Библиотека контейнеров Qt

Читать в книге Шлее (QT 5.10): гл. 4

Библиотека Tulip – встроенный в QT (модуль QtCore) аналог STL (Standard Template Library, стандартная библиотека шаблонов языка C++). Задача любых контейнеров, в сущности, проста – это организация обработки групп элементов.
В основе библиотеки лежат 3 понятия:

- контейнеры - классы, способные хранить в себе элементы различных типов данных;
- алгоритмы – операции преобразования над элементами контейнеров, такие, как сортировка, поиск, сравнение и т.п.;
- итераторы - связывают контейнеры и алгоритмы, позволяют перемещаться по элементами контейнера, абстрагируясь от конкретной структуры данных.

Для демонстрации примеров нам будет достаточно консольного приложения QT с совсем простым шаблоном. Выберем в QT Creator меню Файл, Создать файл или проект…, затем шаблон "Консольное приложение QT" и проделаем стандартные шаги по настройке проекта. Всё содержимое файла main.cpp можно удалить, заменив его на такое:

```
#include <QtCore>
int main() {
 return 0;
}
```

В заголовочном файле QtCore все контейнеры уже прописаны, а "остановку" перед завершением консольного приложения QT обеспечит сам. Весь вывод будем выполнять с помощью стандартной функции qDebug(), например, так:

```
#include <QtCore>
int main() {
    int n=5;
    QVector <int> v(n,100);
    qDebug() << n << v;
    return 0;
}
```

На экран выведется
5 QVector(100, 100, 100, 100, 100)
что очень удобно – функция qDebug() принимает любые типы данных, а для составных типов, таких, как вектор, ещё и печатает имя класса.

Замечание. Переключать режим вывода консольного приложения между нижней панелью "Вывод приложения" и отдельным окном терминала можно с помощью опции Проекты - Запуск - Запускать в терминале.

1. Контейнерные классы (контейнеры) делятся на:
Последовательные (упорядоченные коллекции, в которых каждый элемент занимает определенную позицию):

- QVector<T> — вектор;
- QList<T> — список;
- QLinkedList<T> — двусвязный список;
- QStack<T> — стек;
- QQueue<T> — очередь.

Ассоциативные (коллекции, в которых позиция элемента зависит от его значения):

- QSet<T> — множество;
- QMap<K,T> — словарь, хранящий соответствия ключей типа K и значений типа T (значения хранятся упорядоченными по ключу);
- QMultiMap<K,T> — мультисловарь, может связывать с одним ключом множество значений;
- QHash<K,T> — хэш, набор пар "ключ-значение", данные хранятся в произвольном порядке;
- QMultiHash<K,T> — мультихэш, может связывать с одним ключом хэша множество значений.

Здесь

- <T> - обозначение типа данных (например, QString, int)

- <K> - тип данных ключа, по которому упорядочиваются данные (только для указанных типов контейнеров).

Общие операторы и методы всех контейнеров	
Оператор, метод	Описание
== и !=	Операторы сравнения, равно и не равно
=	Оператор присваивания
[]	Оператор индексации. Исключение составляют только классы QSet<T> и QLinkedList<T>, в них этот оператор не определен
begin() и constBegin()	Методы, возвращающие итераторы, установленные на начало последовательности элементов контейнера. Для класса QSet<T> возвращаются только константные итераторы
end() и constEnd()	Методы, возвращающие константные итераторы, установленные на конец последовательности элементов контейнера
clear()	Удаление всех элементов контейнера
insert()	Операция вставки элементов в контейнер
remove()	Операция удаления элементов из контейнера
size() и count()	Оба метода идентичны — возвращают количество элементов контейнера, но применение первого предпочтительно, т. к. соответствует STL
value()	Возвращает значение элемента контейнера. В QSet<T> этот метод не определен
empty() и isEmpty()	Возвращают true, если контейнер не содержит ни одного элемента. Оба метода идентичны, но применение первого предпочтительно, т. к. соответствует STL

Здесь и далее предполагается, что методы могут иметь различные перегрузки, а конкретные классы, конечно, содержат и уникальные для них методы.

