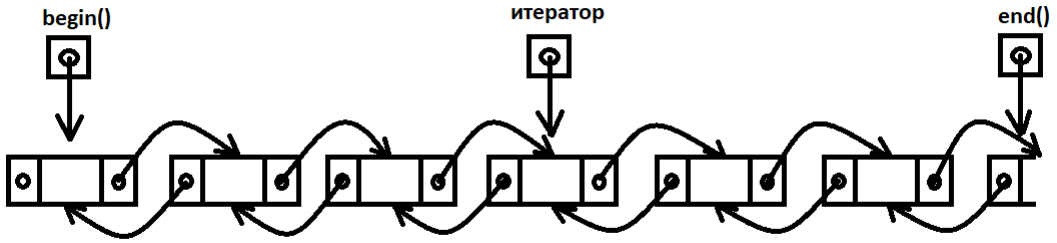
Таким образом, контейнеры с произвольным доступом к элементам используются в когда контейнер содержит небольшое число элементов. При необходимости сохранять значительное число элементов, такие контейнеры используются при нобходимости обеспечить быстрый доступ к элементу по его порядковому номеру и отсутствии операций вставки и удаления новых элементов в середину последовательности. При этом контейнер `array` используется если количество элементов не изменяется и известно на момент трансляции программы, контейнер `deque` используется при необходимости добавлять и удалять элементы из начала и конца последовательности, а во всех остальных случаях используется `vector`.

Связные списки

Связный список это контейнер в котором каждый элемент хранит информацию о местоположении следующего и предыдущего элементов.

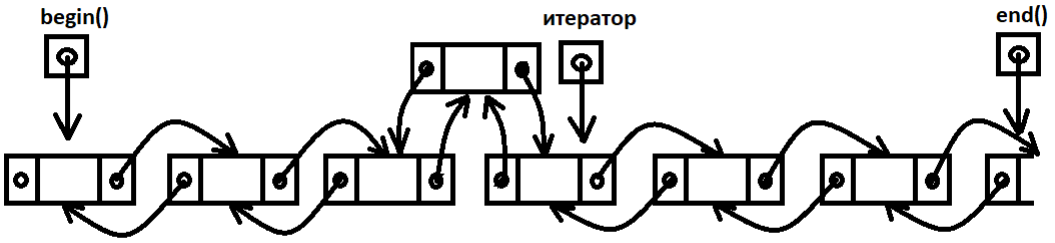


Функция `begin()` возвращает итератор на первый элемент списка, а `end()` — на фиктивный элемент-заглушку конца списка. Полученные итераторы можно перемещать на один элемент вперед или назад с использованием операторов `++` и `--`.



Элементы списка не изменяют своего местоположения, поэтому все итераторы остаются допустимыми после любых операций. Единственным исключением из этого правила является итератор на удаляемый элемент списка. После выполнения операции удаления такой итератор станет недопустимым. Итераторы связных списков не являются итераторами произвольного доступа, поэтому они могут перемещаться по итерируемой последовательности только на один шаг перед или назад. Таким образом, для доступа к элементу, находящемуся на расстоянии k от некоторого допустимого итератора потребуется выполнить k команд смещения, что дает линейное $O(n)$ время доступа к произвольному случайному элементу списка. В связи с этим для списков не реализована операция индексации.

Основным преимуществом списков является возможность их быстрого редактирования. Для вставки одного элемента достаточно разместить его в произвольном месте и перенаправить указатели. Аналогично реализуется и удаление элемента. Более того, для однотипных списков можно вырезать диапазон элементов из одного списка и вставить их в некоторое место другого списка. Все эти операции выполняются со сложностью $O(1)$. Интерфейс для команд редактирования списков совпадает с интерфейсом для дека, но выполняются эти команды для списка значительно быстрее.



Как показано на рисунке, новый элемент вставляется перед тем, на который указывает итератор. Помимо расположения нового элемента в памяти, операции вставки требует перенаправления двух указателей.

Удаление элемента выполняется аналогично. Если итератор используется для удаления элемента, то после выполнения операции он становится недопустимым. Если есть необходимость в его дальнейшем использовании, то можно использовать такую конструкцию.

```
lst.erase(iter++);
```

Выполнение такой строки программы примедет к тому, что итератор сначала будет скопирован, далее итератор сместится к следующему элементу, далее скопированный итератор будет использован для удаления элемента. В результате элемент на который указывал `iter` будет удален, а сам итератор сместится к следующему элементу.

Таким образом, списки применяются в тех случаях, когда не требуется произвольный доступ к элементам, но нужна возможность быстрого редактирования их взаимного расположения.

Производные контейнеры

В программировании часто используются контейнеры stack (стек) и queue (очередь) Очередь используется для хранения элементов по принципу очереди, а именно, новые элементы добавляются в хвост очереди, а извлекаются из начала очереди. Очевидно, что очередь можно реализовать на основе списка или дека, что собственно и делается в стандартной библиотеке C++.

Стек используется для откладывания элементов для последующей обработки и является односторонней очередью, то есть элементы излекаются в обратном порядке относительно добавления.

