Контейнеры, обобщенные алгоритмы

- Понятие контейнера
- Виды контейнеров
- Основные операции над контейнерами
- Требования к хранимым типам данных
- Шаблоны классов
- Шаблоны функций
- Итераторы
- Обобщенные алгоритмы

Понятие контейнера

- Контейнер — это тип данных, который предназначен для хранения однотипных элементов.
- Наибольший интерес представляют обобщенные контейнеры, которые могут хранить объекты произвольного типа.
- В языке Си++ определен только один вид контейнера: массив.
- Однако в библиотеке QT добавлены 10 разновидностей контейнеров.

Виды контейнеров

- Последовательные контейнеры:
  - вектор (QVector);
  - двусторонняя очередь или список (QList);
  - связанный список (QLinkedList).
- Ассоциативные контейнеры:
  - словарь (QMap);
  - хэш-таблица (QHash).

Последовательные контейнеры

- Последовательные контейнеры обеспечивают хранение однотипных величин в виде непрерывной последовательности.
- Обращение к элементам последовательного контейнера обычно выполняется по их индексам.
- Отличительной чертой последовательных контейнеров является возможность задания такого порядка элементов, который нам необходим.

Вектор (QVector)

- Вектор – это структура данных, которая хранит элементы, подобно обычному массиву.
- Все элементы векторы располагаются в непрерывной области памяти.

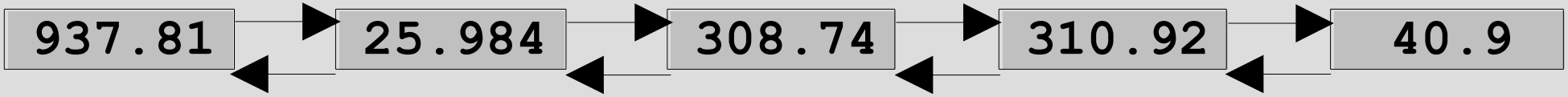
0	1	2	3	4
937.81	25.984	308.74	310.92	40.9

Вектор (QVector)

- Главное отличие вектора от массива Си++ состоит в том, что вектор всегда "знает", сколько элементов он хранит, и может динамически изменять свой размер.
- Добавление новых элементов в конец вектора выполняется очень быстро, но операция по вставке новых элементов в начало или в середину вектора требует значительных затрат по времени.

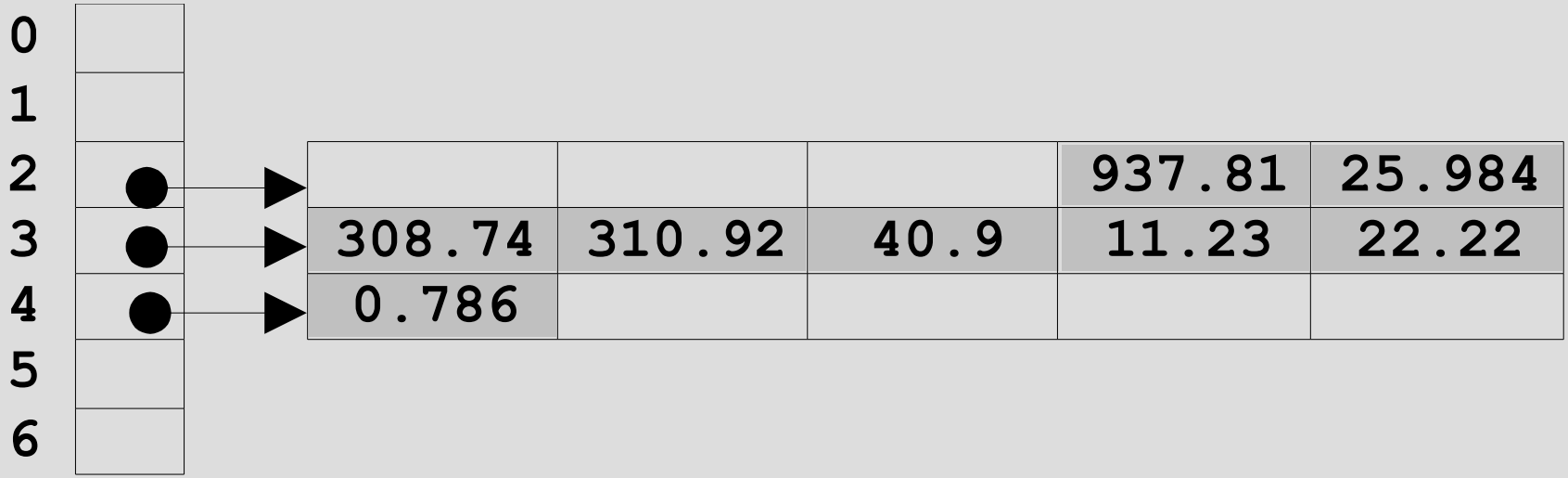
Связанный список (QLinkedList)

