

# Programmieren / Algorithmen & Datenstrukturen 2

## Grafische Benutzeroberflächen



Prof. Dr. Skroch

**Universitatea**  
**BABEȘ-BOLYAI**

# Grafische Benutzeroberflächen

Inhalt.

- ▶ Templates
- ▶ Abgeleitete Klassen
- ▶ Testgetriebene Programmierung
- ▶ Container, Iteratoren und Algorithmen der StdLib
- ▶ Fortgeschrittenes Suchen
- ▶ Fortgeschrittenes Sortieren
- ▶ **Grafische Benutzeroberflächen**

# Grafische Benutzeroberflächen

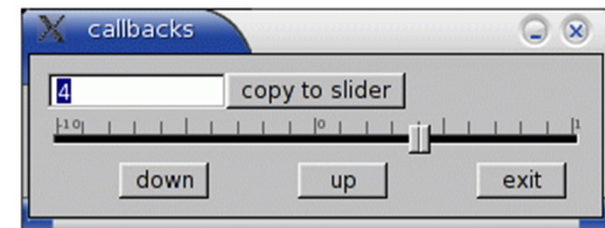
Textuelle Benutzerschnittstelle (der Konsolenprompt).

- ▶ Mit den bisher behandelten Programmen interagiert der Benutzer textbasiert über die Konsole.
- ▶ Typisch ist hier *der Prompt*, an dem das Programm Textausgaben schreiben und benötigte Eingaben als Text lesen kann.
- ▶ Da es nur einen Prompt gibt, ist die Interaktion rein sequenziell.
  - Interaktion erfolgt, wenn der Programmablauf auf entsprechende E/A-Funktionen wie `std::operator>>()` oder `std::operator<<()` trifft.
- ▶ Speziell bei Programmen im professionellen, technischen Umfeld ist diese Art der Benutzer-Interaktion nach wie vor anzutreffen.
  - Die Geräte, auf denen die Programme laufen, verfügen dort oft nicht über Bildschirm, Maus und Tastatur im Sinne eines üblichen PCs.

# Grafische Benutzeroberflächen

Grafische Benutzerschnittstellen (GUIs, graphical user interfaces).

- ▶ Mit einem Programm mit grafischer Benutzerschnittstelle kann der Benutzer im Vergleich dazu deutlich flexibler interagieren.
- ▶ Typisch ist hierbei, dass das Programm
  - *mehrere* interaktive E/A-Elemente anzeigt,
  - die *grafisch* gestaltet sind und mindestens auf Maus/Tastatur reagieren (z.B. "Fenster" oder "Schalter"),
  - und die *voneinander abhängen*.
- ▶ Das Programm kann damit umgehen, dass der Benutzer diese Elemente flexibel bedient (z.B. Eingaben in unterschiedlichen Reihenfolgen tätigt).
  - Man spricht bei solchen Programmen insofern von Ereignis-gesteuertem Programmablauf.
- ▶ Bei Anwendungen, die man von heutigen PCs kennt, erwartet man i.Allg., dass die Bedienung über derartige GUIs erfolgt.
  - Das Design einer GUI ist wichtig:  
Ergonomie (technisch-funktional), aber auch "Look & Feel" (Marketing).



# Ereignis-gesteuerter Programmablauf

Implementierungs-Techniken für Ereignis-Steuerung in Programmen.

- ▶ Es gibt gute GUI-Bibliotheken, die Ereignis-gesteuerte Programme unterstützen, und die man für GUI-Funktionalitäten im eigenen Programm einsetzen kann.
- ▶ Zum Beispiel *FLTK*:
  - Aussprache [fultik], wie engl. full tick, eigentlich "fast light tool kit".
  - Technisches Grundprinzip bei FLTK sind sog. [Rückruf-Funktionen](#).
  - Klassischer, direkter Ansatz, der (nicht nur) zur Implementierung flexibler, Ereignis-gesteuerter Abläufe einsetzbar ist.
- ▶ Zum Beispiel *Qt*:
  - Aussprache [kju:t], wie engl. cute
  - Technisches Grundprinzip bei Qt ist der sog. [Signal & Slot](#) Mechanismus.
  - Erweitert den Ansatz der reinen Rückruf-Funktionen, bessere Typsicherheit, intuitiverer Code, benötigt aber einen eigenen Präprozessor/Code-Generator (den Qt-eigenen "Meta Object Compiler moc").

# Grundlage Funktionszeiger

Funktionszeiger halten Adressen von Funktionen.

- ▶ (Die schon behandelten) Zeiger halten die Adresse eines *Objekts* vom Typ des Zeigers (gehen Sie das Thema ggf. nochmals selbstständig durch).
  - Syntax z.B.:

```
int i { 42 };  
int* pi { &i }; // Zeiger auf int namens pi, zeigt initial auf i  
int j { *pi };
```
- ▶ Mit einer *Funktion* kann man zwei Dinge tun:
  - man kann sie *aufrufen*,
  - man kann ihre *Adresse herausfinden*.
- ▶ Man kann eine Funktion nicht nur über ihren Namen sondern auch über ihre Adresse aufrufen: die Adresse der Funktion wird dazu einem *Funktionszeiger* zugewiesen.
  - Dabei müssen die Signaturen bei Funktion und Funktionszeiger zusammen passen, d.h. (Rückgabe-)Typ und Liste der Parametertypen.
  - D.h. der Funktionszeiger muss neben dem Rückgabetyt auch die Liste der Parametertypen kennen.

# Grundlage Funktionszeiger

Funktionszeiger halten Adressen von Funktionen.

## ► Beispiel:

```
int max( int a, int b ) { return a > b ? a : b; } // liefert int, hat 2 int Parameter
int mult( int a, int b ) { return a * b; } // liefert int, hat 2 int Parameter
```

```
int (*fp)( int, int ) { &max }; // Funktionszeiger namens fp, der int liefert und
                                // 2 int Parameter hat, zeigt initial auf max()
```

```
cout << (*fp)( -7, -3 ) << '\n'; // -3
```

```
fp = &mult;
```

```
cout << (*fp)( -7, -3 ) << '\n'; // 21
```

## ► Hinweise zur Syntax:

- Der Funktionsaufruf `operator()` hat höhere Priorität als die Dereferenzierung `operator*`, also wäre `*fp(-3, 21)` äquivalent zu `*(fp(-3, 21))`, ein Syntaxfehler.
- Möglich ist aber `fp(-3, 21)`, da der Compiler erkennt, dass der Name `fp` ein Funktionszeiger ist und dann die Funktion an der Adresse korrekt aufruft.
- Um die wenig intuitive Syntax zu vermeiden (eigentlich: zu verschieben) werden für Funktionszeiger manchmal eigene Typnamen definiert.

```
using Fptr = int(*) (int,int); // vor C++11: typedef int(*Fptr) (int,int);
```

```
Fptr fp{ &max }; cout << (*fp)(-7,-3) << '\n';
```

```
fp = &mult;      cout << (*fp)(-7,-3) << '\n';
```

# Rückruf-Funktionen

Funktionszeiger als Parameter für andere Funktionen.

