C++ Kurs Teil 4

- Templates
  - Funktions Templates
  - Member Templates
  - Klassen Templates
  - Funktoren
- Qt
  - Layout Managment
  - Qt Widgets und Dialogboxen
  - QPainter und Bildklassen
  - qmake

Warum Templates?

```
// Wie könnten die drei Funktionen implementiert werden,
// ohne drei mal den gleichen Code einzugeben?

int min_int( int a, int b)
{
  if( a < b) return a;
  return b;
}

float min_float( float a, float b)
{
  if( a < b) return a;
  return b;
}

std::string min_string( std::string a, std::string b)
{
  if( a < b) return a;
  return b;
}
```

Warum Templates?
Standard-Lösung mit Vererbung

- Erzeugen einer Basis-Klasse MyObject mit virtueller Funktion isGreaterThan()
- Erzeugen von Klassen MyInt, MyString, etc. die alle von MyObject erben und isGreaterThan() implementieren.
- Dann kann min() für diese Klassen folgendermaßen implementiert werden:

```
MyObject* min( MyObject* a, MyObject* b)
{
  if( a->isGreaterThan(*b) ) return a;
  return b;
}
```

- Das ganze funktioniert nicht für primitive Datentypen (int, float, etc.) oder Klassen, die man nicht selbst geschrieben hat (std::string, etc.) und der Aufruf über eine virtuelle Funktion ist teuer!

Funktions-Template

```
#include <string>

template<typename DataType>
DataType min( DataType a, DataType b)
{
  if( a < b) return a;
  return b;
}

int main( int argc, char** argv)
{
  int a = 5;
  int b = 10;
  int minab = min<int>( a, b);

  std::string c = "bebraham";
  std::string d = "abraham";
  std::string mincd = min( c, d);

  return 0;
}
```

DataType ist der „Platzhalter“ für den Datentyp

'int' ist ein **compile-time-Parameter**, während 'a' und 'b' normale **run-time-Parameter** sind

compile-time Parameter (hier <std::string>) können auch weggelassen werden, wenn der Compiler die Information aus den normalen Parametern herausfinden kann

Funktions-Template

```
#include <string>
using namespace std;

template<typename DataType>
DataType min( DataType a, DataType b)
{
  if( a < b) return a;
  return b;
}

int main( int argc, char** argv)
{
  int a = 5;
  int b = 10;
  int minab = min<int>( a, b);

  string c = "bebraham";
  string d = "abraham";
  string mincd = min( c, d);

  return 0;
}
```

c ist ein string

Aus der „Vorlage“ wird folgender Code erzeugt und compiliert:

```
int min<int>( int a, int b)
{
  if( a < b) return a;
  return b;
}
```

```
string min<string>( string a, string b)
{
  if( a < b) return a;
  return b;
}
```

Funktions-Template (mehrere template Parameter)

```
#include <vector>

template<typename T, typename Iter>
void sumAll( Iter begin, Iter end, T& sum)
{
   sum = T();  // sum auf 0, bzw. “”
   for( Iter p=begin; p != end; ++p)
   {
      sum += *p;
   }
}

int main( int argc, char** argv)
{
   std::vector<int> vals(3);
   vals[0] = 3;
   vals[1] = 2;
   vals[2] = 42;
   double sum;
   sumAll( vals.begin(), vals.end(), sum);
   return 0;
}
```

Aus der „Vorlage“ wird folgender Code erzeugt und compiliert:

vals.begin() ist ein std::vector<int>::const_iterator

```
void sumAll<double, std::vector<int>::const_iterator>(
        std::vector<int>::const_iterator begin,
        std::vector<int>::const_iterator end,
        double& sum)
{
   sum = double();  // sum auf 0, bzw. “”
   for( std::vector<int>::const_iterator p=begin;
        p != end; ++p)
   {
      sum += *p;
   }
}
```

sum ist ein double

typedef in Template Klassen

```
typedef unsigned char uchar;

template<typename T>
class MyVector
{
 public:
   typedef T           value_type;
   typedef T*          iterator;
   typedef const T* const_iterator;
   //...

private:
   T* _data;
};

int main( int argc, char** argv)
{
   //...
   MyVector<int>::const_iterator p;
   //...
}
```

- typedef wird normalerweise genutzt, um irgendwelchen Typen einen kürzeren Namen zuzuweisen
- Bei Template Klassen (wie z.B aus der STL) kann man es nutzen, um weitere Informationen über den Container zur Verfügung zu stellen