Общие методы последовательных контейнеров	
Оператор/метод	Описание
+	Объединяет элементы двух контейнеров
+=	Добавляет элемент в контейнер (то же, что и <<)
<<	Добавляет элемент в контейнер
at()	Возвращает указанный элемент
back() и last()	Возвращают ссылку на последний элемент. Эти методы предполагают, что контейнер не пуст. Оба метода back() и last() идентичны, но применение первого предпочтительнее, т. к. он соответствует STL
contains()	Проверяет, содержится ли переданный в качестве параметра элемент в контейнере
erase()	Удаляет элемент, расположенный на позиции итератора, передаваемого в качестве параметра
front() и first()	Возвращают ссылку на первый элемент контейнера. Методы предполагают, что контейнер не пуст. Оба метода front() и first() идентичны, но применение первого более предпочтительно, т. к. он соответствует STL
indexOf()	Возвращает позицию первого совпадения найденного в контейнере элемента, в соответствии с переданным в метод значением. Внимание: в контейнере QLinkedList этот метод отсутствует
lastIndexOf()	Возвращает позицию последнего совпадения найденного в контейнере элемента, в соответствии с переданным в метод значением. Внимание: в контейнере QLinkedList этот метод отсутствует
mid()	Возвращает контейнер, содержащий копии элементов, задаваемых начальной позицией и количеством
pop_back()	Удаляет последний элемент контейнера
pop_front()	Удаляет первый элемент контейнера
push_back() и append()	Методы добавляют один элемент в конец контейнера. Оба метода идентичны, но применение первого предпочтительно, т. к. он соответствует STL
push_front() и prepend()	Методы добавляют один элемент в начало контейнера. Оба метода идентичны, но применение первого предпочтительно, т. к.

	он соответствует STL
replace()	Заменяет элемент, находящийся на заданной позиции, значением, переданным как второй параметр

Общие методы ассоциативных контейнеров	
Метод	Описание
contains()	Возвращает значение true, если контейнер содержит элемент с заданным ключом. Иначе возвращается значение false
erase()	Удаляет элемент из контейнера в соответствии с переданным итератором
find()	Осуществляет поиск элемента по значению. В случае успеха возвращает итератор, указывающий на этот элемент, а в случае неудачи итератор указывает на метод end()
insertMulti()	Вставляет в контейнер новый элемент. Если элемент уже присутствует в контейнере, то создается новый элемент. Данный метод отсутствует в классе QSet<T>
insert()	Вставляет в контейнер новый элемент. Если элемент уже присутствует в контейнере, он замещается новым элементом. Данный метод отсутствует в классе QSet<T>
key()	Возвращает первый ключ в соответствии с переданным в этот метод значением. Данный метод отсутствует в классе QSet<T>
keys()	Возвращает список всех ключей, находящихся в контейнере. Данный метод отсутствует в классе QSet<T>
take()	Удаляет элемент из контейнера в соответствии с переданным ключом и возвращает копию его значения. Данный метод отсутствует в классе QSet<T>
unite()	Добавляет элементы одного контейнера в другой
values()	Возвращает список всех значений, находящихся в контейнере

1.1. **Вектор QVector** похож на обычный массив, но можно узнать количество элементов внутри вектора (размерность) методом size(), а также динамически расширять вектор. Вектор экономнее по памяти и ресурсам, чем другие виды контейнеров. Добавление элементов в конец вектора делается методом push_back(), а обращение к отдельным элементам – оператором [] или с помощью итератора:

```
QVector<int> vec;
vec.push_back(10); //push_back сам увеличит размерность с 0 до 1
vec.resize(2);
vec[1] = 20; //А квадратные скобки - нет, сначала зарезервировали место
qDebug() << vec;
```

Не сложнее было бы организовать и вектор строк, а элементы можно добавлять операцией "+=":

```
QVector <QString> vs;
vs.append("Item1");
vs += "Item2";
qDebug() << vs;
QVector <int> vi;
vi.push_back(1);
vi += 2;
qDebug() << vi;
```

Некоторые методы контейнера QVector<T>	
Метод	Описание
data()	Возвращает указатель на данные вектора (т. е. на обычный массив)
fill()	Присваивает одно и то же значение всем элементам вектора
reserve()	Резервирует память для количества элементов, в соответствии с переданным значением
resize()	Устанавливает размер вектора в соответствии с переданным значением
toList()	Возвращает объект QList с элементами, содержащимися в векторе
toStdVector()	Возвращает объект std::vector с элементами, содержащимися в векторе

1.2. **QList** – это базовый шаблон списка, имеющий множество методов для работы с его элементами.