Использование стека и очереди строится на трех основных функциях. Это функции добавления `X.push(value)` и удаления `X.pop()` элемента. Для чтения элемента с вершины стека используется метод `X.top()`, аналогичный метод `X.front()` позволяет прочитать элемент из головы очереди.

Контейнеры, не сохраняющие взаимное расположение элементов

В контейнерах этого типа для определения местоположения элемента используется его значение. Это накладывает ряд ограничений на использование таких контейнеров, саммым главным из которых является запрет на изменение элементов в месте хранения. Для изменения элемента в таком контейнере требуется удалить его и снова записать с уже измененным значением.

Приоритетная очередь (куча)

Кучей называется структура данных построенная на основе вектора или дека, которая переупорядочивает элементы так, чтобы на первом месте всегда находился максимальный

(минимальный) элемент. Оставшиеся элементы при этом не обязательно упорядочены. После каждого изменения кучи выполняется операция ее перестроения, сложность которой составляет $O(\log n)$, где n — это количество элементов в куче. Доступ к максимальному элементу производит за $O(1)$.

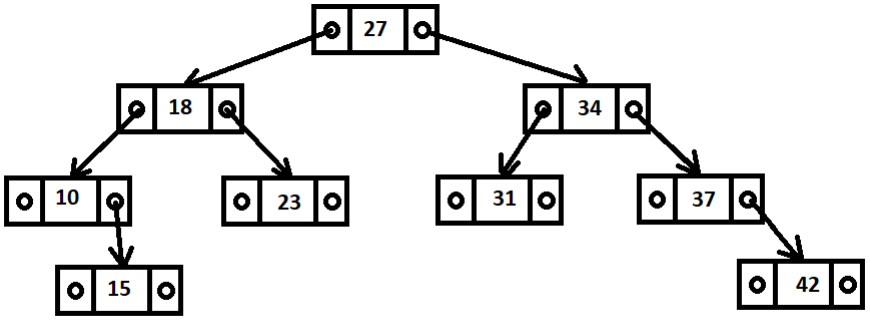
В стандартной библиотеке C++ куча реализована в виде контейнера `priority_queue` (приоритетная очередь) Ее интерфейс совпадает с интерфейсом очереди, но в голове всегда расположен максимальный элемент. Приоритетная очередь может содержать одинаковые элементы. В этом случае в голове будет расположен один из максимальных элементов, выбранных произвольно.

Объявление приоритетной очереди может содержать три шаблонных параметра: тип элементов, способ реализации, бинарное отношение. Например, для того, чтобы создать очередь из вещественных чисел с выбором минимального элемента требуется сделать следующее объявление

```
priority_queue<double,vector<double>,greater<double>> X;
```

Контейнеры на основе деревьев сортировки

Двоичное упорядоченное дерево называется деревом сортировки, если для каждый узел дерева содержит некоторые ключи, причем для ключей определен некоторым образом линейный порядок, и выполняется следующее свойство. Для любого узла дерева с ключем k каждый из ключей, расположенных в левом поддереве, меньше или равен k, а каждый из ключей, расположенных в правом поддереве, больше k. Такая структура хранения информации вместе со специальными алгоритмами балансировки деревьев приводит к тому, что операции поиска, вставки и удаления элемента выполняются за $O(\log n)$ сравнений, где n — это количество узлов в дереве.



Деревья сортирвки реализованы в стандартной библиотеке C++ в виде контейнеров `set` и `multiset`. Их отличие отличается в том, что `multiset` позволяет хранить равные элементы с точки зрения введенного линейного порядка, а `set` — нет.

Помимо ключей, каждый узел дерева может хранить значение произвольного типа, связанное с ключем. В стандартной библиотеке такие контейнеры называются отображениями (`map`) и объявляются в виде `map<T1,T2>`, где `T1` и `T2` типы ключа и значения соответственно. Отметим, что значение, связанное с ключем, не используется для поиска элемента, поэтому, оно может быть изменено в месте хранения.

Алгоритмы, реализованные в деревьях сортировки выполняют балансировку дерева после каждого изменения с целью не допустить появления слишком длинных ветвей. В силу этой особенности любые изменения в контейнерах на основе деревьев сортировки делают все итераторы недопустимыми.

Контейнеры `set`, `multiset` и map() поддерживают следующие операции.

- Поиск элемента в контейнере `X.find(val)`. Метод возвращает итератор на элемент равный `val`, если он существует в контейнере. Иначе возвращается итератор на `end()`. Таким образом, этот метод можно использовать для проверки того, записан ли некоторый элемент в контейнер.

- Добавление элемента в контейнер `X.insert(val)`. Для контейнера типа `set` запрос на добавление элемента, равного некоторому элементу контейнера, не имеет эффекта. Для контейнера `map` требуется указывать пару (`pair`) ключ и значение. Например, `X.insert({"hello",2})`.

- Удаление элемента из контейнера `X.erase(val)` или `X.erase(iter)`. Перввый вариант удаляет элемент по значению, второй — по итератору. Для контейнера `multiset` первый вариант удаляет все элементы равные `val`. Чтобы удалить один элемент требуется сначала взять итератор на него с использованием метода `find`, проверить что итератор не указывает на `end()`, и, только после этого удалить элемент через итератор.