- ▶ Verwendet man Funktionszeiger als Parameter für Funktionen, so spricht man von *Rückruf-Funktionen* (engl. "callbacks").
  - In einer Funktion sind Parameter als Funktionszeiger deklariert und im Quellcode dieser Funktion werden die Funktionszeiger eingesetzt (aufgerufen).
  - D.h.: über den Funktionszeiger wird aus der ersten Funktion diejenige zweite Funktion aufgerufen, die beim Aufruf der ersten als entsprechender Parameter übergeben wurde.
  - Mit anderen Worten:
    - Eine erste Funktion (die man sich auch als "Server" vorstellen kann) wird mit der Adresse einer beliebigen, geeigneten zweiten Funktion (die man sich auch als "Client" vorstellen kann) als Parameter aufgerufen.
    - Der aufgerufene "Server" ruft über den Funktionszeiger wiederum den übergebenen "Client" auf: man spricht von Rückruf-Funktionen.
    - Ohne den "Server"-Quellcode zu ändern kann man die "Server"-Funktion mit beliebigen "Client"-Funktionen syntaktisch korrekt aufrufen, solange nur die Signatur des "Clients" zum Funktionszeiger passt.



# Rückruf-Funktionen

Funktionszeiger als Parameter für andere Funktionen.

## ► Beispiel aus der C-Welt:

```
void std::qsort( void* base, size_t num, size_t size,  
                int (*comp)( const void*, const void* ) );
```

- `qsort()` benötigt als vierten Parameter einen Zeiger auf eine Funktion, die `int` liefert und als Parameter zwei Zeiger auf Konstanten von unbekanntem Typ hat.
  - Mit dieser Funktion vergleicht `qsort()` während der Sortierung je zwei Elemente vom zu sortierenden Typ.
- Um `qsort()` zu verwenden, muss beim Aufruf eine geeignete, tatsächliche Vergleichsfunktion definiert sein und deren Adresse an den Funktionszeiger als vierter Parameter übergeben werden.
  - Die Vergleichsfunktion muss `0` liefern, wenn die beiden Werte gleich sind, `<0` wenn der erste ("kleinere") Wert vor dem zweiten Wert einsortiert werden soll, und `>0` wenn der erste Wert nach dem zweiten Wert einsortiert werden soll.

## ► Bemerkung:

Wie man sieht wirkt das eher altmodisch und nicht sehr intuitiv...

# Rückruf-Funktionen

Funktionszeiger als Parameter für andere Funktionen.

## ► Weiteres Beispiel:

```
// Typname Pred1
using Pred1 = bool(*) ( const int& );

// Rueckruf-Funktion ("Server") fuer einstellige int-Praedikate:
bool pred_1( const int& v, Pred1 ) {
    std::cout << "client says: " << v << " is ";
    return Pred1( v );
}

bool even( const int& i ) { // Einstelliges Praedikat ("Client"): gerade Zahl?
    if( i%2 ) std::cout << "not ";
    std::cout << "even\n";
    return i%2;
}

bool negative( const int& i ) { // Einstelliges Praedikat ("Client"): negative Zahl?
    if( !(i<0) ) std::cout << "not ";
    std::cout << "negative\n";
    return i<0;
}

int main() { // Aufruf z.B.:
    pred_1( -99, even ); // Ist -99 gerade?
    pred_1( 2, negative ); // Ist 2 negativ?
    return 0;
}
```

# Rückruf-Funktionen

"Strong cohesion, loose coupling".

- ▶ Rückruf-Funktionen tragen oft zur Umsetzung eines der wohl grundlegendsten Design-Prinzipien verteilter Systeme bei:  
*schwache (lose) externe Kopplung.*
- ▶ Der "Treiber" oder "Server" `pred_1()` ist unabhängig von den einzelnen Prädikaten.
  - `pred_1()` kennt nur einen Funktionszeiger, d.h. die Signatur eines Funktionsaufrufs.
- ▶ Jedes einstellige Prädikat einer ganzen Zahl wie `even()`, `negative()` oder `prime()`, kann
  - erst als Funktion mit passender Signatur definiert
  - und dann beim Aufruf von `pred_1()` übergeben werden.
- ▶ Die Abhängigkeit (Kopplung) besteht formal nur in der Signatur des Funktionszeigers.

# Rückruf-Funktionen

Funktionszeiger können auch die Adressen von Methoden (Memberfunktionen) halten.

## ► Beispiel:

```
class Counter {
public:
    Counter() : val{} {}
    int get_value() const { return val; }
    void set_value( const int& v ) { val = v; }

private:
    int val;
};

// Rueckruf-Signatur des 2. Parameters, die zum "Getter" passt:
int cb1( Counter& a, int (Counter::*mp)() const ) { return (a.*mp)(); }

// Rueckruf-Signatur des 2. Parameters, die zum "Setter" passt:
void cb2( Counter& a, void (Counter::*mp)( const int& ), const int& i ) { (a.*mp)(i); }

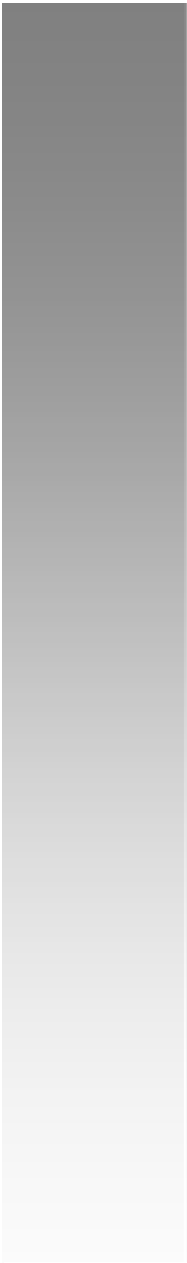
int main() {
    Counter c1 {};
    cb2( c1, &Counter::set_value, 77 );           // c1.set_value(77);
    std::cout << cb1( c1, &Counter::get_value ); // std::cout << c1.get_value();
    return 0;
}
```

Die Rückruf-Funktionen `cb1()` und `cb2()` können mit beliebigen Methoden der `Counter` Klasse aufgerufen werden, solange die Signaturen passen.

# Rückruf-Funktionen

Ereignis-gesteuerte Abläufe mit grafischen Elementen.

- ▶ Derartige Rückruf-Mechanismen werden, v.a. auch in der GUI-Programmierung, zur Ereignis-Steuerung eingesetzt.
- ▶ In FLTK ist beispielsweise jedes grafische Element (Widget) mit einem Funktionszeiger ausgestattet, der bei einer jeweils Widget-spezifischen Aktionen aufgerufen wird.
  - Welche Funktion aufgerufen (zurückgerufen) wird, und was genau diese tut, definiert der Programmierer beim Erstellen der GUI.
- ▶ Beispiel: beim Schließen eines Fensters soll ein Pop-Up mit Sicherheitsabfrage gezeigt werden: Sichern oder Verwerfen der Daten?
- ▶ Die Details der Sicherheitsabfrage – welche Daten, Form des Pop-Ups, etc. – sind über den Funktionszeiger abstrahiert.
- ▶ Es ist lediglich für das Fenster-Widget mit dem Ereignis "Schließen des Fensters" in FLTK ein Funktionszeiger-Aufruf implementiert.
- ▶ Der Programmierer (Nutzer der FLTK Bibliothek) definiert eine entsprechend geeignete Funktion (für die syntaktische Eignung muss nur die Signatur stimmen), und übergibt sie als Funktionszeiger.