Некоторые методы контейнера QList<T>	
Метод	Описание
move()	Перемещает элемент с одной позиции на другую
removeFirst()	Удаляет первый элемент списка
removeLast()	Удаляет последний элемент списка
swap()	Меняет местами два элемента на указанных позициях
takeAt()	Возвращает элемент на указанной позиции и удаляет его
takeFirst()	Удаляет первый элемент и возвращает его
takeLast()	Удаляет последний элемент и возвращает его
toSet()	Возвращает контейнер QSet<T> с данными, содержащимися в объекте QList<T>
toStdList()	Возвращает стандартный список STL std::list<T> с элементами, содержащимися в объекте QList<T>
toVector()	Возвращает объект вектора QVector<T> с элементами, содержащимися в объекте QList<T>

Если вы не собираетесь менять значения элементов, из соображений эффективности не рекомендуется использовать оператор []. Вместо него есть метод at(), так как он возвращает константную ссылку на элемент.
Самая распространённая задача со списком – его обход для последовательного получения значений каждого элемента:

```
QList <int> list;
list << 10 << 20 << 30;
QList <int>::iterator it = list.begin();
while (it != list.end()) {
 qDebug() << "Element: " << *it++;
}
```

Покажем формирование списка строк вводом с консоли и обмен местами первого элемента с последним:

```
#include <QtCore>
#include <iostream>
int main() {
 QList <QString> list;
 std::cout << "Enter the strings, 0 is the end of input" << std::endl;
 QTextStream qtcin(stdin);
 QString s;
 for (;;) {
  s = qtcin.readLine();
  //qtcin >> s; //если чтение до пробела
  if (s.compare("0")==0) break;
  list << s;
 }
 list.swapItemsAt(0,list.size()-1);
 qDebug() << list;
 return 0;
}
```

1.3. Массив байт **QByteArray**, в отличие от вектора, не является шаблоном, в нём допускается хранение только элементов, имеющих размер в один байт. Объекты типа QByteArray можно использовать везде, где требуется промежуточное хранение данных. Количество элементов массива можно задать в конструкторе, а доступ к ним получать при помощи оператора [].

```
QByteArray arr(3,'0'); //Массив байт QByteArray
arr[0] = 0x23; arr[2]=65;
qDebug() << arr.data(); //#0A
```

К данным объектов класса QByteArray можно также применить операцию сжатия и обратное преобразование. Это достигается при помощи двух глобальных функций qCompress() и qUncompress(). Сожмём и разожмем данные строки char *:

```
QByteArray a = "Some text...";
QByteArray aCompressed = qCompress(a);
```

```
a = qUncompress(aCompressed);
qDebug() << a;
```

1.4. **QBitArray** – это булев массив, каждое из значений которого занимает 1 бит, не расходуя лишней памяти. Этот тип используется для хранения большого количества переменных типа bool.

```
QBitArray arr(3);
arr[0] = arr[1] = true; arr[2] = false;
qDebug() << arr;
```

1.5. **Стек QStack** – это класс-наследник QVector, он реализует структуру данных, работающую по принципу LIFO (Last In, First Out — последним пришёл, первым ушёл). То есть, из стека первым удаляется элемент, который был вставлен позже всех остальных. Процесс помещения элементов в стек обычно называется "проталкиванием" (pushing), а извлечение из него верхнего элемента — "выталкиванием" (poping). Каждая операция проталкивания увеличивает размер стека на 1, а каждая операция выталкивания уменьшает его на 1. Для этих операций в классе QStack определены функции push() и pop().

```
QStack <QString> stack;
stack.push("Hedgehog");
stack.push("Woodpecker");
stack.push("Fox");
while (!stack.empty()) {
 qDebug() << "Element: " << stack.pop();
}
```

1.6. **Очередь QQueue** реализует структуру данных, работающую по принципу — FIFO (First In, First Out — первым пришёл, первым ушёл). Класс очереди QQueue унаследован от QList.

```
QQueue <QString> q; //Очередь QQueue
q.enqueue("Str1");
q.enqueue("Str 2");
while (!q.empty()) qDebug() << q.dequeue();
qDebug() << q.size();
```

