technische universität
dortmund

Einführung in die Programmierung

Wintersemester 2012/13

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

Inhalt

- Was ist eine GUI? Was ist QT?
- Erste Schritte: „Hello World!“
- Signals & Slots: SpinBoxSlider
- Anwendung: Temperaturumrechnung
  - Lösung ohne GUI (Ein- und Ausgabe an Konsole)
  - Lösung mit GUI
- Größere Anwendung: Grafik (→ nächste Vorlesung)

GUI = **G**raphical **U**ser **I**nterface (grafische Benutzerschnittstelle)

Funktionalität wird durch Programm-Bibliothek bereit gestellt

- z.B. als Teil der MFC (Microsoft Foundation Classes)
- z.B. X-Window System, Version 11 (X11)

hier: Qt 4.7.3 („**Q**uasar **t**oolkit“) → `ftp://ftp.qt-project.org/qt/source/qt-win-opensource-4.7.3-vs2008.exe`

aktuell: Qt 5.0.0 (abwärtskompatibel?) → `http://www.qt-project.org`

Warum?

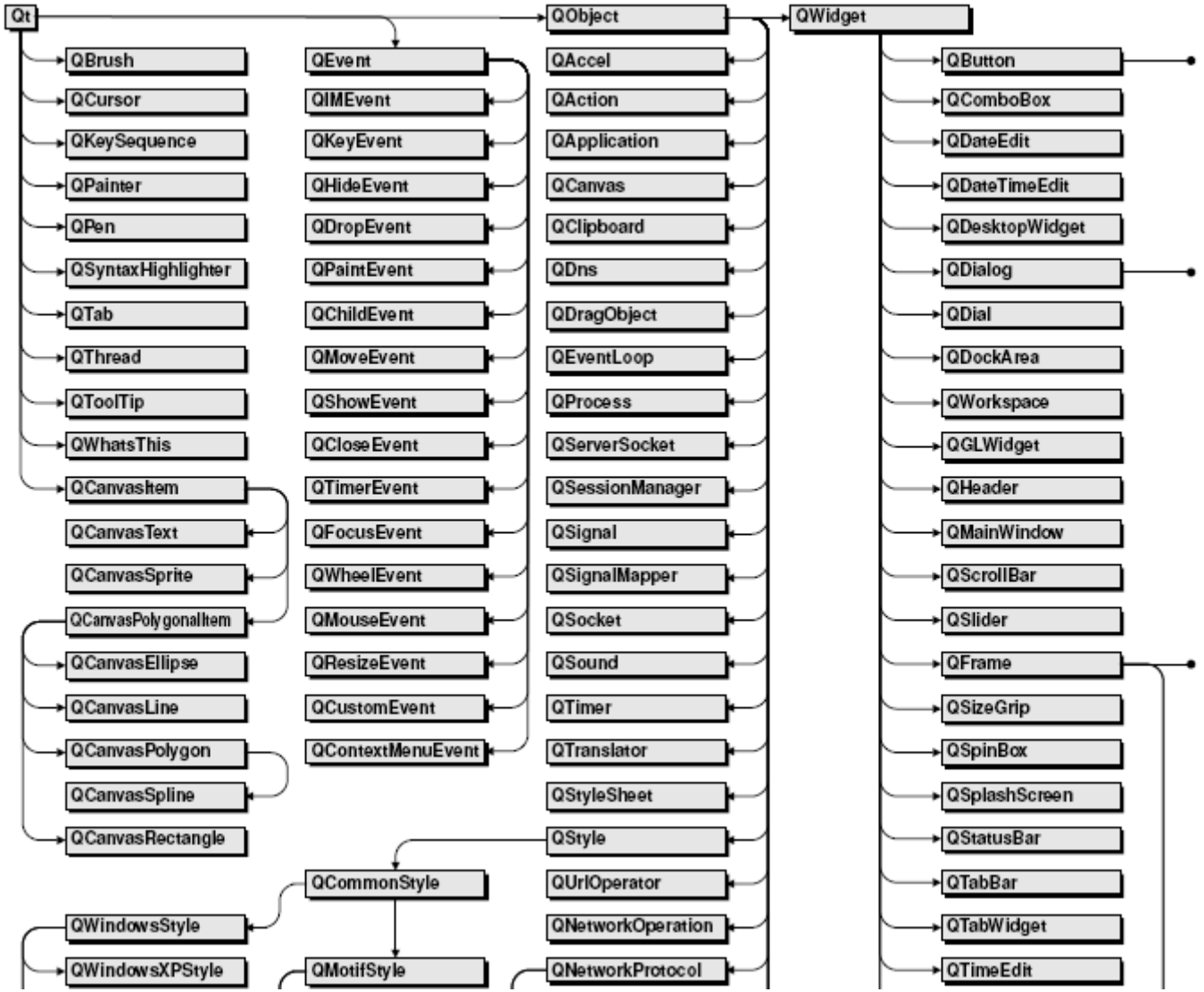
1. Plattform-unabhängig: läuft unter Linux/Unix, Windows, MacOS, u.a.
2. Für nicht-kommerziellen Einsatz frei verfügbar (unter GPL), allerdings ohne Support u.a. Annehmlichkeiten

