Классы контейнеров

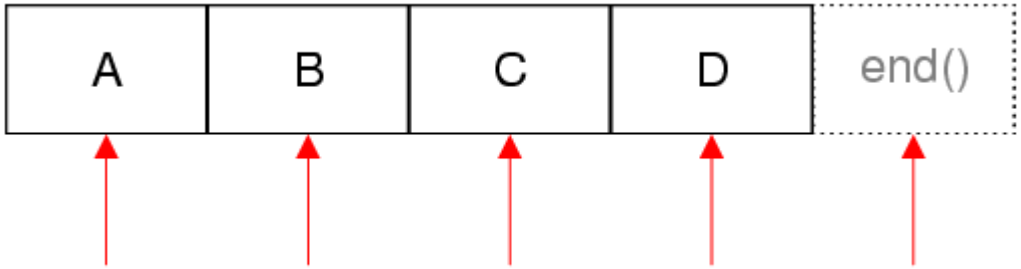
Классы контейнеров Qt

- **Массивы:**
  - **`QList<T>`** – список значений, доступных по индексу
    - **`QQueue<T>`** – расширение QList - для использования как очередь
  - **`QLinkedList<T>`** – список значений, доступных последовательно
  - **`QVector<T>`** – массив значений, последовательно расположенных в памяти
    - **`QStack<T>`** – расширение QVector - для использования как стек

- **Множества и словари:**
  - **`QSet<T>`** – множество элементов (как в математике)
  - **`QMap<Key,T>`** – словарь, хранящий отображение ключ-значение, ключи упорядочены
  - **`QMultiMap<Key,T>`** – словарь, можно хранить несколько значений для одного ключа
  - **`QHash<Key,T>`** – более быстрая версия QMap, ключи не упорядочены.
  - **`QMultiHash<Key,T>`**

Контейнеры – способы доступа

- По индексу
- Итераторы STL-стиля



- foreach()

Доступ по индексу

```
QList<QString> list;
list << "A" << "B" << "C" << "D";

int i;
QTextStream cout(stdout);
for(i=0; i<list.count(); i++)
    cout << list[i] << " ";
cout << endl;
```

Итераторы STL-стиля

- Проход по массиву в прямом порядке:

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i;

for (i = list.begin(); i != list.end(); ++i)
    *i = (*i).toLower();
```

- Проход по массиву в обратном порядке:

```
i = list.end();
while (i != list.begin())
{
   --i;
   *i = (*i).toLower();
}
```

foreach()

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QString str;
QTextStream cout(stdout);

foreach(str, list)
    cout << str;
```

- foreach() делает **копию** списка, который он обходит, поэтому с его помощью **нельзя модифицировать** элементы исходного списка.

QList

QList<QString> list;

- **Добавление**

```
list << "one" << "two" << "three";
list.append(“four”);
list.prepend(“zero”);
```

- **Поиск**

```
int i=list.indexOf(“six”);
if(i!=-1)
    cout<<tr(“первое вхождение ‘six’ в позиции”) << i;
```

- **Модификация**
  - `removeAll()`
  - `removeAt(int i)`
  - `swap(int i,int j)`
  - `move(int from,int to)`
  - `reserve(int alloc)`