- Методы `X.lower_bound(val)` и `X.upper_bound(val)`. Итераторы, предоставляемые контейнерами этого вида не являются итераторами произвольного доступа, поэтому обобщенные функции `lower_bound` и `upper_bound` будут работать с линейной сложностью $O(n)$. Поэтому в контейнере реализованы двойники этих функций, которые работают уже с логарифмической сложностью $O(\log n)$. Разумеется, использовать следует только их. При вызове метода из контейнера требуется указать искомый элемент, например `X.lower_bound(100)`.

Вместе с указанными методами контейнер `map` предоставляет удобный интерфейс для работы с использованием оператора индексации `[]`. Оператор индексации возвращает ссылку на значение по ключу. При этом, если указанного ключа в контейнере нет, то он создается со значением по умолчанию. Таким образом, если `map` используется, например, для связывания строк с целыми числами, то для добавления элемента отображения можно написать `X["hello"]=2`, для изменения — `X["hello"]+=7`. Если же использовать `X["hello"]++`, то число будет увеличиваться на единицу при каждом вополнении выражения, причем при первом вызове, эелемент отображения сначала будет создан со значением ноль, а после будет увеличен.

Вместе с тем использование условия `if␣(X["hello"]==0)` для проверки существования ключа `"hello"` в контейнере является плохим способом, поскольку если такого ключа не было, то он будет создан, что приведет к лишним затратам времени и памяти.

Контейнеры на основе хэш-таблиц

Контейнеры на основе хэш-таблиц, в целом, реализуют те же интерфейсы, как и контейнеры на основе деревьев сортировки, но отличаются от них по способу хранения элементов. В основе этих контейнеров лежит идея хэширования — псевдослучайное отображение элементов заданного множества на некоторый интервал натуральных чисел так, чтобы вероятность того, что два различных элемента отобразятся в одно число была незначительной. В ряде случаев контейнеры на основе хэш-таблиц работают быстрее аналогов на основе деревьев. При удачном выборе хэш-функции сложность выполнения основных операций с этими контейнерами составляет $O(1)$.

Контейнеры этого вида реализованы в стандартной библиотеке под именами `unordered_set`, `unordered_map`, `unordered_multiset`.

Нестандартизованные контейнеры

Персистентное дерево с неявным ключем (rope)

Этот контейнер не входит в состав стандартной библиотеки, но поддерживается рядом трансляторов. Изначально, `rope` является мощным инструментом для работы с длинными строками, но, благодаря применению шаблонов, позволяет работать и с другими типами данных. Контейнер `rope` сохраняет взаимное расположение элементов и предоставляет

итераторы произвольного доступа, поэтому с его помощью можно быстро обращаться к элементу контейнера по его порядковому номеру. Важной особенностью `rope` является то, что он обладает чертами персистентной структуры данных. Это означает, что можно скопировать все элементы или некоторый непрерыный подтрезок элементов в другую переменную и независимо их редактировать. Контейнер `rope` позволяет также удалять и вставлять элементы контейнера по их номерам а также подотрезки элементов. Все операции выполняются не медленнее чем со сложностью $O(\log n)$

Для работы с `rope` требуется подключить библиотеку `ext\rope` и пространство имен `__gnu_cxx`. Далее простой пример программы.

```
#include <iostream>
#include <ext/rope>
using namespace std;
using namespace __gnu_cxx;
int main() {
  rope<char> R="hello world";
  cout<<R;
  return 0;
}
```

Контейнер `rope` не перегружает чтение из входного потока, поэтому написать `cin>>R` нельзя. Но можно прочитать данные в строку а потом присвоить ее.

```
string s;
getline(cin,s);
rope<char> R=s.c_str();
```

Для создания `rope` из других элементов, например, целых чисел, можно использовать добавление в конец с помощью метода `push_back()` или функцию `copy` с адаптером `back_inserter`.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ext/rope>
using namespace std;
using namespace __gnu_cxx;
int main() {
  vector<int>  V={100,200,300,400};
  rope<int> T;
  copy(V.begin(),V.end(),back_inserter(T));
  for (int i=0;i<10;i++)
    T.push_back(i);
  for (auto x:T)
    cout<<x<<' ';
  return 0;
}
```

Примеры использования методов `rope`

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ext/rope>
using namespace std;
using namespace __gnu_cxx;
int main() {
```

```
  rope<int> T;
  for (int i=1;i<=10;i++)
    T.push_back(i);
  // Удалили 2 элемента, начиная с третьего  1 2 3 6 7 8 9 10
  T.erase(3,2);
  // сделали копию 4 элементов, начиная со второго 3 6 7 8
  rope<int> Q=T.substr(2,4);
  // заменили второй элемент 3 6 999 8
  Q.mutable_begin()[2]=999;
  // Вставили Q в седьмую позицию
  T.insert(7,Q);
  for (auto x:T)
    cout<<x<<' ';
  return 0;
}
```