1.7. **Множества QSet** удобны тем, что поддерживают уникальность ключей и стандартные операции над множествами - объединение, пересечение, разность. Контейнер QSet можно использовать в качестве неупорядоченного списка для быстрого поиска данных.

```
QSet <int> set1, set2;
set1 << 1 << 2 << 3;
set2 << 3 << 4 << 2 << 5;
qDebug() << set1.unite(set2); // set1 U set2 = (1, 3, 2, 5, 4)
```

Некоторые методы контейнера QSet <T>

Метод	Описание
intersect()	Удаляет элементы множества, не присутствующие в переданном множестве (пересечение множеств)
reserve()	Задает размер хэш-таблицы множества
squeeze()	Уменьшает объем внутренней хэш-таблицы для экономии памяти
subtract()	Удаляет элементы множества, присутствующие в переданном множестве (разность множеств)
toList()	Возвращает объект типа QList<T>, содержащий элементы множества QSet<T>
unite()	Объединяет элементы двух множеств

Покажем реализацию основных операций над множествами (объединение, пересечение и разность).

```
 QSet <int> set1;
 QSet <int> set2;
 set1 << 1 << 2 << 3;
 set2 << 2 << 3 << 4;
 QSet <int> setUnion = set1;
 setUnion.unite (set2);
 qDebug() << "Union = " << setUnion;
 QSet <int> setIntersect = set1;
 setIntersect.intersect(set2);
```

```
 qDebug() << "Intersection = " << setIntersect;
 QSet <int> setSubtract = set1;
 setSubtract.subtract(set2);
 qDebug() << "Subtraction = " << setSubtract;
```

1.8. **Словари QMap, QMultiMap** – контейнеры, действительно похожи на обычные словари, они хранят элементы одного и того же типа, индексируемые ключевыми значениями. Основное достоинство словарей в том, что они позволяют быстро получать значение, сопоставленное (*ассоциированное*) с заданным ключом. В QMap при этом ключи должны быть уникальными, а QMultiMap допускает дубликаты ключей, то есть, одному ключу может быть сопоставлено несколько значений. И ключами, и элементами могут выступать значения любого типа данных.
При создании объекта QMap нужно указать, какого типа данных будут ключи и значения.

```
QMap <QString,QString> map; //Словарь с ключом-строкой и значением-строкой
map["Piggy"]="+79131234567"; //ключ-имя,значение-телефон
map["Kermit"]="+79539990203";
qDebug() << map;
```

Частый способ обращения к элементам словаря - использование ключа в операторе []. Можно обойтись и без этого, так как ключ и значение можно получить через методы итератора key() и value(), например:

```
QMap <QString,QString> map;
map["Piggy"]="+79131234567";
map["Kermit"]="+79539990203";
QMap <QString,QString>::iterator it;
for (it = map.begin(); it != map.end(); ++it) {
 qDebug() << "Name:" << it.key() << "Phone:" << it.value();
}
```

Если же одному имени можно сопоставлять несколько телефонных номеров, поможет QMultiMap:

```
QMultiMap <QString,QString> mmap; //Словарь, допускающий дублирование ключей
mmap.insert("Mike","6400508");
mmap.insert("Sopha","3332702");
mmap.insert("Sopha","+79531170409");
qDebug() << mmap;
```

1.9. **Хэши QHash и QMultiHash**, в принципе, похожи на QMap, но вместо сортировки по ключу они используют специальную хэш-таблицу, что позволяет искать ключи гораздо быстрее, чем на QMap.
Как и в случае с QMap, нужно соблюдать осторожность при использовании оператора индексации [], так как задание ключа, для которого элемент не существует, приведет к тому, что элемент будет создан. Важно всегда проверять наличие элемента, привязанного к ключу при помощи метода contains() контейнера.

```
QHash <int,QString> array; //Здесь ключами служат целые числа
array[0]="Peppa";
array[10]="Kermit";
if (array.contains(10)) array[10]="TheFrog"; //Проверка, есть ли ключ 10 в хэше
qDebug() << array;
```

Класс QMultiHash унаследован от QHash. Он позволяет размещать значения с одинаковыми ключами и, в целом, похож на QMultiMap, но учитывает специфику своего родительского класса.