Qt

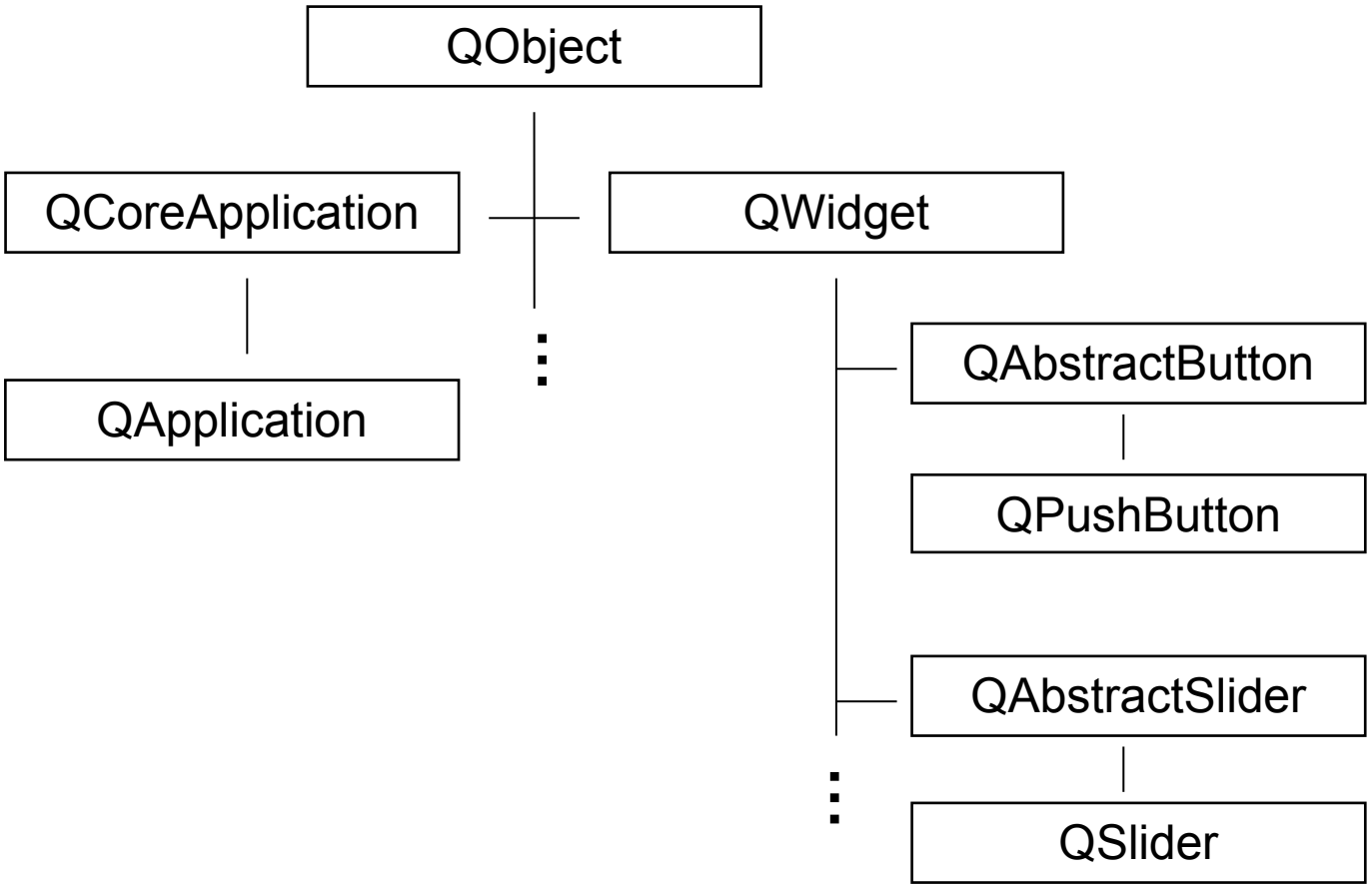
- System übergreifende Bibliothek
- stellt Objekte und Funktionen zur Verfügung, mit denen unabhängig vom Betriebssystem (Linux/Unix, Windows, MacOS) Programme erstellt werden können
- Hauptverwendungszweck: Graphische Benutzeroberflächen (GUIs) für unterschiedliche Betriebssysteme erstellen, ohne den Code für jedes System neu zu schreiben
- Oberfläche KDE (Linux/Mac), Google Earth, Skype basiert auf Qt

Qt Klassen

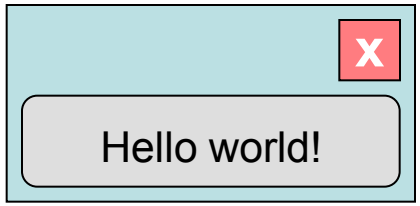
ca. 500



Qt Klassen (Ausschnitt)



Button („Schaltfläche“) mit Text „Hello World!“




```
#include <QApplication.h>
#include <QPushButton.h>

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

  QApplication app(argc, argv);

  QPushButton hello("Hello world!", 0);

  hello.resize(100, 30);

  hello.show();

  return app.exec();
}
```

Jedes Programm hat genau eine Instanz von `QApplication`

Erzeuge Button, 0=kein Elternfenster

Größe in Pixel

Button darstellen!