Member Template Beispiel
Ein Stream, der in Container einliest

Wir erben von std::ifstream

InputDataStream.hh

```
#include <fstream>

class InputDataStream : public std::ifstream
{
public:
  InputDataStream( const char* fileName);

  template<typename Container>
  void readArray( Container& array);
};

#include "InputDataStream.icc"
```

Member Template

Initializer für die Parent-Klasse

InputDataStream.cc

```
InputDataStream::InputDataStream(const char*
                                 fileName)
        :  std::ifstream( fileName)
{
   // nothing more to do
}
```

InputDataStream.icc

```
template<typename Container>
void InputDataStream::readArray( Container&
array)
{
  typename Container::value_type  value;
  while( *this >> value )
  {
    array.push_back( value);
  }
}
```

Anwendung von InputDataStream

```
#include <vector>
#include <string>
#include <iostream>
#include "InputDataStream.hh"

int main( int argc, char** argv)
{
  std::vector<std::string> words;
  InputDataStream ids( "GPL_V2");

  ids.readArray( words);

  // Ausgabe aller Wörter
  for( sizt_t i = 0; i < words.size(); ++i)
  {
    std::cout << i+1 << ". Wort: " << words[i]
              << std::endl;
  }
  return 0;
}
```

Öffnen der Datei GPL_V2

Auch hier findet der Compiler selbst den compile-time-Parameter heraus. Wie müßte der ausführliche Aufruf aussehen?

Klassen Template

```
template<typename DataType, int MaxSize>
class MyFixedVector
{
public:
  typedef const DataType* const_iterator;

  DataType& operator[](unsigned int index)
        {
          return _data[index];
        }

  const_iterator begin() const
        {
          return _data;
        }

  const_iterator end() const
        {
          return _data + MaxSize;
        }

private:
  DataType _data[MaxSize];
};
```

Als compile-time-Parameter können nicht nur Typ-Namen sondern auch Werte übergeben werden

Überladen des [ ] Operators

inline-Methoden können wie hier auch direkt in der Klassen-Deklaration definiert werden.

Anwendung von MyFixedVector

```
#include <iostream>
#include "MyFixedVector.hh"

int main( int argc, char** argv)
{
  MyFixedVector<float,5>  prims;

  prims[0] = 1;
  prims[1] = 3;
  prims[2] = 5;
  prims[3] = 7;
  prims[4] = 11;

  // Ausgabe der Primzahlen
  for( MyFixedVector<float,5>::const_iterator
       p = prims.begin(); p != prims.end(); ++p)
  {
    std::cout << *p << std::endl;
  }
}
```

Erzeugter Code:

```
class MyFixedVector<float, 5>
{
public:
  typedef const float* const_iterator;

  float& operator[](unsigned int index)
        {
          return _data[index];
        }

  const_iterator begin() const
        {
          return _data;
        }

  const_iterator end() const
        {
          return _data + 5;
        }

private:
  float _data[5];
};
```

Funktoren
(Funktionen, die als Parameter übergeben werden)

```
template<typename Iter, typename Comp>
void sort( Iter begin, Iter last, Comp lessThanFunctor)
{
  ...
  if( lessThanFunctor( a, b))  // is a < b ?
  ...
}
```

```
  // Sortieren nach Gehalt
  sort( people.begin(), people.end(), compareSalary);
```

Die Vergleichsfunktion (Funktor) wird extra definiert:

```
inline
bool compareSalary( const Employee &a, const Employee &b)
{
  return (a.salary() < b.salary());
}
```

Spezialisierung

- Wenn man für eine bestimmte Kombination von template-Parametern speziell optimierten Code schreiben möchte, kann man das per „specialization“ erreichen

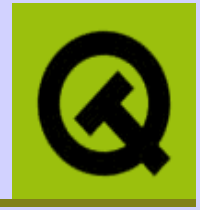
```
// allgemeine Version
template<typename T, int Size>
void MyVector::multWith( double a)
{
   for( int i = 0; i < Size; ++i)
   {
      _data[i] *= a;
   }
}

// spezialisierte (optimierte) Version für 3D float Vektoren
template<>
void MyVector<float,3>::multWith ( double a)
{
   _data[0] *= a;
   _data[1] *= a;
   _data[2] *= a;
}
```

Spezialisierung

```
int main( int argc, char** argv)
{
  MyVector<int,5> a;
  a.multWith( 42);    // standard-Version wird genutzt

  MyVector<float,3> b;
  b.multWith( 137);    // optimierte Version wird genutzt

  return 0;
}
```

- Vorteil: Die optimierten Versionen für bestimmte Kombintaionen können später hinzugefügt werden, ohne daß der aufrufende Code geändert werden muß

Was ist Qt ?