В завершение обзора контейнеров реализуем пример, который везде обсуждается, но нигде не приведён полностью. Если нужно разместить в QHash объекты собственных классов, потребуется реализовать оператор сравнения == и функцию qHash() для вашего класса. Функция qHash() при этом возвращает число, которое должно быть уникальным для каждого находящегося в хэше элемента.

```
#include <QtCore>
class MyClass { //Класс с фамилией и именем
 QString _firstName,_secondName;
public:
 MyClass(QString firstName=0,QString secondName=0) { //Конструктор
 _firstName=firstName; _secondName=secondName;
```

```
 }
 QString firstName() { return _firstName; }
 QString secondName() { return _secondName; }
 QString printable() { //строка для вывода объекта
  return _firstName+" "+_secondName;
 }
 bool operator == (MyClass &e2) { //Оператор сравнения
  return _firstName == e2.firstName() && _secondName == e2.secondName();
 }
 uint qHash(MyClass key, uint seed) { //Функция qHash()
  return qHash(key.firstName(),seed) ^ qHash(key.secondName(),seed);
 }
 MyClass & operator = (MyClass *e2) { //Оператор присваивания
  _firstName = e2->firstName();
  _secondName = e2->secondName();
  return *this;
 }
};
int main() {
 QHash <QString,MyClass> lst; //Ключом будет строка, элементом - объект класса
 lst["Piggy"] = new MyClass("Peppa","Piggy");
 lst["Kermit"] = new MyClass("Kermit","TheFrog");
 QHash <QString,MyClass>::iterator it = lst.begin();
 for (;it != lst.end(); ++it) {
  qDebug() << "Key: " << it.key() << "Value:" << it.value().printable();
 }
 return 0;
}
```

2. Итераторы позволяют перемещаться по элементам контейнера, абстрагируясь от структуры данных.

Методы QListIterator, QLinkedListIterator, QVectorIterator, QHashIterator, QMapIterator , QSetIterator	
Метод	Описание
toFront()	Перемещает итератор на начало списка
toBack()	Перемещает итератор на конец списка
hasNext()	Возвращает значение true, если итератор не находится в конце списка
next()	Возвращает значение следующего элемента списка и перемещает итератор на следующую позицию
peekNext()	Просто возвращает следующее значение без изменения позиции итератора
hasPrevious()	Возвращает значение true, если итератор не находится в начале списка
previous()	Возвращает значение предыдущего элемента списка и перемещает итератор на предыдущую позицию
peekPrevious()	Возвращает предыдущее значение без изменения позиции итератора
findNext(const T&)	Поиск заданного элемента в прямом направлении
findPrevious(const& T)	Поиск заданного элемента в обратном направлении

Например, здесь мы сканируем список строк с помощью итератора

```
 QList <QString> list;
 list << " Item1 " << " Item2 " << " Item3 ";
 QListIterator <QString> it(list);
 while(it.hasNext()) {
  qDebug() << it.next();
 }
```

Если необходимо производить изменения в процессе прохождения итератором элементов, то для этого следует воспользоваться изменяющимися (mutable) итераторами. Их классы называются аналогично, но с добавлением "Mutable": QMutableListIterator, QMutableHashIterator, QMutableLinkedListIterator, QMutableMapIterator и

QmutableVector-Iterator. Метод remove() удаляет текущий элемент, а insert() производит вставку элемента в текущую позицию. При помощи метода setValue() можно присвоить элементу другое значение.
Здесь выполняется замена значения элемента в списке:

```
 QList <QString> list;
 list << "Item1" << "Item2" << "Item3";
 QMutableListIterator <QString> it(list);
 while (it.hasNext()) if (it.next()=="Item3") it.setValue("3Item");
qDebug() << list;
```

В QT также работают итераторы в стиле STL:

```
QVector <QString> vec;
 vec << "Item1" << "Item2" << "Item3";
 QVector<QString>::iterator it = vec.begin();
 for (; it != vec.end(); ++it) { // Прямой проход по вектору строк
  qDebug() << "Element:" << *it;
 }
 QVector<QString>::iterator itb = vec.end();
 for (;itb != vec.begin();) { // Обратный проход по вектору строк
  --itb;
  qDebug() << "Element:" << *itb;
 }
```