# Qt

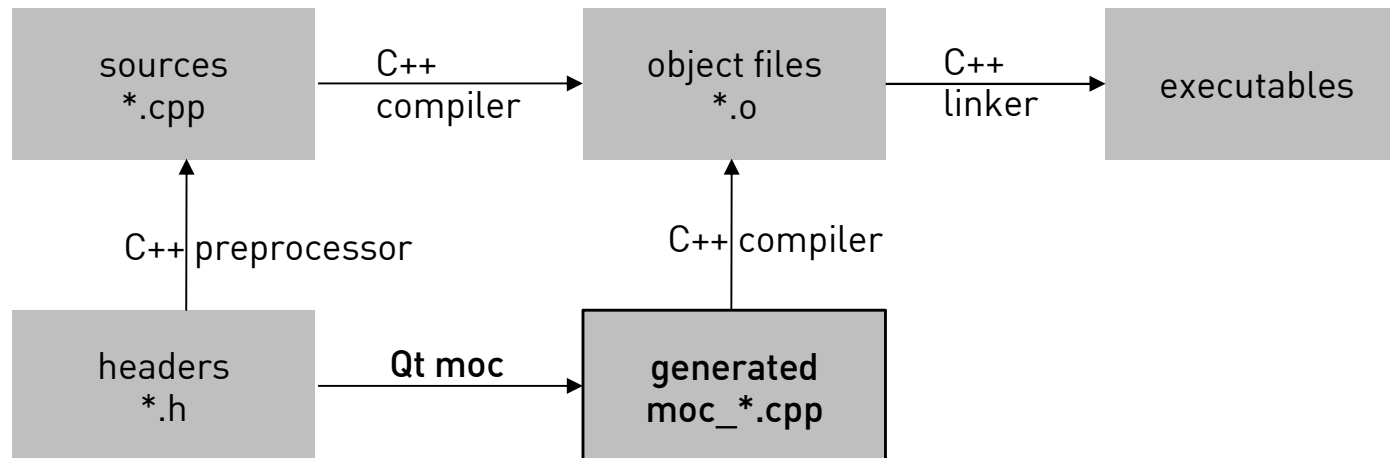
Wir sehen uns die Qt Klassenbibliothek für Programme mit grafischer Benutzerschnittstelle und Ereignis-gesteuertem Ablauf näher an.

- ▶ Installation abrufbar unter <http://qt-project.org>
  - Zum Mitmachen sollten Sie einen Computer benutzen, auf dem
    - Qt Library (Klassenbibliothek),
    - Qt Creator (IDE)in der zugrunde liegenden Version installiert sind.
- ▶ Diesen Unterlagen zugrunde liegende Versionen:
  - [Qt Library 5.4](#)
  - [Qt Creator IDE 3.3.0](#)
- ▶ Lizenz: GPL, LGPL.
- ▶ Zentrale Qt Grundkonzepte:
  - Zusätzlicher Präprozessor/Codegenerator (moc: "Meta Object Compiler"),
  - Introspektion,
  - Signale und Slots.

# Qt Tool Chain mit dem *moc*

Die C++ Tool Chain ist durch den Qt-eigenen *moc* (sog. Meta Object Compiler) erweitert.

- ▶ Qt C++ Tool Chain (vereinfacht):



- ▶ Der moc ist ein zusätzlicher Präprozessor und Code-Generator, der Quellcodedateien parst und Code generiert.
- ▶ Dabei wird, gesteuert durch Qt-eigene Schlüsselwörter und Makros, Quellcode in Standard C++ Syntax generiert, der im Wesentlichen Informationen zur Introspektion verfügbar macht, indem v.a. auch sog. "Meta-Objekte" erzeugt werden.



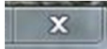
# Introspektion

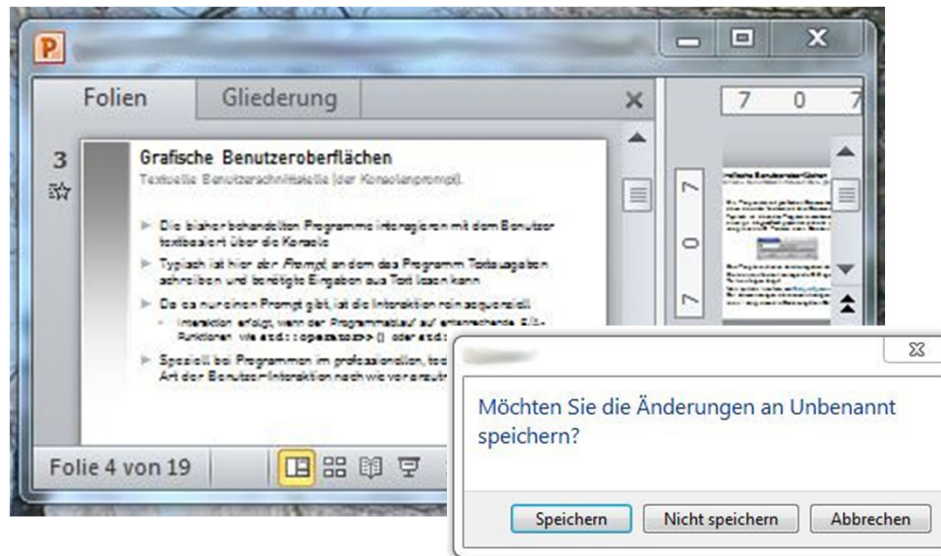
Programme, die zur Laufzeit Informationen über sich selbst erhalten.

- ▶ Man spricht beim Programmieren von Introspektion (oder auch Reflexion), wenn das Programm zur Laufzeit Informationen über sich selbst erhalten kann,
  - wie z.B. den Klassennamen eines Objekts, das im Programm verwendet wird, oder den Rückgabebetyp und die Parameterliste einer Funktion, die im Programm aufgerufen wird.
- ▶ Viele Programmiersprachen stellen dafür extra Sprachmittel bereit,
  - z.B. Python oder Java, aber auch Programme in Maschinencode – letztere können sich zur Laufzeit sogar selbst verändern (weil in Maschinencode Anweisungen und Daten im Prinzip gleich gespeichert werden).
- ▶ C++ unterstützt (fast) keine Introspektion mit eigenen Sprachmitteln.
- ▶ C++ ermöglicht aber die effiziente Implementierung von Introspektion durch geschickten Einsatz seiner grundlegenden Sprachmittel.
  - Hauptaufgabe des Qt moc ist es, Code zu generieren, der bestimmte Informationen zur Introspektion über die Erzeugung von C++ Standardsyntax zur Verfügung stellt (weil das für die Signal-Slot-Technik von Qt erforderlich ist).
  - Dieser Schritt wird von der Qt Creator IDE mit Hilfe des moc automatisch miterledigt.

# Signal & Slot

Qt Objekte kommunizieren untereinander über den sog. Signal-Slot-Mechanismus.

- ▶ In GUIs ist es normalerweise notwendig, dass die unterschiedlichen GUI Elemente (die laut Duden "Widgets" heißen) untereinander kommunizieren.
  - Grund: die einzelnen Elemente hängen meist voneinander ab, wenn in einem Widget etwas passiert, muss das ggf. anderen Widgets mitgeteilt werden.
  - Beispiel: wenn der Benutzer auf  klickt (um das Fenster zu schließen) und die im Programm geöffnete Datei geändert wurde, soll die Sicherheitsabfrage "Möchten Sie die Änderungen speichern?" als Pop-Up erscheinen.



