Einführung in die Programmiersprache C++

... FÜR FORTGESCHRITTENE ...

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

- OpenMP
- Boost
- Boost Graph Library

Problem der Woche: Paddingbytes in Structs

Data Structure Alignment (1)

- x86-Prozessoren mit SSE2 Erweiterung und x64-Systeme greifen auf Daten immer in Blöcken von 128-Bit zu (16-Byte aligned)
- Immer wenn in einen Datenblock der nächste Member nicht mehr ins Alignment passt, werden Padding-Bytes hinzugefügt
- sizeof() zählt die Padding Bytes aber nicht mit...
- C/C++-Compiler füllen in Structs immer auf 4 Byte auf
- Datentypen werden in der Regel folgendermaßen angeordnet:

Typ	Größe	Alignment
`char`	1 Byte	2 Byte
`short`	2 Byte	1 Byte
`int`	4 Byte	4 Byte
`float`	4 Byte	4 Byte
`double`	8 Byte	8 Byte (Win), 4 Byte Linux

Data Structure Alignment (2)

- Beispiel:

```
struct Data{
    char Data1,
    short Data2,
    int Data3,
    char Data4
}
```

- `sizeof` sagt zur Compilezeit 8 Byte
- Es werden aber 12 Byte wegen Alignment angelegt
- Diese Padding Bytes werden beim Speichern mitgeschrieben:

```
Data dat;
write(&dat, sizeof(Data));
```

- macht nicht was es soll...

Data Structure Alignment (3)

- Was passiert ?
- In Wirklichkeit wird in etwa das Folgende generiert:

```
struct Data{
  char Data1,
  char Padding[1],
  short Data3,
  int Data4,
  char Padding2[3]
}
```

- Lösungen: Compiler-Flags, pragma-Trickserei
- Umordnen der Datenmember, so dass das Aligment passt

Data Structure Aligment (4)

- Umordnen:

```
struct Data{
  char Data1,
  char Data4,
  short Data2,
  int Data3
}
```

- Nun passt das Alignment
- Geht fasst immer
- Faustregel: Die großen als letztes...

Gliederung

Implizite Datenkonvertierung (1)

- In C++ lassen sich implizite Datenkonvertierungen verwenden
- Beispiel:

```
class Rational {
public:
  Rational (int num, int denom);
  ...
  // Convert from Rational to double
  operator double() const;
};
```

- Stellt Konvertierung zur Verfügung

```
Rational r(1, 2);     // r is 1/2
double d = 0.5 * r;
```

- `r` wird in `double` konvertiert und anschließend wird `d` berechnet

Implizite Datenkonvertierung (2)

- Schön und gut, jetzt wollen wie eine Zahl mit << ausgeben
- Ausgabe soll so aussehen:

```
%(Zähler) / (Nenner)
```

- Leider wurde vergessen den <<-Operator zu implementieren
- Folgendes Codefragment:

```
Rational r(1, 2);
cout << r;
```

- Was passiert?
- Code übersetzt fehlerfrei
- Ausgabe:

```
% 0.5
```

- Nicht überraschend, aber subtile Fehlerquelle!

Implizite Datenkonvertierung (3)

- Compiler sieht kein << für Klasse Rational
- Aber Rational kann nach double und für double gibt es ein <<
- **Dies verletzt das „Gesetz der geringsten Überraschungen“!!!**
- Single-Argument-Konstruktoren können auch Datenkonvertierungseigenheiten mit sich führen
- Beispiel:

```
Rational(int num = 0, int denom = 1);
```

- Definiert Default-Werte für die Argumente
- Erlaubt es `ints` in `Rational` zu konvertieren

```
Rational r1(3);     // r1 = 3/1
Rational r2 = 5;    // r2 = 5/1
r1 = 6;             // r1 = Rational(6)
```

Implizite Datenkonvertierung (4)

- Wir haben eine Array-Klasse:

```
class Array {
  ...
public:
  Array(int lowBound, int highBound);
  Array(int size);
  int& operator[](int index);
};
```

- Vergleich:

```
Array a(10), b(10);
...
for (int i = 0; i < 10; i++) {
  if (a == b[i]) {
    ...       // Do stuff!
  }
}
```

Implizite Datenkonvertierung (5)