3. Алгоритмы изначально определены в заголовочном файле QtAlgorithms и представляют собой операции, применяемые к контейнерам, такие как сортировка, поиск, преобразование данных и т.д. Следует отметить, что алгоритмы реализованы не в виде методов контейнерных классов, а в виде шаблонных функций, что позволяет использовать их как для любого контейнерного класса, так и для обычных массивов.
В настоящее время многие собственные реализации алгоритмов Qt объявлены устаревшими и вместо них рекомендуется использовать соответствующие алгоритмы STL из пространства имён std.
Например, для **копирования** элементов из одного массива в другой можно задействовать алгоритм std::copy():

```
 QString values[] = {"Xandria", "Therion", "Nightwish", "Haggard"};
 const int n = sizeof(values) / sizeof(QString);
 QString copyOfValues[n];
 std::copy (values, values + n, copyOfValues);
 for (int i = 0; i < n; i++) qDebug() << copyOfValues[i];
```

Сортировка списка строк с помощью алгоритма:

```
 QList<QString> list;
 list << "gamma" << "beta" << "alpha";
 std::sort(list.begin(),list.end());
 qDebug() << "Sorted list=" << list;
```

Поиск в списке строк значения с помощью алгоритма:

```
QList<QString> list;
 list << "alpha" << "beta" << "gamma";
 QList<QString>::iterator it =
 std::find(list.begin(), list.end(), "gamma");
 if (it != list.end()) {
  qDebug() << "Found=" << *it;
 }
 else {
  qDebug() << "Not Found";
 }
```

Если найденных строк может быть несколько, возможна организация типового **цикла поиска** значений:

```
 QList<QString> list;
 list << "gamma" << "alpha" << "beta" << "gamma";
 bool found = false;
 QList<QString>::iterator it = list.begin();
 do {
  it = std::find(it, list.end(), "gamma");
  if (it != list.end()) {
```

```
   qDebug() << "Found=" << *it << " at " << (it-list.begin());
   found = true;
   ++it;
  }
 } while (it != list.end());
 if (!found) {
  qDebug() << "Not Found";
 }
```

Для **перебора** всех элементов контейнера удобен также специально введённый в QT цикл foreach:

```
 QList <QString> list;
 list << "alpha" << "beta" << "gamma";
 foreach (QString item, list) {
   qDebug() << "Item = " << item;
 }
```

QT делает копию контейнера при входе в foreach, поэтому данный цикл не предназначен для изменения контейнера.

4. Примеры реализации задач на контейнеры QT

Задача 4.1. Найти количество вхождений минимального элемента в целочисленный список (консольное приложение).

```
#include <QtCore>

int cnt_min (QVector <int> list) {
 std::sort(list.begin(),list.end());
 int cnt = 0, min = list.at(0);
 QVectorIterator <int> it(list);
 while(it.hasNext() && it.peekNext()==min) { cnt++; it.next(); }
 return cnt;
}

int main() {
 QVector <int> list;
 list << -5 << 5 << -1 << -5 << 3 << 3 << -5;
 qDebug() << cnt_min(list);
 return 0;
}
```

Временная сложность этой реализации будет не ниже, чем для сортировки вектора, то есть, O(N log N). Применив пару стандартных алгоритмов, задачу такого типа можно решить и за время O(2*N):

```
//Определить, сколько раз входит в очередь максимальный элемент
#include <QtCore>

int main() {
 QQueue <int> a;
 a << 4 << 2 << -3 << 4 << -1;
 int max = *std::max_element(a.begin(),a.end());
 qDebug() << std::count(a.begin(),a.end(),max);
 return 0;
}
```

Отметим, что код напрямую не портируется на std::queue из-за разниц реализации контейнеров STL и Qt.

Задача 4.2. Удалить из очереди строк элементы короче 4 символов (консольное приложение).

```
#include <QtCore>

QQueue <QString> delete_ (QQueue <QString> q,int len) {
 QQueue <QString> q2;
 while (!q.isEmpty()) {
```

```
  QString qs=q.dequeue();
  if (qs.size()>=len) q2.enqueue(qs);
 }
 return q2;
}

int main() {
 QQueue <QString> queue;
 queue << "It's" << "my" << "life" << "la-la" << "," << "hello";
 QQueue <QString> queue2 = delete_(queue,4);
 qDebug() << queue2;
 return 0;
}
```

Задачи подобного типа можно было бы решить и с помощью алгоритма std::remove_if:

```
// Сделать копию стека без отрицательных элементов
#include <QtCore>

int main() {
 QStack <int> a;
 a << 1 << 2 << -3 << 4 << -1;
 qDebug() << "Stack: " << a;
 QStack <int> b = a;
 b.erase(std::remove_if(b.begin(), b.end(),
  [](int val){return val < 0;}), b.end());
 qDebug() << "Copy: " << b;
   return 0;
}
```

Не всегда применение стандартных алгоритмов даёт выигрыш в производительности, например, реализация следующей задачи выглядит вполне прозрачной.

