Урок 4

1. Qt контейнеры

Одна из самых распространенных задач в программировании ­– это организация обработки групп элементов. В Qt существует собственная библиотека контейнеров. Она находится в модуле QtCore, поэтому для использования только некоторых классов понадобится подключать дополнительные зависимости.

В примерах используется класс QDebug, позволяющий выводить данные в консоль отладки проекта. Для использования класса подключите его с помощью директивы #include.

**QVector**

*Описание*:

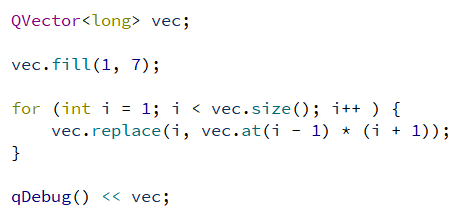
Класс, являющийся моделью динамического массива. К элементам вектора можно обратится как посредством оператора индексации [], так и при помощи итератора.

*Часто используемые методы*:

* push\_back() – append() – вставка в конец;
* push\_front() – prepend() – вставка в начало;
* insert() – вставка в указанную позицию;
* pop\_back() – removeLast() – удаление с конца;
* pop\_front() – removeFirst() – удаление с начала;
* removeAll() – очистка вектора;
* remove() – удаление на указанной позиции;
* replace() – замена значения на указанной позиции;
* size() – получение размера вектора;
* contains() – проверка на наличие элемента в векторе;
* isEmpty() – проверка на пустой вектор;
* fill() – заполнение вектора одним значением.

[*Документация*](https://doc.qt.io/qt-5/qvector.html)

*Пример использования*:



*Вывод*:



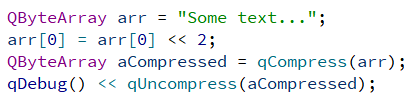
**QByteArray**

*Описание*:

Этот контейнер очень похож на QVector по методам и структуре, но разница заключается в том, что это не шаблонный класс, и в нем допускается хранение только элементов, имеющих размер 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 []. К данным объектов класса QByteArray можно также применить операцию сжатия и обратное преобразование. Это достигается при помощи двух глобальных функций qCompress() и qUncompress().

[*Документация*](https://doc.qt.io/qt-5/qbytearray.html)

*Пример использования*:



*Вывод*:



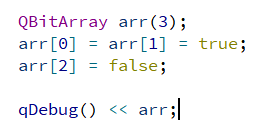
**QBitArray**

*Описание*:

Битовая версия QByteArray. Класс требует подключения с помощью директивы #include.

[*Документация*](https://doc.qt.io/qt-5/qbitarray.html)

*Пример использования*:



*Вывод*:



**QList и QLinkedList**

*Описание*:

Преимущество списков перед векторами и очередями состоит в том, что вставка и удаление элементов в любой позиции происходит эффективнее, так как для выполнения этих операций изменяется только минимальное количество указателей. Исключение составляет только вставка элемента в центр списка. Но есть и недостаток – списки плохо приспособлены для поиска определенного элемента списка по индексу, и для этой цели лучше всего использовать вектор.

Списки реализует шаблонный класс QList. В общем виде данный класс представляет собой массив указателей на элементы. Если не планируется изменять значения элементов, то из соображения эффективности не рекомендуется использовать оператор []. Вместо него следует вызвать метод at(), так как он возвращает константную ссылку на элемент.

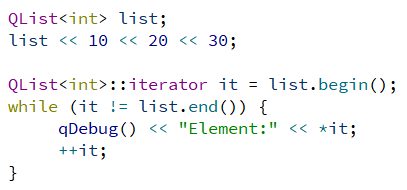
Если для работы программы часто требуется прибегать к операциям вставки и удаления, то эффективнее будет использовать двусвязные списки QLinkedList. Этот контейнер требует больше выделенной памяти, чем QList, но операции вставки и удаления сводятся к переопределению четырех указателей, независимо от позиции удаляемого или вставляемого элемента.