- Statt `a == b[i]` war hier `a[i] == b[i]` gemeint
- Der Code übersetzt!!!
- Compiler schätzt `a == Array(b[i])`
- Ineffizient und falsch!!!
- In solchen Fällen der Datenkonvertierung sollte der Compiler sich beschweren!
- Verbieten von impliziten Konvertierungen
- Konstruktoren können mit dem Schlüsselwort `explicit` versehen werden

```
explicit Array(int size);
```

- C++ verwendet nun keine impliziten Konvertierungen mehr
- Auch in der Klasse `Rational` anwendbar

```
explicit Rational(int num = 0, int denom = 1);
```

- Nun keine Konvertierung nach `int` mehr möglich

Sichtbarkeit bei Ableitungen (1)

»Ich fürchte, es handelt sich hierbei um eine Erbkrankheit.« - »Gut, Herr Doktor. Dann schicken Sie doch bitte Ihre Rechnung an meinen Vater.«

- Bisher haben wir immer das Schlüsselwort `public` beim Ableiten verwendet
- Das legt nahe, das man da auch `private` oder `protected` schreiben kann
- Sollte Euch schon mal aufgefallen sein, wenn ihr das `public` vergessen habt
- Privates Ableiten:
- Alle öffentlichen Element der Basisklasse werden in der Abgeleiteten Klasse privat
- Innerhalb einer Klasse kann man weiterhin auf die vorher erreichbaren Elemente der Oberklasse zugreifen
- Aber Zugriffe von außen sind nicht möglich
- `protected` zeigt ähnliches Verhalten

Sichtbarkeit bei Ableitungen (2)

```
class Base
{
private:
    int pri;
protected:
    int pro;
public:
    int pub;
};
```

```
class Derived : Base // Attention! private!!
{
    int f1() { return pri; }  // Won't work
    int f2() { return pro; }  // Works
    int f3() { return pub; }  // Works
};
```

```
int main()
{
    Base a;
    Derived b;
    i = a.pub;   // Ok, public member
    i = b.pub;   // Won't work.
    a = b;       // Won't work.
}
```

Private Konstruktoren

- Man kann Konstruktoren als `private` deklarieren
- Die Klasse kann dann nicht außerhalb der Klasse instanziert werden
- Anwendungsbeispiel: Singletons

```
class Singleton {

  Singleton::Singleton() {}
  Singleton *m_singleton;
public:

  static Singeton& getInstance() {
     if (!m_instance) {
        m_instance = new Singleton;
     }
     return *m_singleton;
  }
};
```

Up- und Downcasting (1)

- Ein abgeleitetes Objekt ist gewöhnlich größer als ein Objekt der Basisklasse
- Wenn ein Objekt angelegt wird, muss das System seine Größe wissen
- Upcasting entfernt Informationen über das abgeleitete Objekt
- „Object Slicing“

```
class A
{
    int a_var;
};

class B : public A
{
    int b_var;
};
```

```
B b;
A a = b;
// b.b_var is not copied to a

B b2;
A &a = b2;
a = b;
// b.b_var is not copied
// to b2.b_var.
```

Up und Downcasting (2)

- Upcasting wird verwendet, um polymorphes Verhalten zu erzeugen:

```
Fruit f;
Orange o;
f = o;
```

- Downcasting ist unter bestimmten Bedingungen auch möglich
- Gefährlich, denn es wird zu einem spezifischerem Typ gecastet
- Kann unter Umständen nicht korrekt sein
- Sollte nicht benutzt werden, es sei denn es ist unbedingt nötig
- Downcasts geschehen mit `static_cast<>`

Up und Downcasting (3)

- CRTP:

```
template <class Derived> struct Base
{
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }

    static void static_func() {
        Derived::static_sub_func();
    }
};

struct Derived : Base<Derived>{
    void implementation();
    static void static_sub_func();
};
```

Up und Downcasting (4)

- Mit `dynamic_cast<>` kann man prüfen, ob die Zuweisung erfolgreich war
- Beispiel:

```
Orange* pOrange = dynamic_cast<Orange*>(pFruit);
if ( pOrange != 0 ) …   // success
```

- `const_cast<>()` Beispiel:

```
const char *c = "sample text";
const char *d = "new text";
c = const_cast<char*>d;
```

- `reinterpret_cast<>` casted einfach alles:

```
class A {};
class B {};
A *a = new A;
B *b = reinterpret_cast<B*>(a);
```