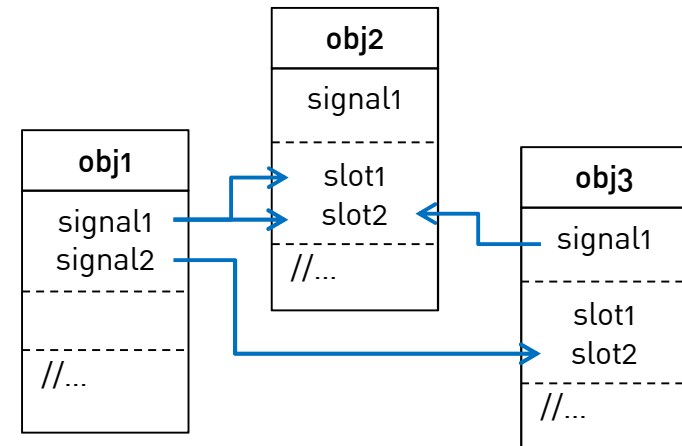
- Der Programmierer (Nutzer der GUI Bibliothek) muss irgendwie eine individuelle Funktion "einhängen" können.
- Der klassische Weg sind die bereits besprochenen Rückruf-Funktionen.
- Qt setzt mit dem Signal & Slot Mechanismus eine erweiterte Technik ein.

# Signal & Slot

Qt Objekte kommunizieren untereinander über den sog. Signal-Slot-Mechanismus.

- ▶ Ein Signal wird gesendet ("emittiert") wenn ein bestimmtes Ereignis eintritt.
- ▶ Ein Slot reagiert auf bestimmte Signale.
- ▶ Damit ein Slot auf ein Signal reagiert, muss man Signal und Slot mittels der Qt Memberfunktion `connect()` verknüpfen:

```
connect( obj1, signal1, obj2, slot1 );  
connect( obj1, signal1, obj2, slot2 );  
connect( obj1, signal2, obj3, slot2 );  
connect( obj3, signal1, obj2, slot2 );
```



- Die Signale und Slots sind `public` Methoden der fraglichen Klassen.
- Die `connect()` Methode gehört `static` zu einer Klasse namens `QObject`, von der aus man Meta-Informationen für die Kopplung erreicht, und `QObject::connect()` wird mit Funktionszeigern auf die zu verknüpfenden Methoden aufgerufen.
- Der *moc* sammelt die erforderlichen Informationen beim Parsen des ursprünglichen Quellcodes und generiert den für die Signal-Slot Rückrufe zusätzlich nötigen Quellcode.
- Tatsächlich aufgerufen wird – über generierte Tabelleneinträge in "Meta-Objekten" – zuerst das Signal und unmittelbar danach die mit dem Signal verknüpften Slots.

# Signal & Slot

Syntax am Beispiel.

## ► Die Counter Klasse mit Qt Signal / Slot:

```
// counter.h
#include <QObject>

class Counter : public QObject {
    Q_OBJECT
    int val;

public:
    Counter();
    int get_value() const;

public slots:
    void set_value( const int& );

signals:
    void value_changed( const int& );
};

// Der moc findet die Qt-eigenen Schluesselwoerter (oben in blau) und erkennt daran,
// fuer welche Methoden Code generiert werden muss. Platz fuer die Meta-Informationen
// zur Introspektion findet sich in den Member-Containern sog. Meta-Klassen.
// Q_OBJECT wird vom moc als Makro expandiert und zeigt dem moc an, dass fuer die Klasse
// Meta-Objekte angelegt werden muessen. Fuer den C++ Praeprozessor ist slots leer
// definiert (d.h. es wird geloescht), signals ist als public: definiert.
```

# Signal & Slot

## Syntax am Beispiel.

### ► Die Counter Klasse mit Qt Signal / Slot:

```
// counter.cpp
#include "counter.h"

Counter::Counter() : QObject{}, val{} {}

int Counter::get_value() const { return val; }

void Counter::set_value( const int& v ) {
    if( val != v ) {
        val = v;
        emit value_changed( v );
    }
}

// Fuer den C++ Praeprozessor ist emit leer definiert (d.h. es wird geloescht).
// Die Signal-Definitionen werden vom moc automatisch generiert.
//     Nur zur Illustration:
//     void Counter::value_changed( const int & _t1 ) {
//         void* _args[] = { nullptr, const_cast<void*>( reinterpret_cast<const void*>(&_t1) ) };
//         QMetaObject::activate( this, &staticMetaObject, 0, _args );
//     }
```

# Signal & Slot

## Syntax am Beispiel.

### ► Einsatz:

```
Counter snd {}; // snd.get_value() == 0
std::cout << snd.metaObject()->className(); // Counter
Counter rcv {}; // rcv.get_value() == 0
std::cout << rcv.metaObject()->className(); // Counter

QObject::connect( &snd, &Counter::value_changed,
                  &rcv, &Counter::set_value );

rcv.set_value( 333 ); // snd.get_value() == 0, rcv.get_value == 333
snd.set_value( 1648 ); // snd.get_value() == 1648, rcv.get_value == 1648
```

### ► Bemerkungen:

- Da das `value_changed` Signal von `snd` mit dem `set_value` Slot von `rcv` verknüpft ist, aber nicht umgekehrt, zeigen sich obige Werte.
  - Das `value_changed` Signal von `rcv` ist mit keinem Slot verbunden und wird daher ignoriert.
- Von `set_value` wird nur dann das Signal emittiert, wenn sich der Wert auch geändert hat.

# Metaklassen und der moc

## Introspektion.

- ▶ Der moc erzeugt für jede Klasse, die das `Q_OBJECT` Makro enthält, ein Objekt vom Typ `QMetaObject`.
  - In Member-Datenfeldern der Meta-Objekte sind die Informationen für die Introspektion enthalten, insbes. sind auch die Signale und Slots (als `public` Memberfunktionen der Klasse) bekannt.
- ▶ Vorsicht: Qt kann nur mit einer Teilmenge der C++ Syntax umgehen, zwei Beispiele für Einschränkungen sind:
  - Templateklassen sind nicht im Signal-Slot Mechanismus möglich.
  - Signale und Slots selbst können keine Funktionszeiger sein.
- ▶ Der Programmierer einer Qt GUI muss sich normalerweise nicht mit den Qt Metaklassen auseinandersetzen.
  - Es handelt sich um Qt Interna, um die sich der moc und die IDE kümmern und damit dem Programmierer viel manuelle Arbeit abnehmen.
  - Sobald aber z.B. ein Skript für die automatisierte GUI-Erstellung programmiert werden soll, werden diese internen Mechanismen wichtig.
  - Daher zumindest dieser grobe, ersten Überblick...

# Die Qt Basisklasse `QObject`

Die Klasse `QObject` ist die Qt Basisklasse für die Umsetzung der Signal-Slot Technik.

- ▶ Klassendiagramm (sehr stark vereinfachter Teilausschnitt):

Die Qt Klassenhierarchie umfasst mehr als 1000 Klassen, so dass selbst ein teilweiser Überblick nicht mehr sinnvoll möglich ist

... `QObject` ... `QMetaObject` ...

- ▶ Weitere zentrale Introspektions-Klassen:
  - `QMetaMethod`, `QMetaProperty`,  
`QMetaEnum`, `QMetaClassInfo`, `QMetaType`



# Die Qt Basisklasse `QObject`

Die Klasse `QObject` ist die Qt Basisklasse für das Design der Signal-Slot Technik.