- Связанный список – это структура данных, которая может хранить элементы списка в областях памяти с произвольным размещением.
- Списки не предоставляют произвольного доступа к своим элементам (по индексу), но с другой стороны, функции «вставка» и «удаление» исполняются очень быстро.



Двусторонняя очередь или список (QList)

- Двусторонняя очередь эффективно реализует произвольный доступ к элементам, добавление в оба конца и удаление из обоих концов.



Основные операции над последовательными контейнерами

Операция	Описание
`operator<<`	Добавляет один элемент или содержимое другого контейнера в конец контейнера
`insert ()`	Вставляет элемент в заданную позицию
`remove ()`	Удаляет один или несколько элементов из заданной позиции
`clear ()`	Очищает контейнер
`last ()`	Возвращает последний элемент
`operator[]`	Возвращает элемент в заданной позиции. Не применим к контейнеру QLinkedList
`size ()`	Возвращает кол-во элементов в контейнере
`isEmpty ()`	Возвращает признак того, что контейнер пуст

Пример использования последовательного контейнера

```
// Создаем пустой вектор
QVector <int> vector;

// Заполняем вектор
vector << 12 << 13 << 3;

// Распечатываем содержимое вектора
for(int i=0; i < vector.size(); i++)
{
   printf("%d ", vector[i]);
}
```

Задание

Имеются два контейнера

```
QVector <int> vector;   // вектор целых чисел
QLinkedList <QString> list; // список строк
```

Перепишите содержимое вектора в связанный список, используя операцию добавления элемента в контейнер

```
operator<<(const T & value)
```

и статический метод преобразования числа в строку

```
QString QString::number (int n, int base=10)
```

Пример использования последовательного контейнера

```
list.clear(); // очищаем контейнер-приемник

// Копируем содержимое одного контейнера
// в другой
for(int i = 0; i < vector.size(); i++)
{
   list << QString::number(vector[i]);
}
```

Ассоциативные контейнеры

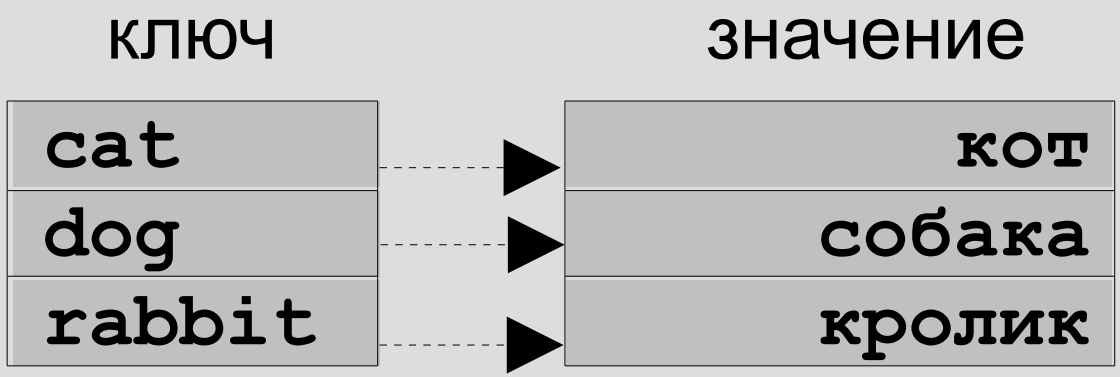
- В ассоциативных контейнерах каждый элемент хранит не только значение, но и уникальный ключ, однозначно определяющий этот элемент.
- В отличие от последовательных контейнеров доступ к элементам ассоциативного контейнера выполняется по ключу.