Mehrfachvererbung (1)

- C++ unterstützt Mehrfachvererbung
- Vererbungskonzept ist an Smalltalk angelegt
- Smalltalk verlangt, dass alles von einem Basis-Objekt abgeleitet wird
- Mehrfachvererbung wurde benötigt, bevor man Templates hatte
- Hauptargument: Container

```
class Container { …
private:
    Object* m_pData;
};
```

- Jeder Typ, der den Container benutzen soll, erbt zum Basisobjekt zusätzlich den Container

```
class Derived : public Container, public Base {…};
```

Mehrfachvererbung (2)

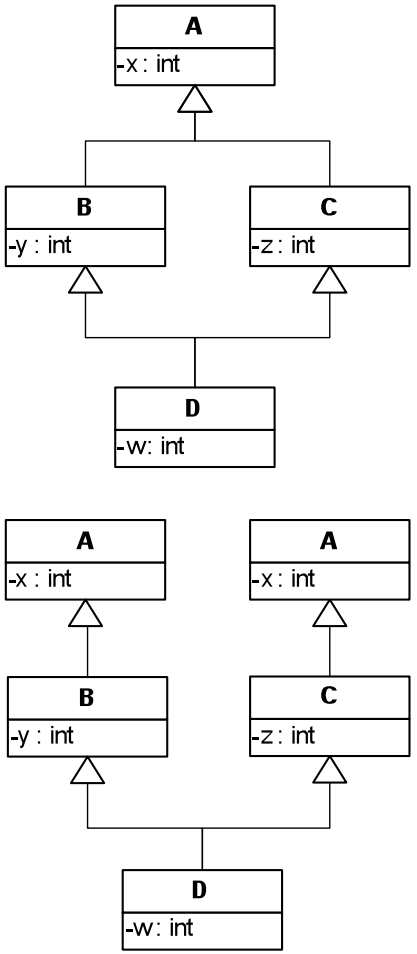
- Mit Mehrfachvererbung kann man Interfaces in C++ realisieren
- Man leitet von einem Interface-Objekt ab
- Die Interface-Objekte enthalten nur pure virtual Methoden
- In manchen Sprachen sind Interfaces Teil des Sprachkonzepts
- Problem bei Mehrfachvererbung: Alles was vererbt wird, taucht in der abgeleiteten Klasse auf
- Dadurch kann es zu Namenskollisionen kommen:

```
class Engine {
public:
Bearing bearing;
};

class Coach {
public:
Bearing bearing;
};
```

```
class car : public Engine,
    public Coach
{
    Bearing a = Engine::bearing;
    Bearing b = Coach::bearing;
}
```

Mehrfachvererbung (3)

- Diamantvererbung (Deadly Diamond Of Death):



Mehrfachvererbung (4)

```
class UltimateBase {
private:
   int value;
public:
   UltimateBase(int value)
     : value(value) {}

  int print() const
  {
    return value;
  }
};

class Derived : public Base1,
  public Base2 {
public:
  Derived(int a, int b)
    : Base1(a), Base2(b) {}
};
```

```
class Base1 : public UltimateBase {
public:
 Base1(int value) :
   UltimateBase(value) {}
};

class Base2 : public UltimateBase {
public:
 Base2(int value) :
   UltimateBase(value) {}
};

int main()
{
  Derived d(3,4);
  cout<<d.Base1::print()<<'\n';
  cout<<d.Base2::print()<<'\n';
}
```

Mehrfachvererbung (5)

- Oft können / wollen wir nicht selber entscheiden, welche Methode aufgerufen werden soll
- Wir wollen in der Regel `UltimateBase` nur einmal haben
- Und nun?
- Wir können mit `virtual` ableiten:

```
class Base2 : virtual public UltimateBase {
public:
  Base2(int value) : UltimateBase(value){}
};
```

- Nun gibt es nur eine `print()`-Methode
- Aber was gibt sie aus?
- 3. Die Klassen wurde zuerst mit 3 initialisiert
- Wer zuerst kommen mahlt zuerst...
- Es herrscht Uneinigkeit, ob man Mehrfachvererbung benötigt
- Virtuelle Basisklassen deuten oft auf ein Designproblem hin

Mehrfachvererbung (6)