Kontrolle an `QApplication` übergeben

Button („Schaltfläche“) mit Text „Hello World!“

Was geschieht wenn Button gedrückt wird? → Anscheinend nichts!

Tatsächlich: Klasse **`QPushButton`** bemerkt die Aktion,
wurde aber nicht instruiert, was sie dann machen soll!

Möglich: Eine Aktion in einem Objekt einer anderen Klasse auslösen.

Klasse QObject

```
static bool connect(
  const QObject *sender,     // Wer sendet?
  const char *signal,        // Bei welcher Aktion?
  const QObject *receiver,   // Wer empfängt?
  const char *member,        // Welche Aktion ausführen?
  Qt::ConnectionType type = Qt::AutoCompatConnection
);
```

Button („Schaltfläche“) mit Text „Hello World!“, Programmende sobald gedrückt

```
#include <QApplication.h>
#include <QPushButton.h>

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

  QApplication app(argc, argv);

  QPushButton hello("Hello world!");
  QObject::connect(&hello, SIGNAL(clicked()),
                   &app,   SLOT(quit())
  );
  hello.resize(100, 30);
  hello.show();

  return app.exec();
}
```

Wenn **hello** angeklickt wird, dann soll in **app** die Methode **quit** ausgeführt werden.

Signals and Slots

Qt-spezifisch!

- Bereitstellung von Inter-Objekt Kommunikation
- Idee: Objekte, die nichts voneinander wissen, können miteinander verbunden werden
- Jede von QObject abgeleitete Klasse kann Signals deklarieren, die von Funktionen der Klasse ausgestoßen werden
- Jede von QObject abgeleitete Klasse kann Slots definieren. Slots sind Funktionen, die mit Signals assoziiert werden können.
- Technisch Umsetzung: Makro Q_OBJECT in Klassendeklaration
- Signals und Slots von Objektinstanzen können miteinander verbunden werden:

  Signal S von Objekt A verbunden mit Slot T von Objekt B →
  Wenn A Signal S ausstößt, so wird Slot T von B ausgeführt.

Signals and Slots

Qt-spezifisch!

- Ein **Signal** kann mit mehreren **Slots** verbunden werden.

  → Ein Ereignis löst mehrere Aktionen aus.

- Ein **Slot** kann mit mehreren **Signals** verbunden werden.

  → Verschiedene Ereignisse können gleiche Aktion auslösen.

- **Signals** können auch Parameter an die **Slots** übergeben.

  → Parametrisierte Aktionen.

- **Signals** können mit **Signals** verbunden werden.

  → Weitergabe / Übersetzung von Signalen.

Button als Teil eines Fensters

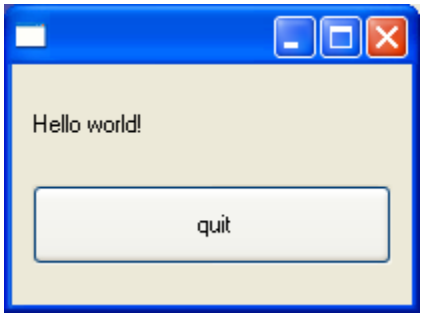
```
#include <QApplication.h>
#include <QPushButton.h>
#include <QWidget.h>

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

  QWidget window;
  window.resize(200, 120);
  QPushButton hello("Hello world!", &window);
  QObject::connect(&hello, SIGNAL(clicked()),
                   &app, SLOT(quit()));
  hello.setGeometry(10, 40, 180, 40);

  window.show();

  return app.exec();
}
```

hello ist Teil von **window**

Button und **Label** als Teile eines Fensters

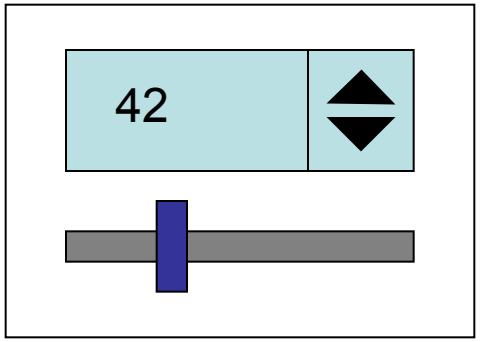
```
#include <QApplication.h>
#include <QPushButton.h>
#include <QLabel.h>
#include <QWidget.h>
int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  QWidget window;
  window.resize(200, 120);
  QLabel hello("Hello world!", &window);
  QPushButton quit("quit", &window);
  QObject::connect(&quit, SIGNAL(clicked()),
                   &app, SLOT(quit())
  );
  hello.setGeometry(10, 10, 180, 40);
  quit.setGeometry(10, 60, 180, 40);
  window.show();
  return app.exec();
}
```

`QLabel` zum Beschriften des Fensterinneren



Slider verbunden mit **SpinBox**

```
#include <QApplication.h>
#include <QSlider.h>
#include <QSpinBox.h>
#include <QWidget.h>

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  QWidget window;
  window.resize(200, 120);

  QSpinBox spinBox(&window);
  spinBox.setGeometry(10, 10, 180, 40);
  spinBox.setRange(0, 130);

  QSlider slider(Qt::Horizontal, &window);
  slider.setGeometry(10, 60, 180, 40);
  slider.setRange(0, 130);
```