- Qt ist eine Bibliothek zur Entwicklung von **G**raphical **U**ser **I**nterfaces

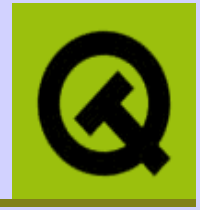
Entwicklung von Trolltech (http://www.trolltech.com/)

Qt unterstützt verschiedene Platformen:

- MS/Windows: 95, 98, NT/2000, XP
- Unix/X11: Linux, Solaris, HP-UX, Digital Unix, AIX, IRIX
- Macintosh: Mac OS X

- Verschiedene Lizenzen:
  - Qt Enterprise Edition und Qt Professional Edition für komerzielle Softwareentwicklung
  - Qt Free Edition: Unix/X11 und MacOS X Version, ausschließlich für Open Source und freie Software, GNU GPL, kostenlos (ab Qt 4.0 auch wieder für Windows)
- Qt ist Basis für KDE
- Aktuelle Version: 3.3.4 (ca. Ende Juni 2005 soll 4.0 erscheinen)

Warum Qt ?

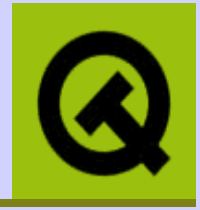


Vollständig objekt-orientiertes Design
Platformunabhängig
Alle Standard GUI Kontrollen + Standarddialoge + OpenGL-Grafik
Innovatives System zur interobjekt-Kommunikation
Thread Unterstützung

... und Seit Qt 3.0

- Modul für Zugriff auf SQL Datenbanken
- Plugin Architektur; z.B. Styles, Datenbanktreiber, Bildformate nachladbar
- Docking Windows/Areas
- Erzeugung und Kontrolle von Kindprozessen
- Verbessert: Druckunterstützung, reguläre Ausdrücke, Unicode, XML

Dokumentation/Literatur



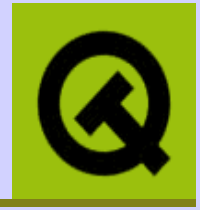
- ON-LINE REFERENCE DOCUMENTATION
  - **API** Referenz: alphabetical class index, Class Inheritance Hierarchy, Member Function Index, API Structure Overview
  - Tutorial und Walkthroughs; ab 3.0: Whitepaper (HTML und PDF)
  - http://doc.trolltech.com/

- Bücher
  - KDE- und Qt-Programmierung (GUI-Entwicklung für Linux)

    Burkhard Lehner; Addison-Wesley
  - Programming with Qt (writing portable GUI applications on UNIX and WIN32)

    Matthias Kalle Dalheimer; O'Reilly

Das erste Programm



```
#include "Hello.h"
#include <qapplication.h>

int main (int argc, char* argv[])
{
  QApplication qapp(argc, argv);

  Hello* h = new Hello;
  h->setCaption("Qt says Hello");

  qapp.setMainWidget( h);
  h->show();
  return qapp.exec();
}
```

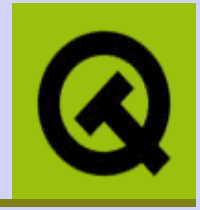
QApplication enthält die *main event loop* und handhabt Initialisierung, Finalisierung sowie *session management;* darf nur genau einmal in einer Qt Anwendung stehen.

h ist das *main widget* dieser Anwendung: d.h. wenn h verschwindet, wird das Programm beendet.
Achtung: QWidgets immer mit „new“ erzeugen

show() zeigt das widget und seine Kinder.

main übergibt Kontrolle an Qt's event loop;
diese läuft bis main widget terminiert oder bis **exit()** aufgerufen wird. **exec()** liefert dessen return code zurück.