- Vermeiden von Mehrfachverarbeitung
- Beantworten Sie folgende zwei Fragen:
  - Muss man beide Basisklassen in der abgeleiteten Klasse zur Verfügung haben?
  - Muss man zu beiden Basisklassen upcasten können?
- Wenn eine der beiden Fragen mit “nein” beantwortet wurde, dann lässt sich Mehrfachvererbung vermeiden:
  - Verwenden Sie Zusammensetzung: Leiten Sie das Interface von der Basisklasse, die am häufigsten benötigt wird ab und fügen Sie die Definition der anderen Klasse als Datentyp / Member ein.
  - Falls man Sie zu einer Basisklasse als Funktionsparameter upcasten müssen, benutzen Sie Zusammensetzung und einen Konvertierungsoperator, um eine Referenz auf das andere Memberobjekt zu erzeugen.

Kompatibilität C / C++ (1)

- C erlaubt es, einen void-Pointer einen anderen Pointertyp ohne casting zuzuweisen:

```
void* ptr;
int *i = ptr;
int *j = malloc(sizeof(int) * 5);
/* Implicit conversion from void* to int* */
```

- g++ wirft dann eine Warning, einige Compiler aber schon einen Error
- Man kann das Problem lösen, indem man explizit castet:

```
void* ptr;
int *i = (int *) ptr;
int *j = (int *) malloc(sizeof(int) * 5);
```

Kompatibilität C / C++ (2)

- Kollisionen mit den C++-Schlüsselwörtern
- Beispiel:

```
struct template
{
    int new;
    struct template* class;
};
```

- Compiliert mit einem C-Compiler, nicht mit C++
- In C++ / C99 dürfen keine Variablen innerhalb eines `switch`-Blocks deklariert werden:

```
switch(foo) {
    case 1:
        int i = 5; // Error
        break;
}
```

Kompatibilität C / C++ (3)

- Ähnliches gilt für `goto`:

```
void fn(void)
{
  goto flack;
  int i = 1;
flack:
  ;
}
```

- Weitere wichtige Probleme:
  - C++ erlaubt verschachtelte `typedefs`
  - C++ erlaubt nicht mehr als einen Underscore in Variablennamen
  - Einige C++-Standardfunktionen geben `const`-Objekte zurück
  - Namesräume bei `structs`
  - „inplicit function declarations“ sind in C++ nicht erlaubt

Gui-Programmierung mit Qt4

- Qt4 ist ein Framework zur plattformunabhängigen GUI-Programmierung
- OpenSource für nichtkommerzielle Anwendungen
- Viele Erweiterungen / Plugins
- Eigene Tools:
  - QtDesginer
  - QtLinguist
  - QtCreator
  - qmake
  - uic
  - moc
  - qrc
  - ...