Gewünschtes Verhalten:

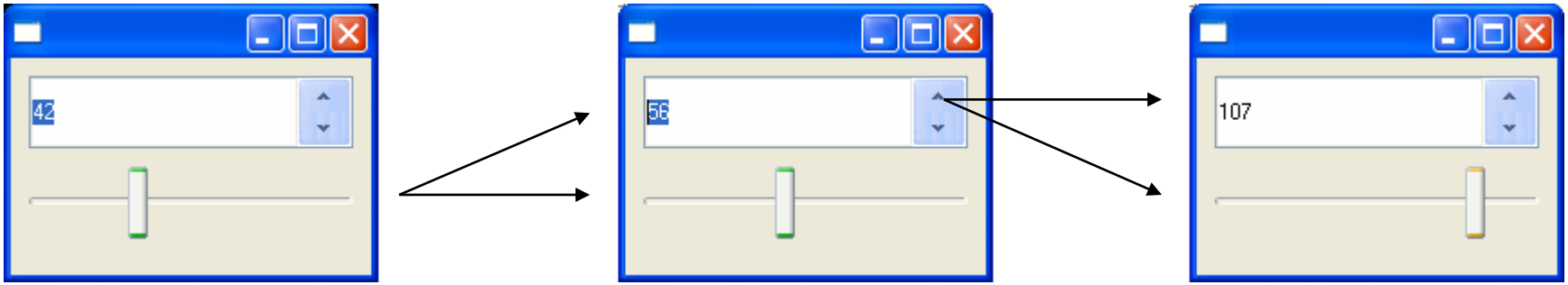
SpinBox wirkt auf Slider und umgekehrt.

Fortsetzung nächste Folie …

Slider verbunden mit **SpinBox**

Fortsetzung

```
  QObject::connect(&spinBox, SIGNAL(valueChanged(int)),
                   &slider, SLOT(setValue(int)));

  QObject::connect(&slider, SIGNAL(valueChanged(int)),
                   &spinBox, SLOT(setValue(int)));

  spinBox.setValue(42);

  window.show();
  return app.exec();
}
```



Anwendung: Temperaturumrechnung

$$x\ [°\text{C}] = \frac{9}{5}x + 32 \quad [°\text{F}]$$

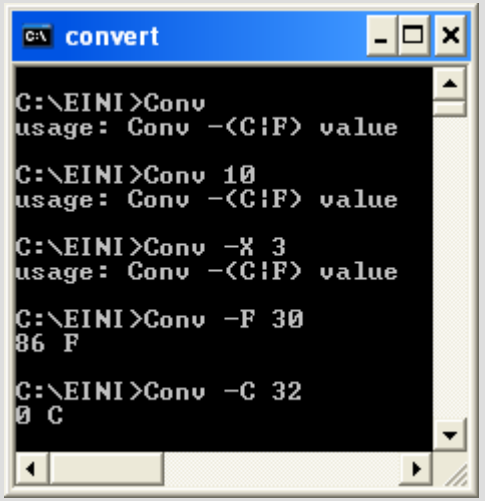
$$y\ [°\text{F}] = \frac{5}{9}(x - 32)\ [°\text{C}]$$

Lösung ohne GUI:

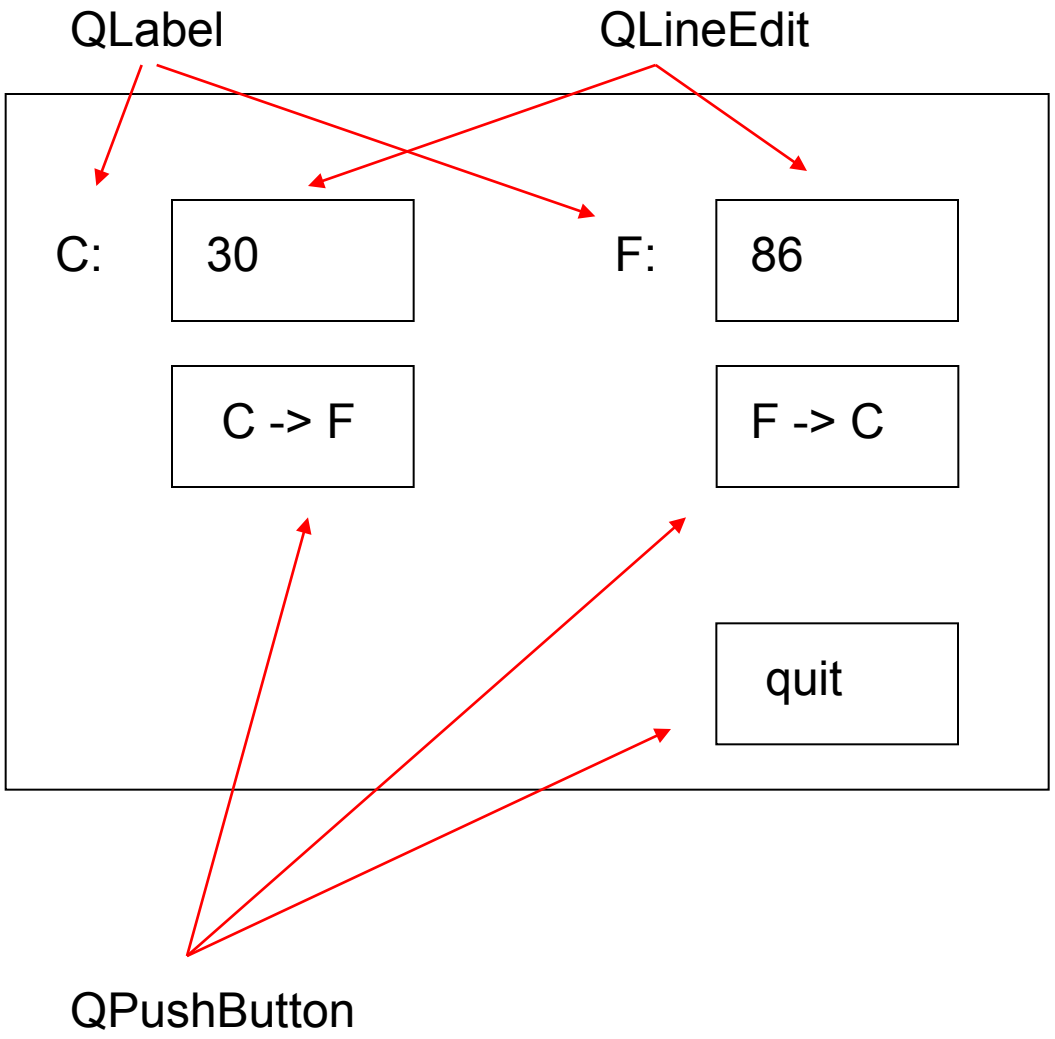
1. Einlesen einer Zahl
2. Angabe der Konvertierungrichtung
3. Ausgabe

Lösung ohne GUI

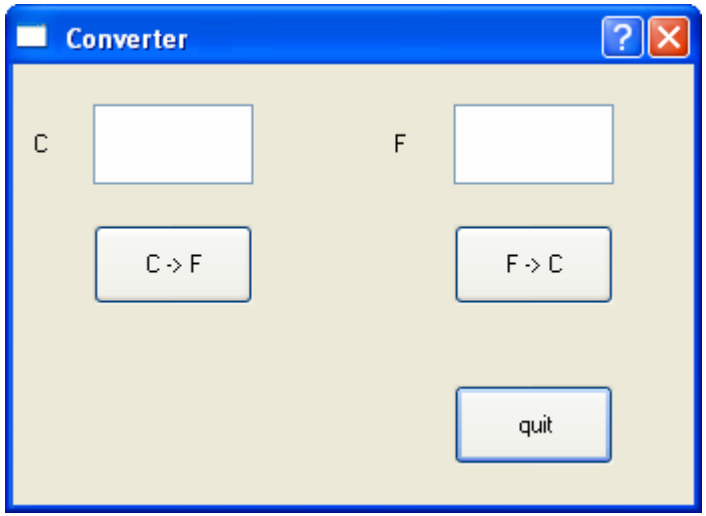
```
#include <iostream>
#include <cstring>

using namespace std;

int main(int argc, char *argv[]) {
  if (argc != 3 || strlen(argv[1]) != 2 || argv[1][0] != '-,
  || (argv[1][1] != 'C' && argv[1][1] != 'F')) {
    cerr << "usage: " << argv[0] << " -(C|F) value\n";
    exit(1);
  }
  double val = atof(argv[2]);
  if (argv[1][1] == 'C')
    val = 5 * (val - 32) / 9;
  else
    val = 9 * val / 5 + 32;
  cout << val << " " << argv[1][1] << endl;
  return 0;
}
```



Lösung mit GUI



```
#include "Converter.h"

int main(int argc, char *argv[]) {
  QApplication app(argc, argv);
  Converter conv(&app);
  conv.show();
  return app.exec();
}
```



So wird die GUI aussehen!

```
#include <QApplication.h>
#include <QObject.h>
#include <QDialog.h>
#include <QPushButton.h>
#include <QLineEdit.h>
#include <QLabel.h>

class Converter : public QDialog {
Q_OBJECT
private:
        QApplication *theApp;
        QPushButton *quit, *f2c, *c2f;
        QLineEdit *editC, *editF;
        QLabel *labelC, *labelF;
public:
        Converter(QApplication *app);
        ~Converter();
public slots:
        void slotF2C();
        void slotC2F();
};
```

Was ist das?

Erst Aufruf von **moc** (*meta object compiler*), der generiert zusätzlichen C++ Code, dann Aufruf des C++ Compilers!