Ein leeres Widget



Hello.h

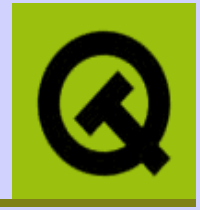
```
#ifndef HELLO_H
#define HELLO_H

#include <qwidget.h>

class Hello : public QWidget
{
   public:
      Hello ();
};

#endif
```

Hello.cpp

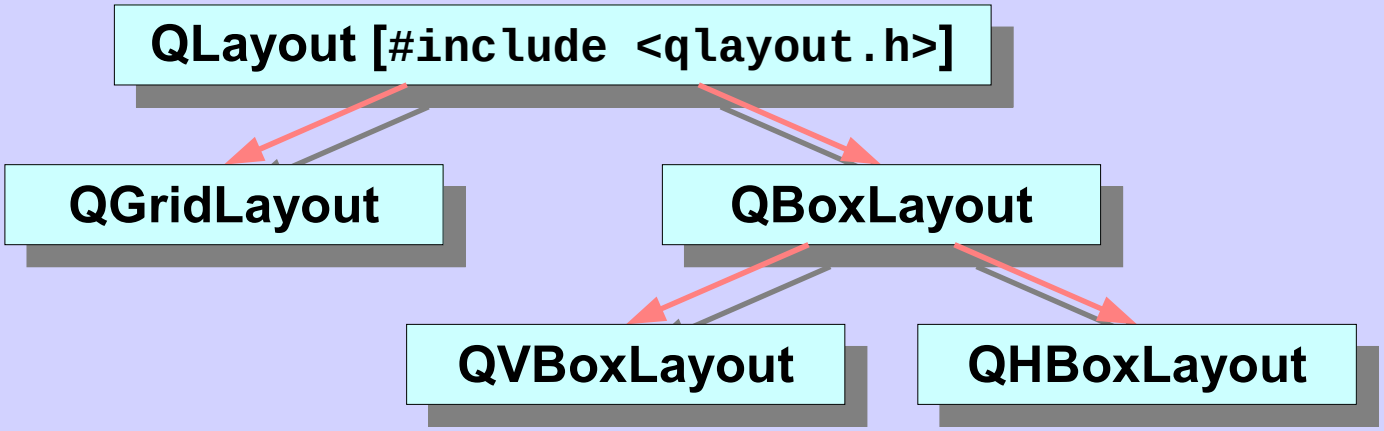
```
#include "Hello.h"

Hello::Hello ()
     : QWidget()
{
    resize(200, 100);
}
```

Widget ist atomare Einheit des UI; empfängt *events* vom Windowsystem und stellt seine graphische Repräsentation dar (Grundfläche stets rechteckig). Widgets werden hierarchisch (parent/child Beziehung) zu komplexeren UIs kombiniert.

Geometry Management

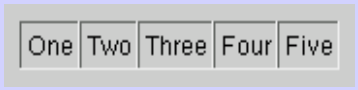


- Layout Klasse verteilt Kinder in Reihenfolge der Einfügung auf bereitgestellten Raum
- für komplexere Arrangements können Layouts ineinander geschachtelt werden
- proportionale Anpassung bei resize; automatisches Update bei Veränderung von Inhalten (Fontgröße, ausblenden von Subwidgets)
- **Achtung:** Widgets denen ein parent $\neq 0$ zugewiesen wird, werden von diesem in Besitz genommen und dürfen nur vom parent-Destruktor freigegeben werden.



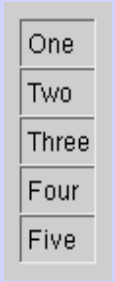
Geometry Management

- **QHBoxLayout**

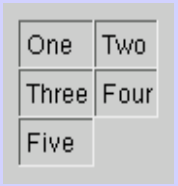


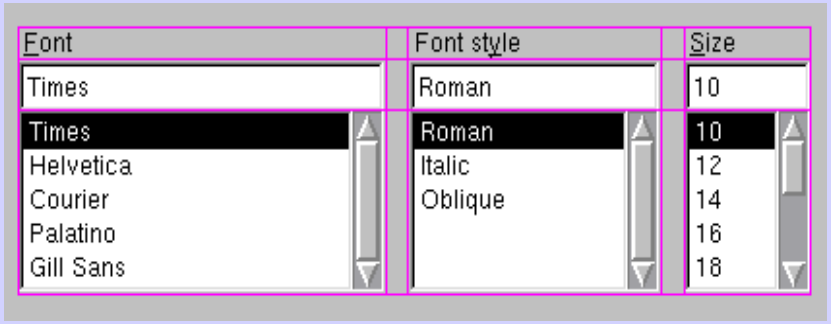
QHBoxLayout(QLayout* parent, ...) : erzeugt horizontale Box und fügt sie ins parent-Layout ein
QHBoxLayout(QWidget* parent, ...) : erzeugt horizontale top-level Box
QHBoxLayout::addWidget(QWidget* widget, ...) : fügt widget am Ende der Box ein