Ассоциативные контейнеры

- Отличительной чертой ассоциативных контейнеров является то, что порядок элементов определяется внутренней реализацией контейнера и его (порядок) практически невозможно изменить.

Словарь (QMap)

- Словари предназначены для хранения произвольного количества элементов в виде пар "ключ-значение".
- Элементы в словаре хранятся в упорядоченном виде в соответствии со значениями ключа.



Словарь (QMap)

- Словари обладают широкими возможностями доступа к произвольным элементам и незначительными накладными расходами на операцию добавления нового элемента.
- Если в словарь вставляется новое значение по существующему ключу, то оно затирает старое значение в паре "ключ-значение".

Хэш-таблица (QHash)

- Хэш-таблица имеет тот же набор операций, что и обычный словарь, но действия над ней выполняются значительно быстрее.
- Это достигается за счет использования хэш-функции, которая каждому значению ключа сопоставляет целочисленное значение.
- Поэтому для использования хэш-таблицы необходимо реализовать хэш-функцию.

Хэш-таблица (QHash)

- Желательно чтобы различным значениям ключа соответствовали различные значения хэш-функции.
- Для стандартных типов данных, а также класса QString хэш-функция уже определена.
- Порядок элементов в хэш-таблице непредсказуем и зависит от многих факторов.

Основные операции над ассоциативными контейнерами

Операция	Описание
`insert ()`	Добавляет элемент с заданным ключом и значением
`remove ()`	Удаляет элемент с заданным ключом
`clear ()`	Очищает контейнер
`value ()`	Возвращает значение элемента с заданным ключом
`operator[]`	Объединяет в себе операции `insert()` и `value()`: возвращает значение элемента с заданным ключом, если такой элемент существует, или добавляет элемент с заданным ключом и значением, если такого элемента не существует
`size ()`	Возвращает кол-во элементов в контейнере
`isEmpty ()`	Возвращает признак того, что контейнер пуст

Пример использования ассоциативного контейнера

```
// Создаем пустой англо-русский словарь
QMap <QString,QString>  dict;

// Заполняем словарь
dict.insert("dog", "собака");
dict.insert("cat", "кот");
dict.insert("rabbit", "кролик");

// Находим перевод слова
QString trnsl= dict.value("elephant");
if(!trnsl.isNull())  puts(qPrintable(trnsl));
else                 puts("no translation");
```

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Удалите из словаря город «Заозерск». Если такого города нет, то распечатайте сообщение «Город не найден». Используйте метод удаления элемента из контейнера по ключу

```
int remove ( const Key & key )
```

Метод возвращает кол-во удаленных элементов.

Пример использования ассоциативного контейнера

```
// Удаляем город «Заозерск»
int ok = cities.remove("Заозерск");

if(!ok)  puts("Город не найден");
```

Понятие шаблона класса

- В QT-библиотеке контейнеры реализованы как шаблоны классов.
- Шаблон класса представляет собой семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра.

Объявление шаблона класса

- Синтаксис объявления шаблона класса:

```
template <class T>
class имя_шаблона_класса
{
  // Тело шаблона класса
};
```

Класс T можно рассматривать как формальный параметр, на место которого при компиляции будет подставлен конкретный тип данных

Порождение нового типа данных на основе шаблона класса

- Для создания типа данных на основе шаблона класса используется следующая синтаксическая конструкция:

```
имя_шаблона_класса <тип>
```

- Поэтому для объявления переменной такого типа используется следующая синтаксическая конструкция:

```
имя_шаблона_класса <тип>
        имя_объекта(параметры_конструктора);
```

Пример создания типа данных на основе шаблона класса

- Объявление шаблона класса:

```
// Шаблон вектора
template <class T> class QVector
{ // Тело шаблона класса
};
```

- Тип данных на основе шаблона класса:

```
// Вектор вещественных чисел
QVector <float>
```

- Переменная нового типа данных:

```
// Экземпляр вектора вещественных чисел
QVector <float> vector;
```

Задание

Шаблон словаря задается следующим образом:

```
template <class Key, class T>
class QMap
{
   // Тело класса
};
```

Создайте новый тип данных - специализированный словарь, в котором хранятся названия месяцев года, а в качестве ключей используются номера месяцев.

Создайте переменную **MonthMap** нового типа данных.