*Часто используемые методы*:

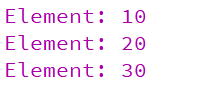
* move() – перемещение элемента с одной позиции на другую;
* removeFirst() – удаление первого элемента списка;
* removeLast() – удаление последнего элемента списка;
* swap() – обмен двумя элементами списка на указанных позициях;
* takeAt() – получение элемента на указанной позиции и удаление его из списка;
* takeFirst() – получение первого элемента и удаление его из списка;
* takeLast() – получение последнего элемента и удаление его из списка;
* toVector() – преобразование в вектор QVector.

[*Документация*](https://doc.qt.io/qt-5/qlist.html)

*Пример использования*:



*Вывод*:



**QStack**

*Описание*:

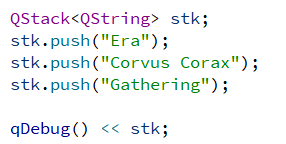
Класс QStack представляет собой реализацию стековой структуры. Этот класс унаследован от QVector. Процесс помещения элементов в стек обычно называется проталкиванием (pushing), а извлечение из него верхнего элемента – выталкиванием (poping). Каждая операция проталкивания увеличивает размер стека на 1, а каждая операция выталкивания – уменьшает на 1. Класс требует подключения с помощью директивы #include.

*Часто используемые методы*:

* push() – проталкивание элемента;
* pop() – выталкивание элемента;
* top() – получение ссылки на верхний элемент.
* empty() – проверка на пустой стек.

[*Документация*](https://doc.qt.io/qt-5/qstack.html)

*Пример использования*:



*Вывод*:



**QQueue**

*Описание*:

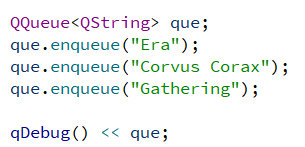
Очередь реализует структуру данных, работающую по принципу – первым пришел, первым ушел. Реализована очередь в классе QQueue, который унаследован от QList. Класс требует подключения с помощью директивы #include.

*Часто используемые методы*:

* enqueue() – проталкивание элемента;
* dequeue() – выталкивание элемента;
* empty() – проверка на пустую очередь.

[*Документация*](https://doc.qt.io/qt-5/qqueue.html)

*Пример использования*:



*Вывод*:



**QMap и QMultiMap**

*Описание*:

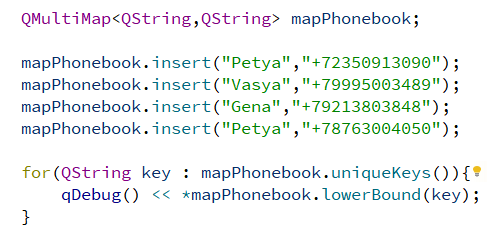
Отображения хранят элементы одного и того же типа, индексируемые ключевыми значениями. Их достоинство в возможности быстро получать значение, ассоциированное с заданным ключом. Ключи должны быть уникальными, за исключением мультиотображения, которое допускает дубликаты. В случае со словарем QMap необходимо следить за тем, чтобы не было занесено двух разных элементов с одинаковым ключом, ведь тогда не удастся извлечь один из этих элементов.

*Часто используемые методы*:

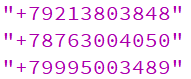
* lowerBound() – получение первого элемента с заданным ключом;
* upperBound() – получение последнего элемента с заданным ключом;
* insert() – добавление элемента в отображение;
* contains() – проверка зарегистрированного ключа или пары ключ-значение;
* keys() / uniqueKeys() – получение всех или только уникальных ключей;
* size() – получение размера словаря.