QVBoxLayout

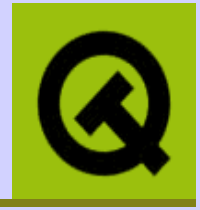


QGridLayout

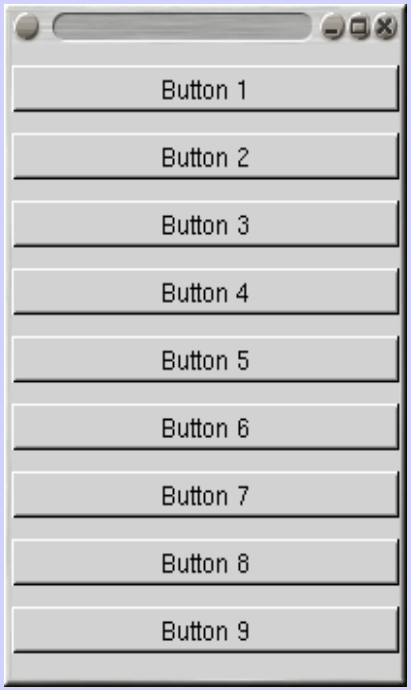




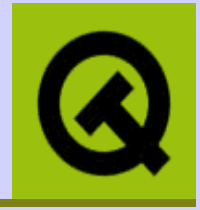
Geometry Management / Beispiel



```
#include <qapplication.h>
#include <qframe.h>
#include <qlayout.h>
#include <qpushbutton.h>

int main (int argc, char* argv[])
{
    QApplication qapp(argc, argv);

    QFrame frame;
    QBoxLayout* l = new QVBoxLayout(&frame);

    for (char c='1'; c<='9'; ++c)
    {
        QPushButton* b = new QPushButton(&frame);
        b->setText(QString("Button ")+QString(QChar(c)));
        l->addWidget(b);
    }

    qapp.setMainWidget(&frame);
    frame.resize(200, 300);
    frame.show();
    return qapp.exec();
}
```



Einige Qt-Widgets



QLabel：kann Text oder Bild anzeigen; keine Benutzerinteraktion.

```
QLabel (QWidget * parent, const char * name=0, WFlags f=0)
QLabel (const QString & text, QWidget * parent, const char * name=0, WFlags f=0 )
QLabel (QWidget * buddy, const QString &, QWidget * parent, const char * name=0, WFlags f=0)
```

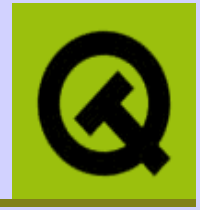
QPushButton : löst vordefinierte Aktionen aus (z.B. "Ok", "Cancel", "Exit")

```
QPushButton (QWidget * parent, const char * name=0)
QPushButton (const QString & text, QWidget * parent, const char * name=0)
QPushButton (const QIconSet & icon, const QString & text, QWidget * parent, const char* name=0)
```

QSlider : kontrolliert int-Wert (frei def. Wertebereich)

```
QSlider (QWidget * parent, const char * name=0)
QSlider (Orientation orientation, QWidget * parent, const char * name=0)
QSlider (int minValue, int maxValue, int pageStep, int value, Orientation, QWidget * parent,
                        const char * name=0)
```

Einige Qt-Widgets



QLineEdit : Eingabe/Editieren einer einzelnen Textzeile

```
QLineEdit ( QWidget * parent, const char * name=0 )
QLineEdit ( const QString contents &, QWidget * parent, const char * name=0 )
```

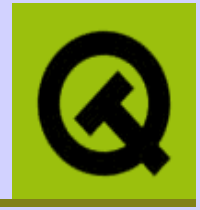
QMainWindow : Typisches Applikationsfenster mit Menubar, Toolbars und Statusbar