- ▶ Die Qt Klassenhierarchie umfasst mehr als 1000 Klassen, so dass ein sinnvoller Überblick praktisch nicht mehr möglich ist.
- ▶ Die Klasse `QObject` ermöglicht, als zentrale Basisklasse, v.a. den Mechanismus für die Kommunikation über Signale und Slots.
  - Die Qt Klassen, die von `QObject` abgeleitet sind, verfügen somit auch über die Signal-Slot Kommunikationstechnik.
- ▶ Signale und Slots dienen allgemein der Ereignis-gesteuerten Kommunikation und sind *keine* reine GUI Funktionalität.
- ▶ Die weiteren Details der Umsetzung von Introspektion mit den Qt Metaklassen sind umfangreich und interessant, gehen aber über diese Lehrveranstaltung hinaus.
- ▶ Die Dokumentation auf <http://qt-project.org> bietet allen Interessierten einen guten Einstieg in die weitere Vertiefung hierzu.

# Die Qt Klasse `QWidget`

Die Klasse `QWidget` ist die GUI-Basisklasse für die GUI-Typen.

- ▶ Ein Widget ist, ganz allgemein/abstrakt, ein Element einer GUI.
  - Eine GUI wird aus mehreren Widgets passend zusammengesetzt,
  - die einzelnen Widgets hängen für gewöhnlich funktional mit anderen Widgets der GUI zusammen.
- ▶ Damit die Programmierung der funktionalen Zusammenhänge zwischen den Widgets vereinfacht wird, können die Widgets in Qt über die Signal & Slot Technik untereinander kommunizieren (d.h. `QWidget` ist auch von `QObject` abgeleitet).
- ▶ Ein Widget wird Fenster genannt, wenn es nicht Teil eines übergeordneten Widgets ist.
  - Ein Fenster ist sozusagen ein unabhängiges "Top-Level-Widget".
- ▶ Es gibt viele von `Widget` abgeleitete Klassen (speziellere Widget-Typen), die echte, nützliche Funktionalität für GUIs bieten,
  - wie z.B. `QLabel` oder `QPushButton`.

# Hallo Qt

## Erstes Anwendungsbeispiel.

### ► Qt Syntax:

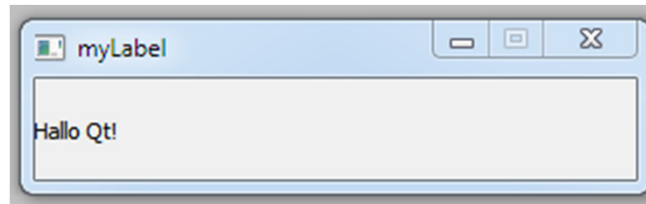
```
// main.cpp
#include <QApplication>
#include <QLabel>

int main( int argc, char** argv ) {
    QApplication app{ argc, argv };    // jede QWidget GUI-Anwendung hat genau
                                       // ein Objekt vom Typ QApplication
                                       // Qt GUI ohne QWidget: QGuiApplication
                                       // Qt ohne GUI: QCoreApplication

    QLabel w{ "Hallo Qt!" };           // ein Objekt von Typ QLabel ist ein (von QWidget
                                       // abgeleitetes) GUI-Element (vgl. unten)

    w.setFixedSize( 300, 50 );
    w.show();

    return app.exec();
}
```



```
// Eine wesentliche Aufgabe des app Objekts ist, die sog. "Ereignisschleife" bereit zu
// stellen, d.h. den Mechanismus, der Maus- und sonstige Ereignisse staendig abfragt
```

# Hallo Qt

Impliziter Aufruf von Konstruktor und Destruktor sichtbar gemacht.

## ► Qt Syntax

```
// main.cpp
#include "myLabel.h"

int main( int argc, char** argv ) {
    myQApp mqa{ argc, argv };
    myLabel mw{ "Hallo Qt!" };
    mw.setFixedSize( 300, 50 );
    mw.show();
    return mqa.exec();
}
```

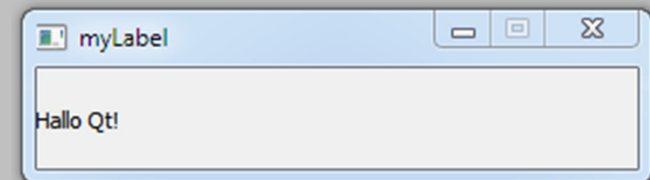
```
// mylabel.h
#include <QApplication>
#include <QLabel>
#include <QString>

class myLabel : public QLabel {
    Q_OBJECT
public:
    myLabel( QString );
    ~myLabel();
};

class myQApp : public QApplication {
    Q_OBJECT
public:
    myQApp( int, char** );
    ~myQApp();
};
```

```
15:44 >myLabel.exe
Constructor myQApp( int, char** )
Constructor myLabel( QString )
Destructor ~myLabel()
Destructor ~myQApp()
```

15:44 >



```
// mylabel.cpp
#include "mylabel.h"
#include <iostream>

myLabel::myLabel( QString s ) : QLabel{s} {
    std::cout << "Constructor myLabel(QString)\n";
}

~myLabel() {
    std::cout << "Destructor ~myLabel()\n";
}

myQApp::myQApp( int i, char** c ) : QApplication{ i,c } {
    std::cout << "Constructor myQApp( int, char** )\n";
}

myQApp::~~myQApp() {
    std::cout << "Destructor ~myQApp()\n";
}
```

# Qt Beispiel: digitaler Zähler

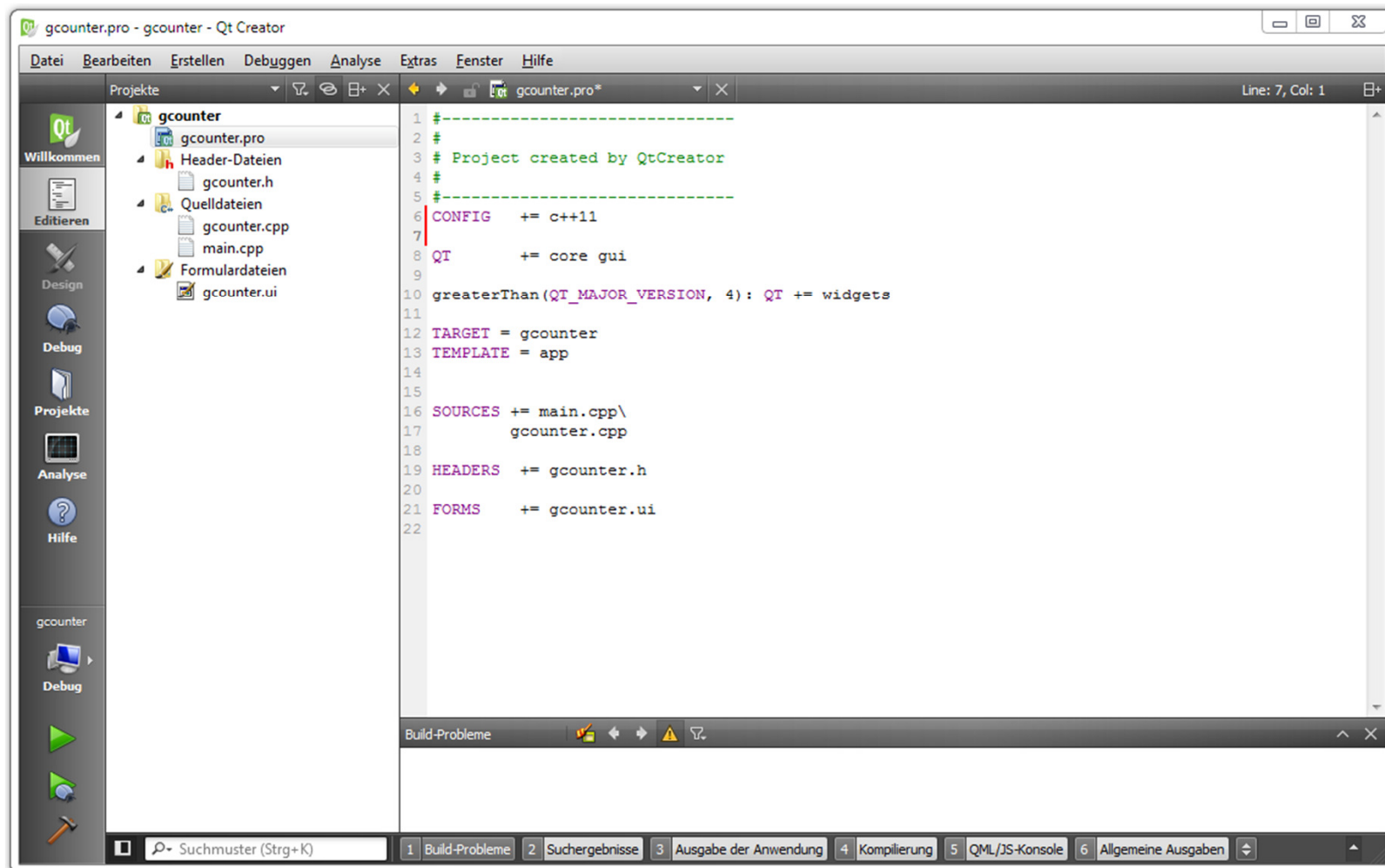
Unser Beispiel mit der `Counter` Klasse, jetzt mit GUI, über die IDE erstellt.