[*Документация*](https://doc.qt.io/qt-5/qmap.html)

*Пример использования*:



*Вывод*:



**QSet**

*Описание*:

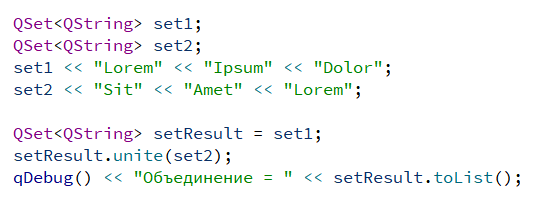
Контейнер записывает элементы в некотором порядке и предоставляет возможность очень быстрого просмотра значений и выполнения с ними операций, характерных для множеств, таких как объединение, пересечение и разность. Необходимое условие – ключи должны быть разными. Контейнер QSet можно использовать в качестве неупорядоченного списка для быстрого поиска данных.

*Часто используемые методы*:

* unite() – выполнение операции объединения;
* intersect() – выполнение операции пересечения;
* subtract() – выполнение операции разности;
* size() – получение размера контейнера.

[*Документация*](https://doc.qt.io/qt-5/qset.html)

*Пример использования*:



*Вывод*:



**QHash**

*Описание*:

Функциональность хэшей очень похожа на QMap с той лишь разницей, что вместо сортировки по ключу этот класс использует хэш-таблицу. Это позволяет ему осуществлять поиск ключевых значений гораздо быстрее, чем это делает QMap.

Следует обязательно 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 containts() контейнера.

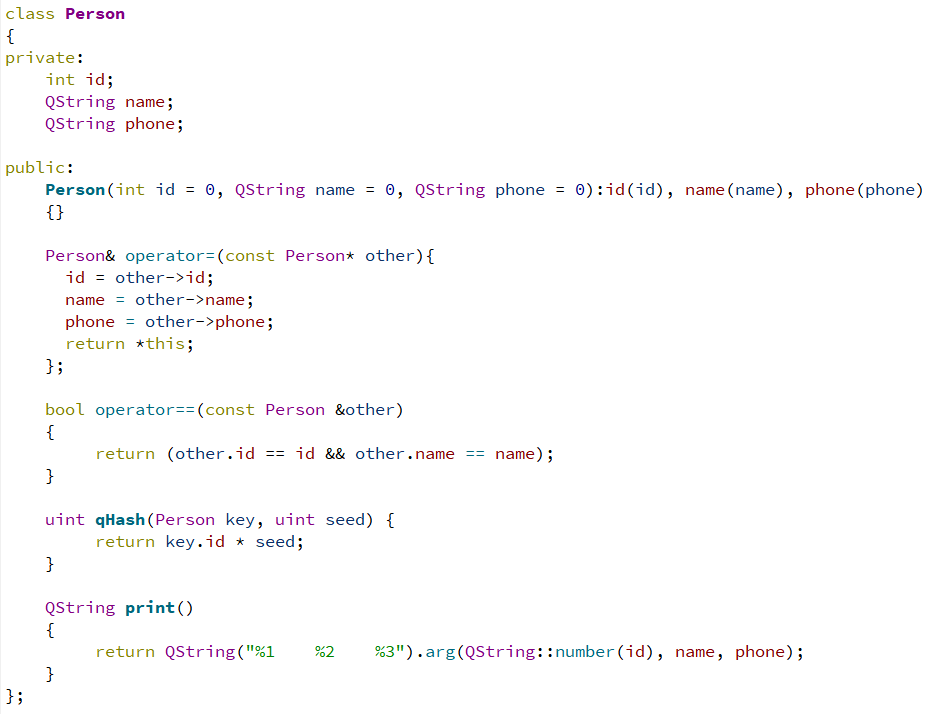
Если требуется разместить в QHash объекты собственных классов, то необходимо реализовать оператор сравнения ==, оператор присваивания = и специализированную функцию qHash() для класса.

*Часто используемые методы*:

Методы унаследованы от QMap.