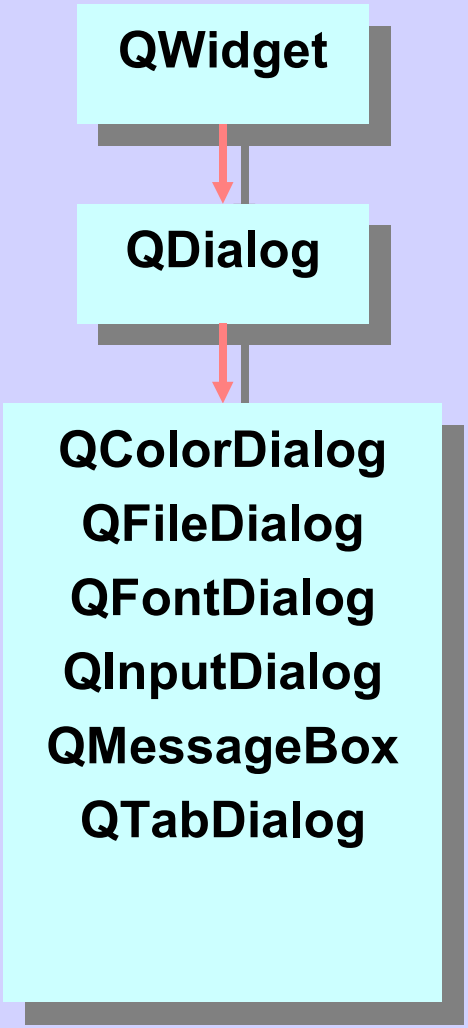
```
QMainWindow ( QWidget * parent = 0, const char * name = 0, WFlags f = WType_TopLevel )
```

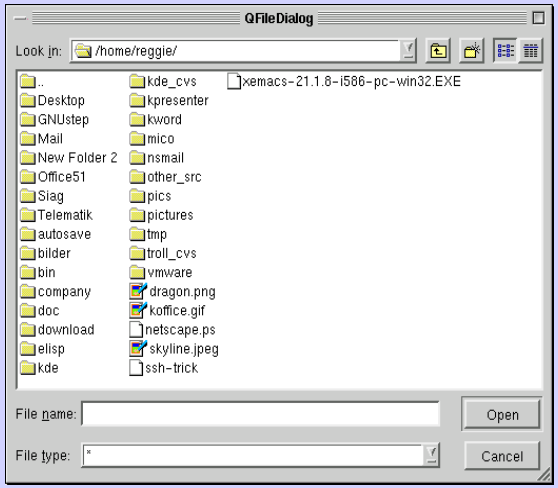
QDialog : Toplevel Fenster für kurze Aufgaben und Kommunikation mit dem Benutzer. Davon abgeleitete Dialoge sind z.B. QFileDialog, QColorDialog und QMessageBox.

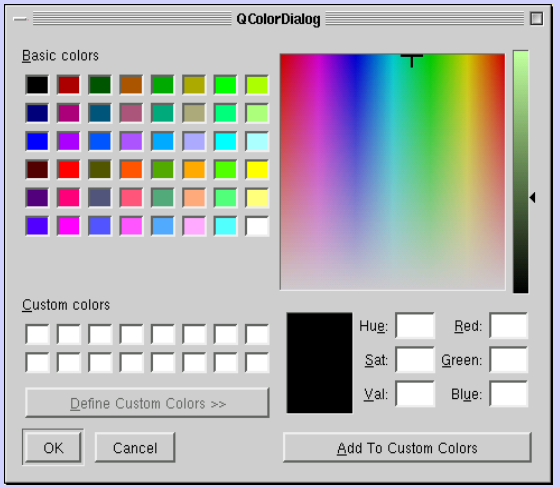
```
QDialog ( QWidget * parent=0, const char * name=0, bool modal=FALSE, WFlags f=0 )
```

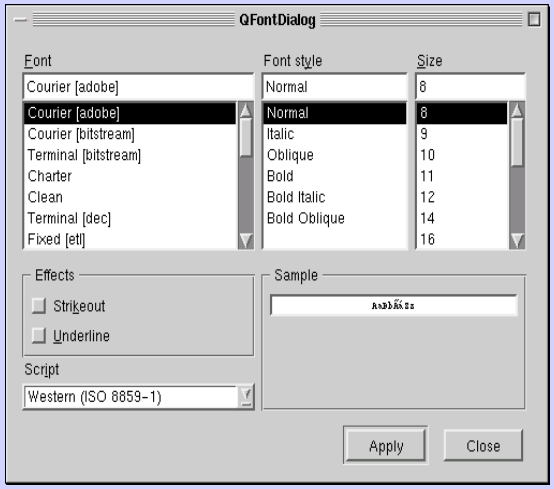
Dialogboxen



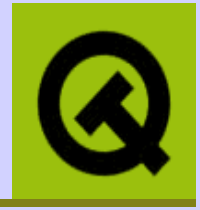








QMessageBox



convenience Methoden (static) für typische Anwendungen:
Information, Warnung, Critical Message, "About", z.B.