Задача 4.3. Определить, есть ли в стеке 2 одинаковых соседних элемента.

```
#include <QtCore>

int main() {
 QStack <int> a;
 a << 4 << 2 << -3 << 4 << -1 << -1;
 bool found = false;
 for (int i = 0; i < a.size()-1; i++)
  if (a[i] == a[i+1]) {
   found = true; break;
  }
 qDebug() << found;
 return 0;
}
```

Недостаток приведённых выше кодов можно увидеть в том, что мы не применяем шаблонных решений, позволяющих работать с контейнерами из элементов различных типов данных одних и тем же кодов. Покажем пример решения, использующего шаблонную функцию.

Задача 4.4. Реализовать циклический сдвиг списка влево или вправо на указанное количество элементов.

```
#include <QtCore>
#include <QtMath>

template <class T>
void shift (QList <T> &numbers, int size) {
 if (size < 0) { //влево
  size = qAbs(size) % numbers.size();
  if (size)
   std::rotate(numbers.begin(),numbers.begin()+size,numbers.end());
 }
 else if (size > 0) { //вправо
```

```
  size %= numbers.size();
  if (size)
   std::rotate(numbers.rbegin(), numbers.rbegin() + size, numbers.rend());
 }
}

int main() {
 QList <int> numbers;
 numbers << 1 << 2 << 3 << 4 << 5;
 shift (numbers,-3); //на 3 элемента влево
 qDebug() << numbers;
 shift (numbers,4); //потом на 4 вправо
 qDebug() << numbers;
 return 0;
}
```

Задача 4.5. Возможна также реализация задач в виде **виджета**, работающего со списком строк QStringList. Класс QStringList удобен, прежде всего, потому, что имеет дополнительные методы для фильтрации данных и легко взаимодействует со стандартными компонентами QT, "понимающими" строки QString. Показанный ниже виджет создан на основе класса QWidget без создания формы (интерфейс спроектирован логически в конструкторе виджета, см. проект Qt6StringListWidget.zip).

Файл widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QtWidgets>
class Widget : public QWidget {
    Q_OBJECT
private:
    QVBoxLayout *layout;
    QTextEdit *textEdit;
    QMenuBar *menuBar;
    QMenu *fileMenu;
    QAction *newAction,*doAction,*exitAction;
private slots:
    void newFile(void);
    void doFile(void);
    void exitFile(void);
public:
    Widget(QApplication *a=0,QWidget *parent = 0);
    ~Widget();
};
#endif // WIDGET_H
```

Файл widget.cpp

```
#include "widget.h"

Widget::Widget(QApplication *a,QWidget *parent) : QWidget(parent) {
 this->menuBar = new QMenuBar(this);
 this->fileMenu = new QMenu(tr("&Файл"));
 this->menuBar->addMenu(this->fileMenu);
 this->newAction = new QAction(tr("&Новый"),this->fileMenu);
 this->doAction = new QAction(tr("&Выполнить"),this->fileMenu);
 this->exitAction = new QAction(tr("В&ыход"),this->fileMenu);
 this->fileMenu->addActions(
  QList <QAction *> () << this->newAction << this->doAction << this->exitAction
 );
 QObject::connect(this->newAction,SIGNAL(triggered()),
  this,SLOT(newFile()));
 QObject::connect(this->doAction,SIGNAL(triggered()),
  this,SLOT(doFile()));
 QObject::connect(this->exitAction,SIGNAL(triggered()),
```

```
  this,SLOT(exitFile()));
 this->textEdit = new QTextEdit(this);
 this->layout = new QVBoxLayout(this);
 this->layout->addWidget(this->textEdit);
 this->layout->setMenuBar(this->menuBar);
 this->setLayout(this->layout);
 this->setWindowTitle("Мой виджет");
 this->setMinimumSize(320,240);
 this->resize(640,480);
 this->setGeometry(QStyle::alignedRect(
  Qt::LeftToRight,Qt::AlignCenter,this->size(),
  a->screens().first()->geometry()));
}