Пример создания типа данных на основе шаблона класса

```
// Словарь месяцев года

QMap <int, QString> MonthMap;
```

Понятие шаблона функции

- Методы шаблона класса автоматически становятся шаблонами функций.
- С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции в виде параметра.
- Аргументы и возвращаемое значение шаблона функции могут иметь произвольный тип.

Пример шаблона функции

- У шаблона класса **QMap** имеется метод

```
const T value ( const Key & key ) const
```

  который является шаблоном функции, т.к. тип аргумента **key** заранее не определен, также не определен тип возвращаемого значения.

- Конкретные типы данных будут заданы при порождении специализированного словаря на основе шаблона класса **QMap**.

Пример шаблона функции

- Создание словаря месяцев года

```
QMap <int, QString> MonthMap;
```

приведет к созданию метода

```
const QString value (const int & key) const
```

Задание

Связанный список задается следующим шаблоном класса:

```
template <class T>
class QLinkedList
{
   ....
   T & last ();
};
```

Укажите заголовок метода **`last()`**, если на основе шаблона класса будет создан связанный список строк языка Си (список массивов символов).

Пример шаблона функции

- Связанный список строк задается как:

```
QLinkedList <char *> strings;
```

- В результате будет создана следующая версия метода **last()**

```
char * last ();
```

Требования к хранимым типам данных

- Типы значений, которые хранятся в контейнерах, должны поддерживать следующие операции:
  - конструктор по умолчанию;
  - конструктор копирования;
  - операция присваивания (operator=);
  - операция «равенство» (operator ==) для выполнения некоторых действий над векторами, списками и очередями.

Требования к хранимым типам данных

- Типы ключей, которые используются в ассоциативных контейнерах, должны поддерживать следующие операции:
  - операция «равенство» (operator ==) и наличие хэш-функции для хэш-таблицы;
  - операция «меньше» (operator <) для словаря.

Задание

Определите, могут ли классы **`QString`** и **`QDate`** использоваться в качестве типов хранимых значений или как тип ключа.

Решение

- Классы **`QString`** и **`QDate`** могут использоваться как типы хранимых значений, т.к. каждый из них имеет конструктор по умолчанию, конструктор копирования и операцию «присваивания».
- Класс **`QString`** может использоваться как тип ключа для словаря или хэш-таблицы, т.к. поддерживает операцию «меньше» и для него определена хэш-функция.

Решение

- Класс **QDate** может использоваться как тип ключа только для словаря, т.к. для него определена операция «меньше», но не определена хэш-функция.

- Чтобы использовать класс **QDate** как тип ключа для хэш-таблицы, должна быть объявлена хэш-функция, например такая:

```
uint qHash(const QDate &key)
{return qHash(key.day()*
          qHash(key.year()));}
```

Понятие итератора

- Во многих задачах требуется последовательный обход всех элементов контейнера.
- Существует два способа обхода контейнера:
  - обращение к элементам контейнера по их индексам;
  - использование итератора.
- Итератор — это класс, который используется для доступа к элементам контейнера.

Преимущества использования итератора

- Итератор существует для всех видов контейнеров, в то время как операция обращения по индексу определена только для последовательных контейнеров и то не для всех.
- Итератор обеспечивает максимально быстрый переход от одного элемента контейнера к другому.

Виды итераторов

- В QT-библиотеке контейнеры поддерживают следующие виды итераторов:
  - Java-Style:
    - константные;
    - модифицирующие.
  - STL-Style:
    - константные;
    - модифицирующие.

Модифицирующие итераторы

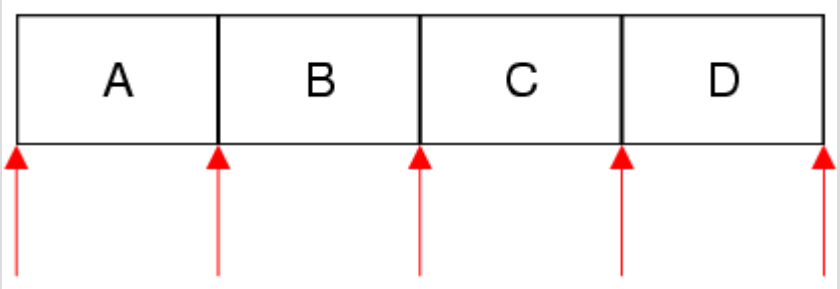
- Модифицирующие итераторы в отличие от константных позволяют:
  - изменять значения элементов контейнера;
  - производить удаление элементов из контейнера;
  - производить вставку элементов в контейнер.