QMap

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

- **Добавление элементов:**

```
map["one"] = 1;
map.insert("twelve", 12);
```

- **Извлечение элементов:**

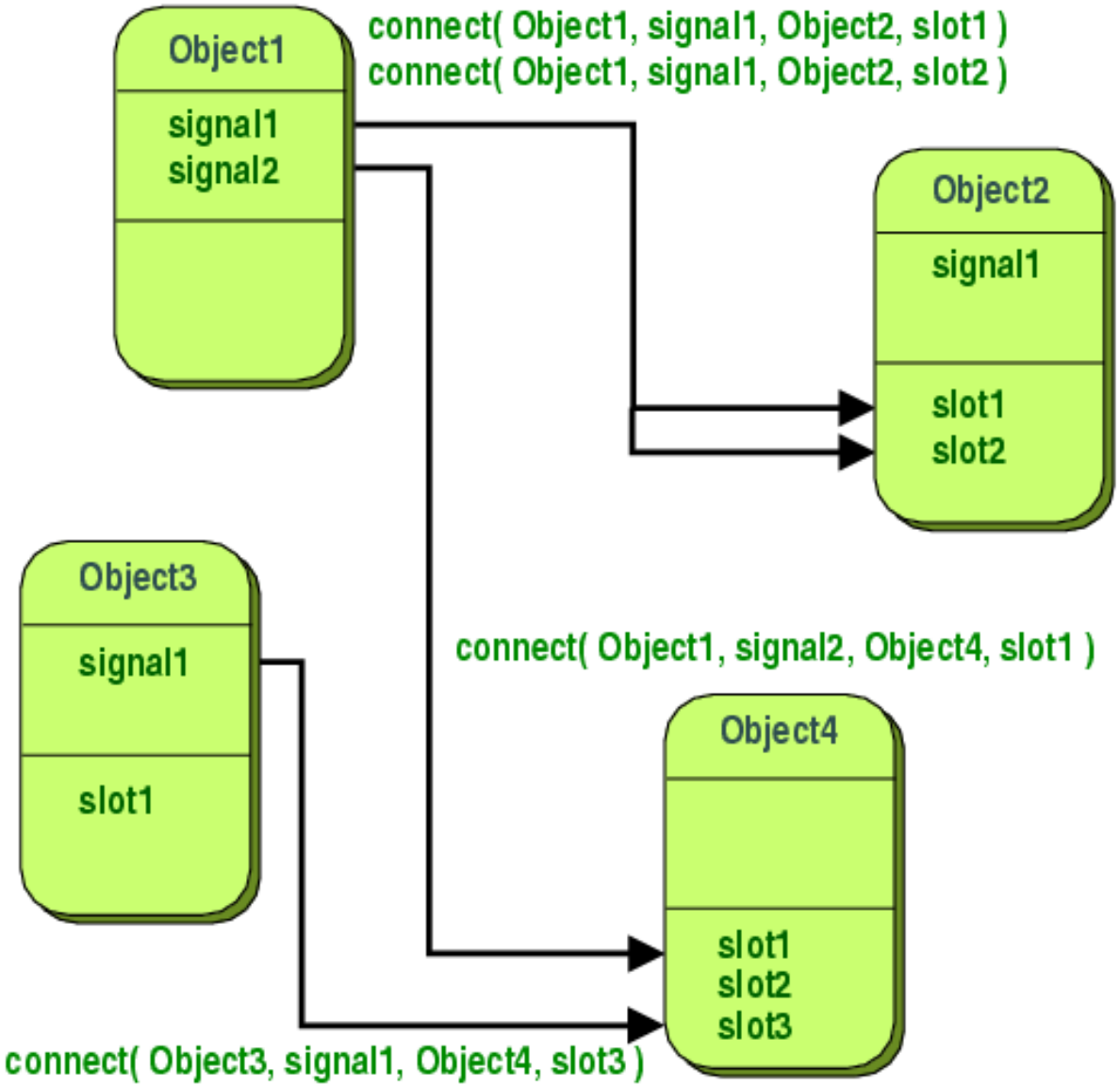
```
int num1 = map["thirteen"];
if(map.contains(“ten”))
     num1=map.value(“ten”);
num2=map.value(“three”,3)
```

Сигналы и слоты

Сигналы и слоты

- Механизм сигналов и слотов служит для того, чтобы одни объекты могли уведомлять других о событиях.
- **Сигнал (SIGNAL)** генерируется, когда происходит определённое ***событие***.
- **Слот (SLOT) -** это функция, которая вызывается в ответ на определённый ***сигнал***.
- Сигналы и слоты слабо связаны:
  - Объект, генерирующий сигнал, не знает, получает ли кто-либо его сигнал.
  - Слот - это обычная функция, она не знает, кто её вызвал, и был ли это сигнал.
  - Можно соединить несколько слотов с одним сигналом и несколько сигналов с одним слотом.