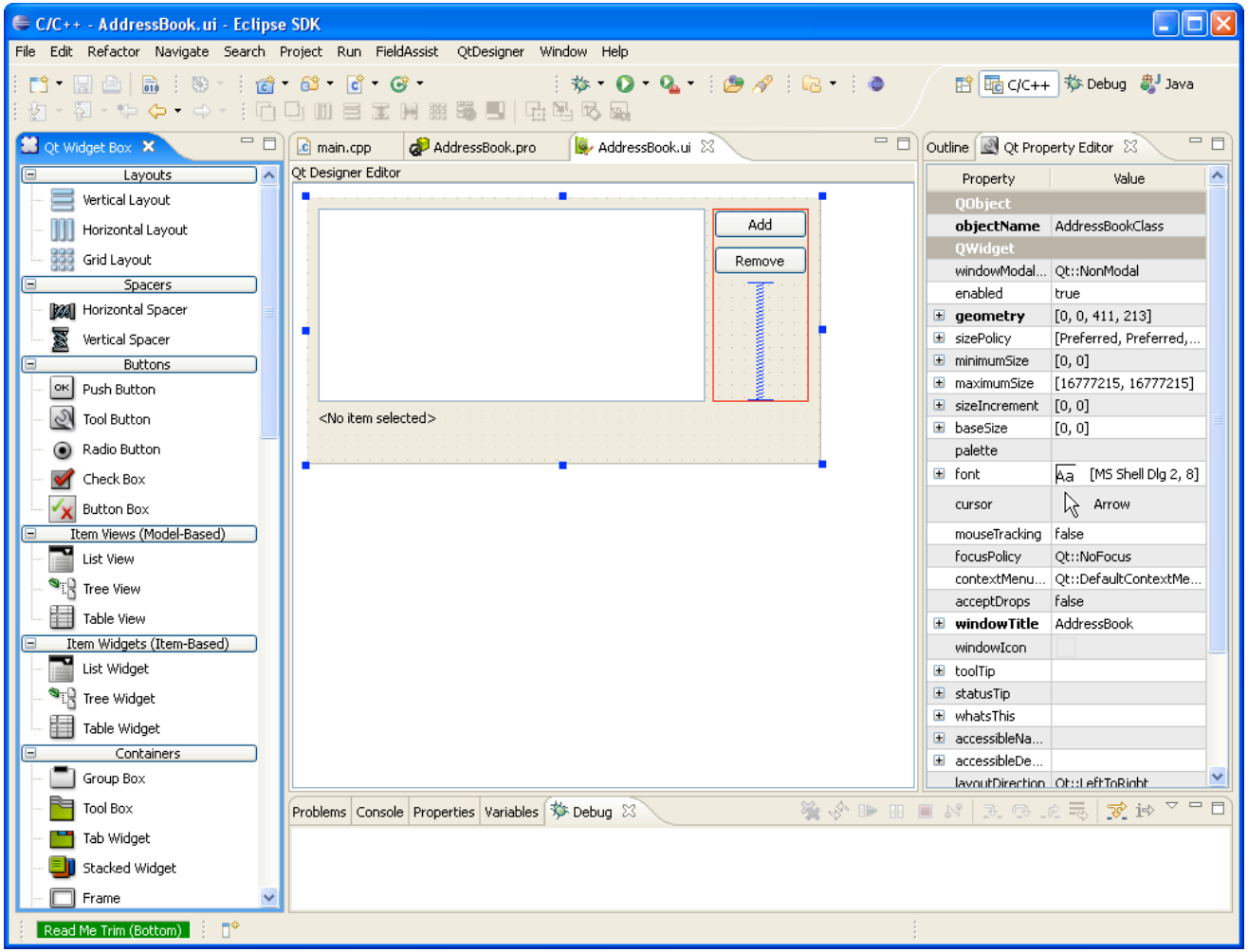
Qt4 „Hello World“

```
#include <QtGui>

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

  QApplication app(argc, argv);
  QMainWindow mainWindow;
  mainWindow.setWindowTitle("Hello World");
  mainWindow.resize(320, 200);
  mainWindow.show();
  return app.exec();
}
```

- Erzeugt ein leeres Fenster mit dem Titel „Hello World“
- Eigene Oberflächen lassen sich am einfachsten mit dem Designer erzeugen

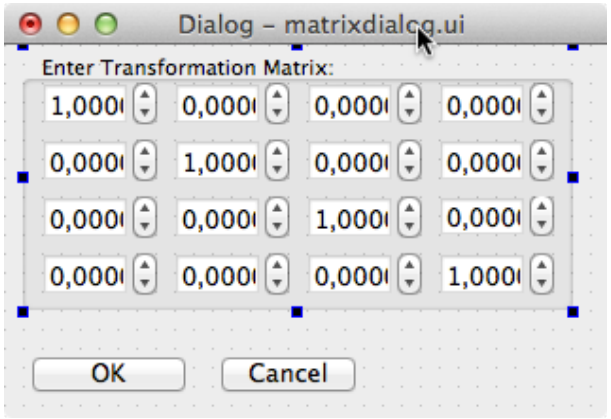
Qt Designer (1)

- Erlaubt interaktiv die Bedienoberflächen von Fenstern, Dialogen usw. zu gestalten:



Qt Designer (2)

- Die Beschreibung der Oberflächen wird in einer .ui-Datei abgelegt
- Das Tool `uic` erzeigt daraus C++-Header-Dateien
- Beispiel:




```
class Ui_MatrixDialog
{
public:
    QGroupBox *groupBox;
    QDoubleSpinBox *doubleSpinBox00;
    QDoubleSpinBox *doubleSpinBox01;
...

void setupUi(QDialog *MatrixDialog)
{
 doubleSpinBox00 = new QDoubleSpinBox(groupBox);
        doubleSpinBox00->setObjectName(QString::fromUtf8("doubleSpinBox00"));
        doubleSpinBox00->setGeometry(QRect(10, 20, 62, 22));
        doubleSpinBox00->setCursor(QCursor(Qt::WaitCursor));
        doubleSpinBox00->setDecimals(4);
        doubleSpinBox00->setMinimum(-1e+09);
        doubleSpinBox00->setMaximum(1e+09);
        doubleSpinBox00->setValue(1);
...
}
```