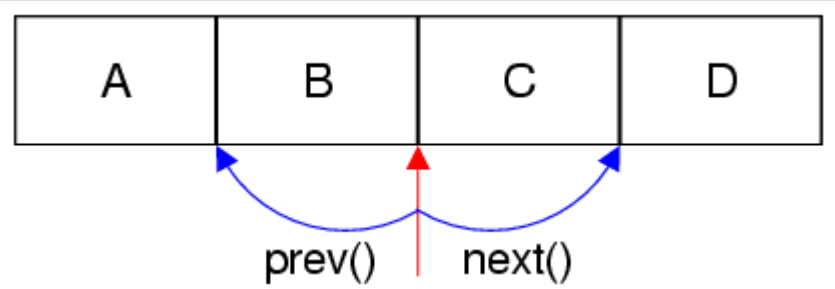
Константные итераторы

- Константные итераторы работают несколько быстрее своих неконстантных аналогов.
- Константные итераторы используются, когда нет необходимости изменять содержимое контейнера, например, при поиске элементов.

Java-итераторы

- Java-итератор указывает на позицию между двумя элементами контейнера.





Создание Java-итератора

- Для каждого вида контейнера существуют свои Java-итераторы, которые описываются отдельными классами-шаблонами

Контейнер	Итератор	
	Константный	Модифицирующий
QList<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

Создание Java-итератора

- Связь между итератором и контейнером задается во время создания итератора - контейнер передается как параметр его конструктора.
- Созданный итератор ссылается на позицию перед первым элементом контейнера.

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Создайте для контейнера константный и модифицирующий Java-итераторы, если известны их конструкторы :

```
QMapIterator ( const QMap<Key, T> & map )
QMutableMapIterator ( QMap<Key, T> & map )
```

Пример создания Java-итератора

```
// Создаем константный итератор для словаря
QMapIterator<QString,int> i1(cities);

// Создаем модифицирующий итератор для
// словаря
QMutableMapIterator<QString,int> i2(cities);
```

Основные операции над константными Java-итераторами

Операция	Описание
`hasNext()`	Возвращает признак существования следующего элемента
`hasPrevious()`	Возвращает признак существования предыдущего элемента
`next()`	Возвращает следующий элемент и перепрыгивает через него
`previous()`	Возвращает предыдущий элемент и перепрыгивает его
`toBack()`	Перемещает итератор в конец контейнера
`toFront()`	Перемещает итератор в начало контейнера

Основные операции над константными Java-итераторами

Операция	Описание
`key()`	Возвращает ключ элемента, через который был выполнен прыжок. Только для ассоциативных контейнеров (QMap и Qhash)
`value()`	Возвращает значение элемента, через который был выполнен прыжок. Только для ассоциативных контейнеров

Пример использования Java-итератора

```
// Создаем итератор для списка целых чисел
QListIterator<int> i(list);

// Распечатываем содержимое списка,
// используя итератор
while (i.hasNext())
{ printf("%d", i.next()); }
```

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Распечатайте название первого города и количество его жителей, используя константный итератор **QMapIterator** и его методы:

```
bool hasNext () const
Item next ()
const Key & key () const
const T & value () const
```

Пример использования Java-итератора

```
// Создаем итератор для словаря
QMapIterator<QString, int> i(cities);

if (i.hasNext())
{
    // Перепрыгиваем через первый элемент
    i.next();
    // Распечатываем информацию
    printf("%s %d", qPrintable(i.key()),
                    i.value());
}
```

Основные операции над модифицирующими Java-итераторами

- Итераторы позволяют не только читать содержимое контейнера, но и изменять его.