- ▶ In der `Qt Creator 3.3.0` IDE ein neues Projekt wie folgt anlegen:
  - Datei – Neu... – Application – Qt-Widgets-Anwendung – Auswählen...
- ▶ Projektname (z.B. `gcounter`) und Projektverzeichnis (z.B. `D:\QtProjects`) eingeben, weiter
- ▶ Das "Kit" (d.h. die Qt Creator-Einstellungen zum Übersetzen) auswählen (z.B. "*Desktop Qt 5.4.0 MinGW 32bit*"), weiter
- ▶ Klassenname: `Gcounter`, Basisklasse: `QWidget`,  
Header-Datei: `gcounter.h`, Quelldatei: `gcounter.cpp`  
Form-Datei generieren: ja, Form-Datei: `gcounter.ui`  
weiter
- ▶ Abschließen

# Qt Beispiel: digitaler Zähler

Unser Beispiel mit der `Counter` Klasse, jetzt mit GUI, über die IDE erstellt.

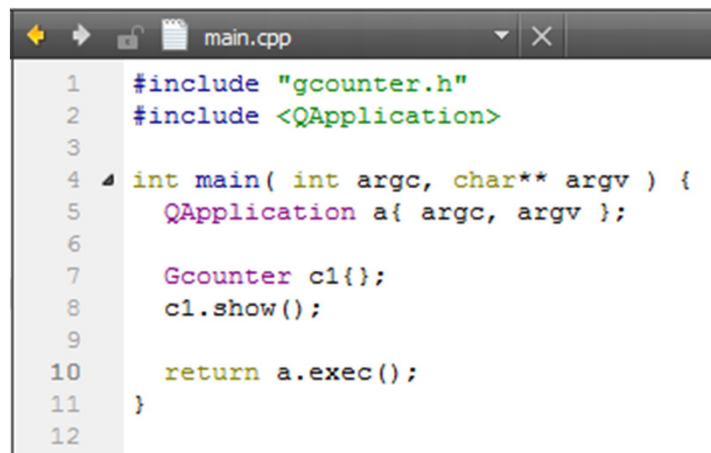
- In die **gcounter.pro** Datei die Zeile `CONFIG += c++11` eintragen:



# Qt Beispiel: digitaler Zähler

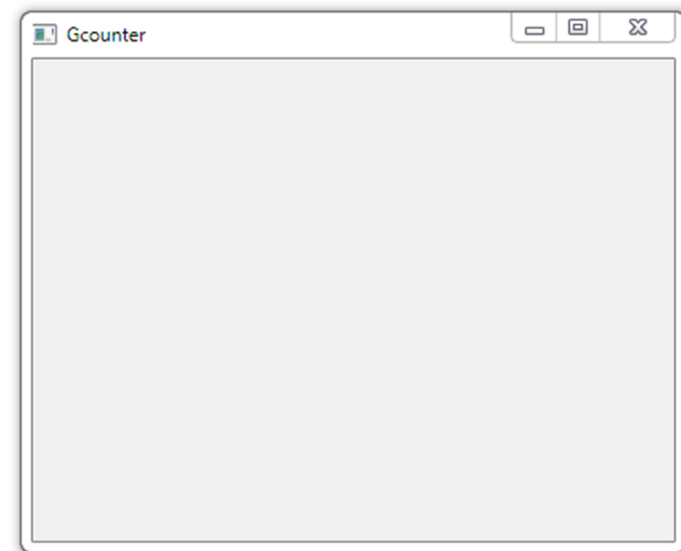
Die IDE erzeugt bereits folgendes Quellcodegerüst.

► Die **main.cpp** Datei.



```
1 #include "gcounter.h"
2 #include <QApplication>
3
4 int main( int argc, char** argv ) {
5     QApplication a( argc, argv );
6
7     Gcounter c1{};
8     c1.show();
9
10    return a.exec();
11 }
12
```

► Ausgabe:



# Qt Beispiel: digitaler Zähler

Die IDE erzeugt bereits folgendes Quellcodegerüst.

- Die Dateien **gcounter.h** und **gcounter.cpp** für die Gcounter Klasse:

```
gcounter.h
1  #ifndef GCOUNTER_H
2  #define GCOUNTER_H
3
4  #include <QWidget>
5
6  namespace Ui {
7      class Gcounter;
8  }
9
10 class Gcounter : public QWidget {
11     Q_OBJECT
12
13     public:
14         explicit Gcounter( QWidget* parent = nullptr );
15         ~Gcounter();
16
17     private:
18         Ui::Gcounter* ui;
19 };
20
21 #endif // GCOUNTER_H
22
```