Qt Designer (3)

- Der Code kann dann genutzt werden, um Oberflächen für Widgets zu erzeugen
- Beispiel: Modaler Dialog

```
// Create new dialog object with main window as
// parent
QDialog matrix_dialog(mainWindow);

// Create user interface
Ui::MatrixDialog matrix_dialog_ui;
matrix_dialog_ui.setupUi(&matrix_dialog);

// Execute as modal dialog
int result = matrix_dialog.exec();
```

- Ergebnis: `QDialog::Accecpted` oder `QDialog::Rejected`
- Solange `matrix_dialog_ui` nicht freigegeben wurde, kann man auf die Inhalte der Gui-Elemente zugreifen

Qt Designer (4)

- Auswertung der Gui-Elemente:

```
if(result == QDialog::Accepted){
     Matrix4 m;
     m.set(0 , matrix_dialog_ui.doubleSpinBox00->value());
     m.set(1 , matrix_dialog_ui.doubleSpinBox01->value());
     m.set(2 , matrix_dialog_ui.doubleSpinBox02->value());
     m.set(3 , matrix_dialog_ui.doubleSpinBox03->value());
     ...
}
```

Qt4 Eigene Widgets (1)

- Oftmals ist es nützlich, eigene Unterklassen von bereits existierenden Widgets zu erstellen
- Beispiel: Eigenes Hauptfenster

```
class MyMainWindow : public QMainWindow, public Ui::MainWindow
{
  Q_OBJECT

public:
  MyMainWindow (QMainWindow *parent = 0);
  ~MyMainWindow();
};
```

Signaling etc.

Eigene Fensteroberfläche

Stellt Basisfunktionalität bereit

Qt4 Eigene Widgets (2)

- Implementierung der eigenen Funktionalität:

```
MyMainWindow::MyMainWindow(QMainWindow *parent) :
QMainWindow(parent)
{
        setupUi(this);
}
```

Qt4 Signaling (1)