Spracherweiterung?

```
#include <QMessageBox.h>
#include "Converter.h"

Converter::Converter(QApplication *app) : theApp(app) {
  quit   = new QPushButton("quit", this);
  f2c    = new QPushButton("F -> C", this);
  c2f    = new QPushButton("C -> F", this);
  editC  = new QLineEdit(this);
  editF  = new QLineEdit(this);
  labelF = new QLabel("F", this);
  labelC = new QLabel("C", this);

  setWindowTitle("Converter");
  resize(340, 220);
  editC->setGeometry(  40,  20, 80, 40);
  editF->setGeometry( 220,  20, 80, 40);
  c2f->setGeometry(    40,  80, 80, 40);
  f2c->setGeometry(   220,  80, 80, 40);
  quit->setGeometry(  220, 160, 80, 40);
  labelC->setGeometry( 10,  20, 20, 40);
  labelF->setGeometry(190,  20, 20, 40);
```

GUI Objekte anlegen

GUI Objekte positionieren

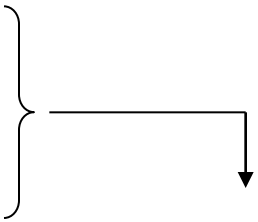
```
  QWidget::connect(quit, SIGNAL(clicked()), app, SLOT(quit()));
  QWidget::connect(c2f, SIGNAL(clicked()), this, SLOT(slotC2F()));
  QWidget::connect(f2c, SIGNAL(clicked()), this, SLOT(slotF2C()));
}
```

Kommunikation zwischen GUI Objekte einrichten

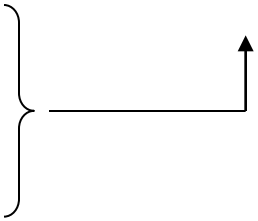
```
Converter::~Converter() {
  delete quit;
  delete f2c;
  delete c2f;
  delete editC;
  delete editF;
  delete labelC;
  delete labelF;
}
```

GUI Objekte freigeben

```
void Converter::slotC2F() {
  editC->selectAll();
  QString s = editC->selectedText();
  bool ok;
  double val = s.toDouble(&ok);
  if (!ok) QMessageBox::information(
    this, "invalid input", "please enter numbers"
  );
  val = 9 * val / 5 + 32;
  editF->setText(QString("%1").arg(val, 0, 'f', 1));
}
void Converter::slotF2C() {
  editF->selectAll();
  QString s = editF->selectedText();
  bool ok;
  double val = s.toDouble(&ok);
  if (!ok) QMessageBox::information(
    this, "invalid input", "please enter numbers"
  );
  val = 5 * (val - 32) / 9;
  editC->setText(QString("%1").arg(val, 0, 'f', 1));
}
```



Fehlerbehandlung unschön
⇓
Ausnahmen wären eleganter!



Qt compilieren und linken (I)

Auszug aus Verzeichnisstruktur nach Installation von Qt 4.7.3:

```
- Qt
  - 4.7.3
    - qt
      - bin
      - include
      - lib
```

Wurzel der Installation
Version (= 4.7.3)
Beginn von Qt
ausführbare Programme
Header-Dateien
Bibliotheken

Stand: Januar 2013

Dem Compiler muss gesagt werden,

- wo er die Header-Dateien zum Compilieren finden kann:

```
C:\Qt\4.7.3\qt\include;C:\Qt\4.7.3\qt\include\QtGui
```

- wo er die statischen Bibliotheken zum Linken finden kann:

```
C:\Qt\4.7.3\qt\lib
```

- welche Bibliotheken er zum Linken verwenden soll: **d** → debug

`QtCore4.lib QtGui4.lib` bzw. `QtCored4.lib QtGuid4.lib` u.v.a.

Qt compilieren und linken (II)

Aufruf des Meta Object Compilers:

```
"$(VADER_QT)\bin\moc.exe" -DUNICODE -DWIN32 -
DQT_LARGEFILE_SUPPORT -DQT_CORE_LIB -DQT_GUI_LIB "-
I$(VADER_QT)\include" "-I$(VADER_QT)\include\qtmain"
"-I$(VADER_QT)\include\QtCore" "-I$(VADER_QT)\QtGui"
"-I." $(InputPath) -o "moc_$(InputName).cpp"
```

Was passiert dort?

- Der moc wird aufgerufen (moc.exe)
- Es werden Präpozessor Definitionen angelegt („-Dxxx“)
- Es werden Verzeichnisse zum Standard Include Pfad hinzugefügt („-Ixxx“)
- Es wird angegeben was übersetzt werden soll ($(InputPath) → VS Makro)
- Es wird angegeben, wie das Resultat heißen soll

→ Eigentlich genau das selbe wie beim Aufruf des „normalen“ Compilers!

Auszug aus Verzeichnisstruktur nach Installation von Qt 4.7.3:

```
- Qt
  - 4.7.3
    - qt
      - bin
      - include
      - lib
```

Wurzel der Installation
Version (= 4.7.3)
Beginn von Qt
ausführbare Programme
Header-Dateien
Bibliotheken