Сигналы и слоты



Соединение сигнала и слота

- Если хочется, чтобы кнопка `cancel` закрывала наш диалог:

```
MyDialog::MyDialog(QWidget *parent)
:QDialog(parent)
{  connect(ui->cancel,SIGNAL(clicked()),SLOT(reject()));
```

или

```
   connect(ui->cancel, &QPushButton::clicked,
                                    this, &MyDialog::reject);}
```

- Для того, чтобы при изменении положения слайдера выбранное значение автоматически отображалось в числовом поле:

```
    connect(ui->slider,SIGNAL(sliderMoved(int)),
                          ui->spinBox,SLOT(setValue(int)));
```

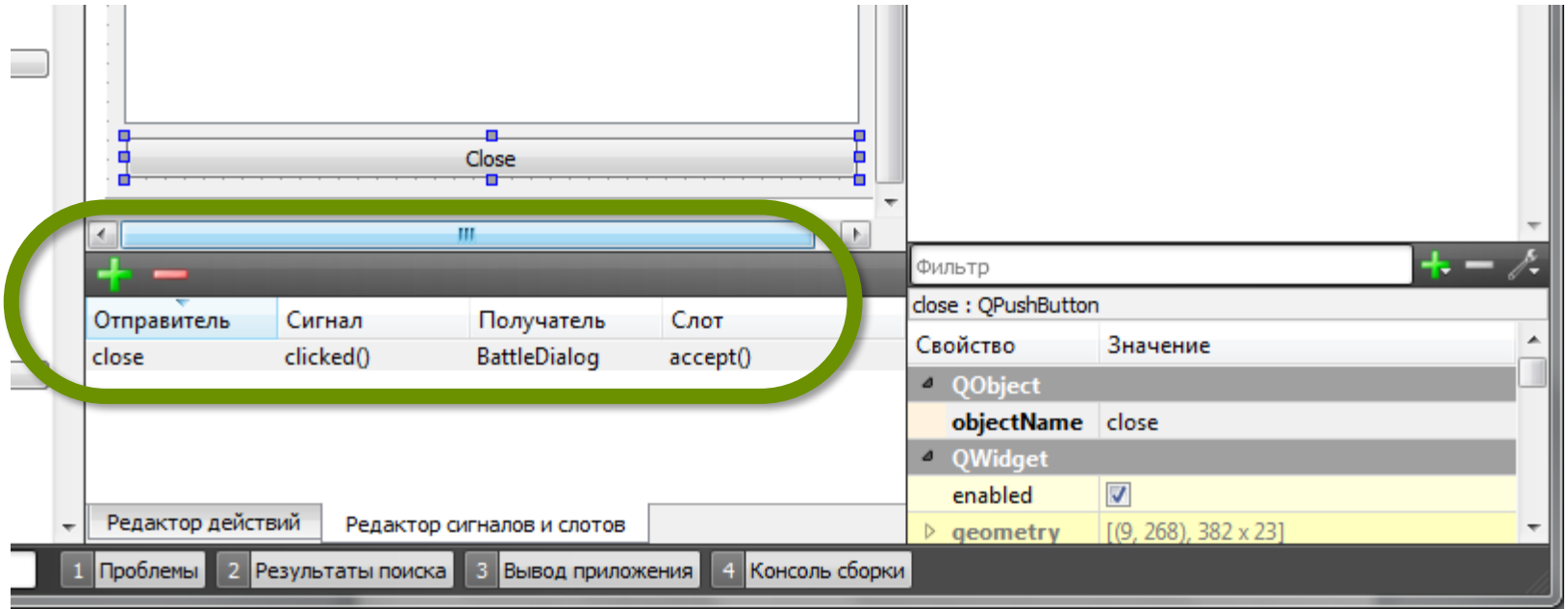
или

```
    connect(ui->slider,&QSlider::sliderMoved,
                          ui->spinBox,&QSpinBox::setValue);
```

Соединение сигнала и слота без программирования

- Слоты, имеющие имя вида **on_объект_сигнал**, автоматически соединяются с сигналом указанного объекта.

- Сигналы и слоты объектов интерфейса можно соединять в **Редакторе сигналов и слотов** в дизайнере.