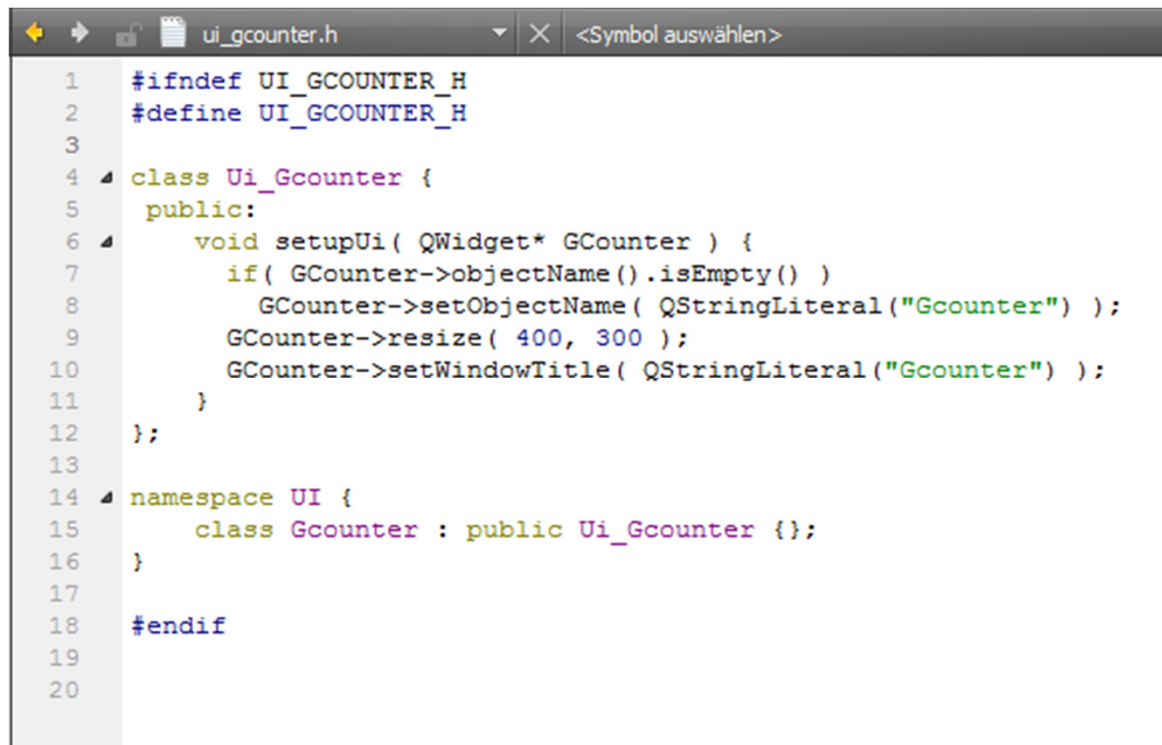
```
gcounter.cpp
1  #include "gcounter.h"
2  #include "ui_gcounter.h"
3
4  Gcounter::Gcounter( QWidget* parent )
5      : QWidget{ parent },
6      ui{ new Ui::Gcounter{} }
7  {
8      ui->setupUi( this );
9  }
10
11 Gcounter::~~Gcounter()
12 {
13     delete ui;
14 }
15
```



# Qt Beispiel: digitaler Zähler

Die IDE erzeugt bereits folgendes Quellcodegerüst.

- Eine zusätzliche Header-Datei `ui_gcounter.h`, die vom Design Tool der IDE bei jedem "Build" neu angelegt wird:



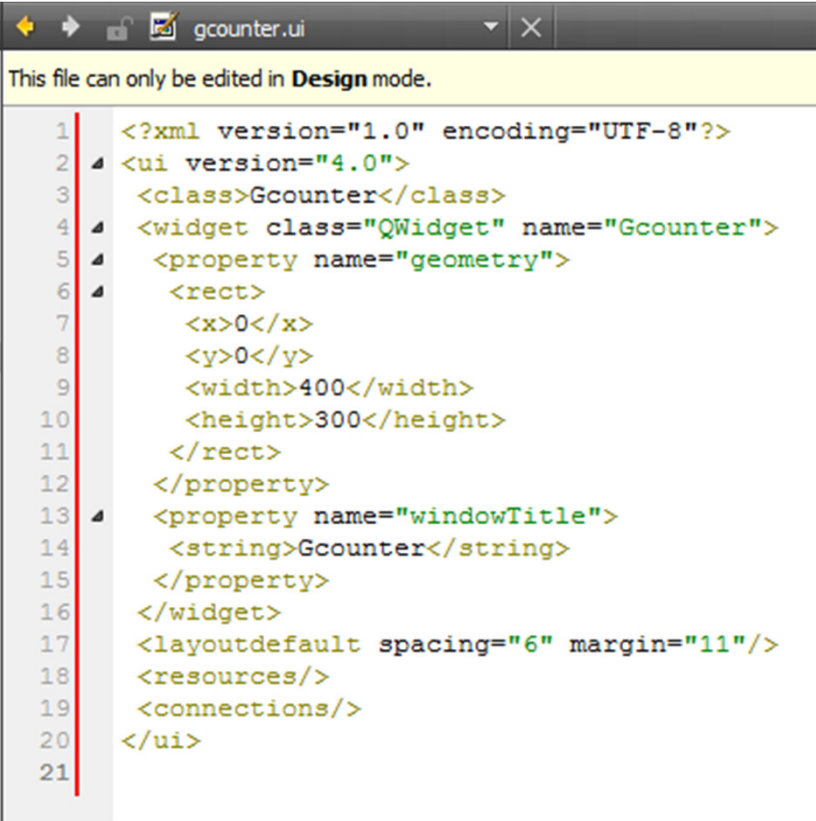
```
1  #ifndef UI_GCOUNTER_H
2  #define UI_GCOUNTER_H
3
4  class Ui_Gcounter {
5  public:
6      void setupUi( QWidget* GCounter ) {
7          if( GCounter->objectName().isEmpty() )
8              GCounter->setObjectName( QStringLiteral("Gcounter") );
9              GCounter->resize( 400, 300 );
10             GCounter->setWindowTitle( QStringLiteral("Gcounter") );
11         }
12     };
13
14     namespace UI {
15         class Gcounter : public Ui_Gcounter {};
16     }
17
18 #endif
19
20
```

(Kann nicht direkt manuell bearbeitet werden, es sind hier nur verkürzt die wesentlichen Quellcodeteile abgebildet.)

# Qt Beispiel: digitaler Zähler

Die IDE erzeugt bereits folgendes Quellcodegerüst.

- ▶ Die **gcounter.ui** Datei mit Layout- und weiteren Informationen:



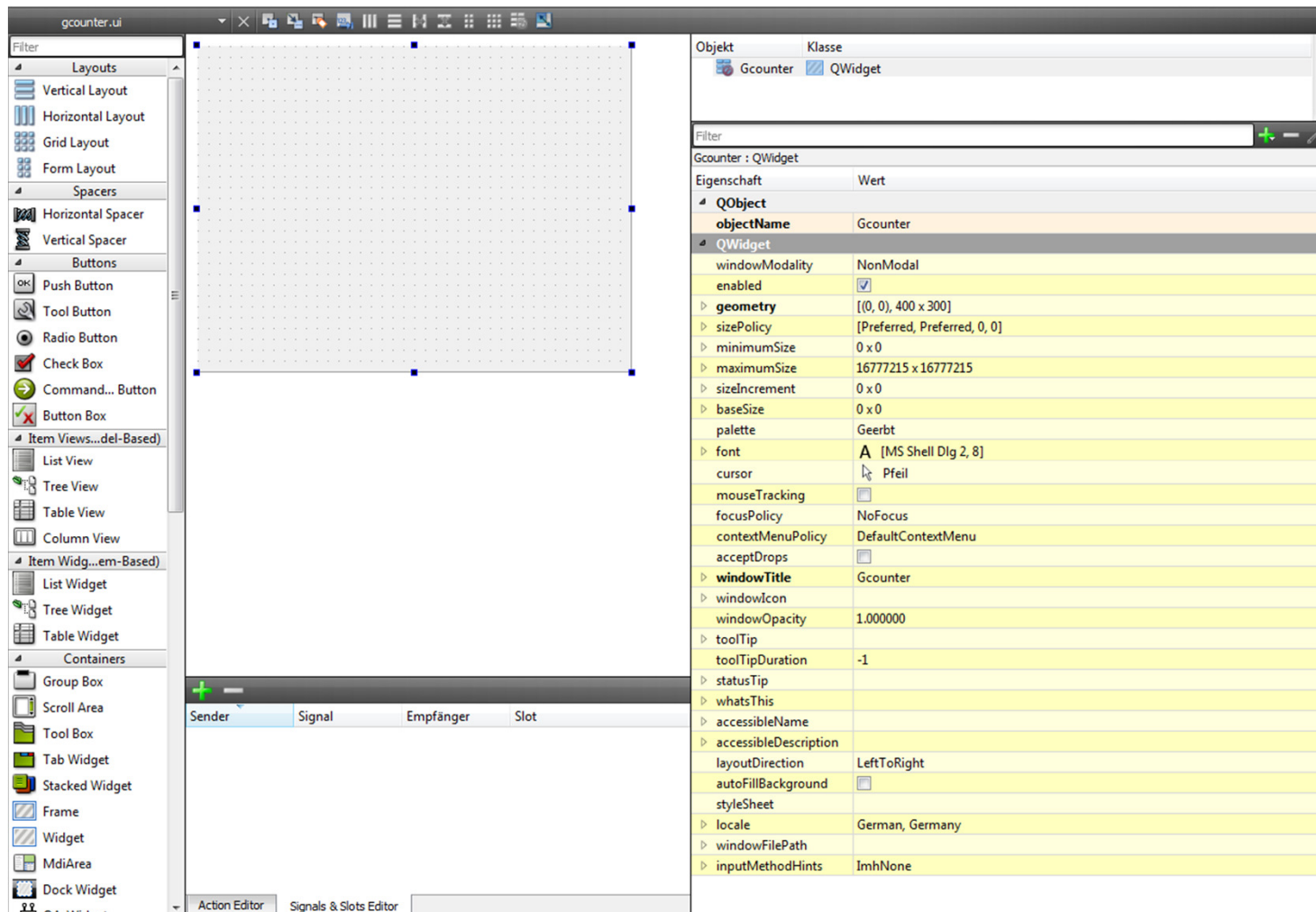
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <ui version="4.0">
3   <class>Gcounter</class>
4   <widget class="QWidget" name="Gcounter">
5     <property name="geometry">
6       <rect>
7         <x>0</x>
8         <y>0</y>
9         <width>400</width>
10        <height>300</height>
11      </rect>
12    </property>
13    <property name="windowTitle">
14      <string>Gcounter</string>
15    </property>
16  </widget>
17  <layoutdefault spacing="6" margin="11"/>
18  <resources/>
19  <connections/>
20 </ui>
21
```

(Kann *nicht* manuell bearbeitet werden.)

# Qt Beispiel: digitaler Zähler

Die IDE erzeugt bereits folgendes Quellcodegerüst.

- Im Design-Modus der IDE sieht die Bearbeitung in etwa so aus:

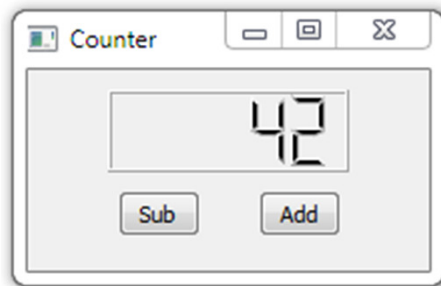


# Qt Beispiel: digitaler Zähler

Aufgabe: Zwei Buttons und ein Digitalzähler hinzufügen, zwei verbundene Zählwerke anlegen.

## ► Übung:

- Fügen Sie im Design-Modus ein Objekt vom Typ `QLCDNumber` und zwei Objekte vom Typ `QPushButton` hinzu.
- Verschönern Sie das Layout.

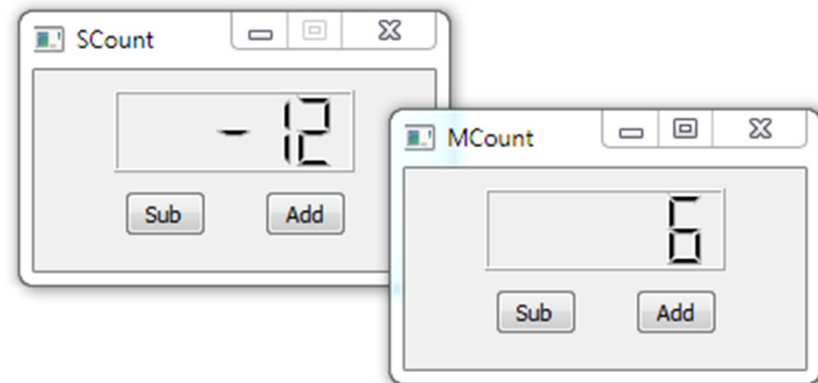


# Qt Beispiel: digitaler Zähler

Aufgabe: Zwei Buttons und ein Digitalzähler hinzufügen, zwei verbundene Zählwerke anlegen.

## ► Übung Teil 2:

- Arbeiten Sie ohne den Designer weiter: fügen Sie dazu in der IDE Ihrem Projekt eine C++ Headerdatei (z.B. namens `gcounter_ui.h`) hinzu. Kopieren Sie dann den Inhalt aus der vom Designer generierten `ui_gcounter.h` in Ihre neue Headerdatei, und entfernen Sie die generierte Datei (auch aus `gcounter.pro`).
- Passen Sie den übrigen Quellcode an und bringen Sie das Zählwerk ohne die generierte Formulardatei wieder zum Laufen.
- Verwenden Sie Signale und Slots, um die Zähleranzeige auf Knopfdruck inkrementieren und dekrementieren zu lassen.
- Erstellen Sie nun zwei Zähler, so dass Klicks vom einen Zählwerk auch das andere mitzählen lassen (aber nicht umgekehrt).



# Einige Beispielfragen

## Grafische Benutzeroberflächen.

- ▶ Wann spricht man von Ereignis-gesteuertem Programmablauf?
- ▶ Nennen Sie zwei technische Grundprinzipien zur Implementierung von Ereignis-gesteuerten Programmabläufen bei GUIs.
- ▶ Warum sind Ereignis-gesteuerte Programmabläufe gerade bei GUIs ein wichtiges Thema?
- ▶ Was sind Funktionszeiger? Wie unterscheiden sie sich von sonstigen Zeigern?
- ▶ Was sind Rückruffunktionen? Was haben Rückruffunktionen mit Funktionszeigern zu tun?
- ▶ Warum sind Rückruffunktionen ein klassisches Konzept zur Programmierung von GUIs? Welche besonderen Vorteile bieten sie? Wo liegen Nachteile?
- ▶ Was bedeutet Introspektion beim Programmieren?

# Einige Beispielfragen

Grafische Benutzeroberflächen.

- ▶ Was macht der Qt moc?
- ▶ Erläutern Sie mit eigenen Worten die Signal & Slot Technik von Qt.
- ▶ Erarbeiten Sie sich selbstständig zwei der vielen, einleitenden Beispiele aus dem Qt Creator 3.3.0 – sehen Sie sich dazu die Kurzbeschreibung der Beispiele an, suchen Sie sich Beispiele aus, die Sie selbst interessant finden

**Das war P / A & D – vielen Dank!**