Операция	Описание
`remove ()`	Удаляет элемент, через который был выполнен прыжок
`setValue ()`	Задает значение элементу, через который был выполнен прыжок
`insert ()`	Выполняет вставку элемента в позицию, определяемую итератором, и перепрыгивает через добавленный элемент. Только для последовательных контейнеров

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Удалите города, количество жителей которых не превышает 1000 человек. Используйте модифицирующий итератор **QMutableMapIterator** и его методы:

```
bool hasNext () const
Item next ()
void remove ()
```

Пример использования Java-итератора

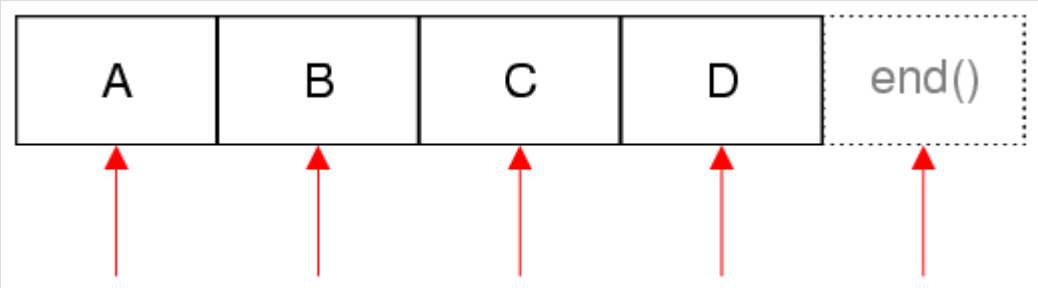
```
// Создаем итератор для словаря
QMutableMapIterator<QString, int> i(cities);
int val = 0;

// Удаляем города, в которых кол-во жителей
// менее 1000
while (i.hasNext())
{   // выполняем переход к следующему
    // элементу даже после удаления
    i.next();   val = i.value();

    if(val <= 1000)         i.remove();
}
```

STL-итераторы

- STL-итератор указывает на элемент контейнера.
- Признаком конца контейнера является «запоследний» элемент.



Создание STL-итератора

- Для каждого вида контейнера существуют свои STL-итераторы
- STL-итераторы объявлены как внутренние классы классов-контейнеров

Контейнер	Итератор	
	Константный	Модифицирующий
QList<T>	QList<T>::const_iterator	QList<T>::iterator
QLinkedList<T>	QLinkedList<T>::const_iterator	QLinkedList<T>::iterator
QMap<Key, T>	QMap<Key, T>::const_iterator	QMap<Key, T>::iterator
QHash<Key, T>	QHash<Key, T>::const_iterator	QHash<Key, T>::iterator

Создание STL-итератора

- Созданный экземпляр STL-итератора является недействительным, т.к. не связан ни с одним контейнером.
- Для получения итератора на первый или «за-последний» элемент контейнера необходимо использовать методы контейнера `begin()`, `constBegin()`, `end()` или `constEnd()`.

Создание STL-итератора

- Методы `begin()` и `constBegin()` возвращают итератор, который ссылается на первый элемент контейнера.
- Методы `end()` и `constEnd()` возвращают итератор на «запоследний» элемент.
- Если контейнер пустой, то методы `begin()`, `constBegin()`, `end()` и `constEnd()` возвращают итератор на «запоследний» элемент.

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Создайте для контейнера константный и модифицирующий STL-итераторы.

Установите константный итератор на первый элемент контейнера, а модифицирующий - на «запоследний».

Пример создания STL-итератора

```
// Создаем константный итератор для словаря
QMap<QString,int>::const_iterator i1;

// Устанавливаем итератор на первый элемент
i1  = cities.constBegin();

// Создаем модифицирующий итератор для
// словаря
QMap<QString,int>::iterator i2;

// Устанавливаем итератор на «запоследний»
// элемент
i2  = cities.end();
```

Основные операции над константными STL-итераторами

Операция	Описание
*	Возвращает значение элемента
==	Возвращает true, если два итератора ссылаются на один и тот же элемент
!=	Возвращает true, если два итератора ссылаются на различные элементы
++	Перемещение итератора на следующий элемент (префиксная и постфиксная операция)
--	Перемещение итератора на предыдущий элемент (префиксная и постфиксная операция)
+=	Перемещение итератора на несколько элементов вперед
-=	Перемещение итератора на несколько элементов назад

Основные операции над константными STL-итераторами

Операция	Описание
`+`	Получение нового итератора, который ссылается на несколько элементов вперед.
`-`	Получение нового итератора, который ссылается на несколько элементов назад.
`key ()`	Возвращает ключ текущего элемента. Только для ассоциативных контейнеров
`value ()`	Возвращает значение текущего элемента. Только для ассоциативных контейнеров