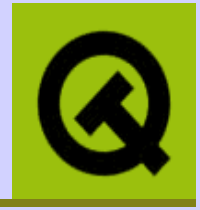
```
int QMessageBox::information (
     QWidget* parent, const QString& caption, const QString& text,
     int button0, int button1=0, int button2=0
);
```

- Öffnet ein Fenster mit Titel, Text und bis zu drei Buttons.
- Gedrückter Button ist Rückgabewert

SpinSlider.cpp:


```
void SpinSlider::info ()
{
  QString text;
  text.setNum(hs->value());
  QMessageBox::information(this, "Info", text);
}
```

QPainter



QPainter bietet Methoden, um elementare 2D-Grafikelemente darzustellen

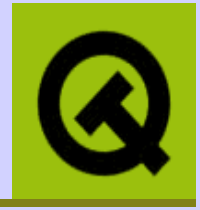
(Punkte, Linien, Rechtecke, Ellipsen (Bögen/Segmente), Polygone, Bezierkurven, Text und Bilder; inkl. clipping und affiner Transformationen)

Arbeitet auf **QPaintDevice**, also insb. auch auf **QWidget**

Typische Anwendung innerhalb von paintEvent:

```
void xyzWidget::paintEvent()
{
    QPainter paint(this);
    paint.setPen(Qt::blue);
    paint.drawText(rect(), AlignCenter, "The Text");
}
```

Qt Bildklassen



QPixmap

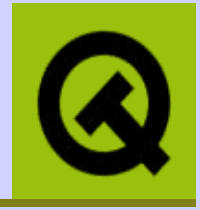
- Implementiert ein off-screen pixel-basiertes paint device
- Optimiert für schnelle Darstellung
- Pixeldaten nicht direkt zugänglich;
- Manipulation nur mittels **QPainter**

QImage

- Hardwareunabhängige Repräsentation von Bilddaten (1,8 und 32 BPP)
- Optimiert für schnelle I/O und Zugriff
- Pixelwerte können direkt ausgelesen und geschrieben werden: pixel(), setPixel(), bits(), scanline()

QPixmap und **QImage** können ineinander konvertiert werden ➞ langsam
Laden von Bildern möglich; unterstützte Formate: PNG, BMP, XBM, XPM, PNM; Optional (JPEG, MNG und GIF)

Qt's C++ Erweiterungen

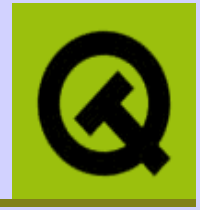


- Qt verwendet „*Meta Object System“;* ermöglicht diverse Erweiterungen, unabhängig von Platform und C++ Compiler

  - **Signal / Slot** - Mechanismus
  - RTTI über Funktionen wie **`inherits()`** oder **`className()`**
  - Dynamische *object properties*
  - Internationalization

- Erweiterungen benötigen separate Übersetzung ➞ **moc** (Meta Object Compiler)
- moc liest C++ Quellcode und produziert bei Bedarf (z.B. **Q_OBJECT**) neue C++ Datei, die Meta Code enthält
- Dieser Code muß übersetzt und gelinkt werden

Qt's qmake



- **qmake:** Werkzeug von Trolltech um platformunabhängige Makefiles zu erstellen

- benötigt evtl. Q**MAKESPEC** Shell-Variable:
  z.B. export QMAKESPEC=$QTDIR/mkspecs/linux-g++ in Bourne shell

- Erzeugen eines .pro -Files einfach mit 'qmake -project'
- Einfaches Beispiel: SpinSlider.pro

```
TEMPLATE = app
CONFIG   = qt warn_on debug
HEADERS  = SpinSlider.h
SOURCES  = main.cpp SpinSlider.cpp
TARGET   = SpinSlider
```

- Aufruf: qmake SpinSlider.pro
- Anschließend: make

Qt GUI Designer

- **Designer** erlaubt interaktives Erstellen von Design und Implementation
- Positionierung der Widgets mit Maus
- Signal/Slot Verbindung mittels D&D
- Editieren aller Widget Properties
- Ausgabe: "human-readable" XML
- User Interface Compiler (ui) erzeugt C++ Code