Как сделать свой слот

1. Ваш объект должен быть унаследован от **QObject** и в начале его объявления должно быть написано Q_OBJECT:

```
class Counter : public QObject
{
    Q_OBJECT
```

2. Объявите функцию-слот:

```
public slots:
    void setValue(int value);
```

3. Реализуйте функцию-слот в вашем .cpp файле как обычный метод:

```
void Counter::setValue(int value)
{
   m_value = value;
}
```

Объекты, унаследованные от QObject, **нельзя** копировать и передавать по значению!

Как сделать свой сигнал

1. Ваш объект должен быть унаследован от QObject, и в начале его объявления должно быть написано Q_OBJECT:

```
class Counter : public QObject
{
    Q_OBJECT
```

2. Объявите сигнал:

```
signals:
    void valueChanged(int value);
```

3. Реализовывать функцию-сигнал не нужно, её реализация автоматически создаётся мета-компилятором Qt moc.

4. Для активации сигнала вызовите его с помощью **emit**:

```
void Counter::setValue(int value)
{
  if (value != m_value) { m_value = value; emit valueChanged(value); }
}
```

Диалоговые окна

Диалоговые окна

Используются для:

- Запроса дополнительной информации у пользователя
- Отображения дополнительной информации/ввода опций во время работы в основном окне.

Типы:

- **Модальные** диалоговые окна
- **Немодальные** диалоговые окна

Могут иметь родителя, но отображаются как отдельное окно верхнего уровня.

Обеспечивается специальная обработка для клавиш Enter и Esc.

Модальные диалоговые окна

- Создаётся для вывода/запроса информации **без возможности** пользователя работать с основным окном приложения.
- Могут быть модальными для всего приложения (по умолчанию) или для одного окна.
- Пользователь должен закончить работу с модальным диалогом прежде, чем он сможет работать с другими окнами приложения.
- При создании становится активным.
- Обычно при работе с диалогом используется код возврата, возвращаемый диалогом после завершения работы.

Немодальные диалоговые окна

- Создаётся для одновременной работы с другими окнами.
- Может становится не активным в процессе работы.
- **Не блокирует** работу пользователя с другими окнами приложения.

Создание диалогового окна

1. Выберите пункт контекстного меню проекта «Добавить новый..».

2. Выберите шаблон «Qt/Qt Designer Form Class» .

3. Выберите название класса и имена файлов. Пусть это будут класс MyDialog и файлы MyDialog.h, MyDialog.cpp, MyDialog.ui.

4. Разместите на форме диалога виджеты, задайте для них разумные имена.

5. При необходимости реализуйте слоты для обработки нажатий на кнопки, etc

Завершение работы диалога

Модальный диалог

Файл MyDialog.cpp:

```
void
MyDialog::on_..._clicked()
{
        …
  accept(); //вернёт Accepted
```

Или

```
  reject(); //вернёт Rejected
```

Или

```
  int code;
  done(code);
}
```

Немодальный диалог

Файл MyDialog.cpp:

```
void
MyDialog::on_..._clicked()
{
        …
  close();
```

/* Если у виджета диалога установлен флаг **Qt::WA_DeleteOnClose**, то объект будет удалён после закрытия окна.

*/

}

Использование диалогового окна

Модальное диалоговое окно

Файл кода главного окна:

```
#include “MyDialog.h”

void
MainWindow::on_..._clicked()
{
  MyDialog dlg(this);

  if(dlg.exec()==Accepted)
//диалог работает только
// «внутри» exec()
  {
   // обработка выбора
       пользователя
  }
}
```

Немодальное диалоговое окно

Файл кода главного окна:

```
#include “MyDialog.h”
void MainWindow::on_..._clicked()
{
 MyDialog *dlg=new MyDialog(this);
 dlg->setAttribute
     (Qt::WA_DeleteOnClose,true);
 dlg->show();  //сделать видимым
 dlg->raise(); //перенести наверх
 dlg->activateWindow();//сделать
                       // активным
//теперь диалог живёт своей
//жизнью,отдельно от главного окна
}
```