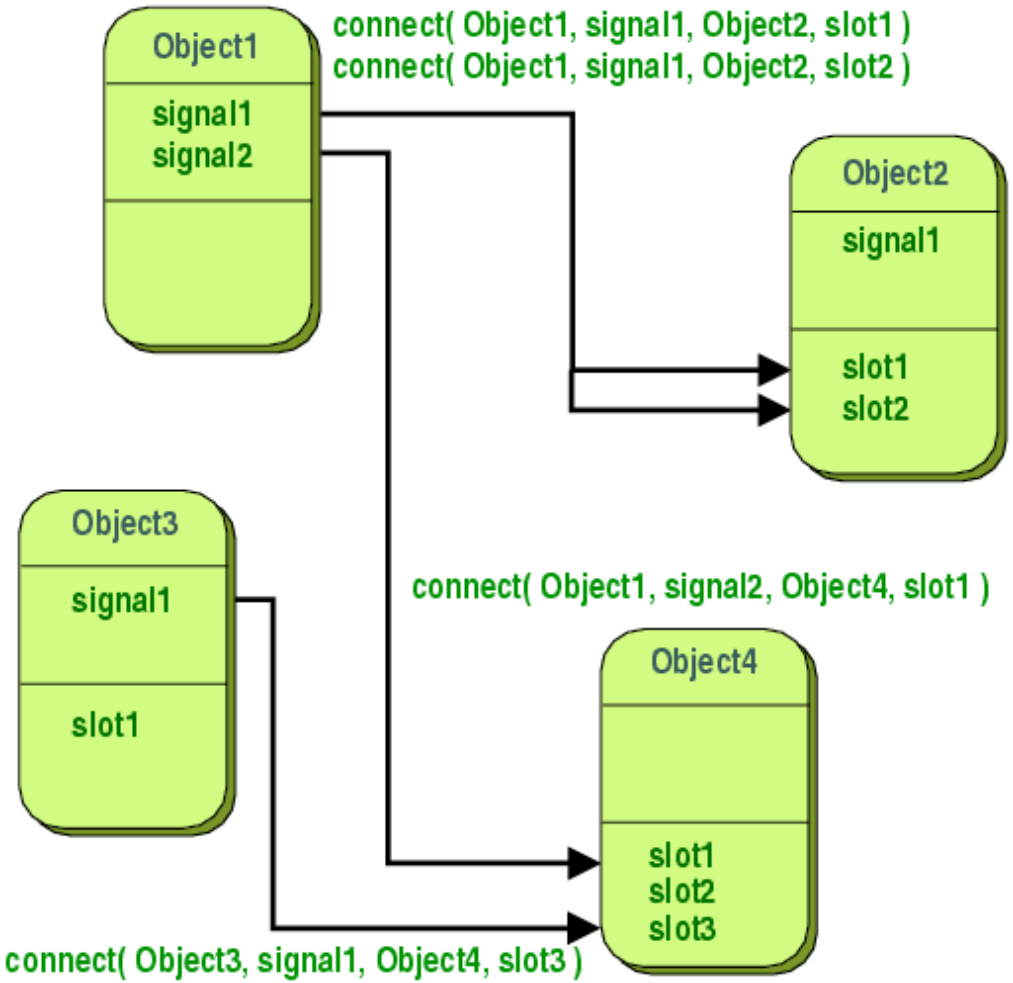
- Signale und Slots realisieren ein eventbasierte Callbacks
- Ein Objekt kann ein Signal zu einem passenden Slot in einem anderen Objekt schicken
- Vorraussetzung: Signaturen passen
- Eigene „Syntax“, realisiert über Makros:

```
class EventHandler : public QObject{

   Q_OBJECT

public:
   EventHandler(ViewerWindow*);
   ...

public slots:
   void editObjects(QListWidgetItem* item);
   void objectSelected(QListWidgetItem* item);
   void touchpad_transform(int, double);

signals:
   void updateGLWidget();
   ...
}
```

Objekt unterstützt Signaling

Signal und Slot Deklarationen

Qt4 Signaling (1)



Quelle: `http://developer.qt.nokia.com`

Qt4 Signaling (2)

- Verbinden von Events

```
QObject::connect(m_listWidget, SIGNAL(itemChanged(QListWidgetItem *)),
                 m_eventHandler, SLOT(action_editObjects(QListWidgetItem *)));

QObject::connect(m_listWidget, SIGNAL(itemClicked(QListWidgetItem *)),
                 m_eventHandler, SLOT(action_objectSelected(QListWidgetItem *)));

QObject::connect(m_touchpad, SIGNAL(transform(int, double)),
                 m_eventHandler, SLOT(touchpad_transform(int, double)));
```

- Eigenes Auslösen von Events:

```
void EventHandler::touchpad_transform(int mode, double d){
  objectHandler->transformSelectedObject(mode, d);
  emit(updateGLWidget());
}
```