Stand:
Januar 2013

Dem *Laufzeitsystem* muss gesagt werden,

- wo es die dynamischen Bibliotheken finden kann:

```
C:\Qt\4.7.3\qt\bin
```

muss u.a. in der Umgebungsvariable **`path`** stehen

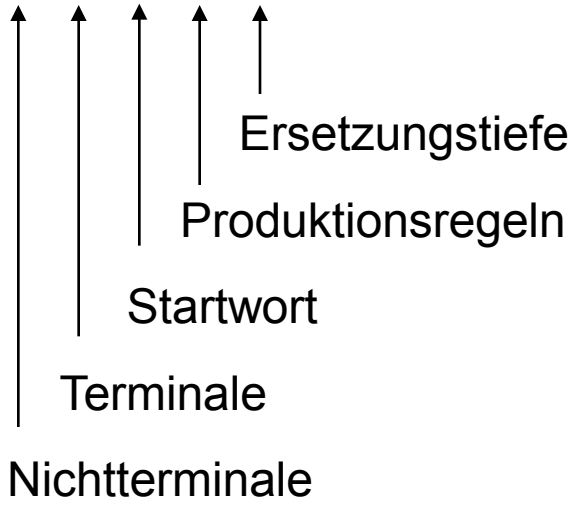
Für Linux-basierte Systeme ist das Prinzip gleich!
Nur die Pfade sehen etwas anders aus.

Lindenmayer-Systeme (L-Systeme) nach Aristid Lindenmayer, theoret. Biologe, Ungarn

Intention: axiomatische Theorie zur biologischen Entwicklung

Formalismus: Ersetzungssysteme ähnlich zu formalen Grammatiken

Quintupel: (N, T, ω, P, n)



hier:

< 6

1 Regel: F → ...

beliebig aus N ∪ T

+ – | [ ]

F

Vorgehensweise (gemäß unserer Einschränkungen): Schritt 1

setze s = ω (Startwort)

while (n > 0)

  initialisiere leere Variable t
  laufe von links nach rechts über s:
    falls Terminal dann nach t kopieren
    falls Nichtterminal F dann rechte Seite der Produktionsregel nach t kopieren
  setze s = t
  setze n = n – 1

endwhile

Bsp: ({ **F** }, { **+**, **-**, **[**, **]**, **|** }, **F+F**, { **F→F--F** }, 2)

F+F → F--F+F--F → F--F--F--F+F--F--F--F

Vorgehensweise (gemäß unserer Einschränkungen): Schritt 2

sei s das erhaltene Wort nach n Ersetzungsrunden

setze (x_0, y_0, α_0) als Startwert fest, setze k = 0, λ = Schrittweite, β = Winkel

laufe über s von links nach rechts

falls F: $(x_{k+1}, y_{k+1}, \alpha_{k+1}) = (x_k + \lambda \cos \alpha_k, y_k + \lambda \sin \alpha_k, \alpha_k)$;
zeichne Linie von (x_k,y_k) nach (x_{k+1},y_{k+1})

falls +: $(x_{k+1}, y_{k+1}, \alpha_{k+1}) = (x_k, y_k, \alpha_k + \beta)$;

falls –: $(x_{k+1}, y_{k+1}, \alpha_{k+1}) = (x_k, y_k, \alpha_k - \beta)$;

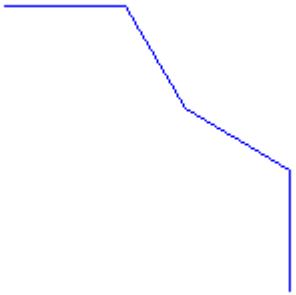
falls | : $(x_{k+1}, y_{k+1}, \alpha_{k+1}) = (x_k, y_k, \alpha_k - 180°)$;

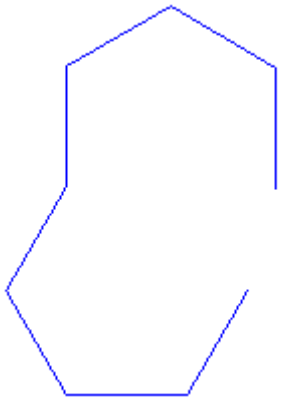
falls [: `push` (x_k, y_k, α_k); $(x_{k+1}, y_{k+1}, \alpha_{k+1}) = (x_k, y_k, \alpha_k)$;

falls] : $(x_{k+1}, y_{k+1}, \alpha_{k+1})$ = `top(); pop()`

setze k = k + 1

Bsp: `F+F` → `F--F+F--F` → `F--F--F--F+F--F--F--F` $\beta = 30°$





... noch nicht spektakulär ...