Пример использования STL-итератора

```
QList<int> list;
...
// Создаем итератор для списка
QList<int>::const_iterator i;

// Распечатываем содержимое списка,
// используя итератор
for (i =  list.constBegin();
      i != list.constEnd(); ++i)
{ printf("%d", *i); }
```

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Распечатайте название первого города и количество его жителей, используя константный STL-итератор и его операции:

```
const Key & key () const
const T & value () const
```

Пример использования STL-итератора

```
// Создаем итератор для словаря
QMap<QString, int>::const_iterator i;

// Распечатываем информацию о первом городе,
// если он имеется
i =  cities.constBegin();
if(i != cities.constEnd())
{
   printf("%s %d", qPrintable(i.key()),
                          i.value());
}
```

Основные операции над модифицирующими STL-итераторами

- Итераторы позволяют не только читать содержимое контейнера, но и изменять его.

Операция	Описание
`*`	Возвращает ссылку на элемент
`value ()`	Возвращает ссылку на текущий элемент. Только для ассоциативных контейнеров

Основные операции над модифицирующими STL-итераторами

- Модифицирующие итераторы используются в операциях контейнера, которые изменяют его содержимое.

`iterator erase (iterator pos)`	Удаляет элемент в заданной позиции. Возвращается итератор на следующий элемент.
`iterator erase (iterator begin, iterator end)`	Удаляет заданную последовательность элементов. Возвращается итератор на следующий элемент. Только для последовательных контейнеров
`iterator insert (iterator before, const T & value)`	Выполняет вставку элемента в заданную позицию. Возвращает итератор на добавленный элемент. Только для последовательных контейнеров

Задание

Имеется словарь городов. В словаре хранится название города и кол-во его жителей

```
QMap <QString,int> cities;
```

Удалите города, количество жителей которых не превышает 1000 человек. Используйте модифицирующий итератор и метод контейнера:

```
iterator erase ( iterator pos )
```

Пример использования STL-итератора

```
// Создаем итератор для словаря
QMap<QString, int>::iterator i;

// Удаляем города, в которых кол-во жителей
// менее 1000
i = cities.begin();
while(i != cities.end())
{
   if(*i <= 1000)      i = cities.erase(i);
   else                ++i;
}
```

Недействительные итераторы

- Если в результате действий над контейнером изменяется расстановка его элементов и/или их количество, то итератор становится недействительным.
- Недействительный итератор либо указывает на несуществующий элемент, либо «перескакивает» на произвольный элемент.

Пример недействительного STL-итератора

```
// Список целых чисел
QList<int>list;

// Итераторы на первый и последний элементы
QList<int>::iterator  first;
QList<int>::iterator  last;

list << 2 << 42 << 12;  // заполняем список
```

Пример недействительного STL-итератора

```
// Устанавливаем итераторы на первый и
// последний элементы
first = list.begin();
last = list.end() - 1;

// Удаляем первый элемент
// Итераторы на первый и последний элементы
// становятся недействительными
list.erase(first);

// Результат не предсказуем
printf("%d\n", *first);
printf("%d\n", *last);
```

Восстановление недействитель-ных Java-итераторов

- Для восстановления Java-итератора исполь-зуется операция присваивания самого итератора:

```
QContainerIterator & operator=
( const QContainer<T> & container )
```

где `Container` — конкретный вид контейне-ра, например,

```
QListIterator & operator=
( const QList<T> & list )
```

Восстановление недействитель-ных STL-итераторов

- Для восстановления STL-итератора исполь-зуются методы контейнера:

`iterator begin ()` - возвращает итератор на первый элемент

`const_iterator constBegin () const` - возвращает итератор на первый элемент

`iterator end ()` - возвращает итератор на последний элемент

`const_iterator constEnd () const` - воз-вращает итератор на последний элемент

Пример восстановления недействительных итераторов

```
// Создаем и заполняем список целых чисел
QList<int> list;   list << 12 << 21 << 5;

// Устанавливаем итераторы на первый элемент
QMutableListIterator<int> java(list);
QList<int>::iterator stl = list.begin();

// Удаляем первый элемент — итераторы
// становятся недействительными
list.removeAt(0);

// Восстанавливаем итераторы
java = list;  stl = list.begin();
```