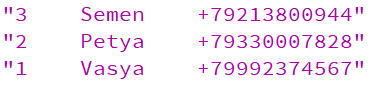
[*Документация*](https://doc.qt.io/qt-5/qhash.html)

*Пример использования*:





*Вывод*:



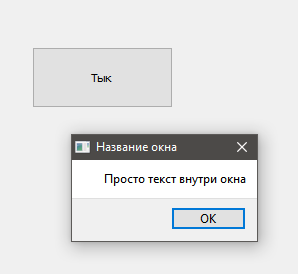
1. Модальные окна

В Qt существует возможность создавать не только однооконные приложения. Самой простой версией отдельного окна является QMessageBox – диалоговое окно. Его можно использовать для информирования или для получения подтверждения действия пользователя.

Для демонстрации работы диалогового окна достаточно одной кнопки и слота on\_clicked(), настроенного для этой кнопки:



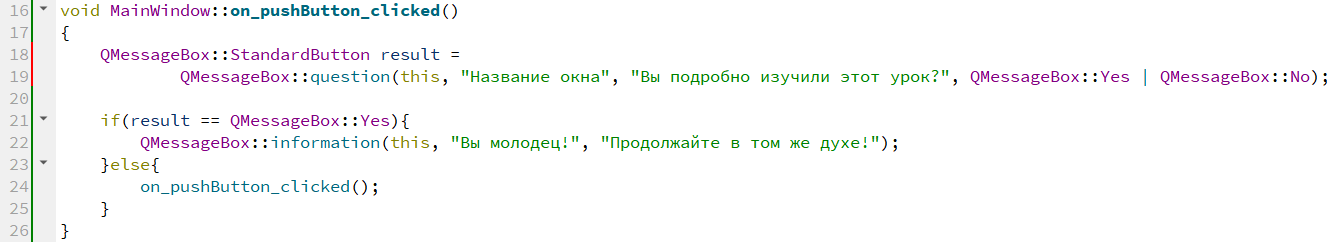
Метод about() класса QMessageBox принимает как аргументы родительский класс, заголовок и содержание окна. Результат работы будет выглядеть следующим образом:



Существует еще несколько типов диалоговых окон:

* critical() – сообщение о критической ошибке;
* information() – информационное сообщение;
* warning() – предупреждающее сообщение;
* question() – сообщение с вариантами ответов.

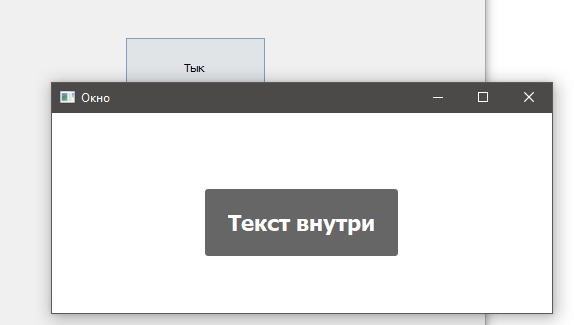
Создание сообщения с вопросом и обработка ответа может выглядеть так:



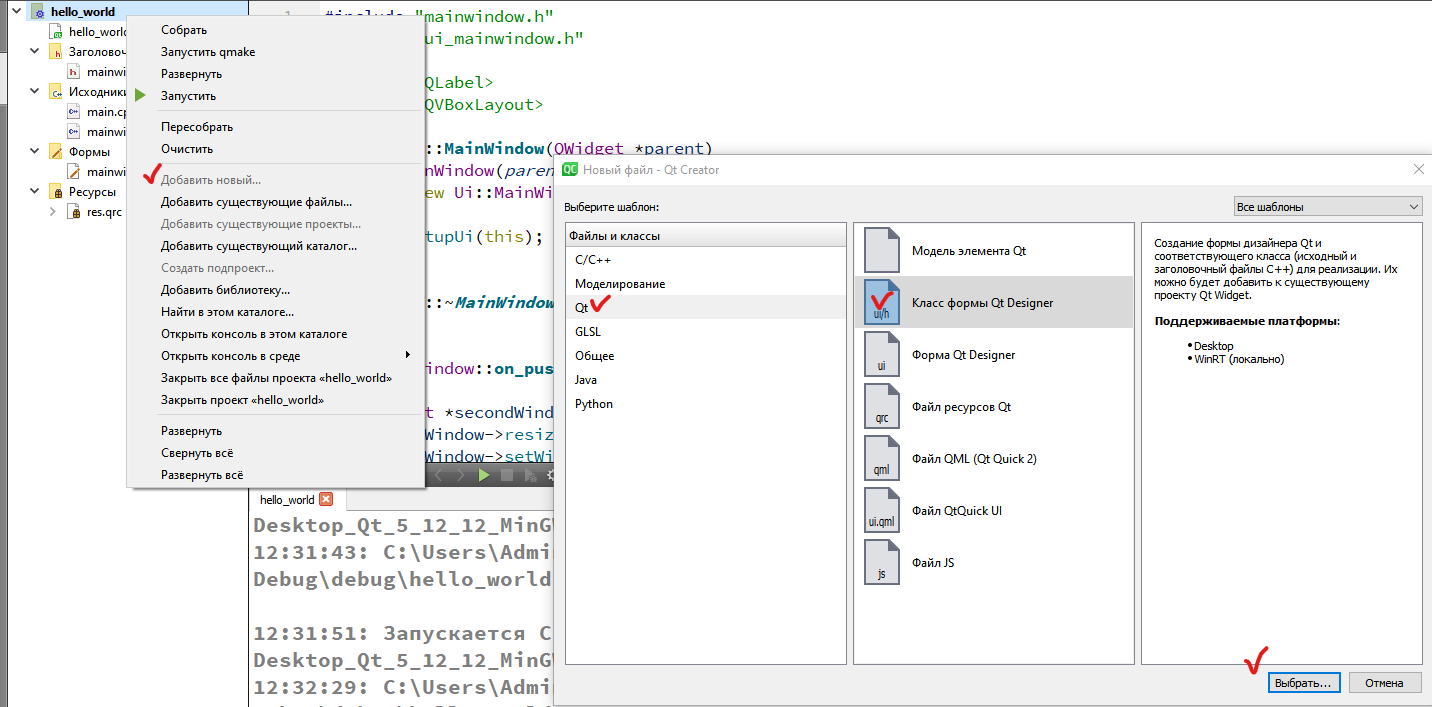
Следует обратить внимание на 18 строку. Результат ответа в себе содержит тип QMessageBox::StandardButton, от которого наследуются все кнопки, использующиеся как четвертый аргумент в методе question(). Одинарный знак модуля является разделителем, через который перечисляются кнопки.

Собственное модальное окно можно создать из экземпляра класса QWidget или его производного класса. Например, это можно сделать следующий образом:

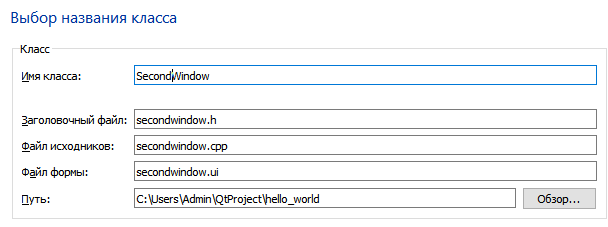




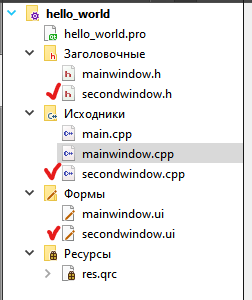
Не модальное, а заранее подготовленное окно приложения можно создать с помощью редактора:



После выбора элемента “Класс формы Qt Designer” на выбор будет представлено несколько шаблонов. Выбор не повлияет на изучение создания окон с помощью редактора. В примере был выбран шаблон “Main Window” со следующими характеристиками:



После настройки класса в древе проекта появятся новые файлы .h , .cpp и .ui:



Для вызова нового окна осталось подключить файл secondwindow.h воспользоваться экземпляром созданного класса:



Стоит отметить, что преимущество данного метода в том, что существует возможность дополнить класс SecondWindow новыми и переопределенными методами, слотами, сигналами и полями.