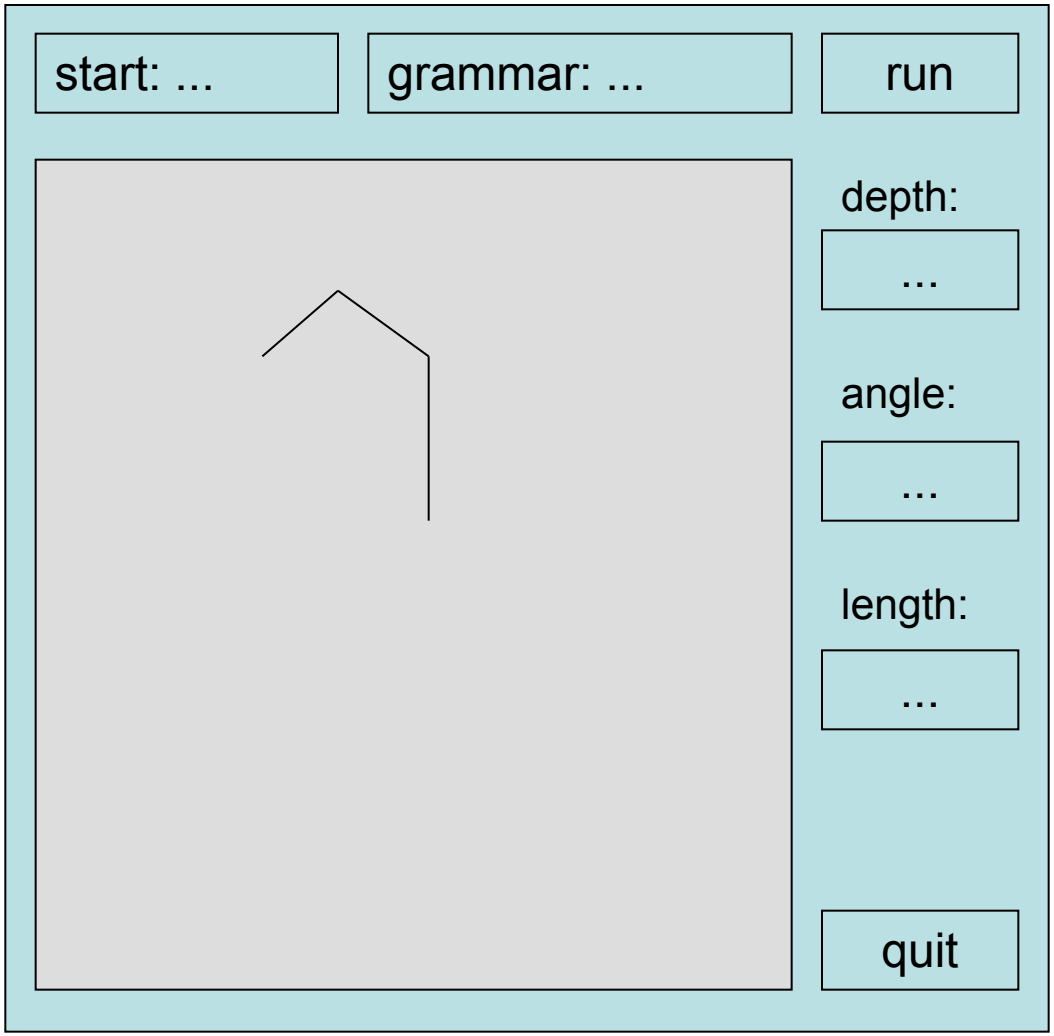
Planung der GUI

```
class Canvas :
  public QWidget
```

sorgt für die Darstellung eines L-Systems

```
class Window :
  public QWidget
```

verwaltet alle Controls



```
class Window : public QWidget {
Q_OBJECT
public:
  Window(QApplication *aApp);
  ~Window();

public slots:
  void run();

protected:
  QApplication *fApp;
  QLineEdit *fStart, *fGrammar, *fLength;
  QPushButton *fRun, *fQuit;
  QSpinBox *fDepth, *fAngle;
  QLabel *fLabelStart, *fLabelGrammar, *fLabelLength,
         *fLabelDepth, *fLabelAngle;
  QSlider *fSliderH, *fSliderV;
  Canvas *fCanvas;
};
```

erfordert Aufruf des Präprozessors `moc` vor eigentlicher C++ Compilierung

Datei

`Window.h`

Auszug aus Verzeichnisstruktur nach Installation von Qt 4.7.3

```
- Qt
  - 4.7.3
    - qt
      - bin
      - include
      - lib
```

Wurzel der Installation
Version (= 4.7.3)
Beginn von Qt
ausführbare Programme
Header-Dateien
Bibliotheken

Aufruf des Prä-Compilers `moc` vor eigentlicher C++ Compilation:

→ als *pre-build event* oder ähnliches eintragen bzw. explizit aufrufen:

```
C:\Qt\4.7.3\qt\bin\moc -o WindowMeta.cpp Window.h
```

Datei, die erzeugt wird

Datei, die `slot` enthält

```
class Canvas : public QWidget {
Q_OBJECT

public:
  Canvas(QWidget *aParent = 0);
  void draw(QString &aStart, QString &aGrammar, int aDepth,
       QString &aLength, int aAngle, int aPosX, int aPosY);

protected:
  void paintEvent(QPaintEvent *aEvent);  // überschrieben

private:
  QString fStart, fGrammar;
  int fDepth, fLength, fAngle;
  QPoint fStartPos;
  QRectF exec(QString &aRule, QPainter *aPainter);
};
```

Implementierung der Klassen

⇒ live demo ... (mit MS Visual Studio 2008)

Demo mit Beispielen

```
start: F                grammar: F → F[-F]F[+F][F]
degrees 20      length 5        depth 5

start: F-F-F-F          grammar: F+F-F-FF+F+F-F
degrees 90      length 5        depth 5

start: F-F-F-F-F-F      grammar: F+F--F+F
degrees 60      length 5        depth 4

start: F                grammar: FF-[-F+F+F]+[+F-F-F]
degrees 20      length 4        depth 4

start: F                grammar: F[+F]F[-F]F
degrees 20      length 4        depth 4
```