- Hier sogar als Verkettung

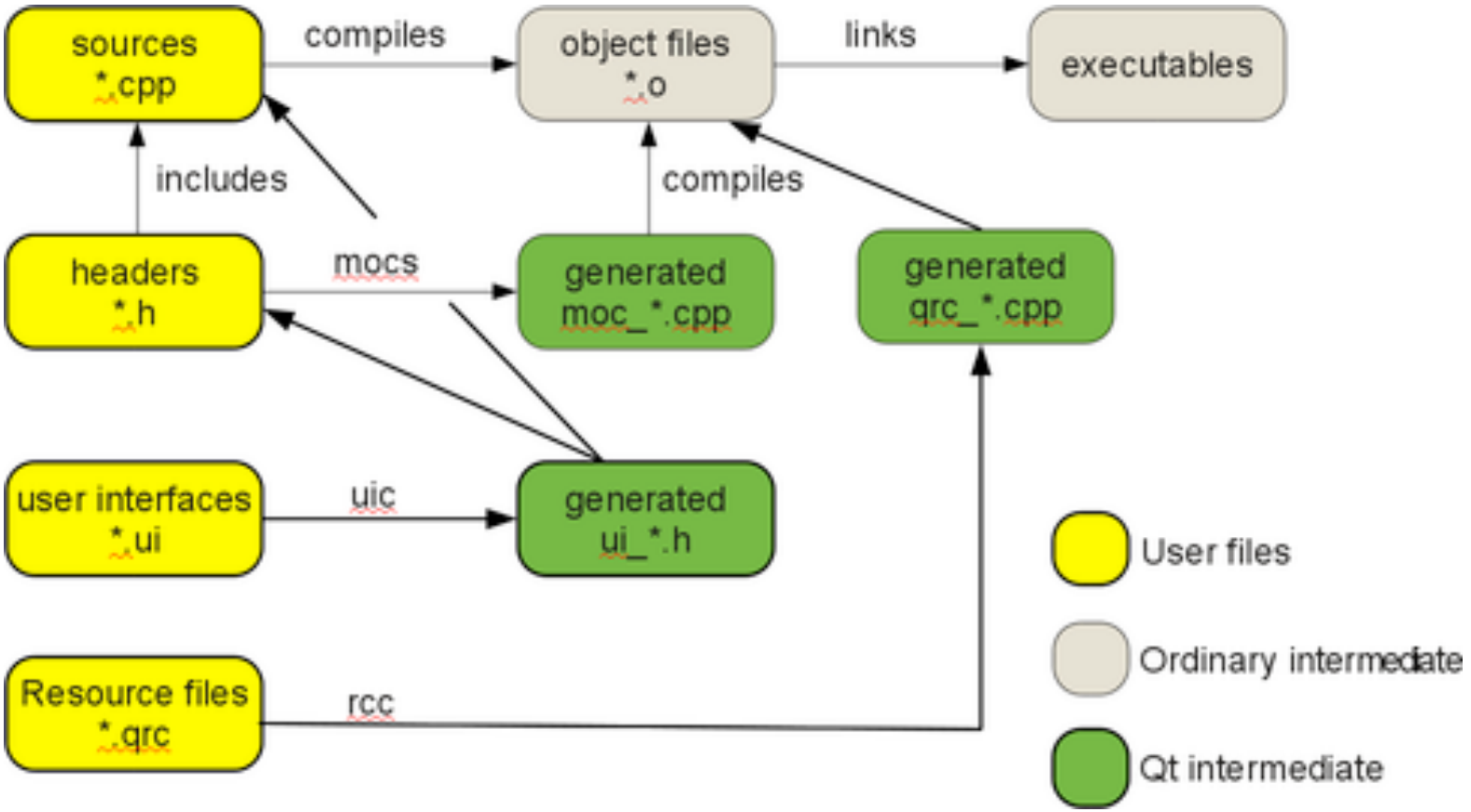
Qt4 Signaling (3)

- `Q_OBJECT` und die `signal` / `slot` makros sind Meta-Code
- Die dazugehörige Lokig muss in C++-Code übersetzt werden
- „Meta Object Code“
- Dazu gibt es das Tool `moc` „Meta Object Comiler“
- Jeder Header, der ein `Q_OBJECT` Makro enthält muss zusätzlich mit `moc` compiliert werden
- `moc` erzeugt C++-Code, der dann von einem C++-Compiler compiliert und zum Programm dazu gelingt werden
- Vergisst man dass, erzeugt man Linker-Fehler...
- Möglichkeiten zur Handhabung:
- Nutzung von qmake (nicht hier)
- cmake unterstützt die Code-Generierung mit moc:

```
set(QVIEWER_MOCS app/ViewerApplication.h ...)
qt_wrap_cpp(qviewer QVIEWER_MOC_SRC ${QVIEWER_MOCS})
```

analog für .ui-Dateien!

Qt4 - Codeerzeugung Zusammenfassung



Quelle: `http://developer.qt.nokia.com`

Qt4 Actions und Menüs

- Menueinträg, Toolbar-Buttons etc. werden mit Actions assoziiert
- Verhindert, dass man sich selber um Synchronisierung (z.B. beim Deaktivieren von Elementen) kümmen muss
- Actions können im Designer angelegt werden (s. Übung!)
- Oder natürlich auch per Hand:

```
MainWindow::createMenu()
{

   QAction *quit = new QAction("&Quit", this);

   QMenu *file;
   file = menuBar()->addMenu("&File");
   file->addAction(quit);

   connect(quit, SIGNAL(triggered()), qApp, SLOT(quit()));

}
```