Widget::~Widget() { }

void Widget::newFile(void) { this->textEdit->clear(); }

void Widget::doFile(void) {
 QString String = this->textEdit->toPlainText();
 QStringList list = String.split("\n");
 //обработка QStringList
 QRegularExpression regExp("^(?!\\s*$).+");
 list = list.filter(regExp); //убрали строки только из разделителей
 list.replaceInStrings(QRegularExpression("\\s+")," ");
  //убрали лишние разделители между словами
 list.replaceInStrings(QRegularExpression("^\\s+|\\s+$"),"");
  //убрали лишние разделители в начале или конце строк
 this->textEdit->clear();
 this->textEdit->append(list.join("\n"));
}

void Widget::exitFile(void) { QApplication::quit(); }
```

Файл main.cpp

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[]) {
 QApplication a(argc, argv);
 Widget w(&a);
 w.show();
 return a.exec();
}
```

Задача 4.6. Работа с другими контейнерами может быть организована аналогично, изменится только наполнение метода doFile(). Например, в показанном ниже коде вводимые пользователем строки вида "ключ:значение", где "ключ" – целое число, а "значение" – строка, записываются в мультихэш (ассоциативный массив, в котором одному ключу может соответствовать несколько значений).
Код вставляется перед оператором this->textEdit->clear();

```
 QMultiHash <int,QString> hash; //Ключ - число, значения - строки
 QStringList::iterator it = list.begin();
 int key=0; //Ключ для элементов, которым его не дал пользователь
 for (;it!=list.end();++it) { //Пройти по списку элементов "ключ:значение"
  QStringList item = (*it).split(":",Qt::SkipEmptyParts);
   //разбить элемент по разделителю ":"
  if (item.size()<2) hash.insert(key++,item.at(0));
  else hash.insert(item.at(0).toInt(),item.at(1));
   //Добавили в хэш ключ (наш или заданный пользователем) и значение
  qDebug() << "Element: " << (*it);
 }
 //Вывести в консоль отладки мультихэш
 QMultiHash<int,QString>::iterator i = hash.begin();
```

```
 for (;i!=hash.end();++i)
    qDebug() << "Key=" << i.key() << "Value=" << i.value();
```

После этого кода можно выполнять любые действия с мультихэшем, например, получить список всех значений, соответствующих ключу "0":

```
QList <QString> lst=hash.values(0);
```

Задача 4.7. Пример выгрузки контейнера в файл и загрузки из файла.
Для простоты создано консольное приложение, в котором показано создание и чтение бинарного файла с содержимым двух списков. При конфигурации QT по умолчанию, файл создастся в папке build-… проекта.
Код файла main.cpp (единственный файл проекта Qt6ConoleContainers.zip):

```
#include <QtCore>

int main() {
 //Пишем 2 списка в файл
 QList <QString> stringList;
 QList <int> intList;
 stringList << "one" << "two" << "three";
 intList << 1 << 2 << 3;
 QFile f("data.dat");
 if (f.open(QIODevice::WriteOnly)) {
  QDataStream stream(&f);
  stream << stringList << intList;
  if (stream.status()!=QDataStream::Ok) {
      qDebug() << "File write error"; return 1;
  }
  else {
      qDebug() << "File writed successfully";
  }
 }
 else {
  qDebug() << "File open error for writing"; return 2;
 }
 f.close();
 //Читаем из файла то, что записали
 stringList.clear(); intList.clear();
 if (f.open(QIODevice::ReadOnly)) {
  QDataStream stream(&f);
  stream >> stringList >> intList;
  if (stream.status()!=QDataStream::Ok) {
      qDebug() << "File read error"; return 3;
  }
  else {
      qDebug() << "File read successfully";
      qDebug() << stringList << "\n" << intList;
  }
 }
 else {
  qDebug() << "File open error for reading"; return 4;
 }
 return 0;
}
```

Задание к лабораторной работе: используя контейнеры, итераторы и алгоритмы, написать приложения, реализующие действия, описанные в вашем варианте задания. Разрешается применять как консольное приложение, так и виджет.